Simulações financeiras em GPU

Thársis Tuani Pinto Souza

Dissertação de Mestrado apresentada
ao
Instituto de Matemática e Estatística
da
Universidade de São Paulo
para
obtenção do título
de
Mestre em Ciências

Programa: Ciência da Computação
Orientador: Prof. Dr. Walter F. Mascarenhas

São Paulo, abril de 2013

Simulações financeiras em GPU

Esta dissertação contém as correções e alterações
sugeridas pela Comissão Julgadora durante a defesa
realizada por Thársis Tuani Pinto Souza em 26/04/2013.
O original encontra-se disponível no Instituto de
Matemática e Estatística da Universidade de São Paulo.

Comissão Julgadora:

- Prof. Dr. Walter F. Mascarenhas - IME-USP
- Prof. Dr. Herbert Kimura - Universidade de Brasília
- Prof. Dr. Marco Dimas Gubitoso - IME-USP

Resumo

SOUZA, T. T. P. **Simulações financeiras em GPU.** Dissertação (Mestrado) - Instituto de Matemática e Estatística, Universidade de São Paulo, São Paulo, 2013.

É muito comum modelar problemas em finanças com processos estocásticos, dada a incerteza de suas variáveis de análise. Além disso, problemas reais nesse domínio são, em geral, de grande custo computacional, o que sugere a utilização de plataformas de alto desempenho (HPC) em sua implementação. As novas gerações de arquitetura de *hardware* gráfico (GPU) possibilitam a programação de propósito geral enquanto mantêm alta banda de memória e grande poder computacional. Assim, esse tipo de arquitetura vem se mostrando como uma excelente alternativa em HPC.

Com isso, a proposta principal desse trabalho é estudar o ferramental matemático e computacional necessário para modelagem estocástica em finanças com a utilização de GPUs como plataforma de aceleração. Para isso, apresentamos a GPU como uma plataforma de computação de propósito geral. Em seguida, analisamos uma variedade de geradores de números aleatórios, tanto em arquitetura sequencial quanto paralela. Além disso, apresentamos os conceitos fundamentais de Cálculo Estocástico e de método de Monte Carlo para simulação estocástica em finanças.

Ao final, apresentamos dois estudos de casos de problemas em finanças: "Stops Ótimos" e "Cálculo de Risco de Mercado". No primeiro caso, resolvemos o problema de otimização de obtenção do ganho ótimo em uma estratégia de negociação de ações de "Stop Gain". A solução proposta é escalável e de paralelização inerente em GPU. Para o segundo caso, propomos um algoritmo paralelo para cálculo de risco de mercado, bem como técnicas para melhorar a solução obtida. Nos nossos experimentos, houve uma melhora de 4 vezes na qualidade da simulação estocástica e uma aceleração de mais de 50 vezes.

Palavras-chave: Métodos Matemáticos em Finanças, Finanças Quantitativas, Modelagem Matemática, Computação Paralela, GPU, GPGPU, Números Aleatórios, Simulação Estocástica, Simulação de Equações Diferencias Estocásticas, Stops, Risco de Mercado, VaR, Value-at-Risk, Precificação de Opções

Abstract

SOUZA, T. T. P. **Finance and Stochastic Simulation on GPU.** Dissertation (M.Sc.) - Instituto de Matemática e Estatística, Universidade de São Paulo, São Paulo, 2013.

Given the uncertainty of their variables, it is common to model financial problems with stochastic processes. Furthermore, real problems in this area have a high computational cost. This suggests the use of High Performance Computing (HPC) to handle them. New generations of graphics hardware (GPU) enable general purpose computing while maintaining high memory bandwidth and large computing power. Therefore, this type of architecture is an excellent alternative in HPC and comptutational finance.

The main purpose of this work is to study the computational and mathematical tools needed for stochastic modeling in finance using GPUs. We present GPUs as a platform for general purpose computing. We then analyze a variety of random number generators, both in sequential and parallel architectures, and introduce the fundamental mathematical tools for Stochastic Calculus and Monte Carlo simulation.

With this background, we present two case studies in finance: "Optimal Trading Stops" and "Market Risk Management". In the first case, we solve the problem of obtaining the optimal gain on a stock trading strategy of "Stop Gain". The proposed solution is scalable and with inherent parallelism on GPU. For the second case, we propose a parallel algorithm to compute market risk, as well as techniques for improving the quality of the solutions. In our experiments, there was a 4 times improvement in the quality of stochastic simulation and an acceleration of over 50 times.

Keywords: Mathematical Methods in Finance, Quantitative Finance, Mathematical Modeling, Parallel Computing, GPU, GPGPU, Random Numbers, Stochastic Simulation, Simulation of Stochastic Differential Equations, Stops, Market Risk, VaR, Value-at-Risk, Options Pricing

Agradecimentos

Em primeiro lugar, à minha amada família: pai, Isaias de Souza Neto; mãe, Maria de Lourdes Pinto de Lacerda Souza; irmão, Heli Samuel Pinto Souza.

Também devo lembrar de inúmeros professores que foram fundamentais na minha formação intelectual. Ao Prof. Dr. Carlile Campos Lavor da Unicamp, responsável pela minha iniciação científica e que me deixou apenas lembranças de sabedoria e gentileza. Ao Prof. Dr. Orlando Stanley Juriaans da USP, que me motivou a lecionar com sua paixão pelo ensino e educação. À Profa. Dra. Yoshiko Wakabayashi, por ter me recebido no IME-USP em meu primeiro ano, sempre cordial e brilhante. Aos professores doutores Marco Dimas Gubitoso, Saulo Rabello Maciel de Barros e Marcel Jackowski pelas críticas em meu exame de qualificação de mestrado e pelas ricas discussões em inúmeros seminários em nosso grupo de estudo. Finalmente, ao Prof. Dr. Walter Mascarenhas, por ter aceito o desafio de me orientar e pelos ensinamentos em tantas áreas de conhecimento.

Não poderia também deixar de agradecer àqueles colegas de trabalho que me motivaram a completar o mestrado acadêmico: Gilberto Burgert, Sandro M. Manteiga e Juan Carlos Ruilova Terán.

Finalmente, agradeço aos amigos e colegas pelas revisões de rascunhos e sugestões: Tiago Montanher, Sandro M. Manteiga, André Valloto Lima, Filipe Polizel e Thiago Winkler

Sumário

Lista de Abreviaturas

API	Interface de Programação de Aplicativos (*Application Programming Interface*)
Cg	*C for Graphics*
CUDA	Arquitetura de Dispositivo de Computação Unificada (*Computer Unified Device Architecture*)
CMRG	*Combined Multiple Recursive Generator*
CPU	Unidade Central de Processamento (*Central Processing Unit*)
DRAM	Memória de Acesso Aleatório Dinâmico (*Dynamic Random-Access Memory*)
ECC	Código de Correção de Erro (*Error Correcting Code*)
FLOPS	Operações de Ponto Flutuante por Segundo (*Floating-Point Operations per Second*)
FMA	Multiplicação e Adição Fundidas (*Fused Multiply Add*)
GDDR	*Graphic Double Data Rate*
GPU	Unidade de Processamento Gráfico (*Graphics Processing Unit*)
GPGPU	Computação de Propósito Geral em GPU (*General Purpose Computing on GPU*)
HLSL	*High Level Shader Language*
HPC	Computação de Alto Desempenho (*High Performance Computing*)
LCG	*Linear Congruential Generator*
LD\ST	Carga\Armazenamento (*Load\Store*)
LFS	*Linear Feedback Shift*
MAD	Multiplicação e Adição
MC	Monte Carlo
MIMD	Múltiplas Instruções e Múltiplos Dados (*Multiple Instructions Multiple Data*)
MISD	Múltiplas Instruções e Dado Único (*Multiple Instructions Single Data*)
MRG	*Multiple Recursive Generator*

NUMA	Acesso não Uniforme à Memória (*Nonuniform Memory Access*)
PCIe	*Peripheral Component Interconnect Express*
PL	Processador Lógico
PRNG	Gerador de Número Pseudo-aleatório (*Pseudorandom Number Generator*)
QRNG	Gerador de Número Quasi-aleatório (*Quasirandom Number Generator*)
RNG	Gerador de Número Aleatório (*Random Number Generator*)
SBD	Sequências de Baixa Discrepância
SFU	Unidade Especial de Função (*Special Function Unit*)
SIMD	Instrução Única e Dados Múltiplos (*Simple Instruction Multiple Data*)
SISD	Instrução Única e Dado Único (*Single Instruction Single Data*)
SMP	Multiprocessadores Simétricos (*Symmetric Multiprocessors*)
SWC	*Substract With Carry*
ULA	Unidade Lógica e Aritmética
UPF	Unidade de Ponto Flutuante
VGA	*Video Graphics Array*

Lista de Símbolos e Notação

B, B_t, W_t	Movimento Browniano
$c_X(t,s), c_X$	Função de covariância do processo X
$cov(X,Y)$	Covariância das variáveis aleatórias X e Y
$correl(X,Y)$	Correlação das variáveis aleatórias X e Y
$\mathbb{E}[X]$	Esperança da variável aleatória X
F	Função de distribuição acumulada de uma variável aleatória
μ	Esperança de uma variável aleatória
μ_X	Esperança da variável aleatória X
$\mathbb{N}$	Conjunto dos números naturais
$\mathcal{N}(\mu, \sigma^2)$	Distribuição normal com média μ e variância σ
ω	$\omega \in \Omega$, evento do espaço de eventos Ω
Ω	Espaço de eventos
P	Medida de probabilidade
ϕ	Função da distribuição normal padrão
$\mathbb{R}$	Linha real
$\mathbb{R}^n$	Espaço Euclidiano n-dimensional
σ	Desvio padrão
σ_X^2	Variância da variável aleatória X
$\sigma_X^2(t)$	Função de variância do processo X
$U(a,b)$	Variável aleatória uniforme em (a,b)
$var(X)$	Variância da variável aleatória X
X	Variável aleatória ou processo estocástico
$\bar{A}$	Representa o complementar do conjunto A
$\sim$	Se $X \sim \mathcal{N}$, significa que a variável aleatória X segue uma distribuição normal
$\stackrel{d}{=}$	Se $X \stackrel{d}{=} Y$, significa que X e Y (variáveis aleatórias, vetores de variáveis aleatórias, processos estocásticos) possuem a mesma distribuição de probabilidade
$a.s.$	quase sempre (*almost surely*)

Lista de Figuras

Lista de Tabelas

Lista de Algoritmos

Parte I

Introdução

Capítulo 1

Introdução

Avanços em computação e algoritmos estabelecem uma nova área interdisciplinar ao combinar finanças e ciência da computação. Com isso, a efetiva exploração de novos métodos computacionais tem ajudado as instituições financeiras em sua tomada de decisões e no seu gerenciamento de risco. Nesse contexto, é comum modelar problemas em finanças utilizando processos estocásticos, dada à incerteza de suas variáveis. Além disso, problemas reais nesse domínio são, em geral, de grande custo computacional. Isso sugere a utilização de plataformas de alto desempenho (HPC) em sua implementação. As novas gerações de arquitetura de GPU possibilitam a programação de propósito geral enquanto mantêm alta banda de memória e grande poder computacional. Assim, essa arquitetura é uma excelente alternativa em HPC. Em (Lee *et al.*, 2010a), os autores estudam a viabilidade da utilização de placas gráficas em simulações estocásticas e concluem:

> "*We believe the speedup we observe should motivate wider use of parallelizable simulation methods and greater methodological attention to their design.*"[1]

Assim, a proposta desse trabalho é estudar o ferramental matemático e computacional necessário para modelagem estocástica em finanças com a utilização de GPUs de propósito geral como plataforma de aceleração.

1.1 Aplicações

Com o recente advento de placas gráficas programáveis, sua aplicação tem sido realizada em diferentes domínios, como: otimização linear (Jung e O'Leary, 2009), (Spampinato e Elstery, 2009); resolução de equações diferenciais parciais (Egloff, 2010); algoritmos de ordenação (Satish *et al.*, 2009); grafos (Buluc *et al.*, 2010), (Dehne e Yogaratnam, 2010), (Harish e Narayanan, 2007); bioinformática (Stojanovski *et al.*, 2012), (Sadiq *et al.*, 2012) ou física (Muller e Frauendiener, 2013), (Saito *et al.*, 2012).

Em finanças, podemos citar aplicações como: precificação de produtos (Solomon *et al.*, 2010), (Pages e Wilbertz, 2010), (Pages e Wilbertz, 2012); cálculo de risco (Dixon *et al.*, 2009) ou econofísica (Preis, 2011). Além disso, há grandes instituições financeiras com casos de sucesso ao migrar sua arquitetura de HPC para GPU. As figuras 1.1 e 1.2 apresentam alguns exemplos. Como podemos ver, em ambos os casos, as instituições obtiveram uma aceleração considerável, com redução de custo e economia de energia.

1.2 Objetivos

Há diferentes níveis de análise de simulações estocásticas financeiras em GPU. Em nível mais baixo, é importante estudar a plataforma computacional utilizada, o modelo de computação paralela, as características da arquitetura alvo e considerações de otimização de desempenho da mesma.

[1] Acreditamos que a aceleração que observamos deve motivar a maior utilização de métodos de simulação paralelizáveis e uma atenção metodológica maior para esse tipo de modelagem.

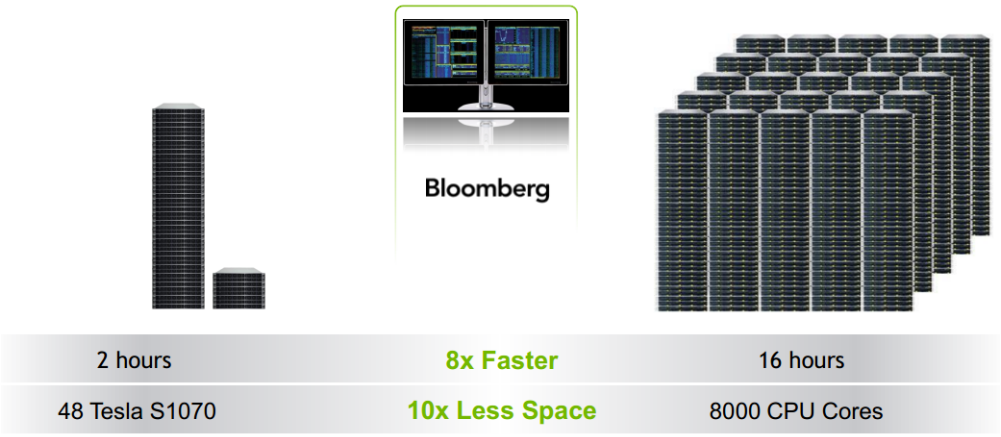

Figura 1.1: *Estudo de caso GPU Tesla: Precificação de produtos - Bloomberg (NVIDIA, 2009b).*

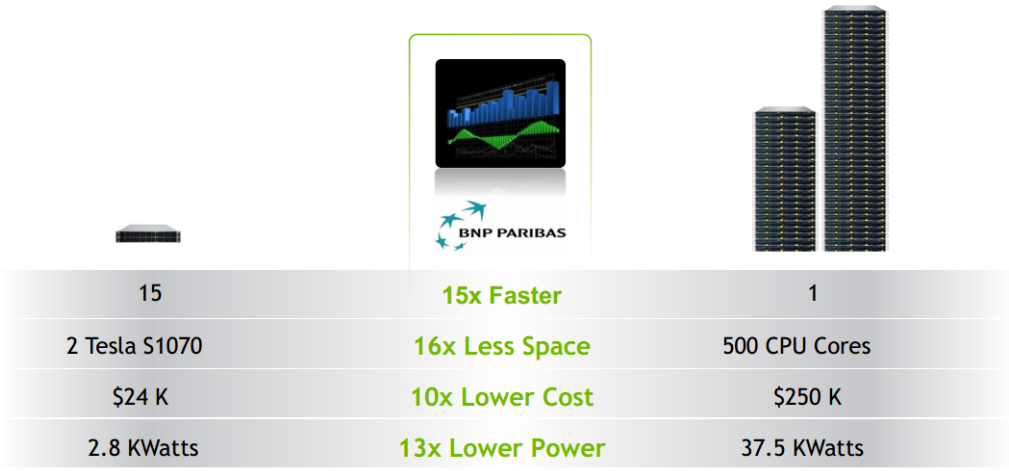

Figura 1.2: *Estudo de caso GPU Tesla: Precificação de produtos - BNP-Paribas (NVIDIA, 2009b).*

Em geral, simulações estocásticas são sensíveis à fonte de aleatoriedade que caracteriza a qualidade estatística de seus resultados. Dessa forma, também é de fundamental importância o estudo de geradores de números aleatórios e a portabilidade dos mesmos na arquitetura utilizada. Em nível mais alto, é necessário um ferramental matemático em processos estocásticos e simulação de Monte Carlo para resolução de grande parte dos problemas em finanças.

Este trabalho é um estudo em todos esses níveis. Ademais, apresentamos dois estudos de casos de problemas reais em finanças. Assim, os objetivos desse trabalho são os seguintes:

- Utilizar um modelo de *hardware* e programação para GPU, apresentando essa plataforma como um verdadeiro co-processador genérico.
- Estudo de geradores de números aleatórios sequenciais e paralelos.
- Apresentação de ferramental matemático fundamental para modelagem estocástica em finanças.
- Resolver problemas reais em finanças ao aplicar a teoria apresentada.

Na última parte do trabalho, apresentamos dois estudos de casos em finanças: "Stops Ótimos" e "Cálculo de Risco de Mercado". No primeiro problema, após uma revisão da literatura, propomos uma alternativa de resolução via simulação estocástica, em vista à escalabilidade da solução e portabilidade em GPU. Para o segundo problema, propomos um algoritmo em GPU com objetivo de aceleração significativa, bem como técnicas de melhoria da qualidade da solução obtida.

1.3 Organização do Trabalho

Além da introdução, o presente trabalho é organizado em quatro partes e um apêndice. Na segunda parte, estudamos a arquitetura computacional de GPU. Para isso, apresentamos modelos tradicionais de computação paralela no Capítulo 2; realizamos uma introdução à computação de propósito geral em GPU no Capítulo 3; estudamos técnicas de otimização nessa arquitetura no Capítulo 4 e finalizamos com um estudo de geradores de números aleatórios e sua portabilidade para GPU. A terceira parte apresenta o ferramental básico para modelagem estocástica em finanças. Assim, na seção 6.1, discutimos fundamentos de Simulações de Monte Carlo. Em seguida, nas seções 6.2 e 6.3, abordamos processos estocásticos e resolução de Equações Diferenciais Estocásticas. Ao final, estudamos métodos práticos de solução numérica dessas equações, na seção 6.4. Na quarta parte descrevemos estudo de casos de dois problemas reais em finanças em uma plataforma de GPU. Na última parte, concluímos o trabalho.

Parte II

Arquitetura Computacional

Capítulo 2

Modelos de Computação Paralela

Computador paralelo é definido por (Almasi e Gottlieb, 1989) como uma coleção de elementos de processamento que cooperam e se comunicam para resolver grandes problemas de forma rápida. Nas próximas seções classificamos os principais tipos de computadores paralelos, abordamos suas arquiteturas de memória e, por fim, apresentamos maneiras de medir o seu desempenho.

2.1 Classificação de Computadores Paralelos

A taxonomia de Flynn (Flynn, 1972) é a maneira mais comum de se caracterizar arquiteturas de computadores paralelos. Segunda ela, há quatro categorias: SISD, SIMD, MISD e MIMD.

Em um sistema SISD (*Single Instruction Single Data*[1]), não há exploração de paralelismo. Nela cada instrução opera sobre um único fluxo de dados, como mostrado na figura 2.1. Essa classificação corresponde à arquitetura clássica de Von Neumann.

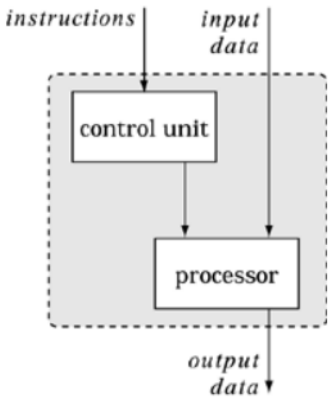

Figura 2.1: *Arquitetura SISD (Mattson et al., 2004)*

Arquiteturas SIMD (*Single Instruction, Multiple Data*[2]) são caracterizadas por um único fluxo de instruções que opera sobre múltiplos dados, como mostrado na figura 2.2. Essa arquitetura é ideal para aplicações que executam o mesmo conjunto de operações em um grande volume de dados. Arquiteturas de GPU se enquadram nessa classificação.

Arquiteturas MISD (*Multiple Instruction, Single Data*[3]) são conceitualmente definidas como de múltiplos fluxos de instrução que operam sobre os mesmos dados. Na prática, não é comum encontrar implementações desse modelo.

[1] em português: instrução única e dados únicos
[2] em português: instrução única e dados múltiplos
[3] em português: instruções múltiplas e dados únicos

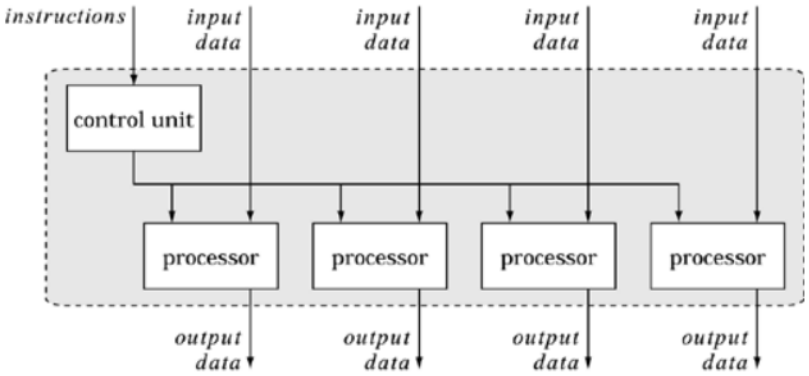

Figura 2.2: *Arquitetura SIMD (Mattson et al., 2004)*

Finalmente, a classificação MIMD (*Multiple Instruction, Multiple Data*[4]) diz respeito a unidades de processamento que executam múltiplas instruções ao mesmo tempo em diferentes conjuntos de dados. Como visto na figura 2.3, essa é a arquitetura mais genérica dentre as nomeadas por Flynn. Na verdade, todas as outras classificações são sub-casos da arquitetura MIMD.

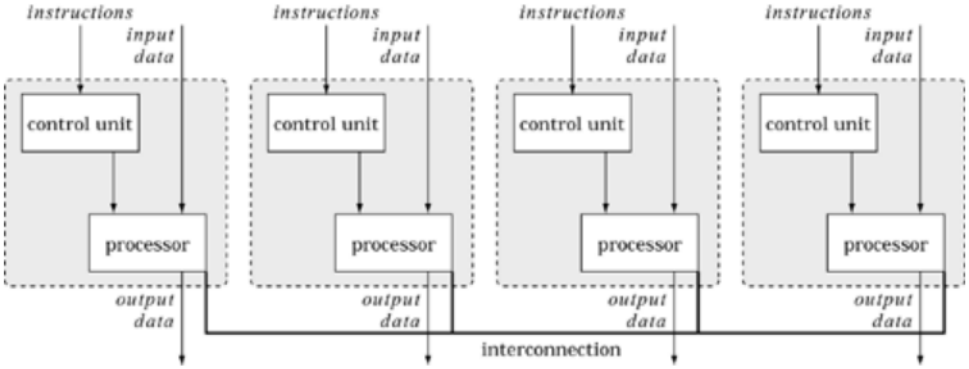

Figura 2.3: *Arquitetura MIMD (Mattson et al., 2004)*

2.2 Arquiteturas de Memória em Computadores Paralelos

A arquitetura MIMD da classificação de Flynn é muito genérica para ser utilizada na prática. Assim, ela é, geralmente, decomposta de acordo com a organização de memória.

2.2.1 Memória Compartilhada

Em um sistema de memória compartilhada, todos os processadores compartilham um mesmo espaço de endereçamento e se comunicam por meio de leitura e escrita em variáveis compartilhadas.

SMPs (*symmetric multiprocessors*[5]) são uma classe comum de sistemas de memória compartilhada. Como mostrado na figura 2.4, todos os processadores compartilham uma conexão com uma memória em comum e acessam todos os locais de memória em mesma velocidade. Nesses sistemas, em geral, não é necessário distribuir estruturas de dados ao longo dos múltiplos processadores, já que os mesmos acessam os dados de maneira compartilhada. Entretanto, como o canal de acesso aos dados é único, o aumento no número de processadores pode tornar a largura de banda em um

[4] em português: instruções múltiplas e dados múltiplos
[5] em português: multiprocessadores simétricos

fator limitante.

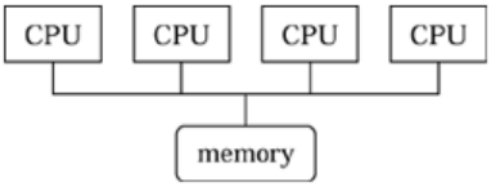

Figura 2.4: *Arquitetura SMP (Mattson et al., 2004)*

NUMA (*nonuniform memory access*[6]) é outra classe importante de sistemas de memória compartilhada. Como visto na figura 2.5, a memória é compartilhada. Contudo, alguns blocos de memória podem ficar fisicamente mais próximos com certos processadores e são naturalmente associados a eles. Isso pode reduzir o problema de largura de banda enunciado anteriormente em sistemas SMP e, portanto, pode permitir a utilização de um número maior de processadores.

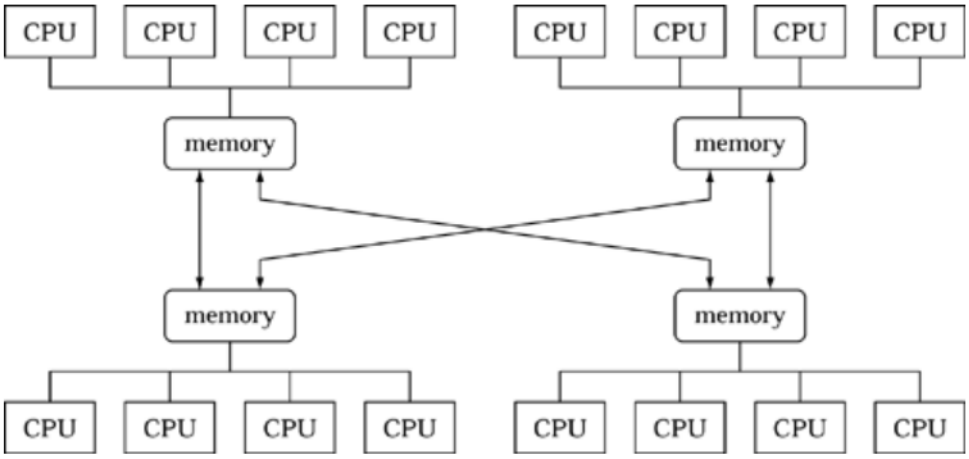

Figura 2.5: *Arquitetura NUMA (Mattson et al., 2004)*

2.2.2 Memória Distribuída

Em sistemas de memória distribuída, cada processo tem seu espaço de memória próprio e a comunicação é feita por meio de troca de mensagens[7]. A figura 2.6 apresenta um exemplo representativo de um computador de memória distribuída.

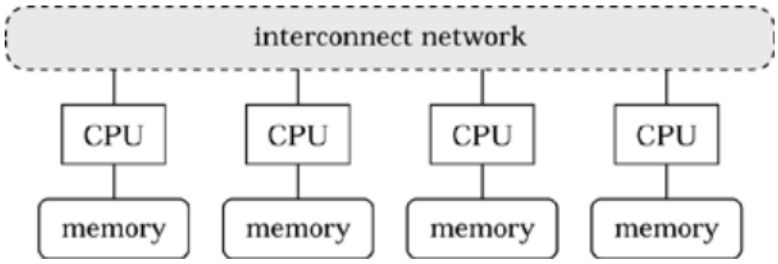

Figura 2.6: *Arquitetura de memória distribuída (Mattson et al., 2004)*

[6] em português: acesso não uniforme à memória
[7] no original: message passing

Dependendo da topologia e da tecnologia utilizadas para interconexão dos processadores, a velocidade de comunicação pode ser tão rápida quanto em uma arquitetura de memória compartilhada (e.g. supercomputadores com alta integração) ou ser de reduzido desempenho (e.g. em um agregado de computadores interconectados via rede *Ethernet*). É importante observar que esta arquitetura requer a configuração explícita da comunicação entre os processadores e a preocupação na distribuição de dados aos mesmos.

2.2.3 Sistemas Híbridos

Como o nome sugere, sistemas híbridos são aqueles que utilizam ambas as formas de organização de memória aqui citadas. Esses sistemas, geralmente, correspondem a agregados de nós com espaços de endereçamento separados, onde cada nó contem múltiplos processadores que compartilham memória.

2.3 Métricas de Desempenho

Em uma arquitetura sequencial, o tempo de execução de um programa se dá em função do tamanho da entrada e do espaço (memória). Já em arquiteturas paralelas, além dessas grandezas, o número de processadores e parâmetros específicos de comunicação da arquitetura alvo influenciam o tempo de execução. Isso implica que é necessário analisar algoritmos paralelos tendo em vista a arquitetura alvo particular.

Nesse sentido, há algumas métricas de desempenho comumente utilizadas em sistemas paralelos como:

- *Speedup*
- Eficiência[8]
- Escalabilidade[9]
- Taxa sustentada de FLOPS[10]

2.3.1 *Speedup*

Razão entre o tempo de execução do algoritmo executado em um único processador e o tempo de execução do mesmo algoritmo em múltiplos processadores:

$$S_p = \frac{T_1}{T_p}$$

onde,

- p é o número de processadores
- T_1 é o tempo de execução do algoritmo sequencial
- T_p é o tempo do algoritmo paralelo em p processadores

Em comparação com o número de processadores, o *speedup* pode ser classificado assim:

- $S_p = p$, *Linear speedup*
- $S_p < p$, *Sub-linear speedup*

[8] no original: *efficiency*
[9] no original: *scalability*
[10] no original: *sustained FLOP rate*

- $S_p > p$, *Super-linear speedup*

A Lei de Amdahl descreve o máximo *speedup* (S) esperado ao paralelizar uma certa porção de um programa sequencial:

$$S = \frac{1}{(1-P) + \frac{P}{p}}$$

onde P é a fração do tempo gasto pelo programa serial da parte do código que pode ser paralelizado e p é o número de processadores sobre o qual o código paralelizável é executado.

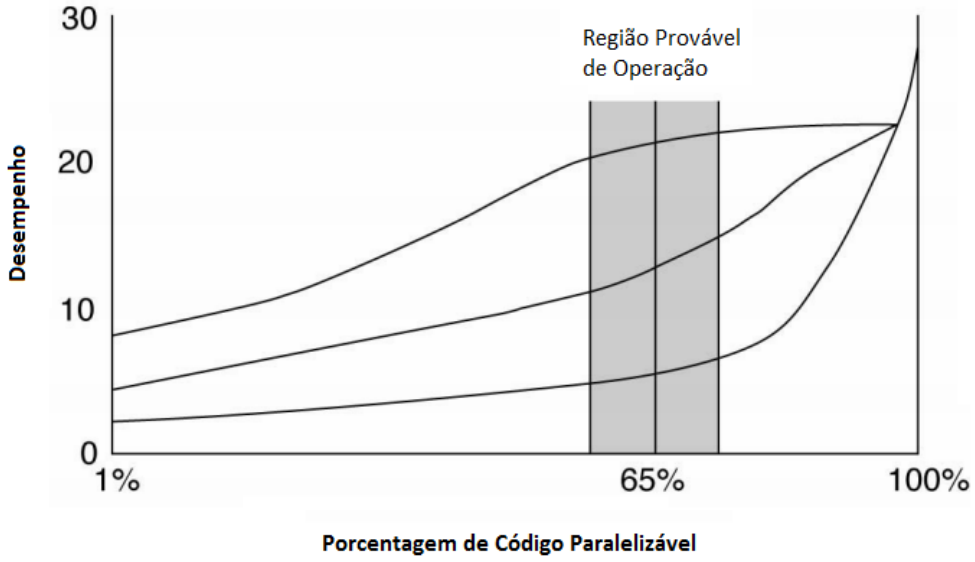

Figura 2.7: *Lei de Amdahl (Amdahl, 1967)*

2.3.2 Eficiência

Razão entre o *speedup* e o número de processadores:

$$E_p = S_p/p = \frac{T_1}{pT_p}.$$

Estima quão bem os processadores estão sendo utilizados, tendo em vista o tempo gasto em sobrecarga como: sincronização e troca de mensagens.

2.3.3 Escalabilidade

É a capacidade do algoritmo de resolver um problema n vezes maior em n vezes mais processadores:

$$Escalabilidade(p,n) = \frac{\text{Tempo para resolver um problema de tamanho } m \text{ em } p \text{ processadores}}{\text{Tempo para resolver um problema de tamanho } nm \text{ em } np \text{ processadores}}.$$

2.3.4 Taxa Sustentada de FLOPS

A taxa sustentada de FLOPS (*Floating-point Operations per Second*) mede a capacidade de execução de operações de ponto flutuante por segundo e quão bem uma implementação específica explora a arquitetura alvo. Vale notar que essa métrica, apesar de ser comumente utilizada em HPC, não indica, necessariamente, que um algoritmo é eficiente. É fato que um algoritmo alternativo com menor taxa sustentada de FLOPS pode resolver um mesmo problema mais rapidamente.

Capítulo 3

Computação em GPU

Nos últimos anos, a capacidade de *hardware* de processamento gráfico (GPU) teve um crescimento sem precedentes. As novas gerações de arquitetura de GPU possibilitam a programação de propósito geral enquanto mantêm alta banda de memória e grande poder computacional. Tal potencial de paralelismo é devido ao maior número de transistores dedicados a ULAs (Unidades Lógicas Aritméticas), em vez de controle de fluxo e cache, comparado com CPUs. Essa conformação é propícia para processamento massivamente paralelo. Porém requer maior conhecimento da arquitetura utilizada para aproveitamento da capacidade computacional. A figura 3.1 compara o crescimento de poder computacional de CPUs e GPUs.

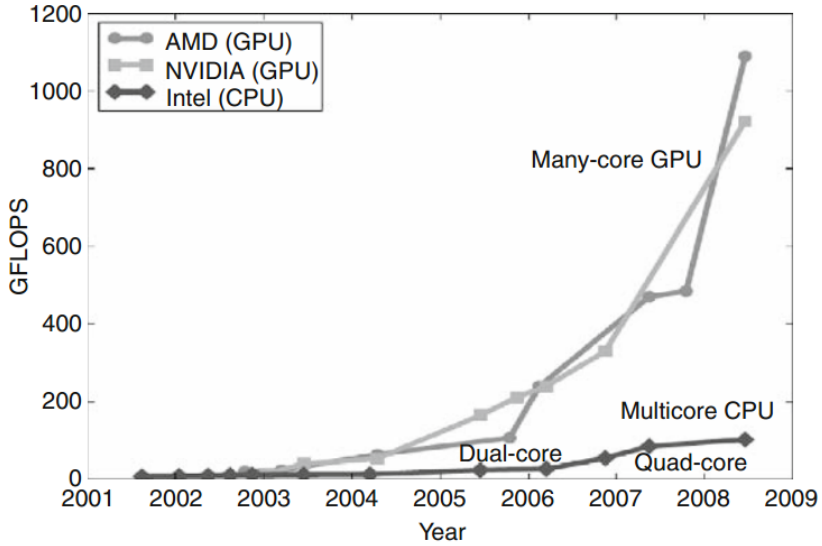

Figura 3.1: *Poder computacional (GFLOP/s) CPU X GPU (Kirk e Hwu, 2010)*

A figura 3.2 compara a utilização da área do chip entre CPU e GPU. A primeira arquitetura reserva maior área para controle de fluxo e cache e possui mais memória principal. Dessa forma, a CPU é mais apropriada para tarefas sequenciais. No caso da GPU, temos uma área maior dedicada para unidades lógicas aritméticas, o que resulta em maior capacidade de execução de operações de ponto flutuante. Dessa forma, essa arquitetura é mais adequada para tarefas com paralelismo de dados (modelo SIMD).

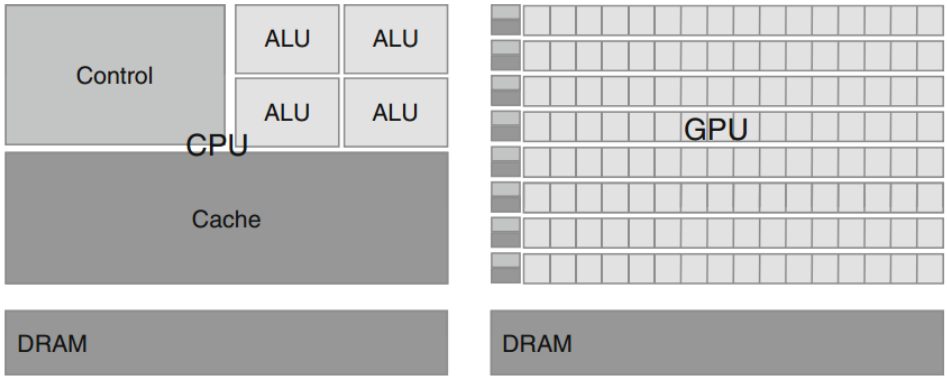

Figura 3.2: *Representação de Chip CPU X GPU (Kirk e Hwu, 2010)*

A utilização de GPUs para computação de propósito geral é conhecida como GPGPU (*General Purpose Computing GPU*). Esse paradigma teve seu início com o advento de APIs gráficas como HLSL e Cg (Fernando, 2003), que trouxeram maior flexibilidade na programação. Contudo, ainda havia muitas dificuldades nessa abordagem, dado que o programador precisava modelar o problema manipulando estruturas de computação gráfica. Em 2006, a NVIDIA introduziu a arquitetura CUDA (*Compute Unified Device Architecture*), que propiciou um nível de abstração adequado para GPGPU, já que o programador não precisava mais utilizar a API gráfica e poderia desenvolver a aplicação em uma linguagem de mais alto nível.

Assim, neste capítulo apresentamos uma visão geral de uma arquitetura moderna de GPU, bem como uma introdução à plataforma de desenvolvimento a ser utilizada nesse trabalho.

3.1 Arquitetura de GPU

3.1.1 Breve Evolução Histórica

No começo dos anos 1990 não havia GPUs. Nesse período, controladores VGA (*Video Graphics Array*) geravam visualizadores gráficos 2D para PCs para acelerar interfaces gráficas. Em 1997, a NVIDIA lançou o RIVA 128, um acelerador gráfico 3D para jogos e aplicações de visualização tridimensional configuráveis via Microsoft Direct3D e OpenGL. A GeForce 256 foi a primeira GPU a ser lançada em 1999. Ela é um processador gráfico 3D de tempo real e pode ser configurada com OpenGL e APIs Microsoft DirectX (DX) 7.

Em 2001, a NVIDIA lançou a GeForce 3, que representou um marco na indústria por se tratar da primeira GPU programável no mercado. Em seguida, houve o lançamento das versões GeForce FX e GeForce 6800, que eram programáveis via Cg, além de DX9 e OpenGL. Esses processadores eram altamente paralelizáveis e sua arquitetura facilitou implementações em GPU com diferentes números de núcleos.

Além de renderizar gráficos em tempo real, programadores também utilizavam Cg para simulações físicas e computação de propósito geral em GPU. Esses primeiros programas de GPGPU já alcançavam alto desempenho, contudo eram de difícil escrita, pois programadores tinham que expressar cálculos com elementos não gráficos utilizando APIs gráficas como OpenGL.

A GeForce 8800, introduzida em 2006, foi outro marco na indústria de placas gráficas ao apresentar a primeira GPU de arquitetura gráfica unificada. Isso quer dizer que o programador poderia utilizar a GPU para cálculos de propósito geral sem usar estruturas gráficas específicas. Essa placa é capaz de executar eficientemente até 12.228 *threads* concorrentemente em 128 núcleos de processadores. Além disso, ela adicionou suporte à linguagem C de programação e outras linguagens de propósito geral. A NVIDIA disponibilizou essa arquitetura escalável em uma família de GPUs GeForce com diferentes números de núcleos de processadores para cada segmento de mercado particular.

Até esse período, usuários construíam supercomputadores pessoais ao montar agregados compostos por nós de PCs com múltiplas GPUs. Em resposta a essa demanda, em 2007, a NVIDIA lançou a série Tesla C870, D870 e S870, que era composta por agregados de GPU baseados na GeForce 8800.

Em 2009 foi lançada a geração Fermi de GPUs NVIDIA. Essa é a arquitetura alvo desse trabalho e será vista com maiores detalhes na seção seguinte. A figura 3.3 resume a evolução história de placas gráficas NVIDIA, enquanto que a figura 3.4 apresenta a evolução das APIs nesse período.

Table 1. NVIDIA GPU technology development.

Date	Product	Transistors	CUDA cores	Technology
1997	RIVA 128	3 million	—	3D graphics accelerator
1999	GeForce 256	25 million	—	First GPU, programmed with DX7 and OpenGL
2001	GeForce 3	60 million	—	First programmable shader GPU, programmed with DX8 and OpenGL
2002	GeForce FX	125 million	—	32-bit floating-point (FP) programmable GPU with Cg programs, DX9, and OpenGL
2004	GeForce 6800	222 million	—	32-bit FP programmable scalable GPU, GPGPU Cg programs, DX9, and OpenGL
2006	GeForce 8800	681 million	128	First unified graphics and computing GPU, programmed in C with CUDA
2007	Tesla T8, C870	681 million	128	First GPU computing system programmed in C with CUDA
2008	GeForce GTX 280	1.4 billion	240	Unified graphics and computing GPU, IEEE FP, CUDA C, OpenCL, and DirectCompute
2008	Tesla T10, S1070	1.4 billion	240	GPU computing clusters, 64-bit IEEE FP, 4-Gbyte memory, CUDA C, and OpenCL
2009	Fermi	3.0 billion	512	GPU computing architecture, IEEE 754-2008 FP, 64-bit unified addressing, caching, ECC memory, CUDA C, C++, OpenCL, and DirectCompute

Figura 3.3: *Evolução Arquitetura GPU NVIDIA (Dally e Nickolls, 2010)*

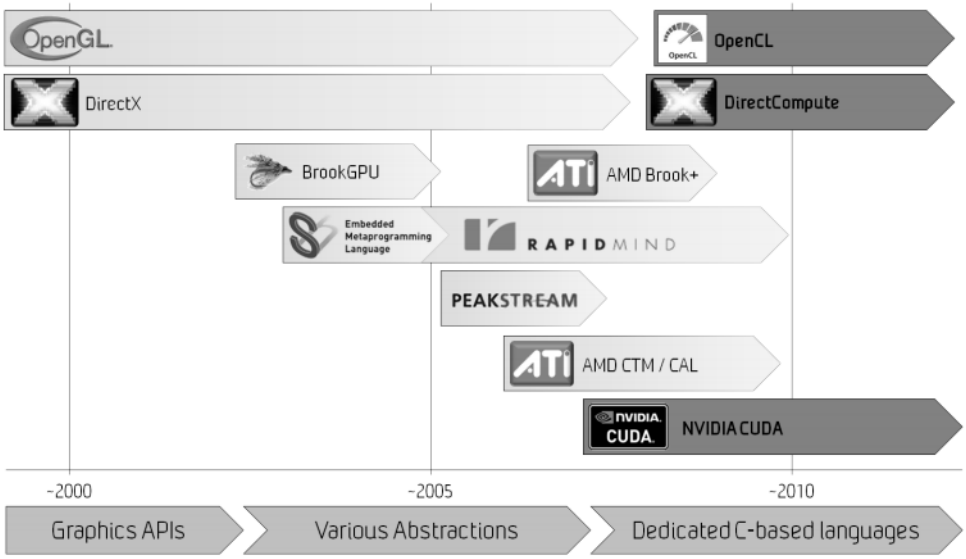

Figura 3.4: *Evolução APIs para GPU (Brodtkorb et al., 2012)*

3.1.2 Fermi

A arquitetura Fermi implementa 3 bilhões de transistores com um total de 512 núcleos CUDA. Um núcleo CUDA corresponde à menor unidade de computação da GPU e executa uma instrução de ponto flutuante ou inteiro por *clock*[1] para uma *thread*. Como podemos ver na figura 3.5, os 512 núcleos CUDA são organizados em 16 unidades chamadas de *Streaming Multiprocessor* (SM), que contem 32 núcleos CUDA cada. A GPU tem seis partições de memória de 64-bits, totalizando uma interface de memória de 384-bits, que suporta até 6GB de memória GDDR5 DRAM. Uma interface conecta a GPU e sua memória (*device*) com a CPU e sua memória (*host*) via PCIe. O componente *GigaThread* é o escalonador que distribui blocos de *threads* ao longo dos SMs. A figura 3.6 apresenta a configuração geral de um SM e de um núcleo CUDA. Cada um dos 32 núcleos CUDA de um SM contem uma ULA e uma UPF (Unidade de Ponto Flutuante).

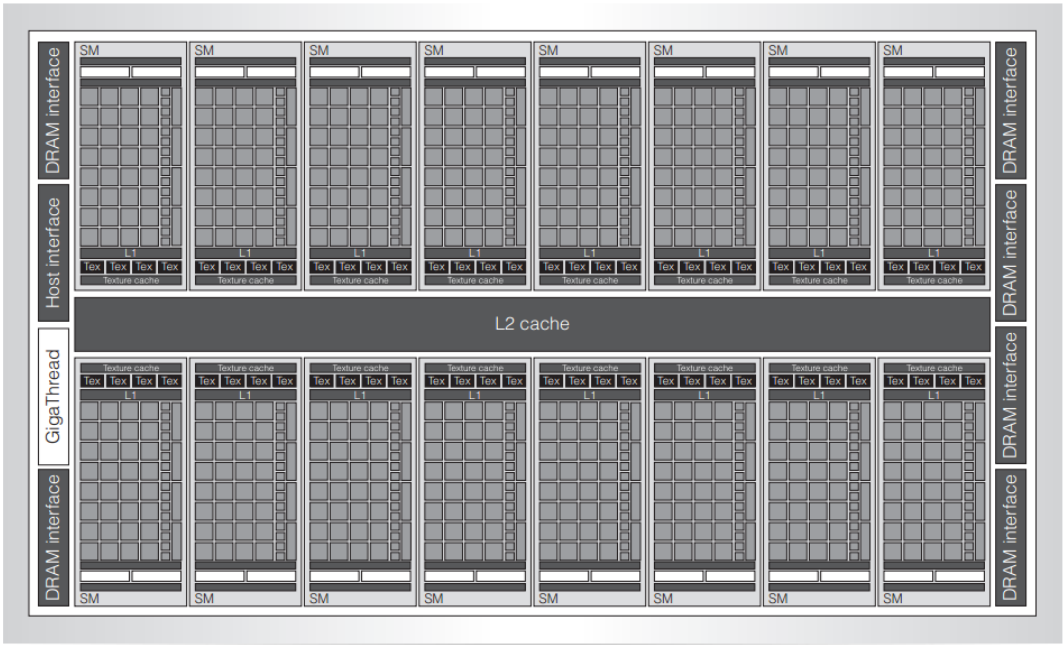

Figura 3.5: *Arquitetura Fermi (Dally e Nickolls, 2010)*

GPUs de gerações anteriores utilizavam o padrão IEEE 754-1985 (IEEE Task P754, 1985) para aritmética de ponto flutuante, ao passo que Fermi implementa o novo padrão IEEE 754-2008 (IEEE Task P754, 2008), que fornece maior precisão para operações de ponto flutuante de precisão dupla. Nesse novo padrão, foi disponibilizada a instrução FMA (*Fused Multiply-Add* [2]). Ela otimiza a instrução clássica MAD (Multiplicação-Adição) ao executar a multiplicação e adição com um passo adicional de arredondamento no final, de tal forma que não se perca precisão em truncamento. Assim, FMA é mais precisa do que se as operações fossem executadas de modo separado. A figura 3.7 demonstra o comportamento dessa instrução e a figura 3.8 exemplifica o ganho de desempenho da arquitetura Fermi em operações de dupla precisão em comparação com a geração anterior de GPU.

Cada SM possui 16 unidades de carga/armazenamento[3] (LD/ST), isso permite que endereços de origem e destino sejam calculados para dezesseis *threads* por *clock*. Unidades de Funções Especiais[4] (SFUs) executam instruções transcendentais como seno ou cosseno . Cada SFU executa uma instrução por *thread* a cada *clock*.

[1] Indica uma unidade de tempo de execução.
[2] em português: multiplicação-adição fundidas
[3] no original: load/store
[4] no original: *Special Function Units*

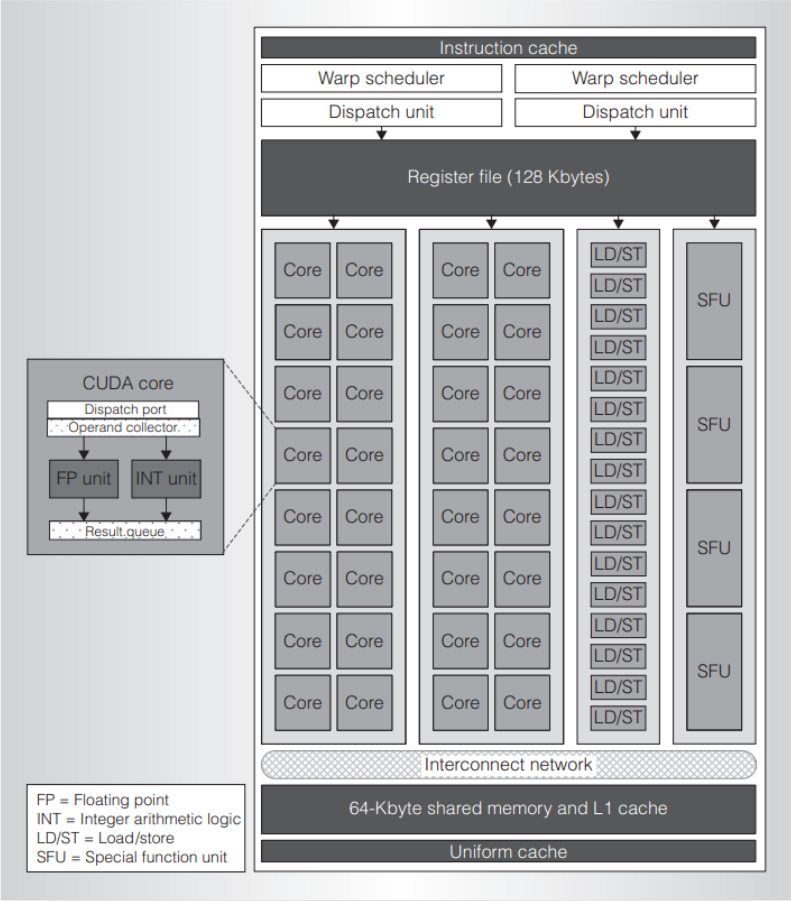

Figura 3.6: *Streaming Multiprocessor Fermi (Dally e Nickolls, 2010)*

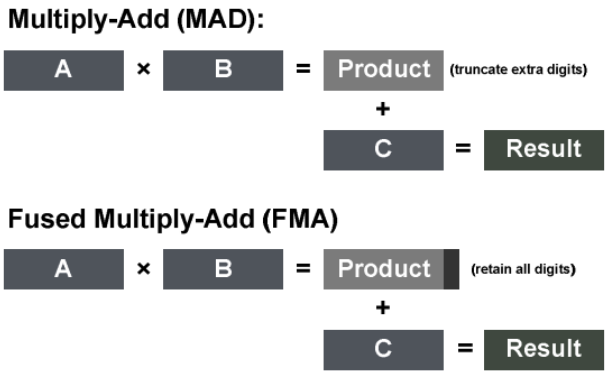

Figura 3.7: *Instrução FMA (NVIDIA, 2009a)*

O SM agenda *threads* para execução em grupos de 32 *threads* paralelas chamados de *warps*. Como cada SM possui dois escalonadores de *warp*, então é possível executar até 2 *warps* de maneira concorrente. O escalonador Fermi seleciona, então, dois *warps* e envia instruções de cada um para um grupo de 16 núcleos CUDA, dezesseis unidades LD/ST e quatro SFUs. Como os *warps* executam de maneira independente, o agendador da Fermi não precisa checar dependências entre instruções. A figura 3.9 demonstra esse funcionamento, que pode ser classificado como uma arquitetura SIMD.

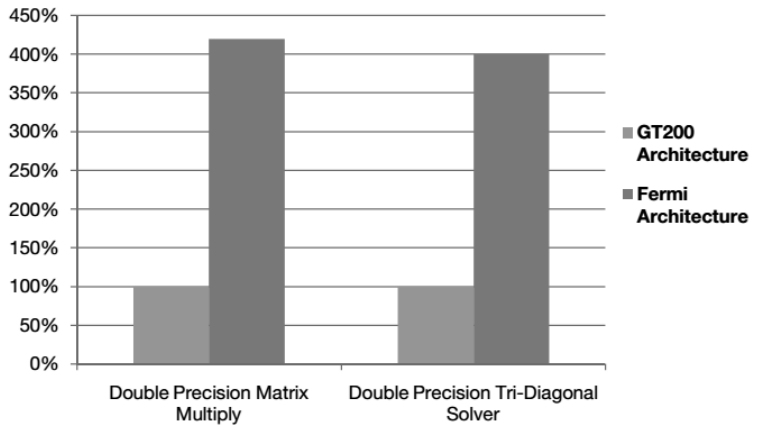

Figura 3.8: *Exemplo de desempenho Fermi em aplicação de dupla precisão (NVIDIA, 2009a)*

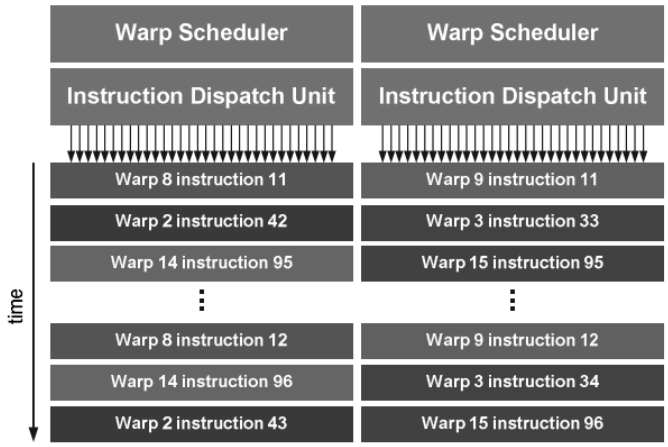

Figura 3.9: *Agendador Fermi de warps (NVIDIA, 2009a)*

Adicionalmente à memória compartilhada, a arquitetura Fermi possui dois níveis de memória cache. O cache L1 é individual ao SM e é configurável para suportar tanto memória compartilhada quanto cache das memórias local ou global. Essa memória de 64KB posse ser configurada tanto com 48KB de memória compartilhada com 16KB de cache L1 quanto com 16KB de memória compartilhada com 48KB de cache L1. Aplicações que utilizam memória compartilhada com frequência podem tirar grande proveito da primeira configuração, enquanto que a segunda alternativa mostra-se muito útil em aplicações que tem acessos com pouca previsibilidade. Para um estudo de algoritmos eficientes em utilização de cache em GPU veja (Govindaraju e Manocha, 2007).

Além de grandes avanços em questões de desempenho, Fermi introduziu uma nova característica às placas modernas da NVIDIA: proteção de integridade de dados. Com o suporte a ECC (*Error Correcting Code*), essa arquitetura se tornou capaz de corrigir erros de único *bit* e detectar erros de dois bits em memória DRAM, caches L1 e L2, e registradores. Essa característica mostra-se de importância para grandes aplicações de HPC, que integram centenas de GPUs em um único sistema.

A NVIDIA classifica a evolução da tecnologia de GPU em *compute capability*. Como a arquitetura Fermi corresponde à *compute capability* 2.x, outras especificações técnicas podem ser vistas no apêndice A.1. Na tabela 3.1 sumarizamos as principais características da arquitetura Fermi vistas até aqui.

Transistores	3 bilhões
Máximo Núcleos CUDA	512
Ponto flutuante de dupla precisão	256 FMA operações / *clock*
Ponto flutuante de precisão simples	512 FMA operações / *clock*
Unidades de Funções Especiais	4
Escalonador de *warp* por SM	2
Memória Compartilhada por SM	Configurável: 48KB/16KB
Cache L1 por SM	48KB/16KB
Cache L2	768KB
Suporte a ECC	Sim

Tabela 3.1: *Sumário de Características Arquitetura Fermi*

3.2 CUDA

CUDA é uma arquitetura de computação paralela para propósito geral. Por meio dessa, a GPU pode ser acessada como um conjunto de multiprocessadores (SMs) que são capazes de executar um grande números de *threads*. Atualmente, diversas APIs suportam CUDA, como: C, C++, Java e Fortran (vide figura 3.10). Este trabalho baseia-se na API de CUDA para C (CUDA C).

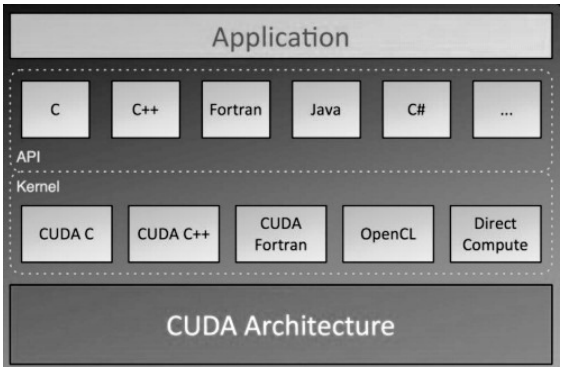

Figura 3.10: *CUDA API (NVIDIA, 2011a)*

3.2.1 Modelo de Programação

Uma função executada em GPU é chamada de *kernel*. O conjunto de *threads* de um *kernel* define uma grade[5] que, por sua vez, é subdividida em blocos de *threads*. Todas as *threads* em uma mesma grade executam o mesmo *kernel*. Com isso, para distinção do endereçamento, CUDA define as seguintes variáveis de sistema:

- *blockIdx*: índice do bloco
- *threadIdx*: índice da *thread*
- *gridDim*: dimensão da grade
- *blockDim*: dimensão dos blocos

Uma grade é bidimensional, portanto o índice de bloco é constituído pelo par *(blockIdx.x, blockIdx.y)*, ao passo que o índice de *thread* pode ter até três dimensões (*threadIdx.x, threadIdx.y,*

[5] no original: *grid*

threadIdx.z). A figura 3.11 representa tal indexação em uma organização bidimensional de blocos e *threads*.

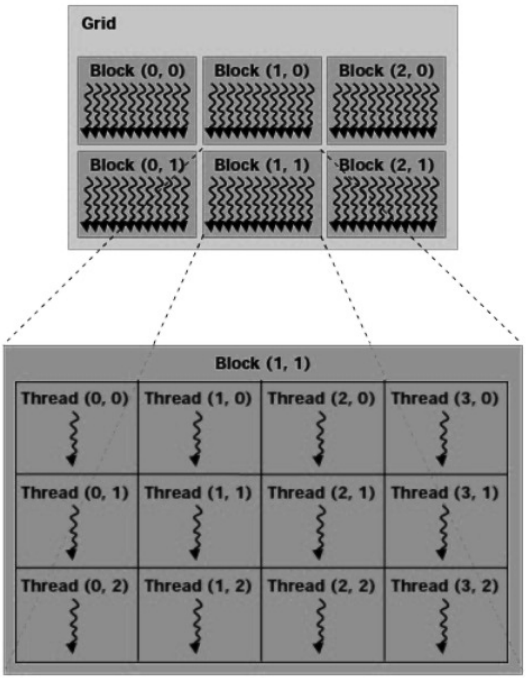

Figura 3.11: *Hierarquia de threads (NVIDIA, 2011a)*

Um *kernel* é definido pela extensão *__global__* e sua configuração deve determinar o número de blocos na grade (*numBlocos*) e de *threads* por bloco (*numThreads*) de sua composição. Tal configuração possui a seguinte sintaxe: <<<*numBlocos*, *numThreads*>>>. O código 3.1 exemplifica o lançamento de um *kernel* com um bloco e N *threads* responsável por somar dois vetores de N elementos.

```
1 //GPU
2 __global__ void VecAdd(float* A, float* B, float* C)
3 {
4   int i = threadIdx.x;
5   C[i] = A[i] + B[i];
6 }
7
8 //CPU
9 int main()
10 {
11   ...
12   //Lançamento do Kernel
13   VecAdd<<<1, N>>>(A, B, C)
14 }
```

Código 3.1: *Exemplo kernel*

CUDA também dispõe de outras duas extensões de função: *__device__* e *__host__*. A tabela 3.2 sumariza a utilização das mesmas.

Função	Executado em	Possível de ser chamado em
__*device*__ float DeviceFunc()	*device*	*device*
__*global*__ void KernelFunc()	*device*	*host*
__*host*__ float HostFunc()	*host*	*host*

Tabela 3.2: *Extensões de função em CUDA*

3.2.2 Modelo de Memória

A hierarquia de *threads* enunciada incorre em diferentes tipos possíveis de memória e em formas distintas de utilização da mesma. Desses tipos, podemos enunciar:

- Registradores: são rápidos, porém escassos. Cada *thread* possui um conjunto privado de registradores.
- Memória Compartilhada[6]: *threads* em um mesmo bloco compartilham um espaço de memória, o qual funciona como um cache manipulado explicitamente pelo programa.
- Memória Local[7]: cada *thread* possui acesso a um espaço de memória local, além de seus registradores. Essa área de memória está fora do micro-chip de processamento, junto à memória global, e portanto, ambas essas memórias possuem tempos de acessos similares.
- Memória Global[8]: esta memória está disponível para todas as *threads* em cada bloco e em todas as grades. Trata-se da única maneira de *threads* de diferentes blocos colaborarem.

A tabela 3.3 sumariza as características dos diferentes tipos de memória enunciados.

Declaração da Variável	Tipo de Memória	Local	Escopo	Tempo de vida
int var	Registrador	No chip	*Thread*	*Thread*
int var[10]	Memória Local	Fora do chip	*Thread*	*Thread*
__*shared*__ *int var*	Memória Compartilhada	No chip	Bloco	Bloco
__*device*__ *int var*	Memória Global	Fora do chip	Grade	Aplicação

Tabela 3.3: *Tipos de Memória*

[6] no original: *Shared Memory*
[7] no original: *Local Memory*
[8] no original: *Global Memory*

Capítulo 4

Técnicas de Otimização de Desempenho em CUDA

A programação de centenas ou até milhares de núcleos de processamento pode se mostrar como um grande desafio. Contudo, otimizar o desempenho desses programas em uma arquitetura de GPU moderna pode ser um problema ainda maior. Assim, este capítulo tem o objetivo de apresentar boas práticas de utilização da arquitetura CUDA. Ao final, dispomos um sumário das melhores práticas de otimização de desempenho classificadas por ordem de prioridade.

4.1 Medição Tempo em CUDA

Podemos medir tempos de execução em CUDA utilizando tanto métodos tradicionais de CPU quanto métodos específicos de GPU. Entretanto, há alguns pontos a considerar nessas escolhas. Primeiro, é importante notar que chamadas à API CUDA podem ser assíncronas. Nesse caso, é necessário garantir a sincronia de *threads* ao utilizar temporizadores em CPU. Além disso, deve-se tomar cuidado ao criar pontos de sincronização em CPU. Isso pode causar paradas em GPU.

Ao utilizar temporizadores de CPU, para medir corretamente o tempo transcorrido em uma chamada ou sequência de chamadas em CUDA, é necessário sincronizar a *thread* da CPU com a GPU com a instrução *cudaThreadSynchronize()* imediatamente antes e depois de iniciar o temporizador na CPU. No código 4.1 exemplificamos esse tipo de medição. Medições realizadas com temporizadores de GPU comumente utilizam a biblioteca *GPU Events*. No código 4.2 demonstramos a utilização dessa biblioteca.

```
1 // temporizadores de CPU
2 struct timeval t1_start, t1_end;
3 double time_d, time_h;
4
5 // trecho de codigo a ser cronometrado
6
7 // marca inicio de temporizacao
8 gettimeofday(&t1_start,0);
9
10 kernel_exemplo<<<dimGrid, dimBlock>>>(data);
11
12 // sincronizacao de threads
13 cudaThreadSynchronize();
14
15 // finaliza temporizacao
16 gettimeofday(&t1_end, 0);
17
18 // tempo transcorrido
19 time_d = (t1_end.tv_sec-t1_start.tv_sec)*1000000 + t1_end.tv_usec - t1_start.
      tv_usec;
```

Código 4.1: *Temporizador CPU*

```
1 // declaracao dos temporizadores
2 cudaEvent_t start, stop;
3
4 // tempo transcorrido
5 float time;
6
7 //aloca temporizadores
8 cudaEventCreate(&start);
9 cudaEventCreate(&stop);
10
11 // inicia temporizacao para stream 0 (default)
12 cudaEventRecord(start, 0);
13
14 // lancamento de kernel
15 kernel<<<grid, threads>>>(data);
16
17 // finaliza temporizacao para stream 0
18 cudaEventRecord(stop, 0);
19 cudaEventSynchronize(stop);
20
21 // tempo total transcorrido em milisegundos
22 cudaEventElapsedTime(&time, start, stop);
23
24 // libera memoria
25 cudaEventDestroy(start);
26 cudaEventDestroy(stop);
```

Código 4.2: *Temporizador GPU Events*

4.2 Execução Concorrente Assíncrona

Para facilitar a execução concorrente entre *host* e *device*, algumas chamadas de função são assíncronas: o controle é retornado para *thread* do *host* antes que o *device* complete as tarefas requisitadas.
Dois tipos de execução concorrente são particularmente importantes:

- Sobreposição em transferência de dados
- Execução de *Kernels* paralelos

Para verificar se o *device* permite tais tipos de concorrência, deve-se executar a chamada *cudaGetDeviceProperties()* e checar os atributos *deviceOverlap* e *concurrentKernels*.

4.2.1 Comunicação *Host-Device*

Ao alocar memória em CPU que vai ser utilizada para transferir dados para GPU, há dois tipos de memória possíveis: memória pinada[1] e memória não pinada[2]. A primeira possibilita transferências via PCIe mais rápidas com cópias assíncronas de memória. Seu uso se dá pelas funções *cudaHostAlloc* e *cudaFreeHost* no lugar das funções de CPU *malloc* e *free*, respectivamente.

Apesar da utilização de memória pinada poder acelerar as transferências via PCIe, essa memória não deve ser super utilizada, pois há considerável sobrecarga em sua utilização e seu uso não planejado pode causar perda de desempenho. Assim, sua quantidade ótima depende da aplicação específica.

[1] no original: *pinned memory*
[2] no original: *non-pinned memory*

4.2.2 *Kernels* Paralelos

Em *compute capability* 1.x, o único modo de se utilizar todos os multiprocessadores é lançar um único *kernel* com pelo menos o número de blocos igual ao de SMs. A partir da *compute capability* 2.x, CUDA permite a execução de múltiplos *kernels*. Tal execução é feita utilizando CUDA *streams*. No código 4.3 exemplificamos a declaração de *streams* para sobreposição entre transferência de dados e computação em GPU. Nas linhas 2 e 3 dois streams são criados para explicitar independência de execução. Na linha 5 é feita uma cópia assíncrona de dados para GPU e na linha 6 o *kernel* é lançado em um *stream* diferente e, portanto, executado de forma paralela com a transferência de dados.

```
1 cudaStream_t stream1, stream2;
2 cudaStreamCreate(&stream1);
3 cudaStreamCreate(&stream2);
4
5 cudaMemcpyAsync( dst, src, sice, dir, stream1 );
6 kernel<<<grid, block, 0, stream2>>>(...);
```

Código 4.3: *CUDA streams para sobreposição entre transferência de dados e computação em GPU*

O código 4.4 fornece um exemplo de execução concorrente entre *kernels*. Ele realiza os seguintes passos:

1. Cria *streams* para explicitar inter-independencia
2. Aloca memória pinada
3. Associa cópia de dados e execução de *kernels* com seus respectivos *streams*
4. Desaloca memória utilizada

Entre as linhas 1 e 8 dois *streams* são definidos ao criar objetos do tipo *cudaStream* e um vetor de *floats* é alocado em memória pinada. Entre as linhas 10 e 16 cada *stream*: copia sua porção de dados do vetor *hostPtr* para o vetor *inputDevPtr* em memória de *device*; processa de modo concorrente *inputDevPtr* no *device* ao chamar *MyKernel()*; copia o resultado obtido em *outputDevPtr* de volta para o *host* em *hostPtr*. Ao final, na linha 18, cada *stream* é liberado com a chamada de *cudaStreamDestroy*.

```
1 cudaStream_t stream[2]
2
3 for (int i = 0; i < 2; ++i)
4   cudaStreamCreate(&stream[i]);
5
6 float* hostPtr;
7
8 cudaMallocHost(&hostPtr, 2 * size);
9
10 for (int i = 0; i < 2; ++i){
11   cudaMemcpyAsync(inputDevPtr + i * size, hostPtr + i* size, size,
        cudaMemcpyHostToDevice, stream[i]);
12
13   MyKernel<<<100, 512, 0, stream[i]>>>(outputDevPtr + i * size, inputDevPtr + i*
         size, size);
14
15   cudaMemcpyAsync(hostPtr + i * size, outputDevPtr + i * size, size,
        cudaMemcpyDeviceToHost, stream[i]);
16 }
17
18 for (int i = 0; i < 2; ++i)
19   cudaStreamDestroy(stream[i]);
```

Código 4.4: *Exemplo de execução concorrente entre kernels*

Um *stream* é uma sequência de comandos que são executados em ordem. Não há ordenação entre comandos de diferentes *streams*. Associar diferentes *streams* a diferentes operações significa explicitar independência dessas. Isso possibilita intervalar operações. Dessa forma, um lançamento de *kernel* e uma cópia de memória podem ter sobreposição. No código 4.5 apresentamos um exemplo de divisão de cópia de dados e execução de *kernel* em passos intermediários para esconder a latência. A figura 4.1 ilustra tal comportamento.

```
1 size = (N*sizeof(float))/nStreams;
2
3 for (i=0;; i<nStreams; i++){
4   offset = (i*N)/nStreams;
5   cudaMemcpyAsync(a_d+offset, a_h+offset, size, dir, stream[i]);
6 }
7 for (i=0;; i<nStreams; i++){
8   offset = (i*N)/nStreams;
9   kernel<<<N/(nThreads*nStreams), nThreads, 0, stream[i]>>>(a_d+offset);
10 }
```

Código 4.5: *Exemplo de divisão de cópia de dados e execução de kernel*

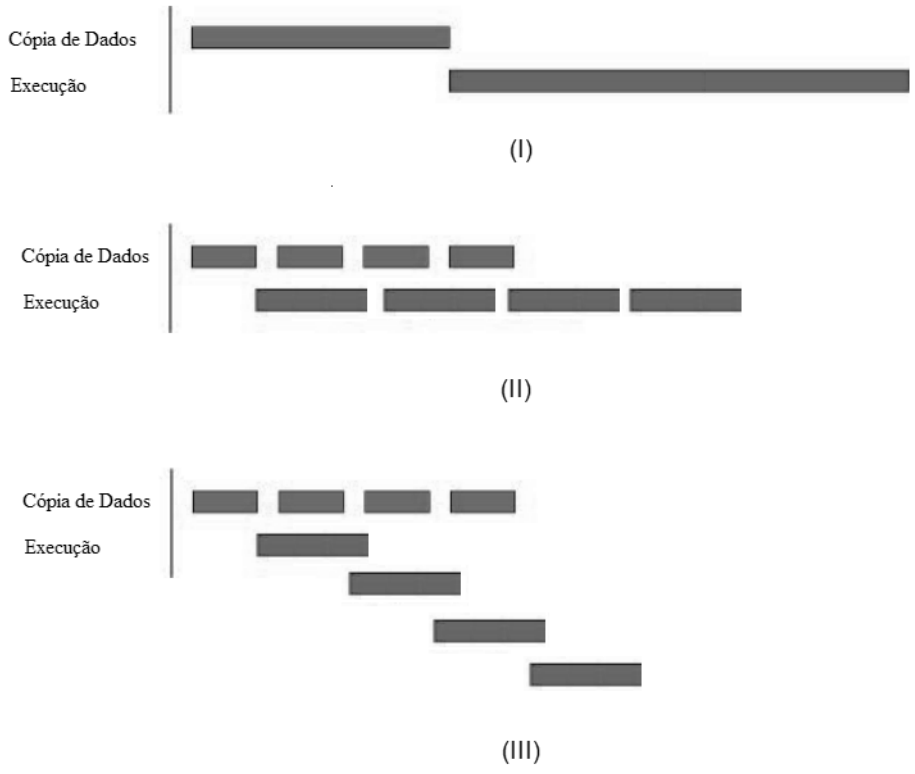

Figura 4.1: *(I) Execução e cópia sequenciais; (II) Execução e cópia paralelos em compute capability 1.x; (III) Execução e cópia paralelos em compute capability 2.x;*

4.3 Otimizações de Memória

4.3.1 *Coalesced Memory*

Uma das mais importantes considerações de otimização na arquitetura CUDA é *Coalesced Memory*. Em um dispositivo de *compute capability* 2.x, os acessos à memória global realizados por *threads* dentro de um mesmo *warp* são reunidos em uma mesma transação de memória quando alguns requisitos de acesso são satisfeitos. Para entendê-los, devemos ver a memória global em termos de segmentos alinhados de 32 palavras de memória. Assim, um padrão de acesso de *coalesced memory* acontece quando a k-ésima *thread* acessa a k-ésima palavra de um segmento, com a observação que nem todas as *threads* precisam participar do acesso. A Figura 4.2 ilustra tal situação.

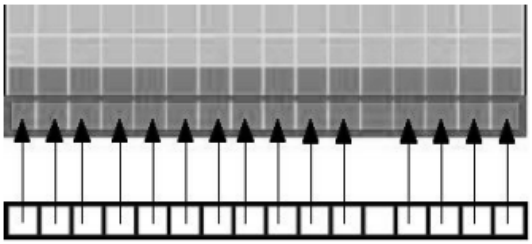

Figura 4.2: *Acesso Coalesced Memory (NVIDIA, 2011a)*

4.3.2 Memória Compartilhada

Como a memória compartilhada é interna ao *chip*, ela é muito mais rápida que a memória local e a memória global. De fato, tal memória tem latência, no mínimo, 100x menor que a memória global, quando não há nenhum conflito de bancos de memória entre as *threads* (NVIDIA, 2010b).

Para alcançar banda de memória máxima para acessos paralelos, a memória compartilhada é dividida em módulos (bancos) de igual tamanho que podem ser acessados simultaneamente. Assim, quaisquer leituras ou escritas em n endereços distribuídos por n bancos distintos de memória são servidos ao mesmo tempo, correspondendo em uma banda efetiva n vezes maior que a banda de um único banco.

Entretanto, se diferentes endereços de requisição de memória são mapeados para um mesmo banco, os acessos são serializados. O *hardware* divide tal requisição em sub-requisições sem conflitos de acesso, o que diminui a banda efetiva em fator igual ao número de sub-requisições. Para exemplificar essa sobrecarga dessa serialização, podemos criar um *kernel* (Che *et al.*, 2008) que transpõe tanto colunas quanto linhas em paralelo em uma matriz 16 X 16. Ao transpor colunas, o *kernel* exibe um número maximal de conflitos de acesso a bancos de memória, enquanto que ao transpor linhas, não há conflitos. A figura 4.3 mostra que a existência de conflitos em bancos de memória compartilhada dobra, aproximadamente, o tempo de execução do *kernel*.

4.3.3 Registradores

Em geral, acesso a registradores consome zero ciclos de *clocks* extras por instrução, mas atrasos podem ocorrer devido a, por exemplo, conflitos em acessos a bancos de registro de memória. O compilador e o *hardware* escalonador de *threads* controlam as instruções tão ótimo quanto possível para evitar conflitos em bancos de memória. Contudo, para obter melhores resultados é necessário configurar o número de *threads* por bloco como um múltiplo de 64 (NVIDIA, 2010b).

4.3.4 Memória Constante

Há um total de 64KB de memória constante tanto em dispositivos de *compute capability* 1.x quanto 2.x. Esse tipo de memória fornece ganho de desempenho, pois é mantida em cache. Assim, uma leitura de memória constante custa apenas uma leitura de memória do *device*.

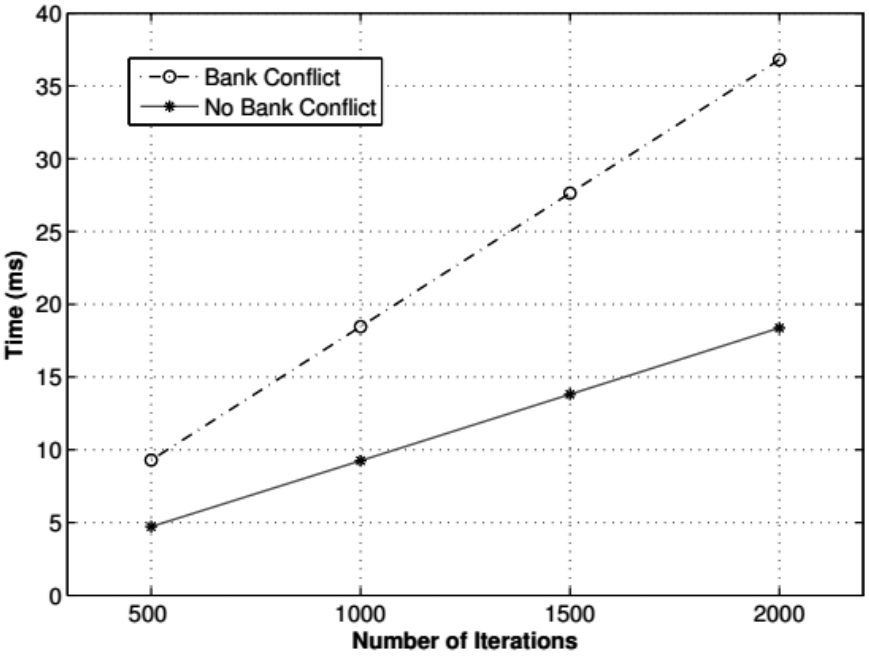

Figura 4.3: *Sobrecarga em conflitos de banco em memória compartilhada (Che et al., 2008).*

Para *threads* de um mesmo *warp*, ler memória constante é tão rápido quanto ler registradores, desde que todas as *threads* leiam do mesmo endereço. Acessos a diferentes endereços de memória por *threads* de um mesmo *warp* são serializados. Assim, o custo cresce linearmente com o número de endereços diferentes acessados. Alternativamente, dispositivos de *compute capacility* 2.x suportam a instrução LDU (*LoaD Uniform*), que o compilador utiliza para carregar variáveis que são somente leitura, não dependem de identificador de *thread* ou apontam para memória global.

4.3.5 Memória Local

É importante explicitar enganos comuns sobre a utilização de memória local. Essa memória tem esse nome devido ao seu escopo ser local à *thread*, não devido a sua localidade física. Na verdade, a memória local é externa ao *chip*. Assim, seu acesso tem custo da mesma ordem de grandeza que o acesso à memória global. Em *compute capability* 1.x, como em memória global, a memória local não é mantida em cache. Portanto, o a palavra "local" em seu nome não significa necessariamente acesso com maior desempenho.

4.4 Controle de Fluxo Condicional

Em CUDA, instruções de controle de fluxo condicional podem impactar significativamente a largura de banda de memória de instruções se *threads* dentro de um mesmo *warp* seguirem controles de fluxo diferentes. Ao executar caminhos de fluxo divergentes, todas as *threads* dentro de um mesmo *warp* executam cada instrução de cada caminho divergente, o que causa uma potencial perda de desempenho. Para obtenção de máximo desempenho em casos onde o controle de fluxo depende do identificador da *thread*, a condição de controle deve ser escrita visando à minimização de *warps* divergentes. A figura 4.4 de (Che *et al.*, 2008) demonstra que a sobrecarga aumenta de linearmente com o aumento do número de *threads* divergentes.

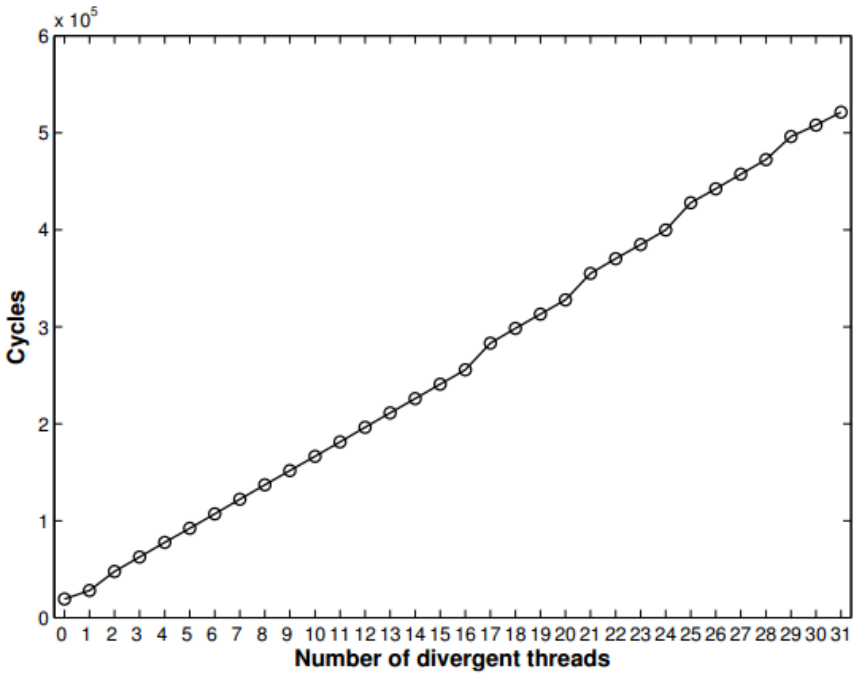

Figura 4.4: *Sobrecarga em threads divergentes (Che et al., 2008).*

4.5 Configuração de Execução

Um princípio para obtenção de bom desempenho é manter o multiprocessador do *device* tão ocupado quanto possível. Uma GPU na qual o trabalho é distribuido de forma desigual ao longo dos multiprocessadores em geral resulta em desempenho reduzido. Assim, é importante construir a aplicação para utilizar *threads* e blocos de forma a maximizar o uso do *hardware* e limitar práticas que impeçam a livre distribuição das instruções a executar. Para isso, dois conceitos têm papel fundamental: a ocupação do multiprocessador e o gerenciamento dos recursos alocados para uma atividade particular.

4.5.1 Ocupação

Instruções de uma mesma *thread* são executadas sequencialmente em CUDA. Por isso, executar outros *warps* quando um *warp* está pausado é a única forma de esconder latência e manter o *hardware* ocupado. Assim, uma métrica relacionada ao número de *warps* ativos em um multiprocessador é importante para determinar quão eficientemente o *hardware* está sendo utilizado. Para isso, uma métrica comumente utilizada é Ocupação[3]. Ela é definida como a razão entre o número de *warps* ativos por multiprocessador e o número máximo possível de *warps* ativos no mesmo.

Intuitivamente, Ocupação mede percentualmente a capacidade do *hardware* em processar *warps* ativos. A capacidade máxima de *warps* ativos em um multiprocessador pode ser obtida no apêndice A.1.

Com isso, é boa prática maximar a ocupação dos SMs. Como um auxílio nesse intuito, a NVIDIA fornece uma calculadora de Ocupação de fácil uso, disponível em (NVIDIA, 2010a). Contudo, tal estratégia é apenas uma heurística, dado que maior ocupação não significa, necessariamente, maior desempenho, já que outros recursos da placa gráfica podem se tornar fatores limitantes de desempenho, como comprovado em (Hong e Kim, 2009).

[3]no original: *occupancy*

4.5.2 Configuração de Registradores

A disponibilidade de registradores é um dos fatores que determina a Ocupação. Alocação de registradores permite que *threads* mantenham suas variáveis locais mais próximas para um acesso de menor latência. Entretanto, registradores são recursos limitados que *threads* de um mesmo multiprocessador devem compartilhar.

Registradores são alocados em escopo de bloco. Assim, se cada bloco de *threads* utiliza muitos registradores, o número total de blocos permitido em um multiprocessador pode ser reduzido e, assim, a Ocupação pode ser prejudicada. O número máximo de registradores por *threads* pode ser definido em tempo de compilação utilizando a opção *-maxrregcount*. Veja (NVIDIA, 2011a) para fórmulas de alocação de registradores por *compute capability* e o apêndice A.1 para o número total de registradores disponíveis nesses dispositivos.

4.6 Otimização de Instruções

Tomar conhecimento de conjunto de instruções, frequentemente, permite otimizações de mais baixo nível, especialmente interessantes em trechos de código que são executados repetidas vezes. A seguir, discutimos instruções de mais baixo nível presentes na biblioteca CUDA.

4.6.1 Bibliotecas Matemáticas

CUDA fornece duas versões de funções em sua biblioteca matemática. Elas podem ser distinguidas pelo prefixo "__" (por exemplo: nomeFuncao() e sua correspondente __nomeFuncao()). Funções que seguem a convenção __nomeFuncao() utilizam instruções de mais baixo nível. Elas são mais rápidas, contudo de menor precisão (e.g. __sinf(x) e __expf(x)). A opção de compilação *-use_fast_math* converte funções do tipo *nomeFuncao()* em sua respectiva versão *__nomeFuncao()*. Assim, esse tipo de compilação deve ser executado sempre que precisão tiver menor prioridade que desempenho. O apêndice A.2 lista funções afetadas pelo uso de *-use_fast_math*.

4.6.2 Instruções Aritméticas

Instruções *float* de precisão simples fornecem melhor desempenho que instruções correspondentes de precisão dupla e, portanto, devem ser consideradas quando possível. Em (NVIDIA, 2011a) é apresentada a largura de banda de operações aritméticas por *compute capability*.

Dentre as instruções aritméticas, alguns usos devem ser evitados, como por exemplo operações de módulo ou divisão. Essas operações são particularmente custosas e seu uso deve ser controlado. Uma opção é a substituição por operações bit-a-bit (e.g. utilização de *shift* $i >> 1$ no lugar de divisão $i/2$). Conversões automáticas de tipos também devem ser evitadas, já que isso gera instruções a mais pelo compilador. Alguns exemplos são a utilização de *char* ou *short*, que são convertidas em *int*, ou a utilização de constantes de dupla precisão como argumentos em cálculos de precisão simples.

4.7 Sumário de Melhores Práticas

Como visto, técnicas de otimização de desempenho em CUDA baseiam-se em três estratégias fundamentais:

- Maximar a quantidade de código paralelizada
- Otimizar uso de memória para alcançar maior largura de banda possível
- Otimizar uso de instruções para alcançar maior quantidade de execução de operações por unidade de tempo

Na tabela 4.1 sumarizamos as principais boas práticas em CUDA vistas até aqui em ordem de prioridade.

Prioridade	Tipo de Otimização	Estratégia de Otimização
Alta	Paralelização Geral	Para maximar desempenho, encontre formas de paralelizar o código sequencial
Alta	Memória	Acesso à memória global deve ser *coalesced* (4.3.1)
Alta	Memória	Minimize utilização de memória global. Utilize memória compartilhada sempre que possível (4.3.2)
Alta	Controle de Fluxo	Evite caminhos divergentes dentro de um mesmo *warp* (4.4)
Média	Configuração de Execução	Número de *threads* por bloco deve ser múltiplo de 32 (4.3.3)
Média	Instruções Aritméticas	Utilize versões de baixo nível da biblioteca de matemática sempre que precisão não for uma prioridade (4.6.2)
Baixa	Instruções Aritméticas	Utilize operadores de *shift* no lugar de operações de divisão ou módulo (4.6.2)
Baixa	Instruções Aritméticas	Evite conversões automáticas de tipo de variáveis (4.6.2)

Tabela 4.1: *Sumário de estratégias de otimização de desempenho*

Capítulo 5

Geração de Números Aleatórios em GPU

Em geral, simulações estocásticas são sensíveis à fonte de aleatoriedade, que caracteriza a qualidade estatística de seus resultados. Dessa forma, são necessários geradores de números aleatórios (RNGs) de alta confiança para alimentar tais aplicações. Desenvolvimentos recentes utilizam GPUs de propósito geral (GPGPUs) para acelerá-las. Esse *hardware* oferece novas possibilidades de paralelização, mas as mesmas causam não somente a reescrita dos algoritmos de simulação existentes como, também, mudanças nas ferramentas que os mesmos utilizam. Como RNGs são a base de qualquer simulação estocástica eles necessitam ser portados para GPGPU.

A seleção de um RNG não é simples. Contudo, há trabalhos que os estudam em uma arquitetura sequencial, como (Knuth, 1997) e (L'Ecuyer, 2007). Em um contexto paralelo, a qualidade estatística é um requisito necessário, mas não suficiente para selecionar um RNG, pois os *streams* paralelos devem ser independentes. Assim, fornecer um gerador de números aleatórios de alta qualidade é mais difícil em arquiteturas paralelas. Com isso, precisamos levar em conta a técnica de paralelização ao determinar como os *streams* aleatórios podem ser particionados dentre os elementos paralelos (*threads* ou processadores) com a garantia de independência entre os mesmos para evitar resultados enviesados nas simulações.

Nas próximas seções apresentaremos um estudo dos principais RNGs disponíveis, bem como sua portabilidade para GPU. Na seção 5.2, apresentamos as principais técnicas de paralelização desses geradores e, baseado no trabalho de (Nguyen, 2007), finalizamos o capítulo com um estudo de otimização em GPU dos geradores de números aleatórios MRG32k3a e Sobol.

5.1 RNGs

Geradores de números aleatórios podem ser classificados em três grandes grupos de acordo com sua fonte de aleatoriedade:

- TRNGs (*True random number generators*): utilizam uma fonte de aleatoriedade física para prover números verdadeiramente imprevisíveis. TRNGs são utilizados principalmente para criptografia. Eles são, em geral, muito lentos para simulação.

- QRNGs (*Quasirandom number generators*): visam a preencher um espaço n-dimensional com pontos de forma mais uniforme possível.

- PRNGs (*Pseudorandom number generators*): simulam TRNGs, mas podem ser implementados em *software* determinístico, tendo seu estado e função de transição previsíveis.

Há dois requisitos básicos desejáveis para PRNGs:

- Longo Período. Todo gerador determinístico eventualmente entra em ciclo. O objetivo, então, é ter o período do ciclo o maior possível.

- Boa Qualidade Estatística. É desejável que a saída de um PRNG seja praticamente indistinguível em relação a um TRNG de mesma distribuição. Além disso, também não devem exibir correlações ou padrões de geração.

Há inúmeros testes para verificar tais requerimentos de qualidade em algoritmos de arquitetura serial (Knuth, 1997), (L'Ecuyer, 2006) e (Marsaglia, 1995).

5.1.1 Algoritmos Sequenciais de PRNGs

Em geral, um gerador de números pseudo-aleatórios consiste em um conjunto finito de estados e uma função de transição f que leva o PRNG de um estado S para um próximo estado $S^{'}$,

$$S^{'} = f(S),$$

a partir de um estado inicial chamado semente S_0. Dado um estado S_n do PRNG, há uma função g que retorna o correspondente número aleatório x_n:

$$x_n = g(S_n).$$

A memória utilizada para armazenar o estado de um PRNG é finita. Portanto, o espaço dos estados é finito. Dessa forma, após suficientes passos, tal gerador fecha um ciclo. O tamanho desse ciclo é chamado de *período* do PRNG.

Linear Congruential Generator

Linear Congruential Generator (LCG) (Knuth, 1997) é um gerador clássico de PRNGs que possui uma função de transição da forma

$$x_{n+1} = (ax_n + c) \bmod m.$$

Assim, o período máximo desse gerador é m. Com isso, por exemplo, em uma representação de inteiro de 32-bits, o período pode ser no máximo 2^{32}. Isso é pouco para simulações estocásticas modernas.

Multiple Recursive Generator

Multiple Recursive Generator (MRG) é um algoritmo derivado do LCG, que, de modo aditivo, combina dois ou mais geradores. Se n geradores com períodos primos entre si $m_1, m_2, \ldots, m_n$ são combinados, então o período resultante é LCG$(m_1, m_2, \ldots, m_n)$. Assim, o período pode ser, no máximo, $m_1x, m_2x, \ldots, m_nx$. Em geral, esses geradores têm boa qualidade estatística e longos períodos. Contudo, os mesmos podem ter muitas multiplicações e divisões o que pode causar perda de desempenho em GPUs.

Lagged Fibonacci Generator

O *Lagged Fibonacci Generator* (Knuth, 1997) é comumente utilizado em simulações de Monte Carlo distribuídas. Ele é similar ao LCG, mas introduz um atraso na dependência de estados na função de transição:

$$x_{n+1} = x_n \otimes x_{n-k} \bmod m$$

onde $\otimes$ é, tipicamente, um operador de multiplicação ou adição. Apesar da simplicidade e pequeno número de variáveis envolvidas, esse gerador necessita que a constante k seja muito grande. Consequentemente, muitas palavras são necessárias para manter o estado e, assim, eventualmente seria necessário utilizar-se de memória global da GPU para transição dos estados, o que implicaria em perda de desempenho.

Mersenne Twister

Mersenne Twister (Matsumoto e Nishimura, 1998) é um dos geradores mais respeitados. Ele tem período $2^{19.937}$ e excelente qualidade estatística. Entretanto, apresenta um problema similar ao *Lagged Fibonacci*, pois possui um grande estado que precisa ser atualizado serialmente.

5.1.2 Algoritmos Sequenciais de QRNGs

Logo após a introdução do método de Monte Carlo no final dos anos 1940, pesquisadores se interessaram pela possibilidade de substituir amostragens aleatórias em simulações por séries determinísticas. Essa é a ideia fundamental dos QRNGs. Essas sequências são conhecidas como sequências de baixa discrepância. O conceito de baixa discrepância diz respeito à diferença relativa entre séries originadas por sequências determinísticas e aquelas obtidas por variáveis aleatórias uniformes. Assim, essas sequências tem a propriedade de uniformidade dentro de um intervalo de domínio.

Nessa seção, apresentamos algoritmos clássicos para geração de sequências de baixa discrepância (SBD) ou QRNGs. A sequência de Van Der Corput é a mais simples e serve como base na construção de QRNGs em finanças. As sequências mais conhecidas são a de Halton, Faure e, principalmente, Sobol. O princípio de construção dessas sequencias é dividir um hipercubo unitário em hipercubos menores cujas faces são paralelas às faces do hipercubo original. Cada ponto da sequência gerada é colocado em cada hipercubo menor e, assim, a sequência é gerada sucessivamente. O grande desafio na construção de boas SBDs é evitar a aglomeração de pontos em regiões específicas, o que é indesejável para um QRNG.

Sequência de Van Der Corput

Suponha que desejamos criar uma sequência de baixa discrepância de dezesseis números dentro do intervalo $[0, 1)$. Para isso, poderíamos simplesmente escrever a sequência $0, 1/16, 2/16, \ldots, 15/16$. Essa solução trivial é ruim, pois no meio da sequência já negligenciamos toda a outra metade do intervalo. É preferível uma geração uniforme independente do tamanho da sequência gerada até um dado momento. Para isso, Van Der Corput propôs um algoritmo similar que transforma o número em uma base b e, a cada passo, inverte seus bits (e.g. 001_2 se torna 100_2). Isso garante que o bit mais significativo sempre é alternado a cada número gerado. Assim, se o i-ésimo número da sequência pertence ao intervalo $[0, 1/2]$, então o $(i + 1)$-ésimo pertencerá ao intervalo $[1/2, 1]$. A seguir, apresentamos o Algoritmo 1 de Van Der Corput, que encontra o n-ésimo número da sequência gerada na base b. Um exemplo de execução desse algoritmo para uma sequência na base binária é fornecido na figura 5.1.

Algoritmo 1 Sequência de Van Der Corput

Input: n, b
Output: b_n: n-ésimo termo da sequência da Van Der Corput

Escreva n na base b
$n = \sum_{j=0}^{m} a_j(n) b^j$,
onde m é o menor inteiro tal que $a_j(n) = 0$ para todo $j > m$.

Inverta os bits do número n:
$b_n = \sum_{j=0}^{m} \frac{a_j(n)}{b^{j+1}}$.

i	Sequência Trivial	Binário	Binário Invertido	Van Der Corput	Pontos no intervalo [0, 1]
1	0	.0000	.0000	0	
2	1/16	.0001	.1000	1/2	
3	1/8	.0010	.0100	1/4	
4	3/16	.0011	.1100	3/4	
5	1/4	.0100	.0010	1/8	
6	5/16	.0101	.1010	5/8	
7	3/8	.0110	.0110	3/8	
8	7/16	.0111	.1110	7/8	
9	1/2	.1000	.0001	1/16	
10	9/16	.1001	.1001	9/16	
11	5/8	.1010	.0101	5/16	
12	11/16	.1011	.1101	13/16	
13	3/4	.1100	.0011	3/16	
14	13/16	.1101	.1011	11/16	
15	7/8	.1110	.0111	7/16	
16	15/16	.1111	.1111	15/16	

Figura 5.1: *Sequência de Van Der Corput em base binária*

Sequência de Halton

A sequência de Halton é uma extensão multi-dimensional da sequência de Van Der Corput. Assim, cada dimensão em uma sequência de Halton corresponde a uma sequência de Van Der Corput construída em uma base diferente. Mais especificamente, a i-ésima dimensão de Halton corresponde à sequência de Van Der Corput na base do i-ésimo número primo ordenado crescentemente. A tabela 5.1 exibe os primeiros termos dessa sequência. Na figura 5.2 podemos observar mil pontos

Termo	Dimensão 1	Dimensão 2	Dimensão 3	Dimensão 4
	Base 2	Base 3	Base 5	Base 7
1	1/2	1/3	1/5	1/7
2	1/4	2/3	2/5	2/7
3	3/4	1/9	3/5	3/7
4	1/8	4/9	4/5	4/7
5	5/8	7/9	1/25	5/7

Tabela 5.1: *Sequência de Halton*

gerados pela função padrão de Matlab para geração de números aleatórios uniformes. Na figura 5.3 são exibidos números bi-dimensionais gerados pela sequência de Halton. É fácil notar, visualmente, que a Sequência de Halton resultou em uma amostra mais uniformemente distribuída no quadrado unitário.

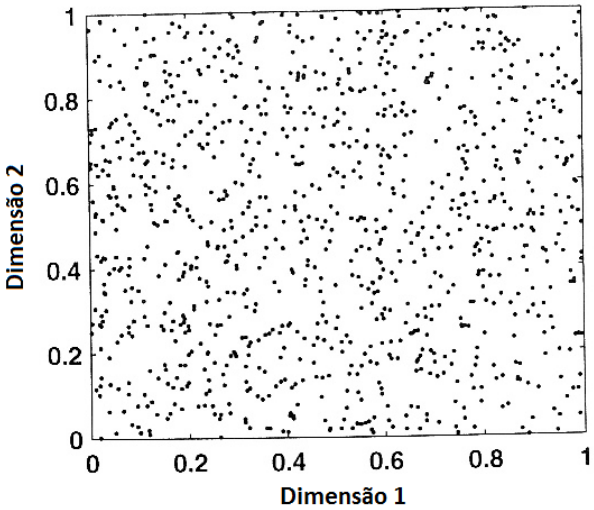

Figura 5.2: *Função Matlab para geração de números aleatórios uniformes (Huynh et al., 2011)*

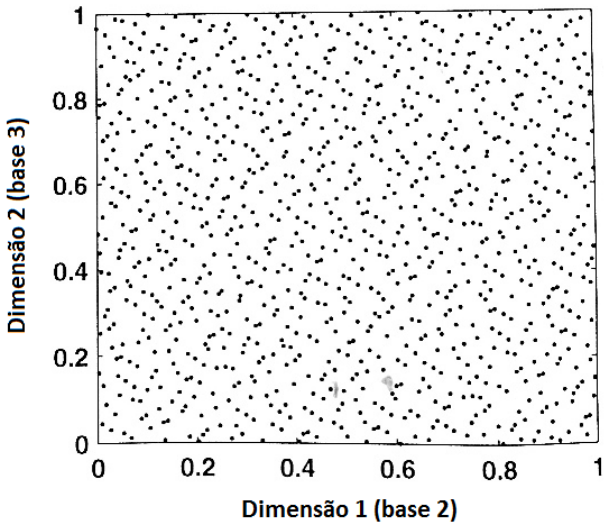

Figura 5.3: *Sequência de Halton bi-dimensional (Huynh et al., 2011)*

Sequência de Faure

A sequência de Faure é parecida com a sequência de Halton. Ela também utiliza a sequência de Van Der Corput de forma multidimensional. Contudo, Faure utiliza a mesma base para todas as dimensões. Além disso, os n-ésimos termos gerados são resultado de uma permutação dos $(n-1)$-ésimos termos gerados, conforme recursão:

$$\begin{pmatrix} a_0^k(n) \\ a_1^k(n) \\ a_2^k(n) \\ \vdots \end{pmatrix} = \begin{pmatrix} \binom{0}{0} & \binom{1}{0} & \binom{2}{0} & \binom{3}{0} & \cdots \\ 0 & \binom{1}{1} & \binom{2}{1} & \binom{3}{1} & \cdots \\ 0 & 0 & \binom{2}{2} & \binom{3}{2} & \cdots \\ \vdots & \vdots & \vdots & \vdots & \ddots \end{pmatrix} \begin{pmatrix} a_0^{k-1}(n) \\ a_1^{k-1}(n) \\ a_2^{k-1}(n) \\ \vdots \end{pmatrix}$$

5.1.3 Distribuições não Uniformes

Apesar dos geradores uniformes apresentados na seção 5.1.1 formarem a base para a maioria dos procedimentos estocásticos, outras sequências de distribuição não uniforme podem ser construídas

a partir dessas.

Método de Inversão

O método mais simples de geração de distribuições não uniformes é por inversão direta da função de distribuição, quando possível. Suponha que desejamos gerar uma sequência distribuída de acordo com a função de densidade $p(x)$, onde x é um escalar real. A distribuição de probabilidade acumulada é

$$P[x \leq z] = P(z) = \int_{-\infty}^{z} p(t)dt.$$

Se for possível obter P^{-1}, então podemos gerar uma sequência de números distribuídos não uniformemente e mutuamente independentes como

$$x_i = P^{-1}(u_i),$$

onde $u_1, u_2, \ldots$ é uma sequência de números uniformemente distribuídos e mutuamente independentes de números reais pertencentes ao intervalo $(0, 1)$.

Método de Aceitação e Rejeição

Suponha que conheçamos um método para simular uma variável aleatória Y com função de densidade de probabilidade $g(x)$. O método de aceitação e rejeição utiliza essa função como base para simular a distribuição de X que tem como função de densidade de probabilidade $f(x)$. Para isso, devemos simular Y e aceitar esse valor com uma probabilidade proporcional a $f(Y)/g(Y)$.

Assim, seja c uma constante tal que

$$\frac{f(y)}{g(y)} \leq c, \quad \forall y.$$

Deve-se simular Y tantas vezes quanto se queira e devemos aceitar o valor dessa variável aleatória, em cada simulação, como uma amostra para X se

$$U \leq \frac{f(Y)}{cg(Y)}$$

onde $U \sim U[0, 1]$.

Transformada Gaussiana

A distribuição Gaussiana é um importante exemplo de distribuição não uniforme. Por isso, há diversas formas de transformação de uma distribuição uniforme para uma distribuição normal, como o método de Ziggurat (Marsaglia e Tsang, 2000) e o método Polar (Miller *et al.*, 2010).

Como GPUs são sensíveis a laços e desvios de fluxo, o método de Box-Muller (Box e Muller, 1958) é uma excelente escolha para transformação Gaussiana. Nesse método, duas amostras uniformes u_0 e u_1 pertencentes ao intervalo $(0, 1)$ são tomadas e em seguida transformadas em duas amostras normais n_0 e n_1 por meio da relação:

$$n_0 = \sin(2\pi u_0)\sqrt{-2log(u_1)}$$
$$n_1 = \cos(2\pi u_0)\sqrt{-2log(u_1)}$$

Em (Lee *et al.*, 2009) foi demonstrado um *speedup* de 170 vezes no cálculo do método de Box-Muller em GPU.

Para uma comparação detalhada de diferentes técnicas para geração de variáveis aleatórias normais veja (Roy, 2002).

5.2 Técnicas de Paralelização de PRNGs

Para gerar números aleatórios em um computador paralelo podemos nos basear em um RNG serial que distribui sua sequência gerada sequencialmente ao longo dos processadores disponíveis. Uma maneira mais moderna seria parametrizar o RNG de modo diferente de acordo com o processador escolhido. Com esse objetivo, há diversas técnicas de obtenção de *streams* paralelos de números aleatórios que exploram a divisão do ciclo do RNG em saltos na transição de estados ou mesmo no particionamento da sequência principal em um dado gerador de sub-sequencias. A seguir, apresentamos algumas das principais técnicas para esse fim.

5.2.1 *Central Server*

A técnica *Central Server* consiste em um servidor central rodando um RNG provendo números pseudo-aleatórios por demanda para diferentes processadores lógicos (PL). Algumas desvantagens de essa técnica são: simulações nessa abordagem são de difícil reprodução e o servidor central pode se tornar um gargalo ao se considerar muitos PLs.

5.2.2 *Sequence Splitting*

Também conhecido como *Regular Spacing*, a técnica *Sequence Splitting* consiste em dividir de modo determinístico uma sequência de estados em blocos contíguos sem sobreposição. Dada uma sequência $S_i, i = 0, 1, ...$ e N *streams*, o j-ésimo *stream* contem a sequência $S_{p+(j-1)m}, p = 0, 1, ...$, onde m é o tamanho do *stream*. Assim, se a sequência original é

$$S_0, S_1, S_2, \ldots, S_{N-1}, S_N, \ldots, S_{2N-1}, S_{2N},$$

então a sequência gerada para o *stream* 0 é

$$S_0, S_1, S_2, \ldots, S_{N-1}.$$

Apesar de geradores sequenciais com correlações de grande intervalo serem comuns, tal particionamento contíguo pode levar a correlação em intervalos menores (Matteis e Pagnutti, 1990).

5.2.3 *Random Spacing*

Random Spacing constrói partições de N *streams* ao inicializar o mesmo gerador com N estados aleatórios. Dessa forma, as sementes do gerador são produzidas por outro RNG. Essa técnica é interessante quando o período dos geradores é muito grande, o que diminui a probabilidade de sobreposição nos *streams*. Apesar dessa técnica ser de fácil uso, sempre há o risco de má inicialização das sementes. Em 2009, (Reuillon *et al.*, 2008) propuseram um milhão de estados para o Mersenne Twister e demonstraram uma perda de qualidade estatística quando há um número desbalanceado de 0s e 1s nos estados iniciais binários do algoritmo.

5.2.4 *Leap Frog*

Dados n *streams*, cada *stream* na técnica *Leap Frog* seleciona números que estão n posições distantes entre si em relação a sequência original gerada. Assim, dada uma sequência $S_i, i = 0, 1, ...$ e N *streams*, o j-ésimo *stream* contem a sequência $S_{pN+(j-1)}, p = 0, 1, ...$, onde p é o período da sequência original e p/N o período de cada *stream*. Dessa forma, se a sequência original é

$$S_0, S_1, S_2, \ldots, S_{N-1}, S_N, \ldots, S_{2N-1}, S_{2N},$$

então a sequência gerada para o *stream* 0 é dado por:

$$S_0, S_{N-1}, S_{2N-1}, \ldots$$

A geração de números aleatórios em saltos como esse é muitas vezes referenciado como uma técnica de *skip-ahead*.

5.3 *Multiple Recursive Generator* MRG32k3a em GPU

5.3.1 Formulação

L'Ecuyer estudou CMRGs (*Combined Multiple Recursive Generators*) visando a um gerador que tivesse boas propriedades aleatórias e uma implementação simples. O mais conhecido CMRG é o MRG32k3a (Fischer *et al.*, 1999), que é definido por

$$\begin{array}{rcll} y_{1,n} & = & (a_{12}y_{1,n-2} + a_{13}y_{1,n-3}) & \text{mod} \quad m_1, \\ y_{2,n} & = & (a_{21}y_{2,n-1} + a_{23}y_{2,n-3}) & \text{mod} \quad m_2, \\ x_n & = & (y_{1,n} + y_{2,n}) & \text{mod} \quad m_1, \end{array} \tag{5.1}$$

para todo $n \geq 3$, onde

$$\begin{array}{rclrclrcl} a_{1,2} & = & 14403580, & a_{1,3} & = & -810728, & m_1 & = & 2^{32} - 209 \\ a_{2,1} & = & 527612, & a_{2,3} & = & -1370589, & m_2 & = & 2^{32} - 22853 \end{array}$$

e x_n representa a n-ésima saída do gerador. Dessa forma, o estado do gerador é definido pelo par de vetores

$$Y_{i,n} = \begin{pmatrix} y_{i,n} \\ y_{i,n-1} \\ y_{i,n-2} \end{pmatrix}$$

para $i = 1, 2$. Com isso, a recorrência da equação 5.1 pode ser expressa como

$$Y_{i,n} = A_i Y_{i,n} \text{ mod } m_i,$$

onde A_i é uma matriz de constantes e, portanto,

$$Y_{i,n+p} = A_i^p Y_{i,n} \text{ mod } m_i \tag{5.2}$$

para um parâmetro p constante.

5.3.2 Paralelização

A equação 5.2 é uma forma eficiente de transição arbitrária entre estados do MRG. Assim, a técnica *Sequence Splitting* poderia ser utilizada apropriadamente para paralelização de tal gerador. Nessa estratégia, cada *thread* tem uma cópia do estado e produz um segmento contíguo da sequência MRG32k3a independente de todas as outras *threads*.

Para um cálculo eficiente do produtório matricial A_i^p com grandes valores de p, pode-se utilizar uma estratégia de "Divisão e Conquista" de elevar ao quadrado iterativamente a matriz A_i (Knuth, 1997). A começar pela decomposição em fatores binários do expoente

$$p = \sum_{j=0}^{k} g_j 2^j,$$

onde $g_j \in \{0, 1\}$, calcula-se a sequência

$$A_i, A_i^2, A_i^4, A_i^8, A_i^{16}, \ldots, A_i^{2^k}, \text{ mod } m_i$$

Com isso, chegamos a

$$A_i^p Y_i = \prod_{j=0}^{k} A^{g_j 2^j} \bmod m_i,$$

Dessa forma, o cálculo pode ser efetuado em $O(\log_2 p)$ passos em vez de $O(p)$, como no algoritmo original. Tal procedimento pode ser generalizado para uma decomposição de p em uma base b qualquer:

$$p = \sum_{j=0}^{k} g_j b^j,$$

com $g_j \in \{0, 1, \ldots, b-1\}$. Uma base maior aumentaria a velocidade do procedimento ao custo de mais memória para armazenar as potências de A_i na expansão de p.

5.3.3 Implementação

Primeiramente, é necessário armazenar em memória de *host* o conjunto de matrizes $A_i^{g_j b^k}$:

$$\begin{array}{cccc} A_i & A_i^2 & \cdots & A_i^{b-1} \\ A_i^b & A_i^{2b} & \cdots & A_i^{(b-1)b} \\ \vdots & \vdots & \cdots & \vdots \\ A_i^{b^k} & A_i^{2b^k} & \cdots & A_i^{(b-1)b^k} \end{array}$$

Como um exemplo, podemos tomar $b = 8$ e $k = 20$. Como o estado é pequeno, aproximadamente 10KB de memória, toda a matriz de potências pode ser copiada para a memória constante da GPU, o que provê grande aumento de desempenho.

Para gerar um total de N números aleatórios, um *kernel* pode ser configurado com T *threads* no total, e a i-ésima *thread* com $1 \leq i \leq T$ deve avançar seu estado em $(i-1)N/T$ posições e gerar $p = N/T$ números. Assim, o j-ésimo número aleatório gerado seria indexado por

$$prngID = j + p * threadIdx.x + p * blockIdx.x * blockDim.x.$$

Note que, como cada *thread* gera um segmento contíguo de números aleatórios, tal armazenamento em memória global não é *coalesced*. *Coalesced memory* pode ser obtida indexando o j-ésimo número aleatório por

$$prngID = threadIdx.x + j * blockDim.x + p * blockIdx.x * blockDim.x$$

5.4 Sobol em GPU

Sobol (Sobol, 1967) propôs sua sequência como um método alternativo para integração numérica em um hipercubo unitário. A ideia é construir uma sequência que preenche um cubo de maneira regular. Então, a integral é aproximada como uma média dos valores da função desses pontos. Essa abordagem é eficiente em grandes dimensões onde métodos tradicionais de quadratura numérica são custosos.

5.4.1 Formulação

Para gerar o j-ésimo componente de pontos de uma sequência de Sobol, escolhemos um polinômio primitivo de um grau s_j no corpo $\mathbb{Z}_2$,

$$x^{s_j} + a_{1,j} x^{s_j - 1} + a_{2,j} x^{s_j - 2} + \ldots + a_{s_j - 1, j} x + 1 \tag{5.3}$$

onde os coeficientes $a_{1,j}, a_{2,j}, \ldots, a_{s_j-1,j} \in \{0,1\}$. Definimos uma sequência de inteiros positivos $\{m_{1,j}, m_{2,j}, \ldots\}$ pela relação de recorrência

$$m_{k,j} = 2a_{1,j}m_{k-1,j} \oplus 2^2 a_{2,j}m_{k-2,j} \oplus \ldots \oplus 2^{s_j-1}a_{s_j-1,j}m_{k-s_j+1,j} \tag{5.4}$$

onde $\oplus$ denota o operador OR exclusivo bit-a-bit. Os valores iniciais $m_{1,j}, m_{2,j}, \ldots$ podem ser escolhidos livremente contanto que cada $m_{k,j}$, $1 \leq k \leq s_j$, seja ímpar e menor que 2^k. As chamadas direções da sequência $\{v_{1,j}, v_{2,j}, \ldots\}$ são definidas por

$$v_{k,j} = \frac{m_{k,j}}{2^k}$$

Assim, o j-ésimo componente do i-ésimo ponto de uma sequência de Sobol é dado por

$$x_{i,j} = i_1 v_{1,j} \oplus i_2 v_{2,j} \oplus \ldots$$

onde i tem sua representação binária dada por $i = (\ldots i_3 i_2 i_1)_2$.

Para obter de sequências multidimensionais de Sobol, tomamos direções diferentes para cada dimensão. Porém, é preciso cuidado, pois escolhas ruins nas direções levam à não uniformidade na distribuição dos pontos da sequência.

5.4.2 Paralelização

Utilizando código de Gray, Antonov e Saleev (Antonov e Saleev, 1979) encontraram uma forma simples de obter o $(i+1)$-ésimo elemento de uma sequência de Sobol a partir do i-ésimo elemento. Eles mostraram que

$$\begin{aligned} x_n &= g_1 m_1 \oplus g_2 m_2 \oplus \ldots && (5.5) \\ &= x_{n-1} \oplus m_{f(n-1)} && (5.6) \end{aligned}$$

onde, o código de Gray de n é dado por $n \oplus (n/2) = \ldots g_3 g_2 g_1$ e $f(n)$ retorna o índice do bit zero mais a direita na expansão binária de n.

Para geração de até 2^k pontos é necessário um conjunto de k números de direção. Assim, para obtenção do elemento x_n a partir do elemento x_{n-1} seriam necessárias, no máximo, k iterações, onde cada iteração faria uma inversão de bit e, possivelmente, uma operação XOR. Dessa forma, podemos concluir que a equação 5.5 fornece um modo eficiente de implementação de uma estratégia do tipo *skip-ahead*, já que é possível realizar saltos sem muito custo. Portanto, o algoritmo poderia ser paralelizado tal qual na paralelização do MRG32k3a apresentada, onde cada *thread* gera um bloco de pontos e depois realiza um *skip-ahead*.

Ao adicionarmos 2^p em $n = (\ldots b_3 b_2 b_1)_2$, podemos notar que os p primeiros bits de n se mantem inalterados: adicionar 1 ao número n em 2^p vezes resulta em alternar b_1 em 2^p vezes, b_2 em 2^{p-1} vezes e, assim, por diante. Considere, agora, o comportamento da função $f(n+i)$ para $1 \leq i \leq 2^p$: como estamos enumerando todas as permutações dos primeiros p bits de n, então, $f(n+i) = 1$ em 2^{p-1} vezes, $f(n+i) = 2$ em 2^{p-2} vezes e assim, por diante. Portanto, $f(n+i) = j$ em 2^{p-j} vezes, para $j \in \{1, 2, \ldots, p\}$. Note que, $f(n+1)$ retornará um índice q maior do que p apenas uma vez, já que os primeiros p bits valem todos 1 uma única vez. Sabendo que duas operações de XOR se cancelam, podemos reescrever a equação 5.5 como:

$$\begin{aligned} x_{n+2^p} &= x_n \oplus \overbrace{m_1 \oplus \ldots \oplus m_1}^{2^{p-1} vezes} \oplus \overbrace{m_2 \oplus \ldots \oplus m_2}^{2^{p-2} vezes} \oplus \ldots \oplus m_p \oplus m_q \\ &= x_n \oplus m_p \oplus m_q && (5.7) \end{aligned}$$

Assim, podemos utilizar a equação 5.7 em uma estratégia eficiente do tipo *leapfrog*, na qual cada *stream* possui um espaçamento de tamanho 2^p.

5.4.3 Implementação

As direções m_i podem ser pré-computadas no *host* e copiadas para o *device*. Em uma sequência de Sobol de 32-bits temos no máximo 32 valores de m_i. Como as dimensões de uma sequência D-dimensional de Sobol são independentes, podemos utilizar um bloco para cada dimensão. Assim, para cada dimensão, cada bloco deve lançar 2^p *threads* e os 32 valores de m_i podem ser armazenados na memória compartilhada, que é mais veloz. Dentro do bloco, a i-ésima *thread* deve computar x_i pela equação original 5.5 e, então, deve fazer o *skip-ahead* dado pela equação 5.7. Note que *threads* sucessivas geram números de Sobol sucessivos, logo obtemos *Coalesced Memory*.

5.5 Bibliotecas para RNGs em GPU

5.5.1 NVIDIA CURAND

Introduzido na versão CUDA 3.2, a biblioteca CURAND foi projetada para gerar números aleatórios de modo simples em GPGPUs com suporte a CUDA. Ela gera números quasi-aleatórios e pseudo-aleatórios tanto em GPGPU ou CPU. Exceto pela fase de inicialização do gerador, a API é praticamente a mesma em ambas plataformas.

Em uma versão mais recente, com CUDA SDK 4.0, CURAND vem com uma implementação de Sobol para geração de números quasi-aleatórios e XORWOW (Marsaglia, 2003) para números pseudo-aleatórios. Ambos podem apresentar uma aceleração de oito vezes em relação à biblioteca MKL da Intel (NVIDIA, 2011). O algoritmo PRNG escolhido pela NVIDIA é conhecido pela sua velocidade e pelo baixo uso de memória.

5.5.2 *Thrust::random*

Thrust::random faz parte de uma biblioteca de propósito geral disponível para GPGPU chamada *Thrust*. Esse projeto de código aberto tem a intenção de prover uma biblioteca para GPGPU equivalente a bibliotecas clássicas de C++ como STL ou *Boost*. Classes são divididas em diversos *namespaces*, dos quais *Thrust::random* é um exemplo. Essa biblioteca implementa três tipos de PRNGs: *Linear Congruential Generator* (LCG), *Linear Feedback Shift* (LFS) e *Substract With Carry* (SWC).

5.5.3 *ShoveRand*

ShoveRand (C. Mazel e Hill, 2011) é uma proposta de meta-modelo de representação de RNGs em qualquer plataforma. Assim, ele foi desenhado como um arcabouço para que desenvolvedores integrassem suas implementações de RNGs em GPGPU. O modelo apresenta uma hierarquia constituída por quatro classes principais: *RNG*, *Algorithm*, *ParameterizesStatus* e *SeedStatus*, como pode ser visto na figura 5.4. As duas últimas são agregadas pelas diferentes possibilidades de implementação da classe *Algorithm*. A classe *RNG*, associa-se à classe *Algorithm* e expõe uma interface ao usuário.

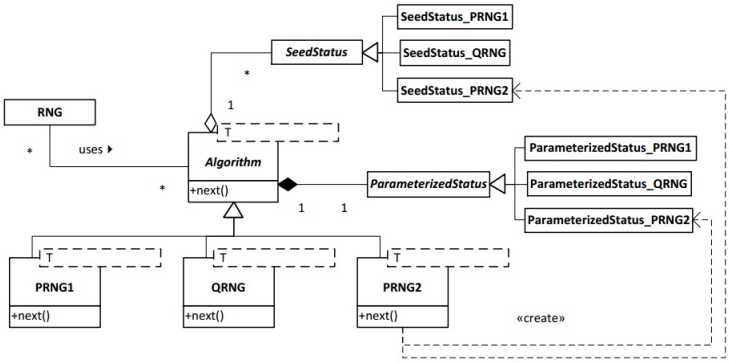

Figura 5.4: *ShoveRand meta-modelo (C. Mazel e Hill, 2011)*

Parte III

Fundamentos da Modelagem Matemática

Capítulo 6

Simulação Estocástica

A presença de incerteza é um fato em modelagem financeira. A menos em problemas de baixa complexidade, essa característica não pode ser ignorada. Assim, a modelagem de sistemas complexos e com características estocásticas vem sendo utilizada cada vez mais. Com isso, este capítulo tem o objetivo de apresentar de forma sucinta e elementar alguns aspectos da área de modelagem estocástica em finanças que serão usados nos casos de estudo do trabalho. A teoria apresentada está diretamente relacionada com aplicações e permite, por meio do método de Monte Carlo, a implementação numérica em problemas práticos.

Assim, na seção 6.1, apresentamos fundamentos de Simulações de Monte Carlo. Em seguida, nas seções 6.2 e 6.3, abordamos processos estocásticos e resolução de Equações Diferenciais Estocásticas. Na seção 6.4, estudamos métodos práticos de solução numérica dessas equações.

6.1 Fundamentos de Simulações de Monte Carlo

Em finanças, a precificação de ativos geralmente consiste em computar uma expectativa de ganho. Nesta seção, descrevemos como técnicas de simulação estocástica podem ser utilizadas para aproximar essas esperanças matemáticas.

6.1.1 Integração de Monte Carlo

O método de Monte Carlo (MC) é um método utilizado para estimar numericamente funções de variáveis aleatórias. Suas origens remontam à computação de π por Laplace e Bouffon, por meio de um experimento aleatório. Mais tarde, durante a construção da bomba atômica em Los Alamos, Von Newman and Ulam aprimoraram essa técnica para calcular integrais complexas.

A integração de Monte Carlo é motivada pela Lei dos Grandes Números: se X_i é uma coleção de variáveis aleatórias independentes e identicamente distribuídas de função densidade $q(x)$ e pertencem ao intervalo $[0, 1]$, então

$$\lim_{N\to\infty} \frac{1}{N}\sum_{i=1}^{N} X_i = \int_0^1 xq(x)dx, \quad a.s.$$

Além disso,

$$var\left(\frac{1}{N}\sum_{i=1}^{N} X_i\right) = \frac{\sigma_x^2}{N},$$

onde $\sigma_x^2 = var(X_i)$. Se σ_x^2 é desconhecido, podemos estimá-lo como

$$\hat{\sigma}_x^2 = \frac{1}{N-1}\sum_{i=1}^{N}\left(X_i - \hat{X}\right)^2,$$

onde $\hat{X}$ é a média amostral

$$\hat{X} = \frac{1}{N}\sum_{i=1}^{N} X_i.$$

Assim, a Lei dos Grandes Números sugere um procedimento de integração numérica, como demonstramos a seguir.

Suponha que desejamos computar $I_f = \int_0^1 f(x)dx$. Se $X \sim U[0,1]$, então

$$\mathbb{E}[f(x)] = \int_0^1 f(x)dx,$$

ou seja, a integral é estimada pela esperança da variável aleatória $Y = f(X)$. Para isso, o método de Monte Carlo gera N amostras $x_i \sim U[0,1]$, $i = 1, \ldots, N$ e considera

$$\hat{I}_f = \frac{1}{N}\sum_{i=1}^{N} f(x_i)$$

como uma estimativa de $\int_0^1 f(x)dx$. Dessa forma, essa aproximação é, por sua vez, uma variável aleatória, $\hat{I}_f$, com variância

$$\sigma^2_{\hat{I}_f} = \frac{1}{N}\int_0^1 (f(x) - I_f)^2 dx = \frac{\sigma_f^2}{N}$$

Como σ_f^2 é desconhecido, tomamos $\hat{\sigma}_f^2$ como estimativa de σ_f^2:

$$\hat{\sigma}_f^2 = \frac{1}{N-1}\sum_{i=1}^{N}(f(x_i) - \hat{I}_f)^2.$$

Pelo teorema central do limite, podemos afirmar que

$$\hat{I}_f \sim \mathcal{N}(\mathbb{E}[f(x)], \frac{1}{N}var(f(x))).$$

Assim, o valor estimado $\hat{I}_f$ terá um desvio da verdadeira esperança $\mathbb{E}[f(x)]$ na ordem de $1/\sqrt{N}$. Dado que $P(|Z| < 1,96) \approx 0,95$, $Z \sim \mathcal{N}(0,1)$, podemos construir o seguinte intervalo de confiança a um nível de 95% para estimativa de $\hat{I}_f$:

$$\left(\hat{I}_f - 1,96\frac{\hat{\sigma}}{\sqrt{N}}, \hat{I}_f + 1,96\frac{\hat{\sigma}}{\sqrt{N}}\right).$$

6.1.2 Técnicas de Redução de Variância

O método tradicional de Monte Carlo é comumente utilizado sem modificações. Apesar de ser um método não enviesado, sua variância pode ser muito grande. Há uma variedade de técnicas que podem reduzir substancialmente a variância da estimativa, enquanto mantém a característica importante de estimador não enviesado do método.

Variáveis Antitéticas

Para se obter a média de uma variável aleatória com grande variância pode ser necessário um grande número de simulações para atingir a precisão desejada. Em contraste, quando a variância é pequena, o número de simulações exigidas é baixo. Assim, é de grande importância encontrarmos transformações que reduzam a variância do resultado da simulação.

Definição 1. *Dizemos que a variável aleatória X^a é antitética à variável aleatória X se correl$(X, X^a) = -1$.*

A utilização de variáveis antitéticas é umas das técnicas mais simples. Ela gera N variáveis aleatórias independentes e identicamente distribuídas (i.i.d.) X_i e constrói N outras variáveis i.i.d. a partir dessas com a mesma distribuição mas com correlações perfeitamente negativas com as primeiras.

Por exemplo, seja f uma função monotonicamente crescente. Se $X \sim U[0,1]$, então $f(x)$ e $f(1-x)$ são negativamente correlacionados. Assim, a ideia da técnica é reduzir a variância de $\hat{I}_f$ ao estimar I_f com a estimativa antitética

$$\hat{I}_f^a = \frac{1}{2N}\sum_{i=1}^{N}(f(x_i) + f(1-x_i)).$$

Esse estimador antitético é um estimador não enviesado de I_f, mas possui variância menor que o estimador tradicional dado que os termos do somatório são negativamente correlacionados.

Proposição 1. *Considere um conjunto de $2N$ variáveis aleatórias i.i.d. $\{X_i\}_{i=1}^{2N}$ e a geração de N variáveis aleatórias i.i.d. $\{X_i^a\}_{i=1}^{N}$, tal que, X_i é antitética a X_i^a e $\mathbb{E}[X_i] = \mathbb{E}[X_i^a]$. Sejam $\hat{I}_f$ e $\hat{I}_f^a$, respectivamente, os estimadores de média sobre os conjuntos $\{X_i\}_{i=1}^{2N}$ e $\{X_i\}_{i=1}^{N} \cup \{X_i^a\}_{i=1}^{N}$, então $var(\hat{I}_f^a) = var(\hat{I}_f)/2$.*

Demonstração. Temos que,

$$\hat{I}_f^a = \frac{1}{2N}\sum_{k=1}^{N}(X_k + X_k^a).$$

Seja σ_X^2 a variância do estimador sobre o conjunto $\{X_i\}_{i=1}^{N}$, então

$$\begin{aligned} var(\hat{I}_f^a) = &\frac{1}{4N^2}\sum_{k=1}^{N}\sigma_X^2 + \frac{1}{4N^2}\sum_{k=1}^{N}\sigma_X^2 \\ &+ \frac{1}{4N^2}\sum_{k=1}^{N}\sum_{j=1}^{N}\mathbb{E}[(X_k - \mu_X)(X_j^a - \mu_X)]. \end{aligned}$$

Todas as variáveis X_k são independentes duas a duas, bem como as variáveis X_k^a. Assim, todas as $2N$ variáveis são independentes, exceto pelos pares X_k e X_k^a. Dessa forma, a esperança do último termo da equação anterior é zero, exceto quando $k = j$, onde

$$\mathbb{E}[(X_k - \mu_X)(X_j^a - \mu_X)] = -\sigma_X^2.$$

Assim,

$$var(\hat{I}_f^a) = \frac{1}{4N}\sigma_X^2 + \frac{1}{4N}\sigma_X^2 - \frac{1}{4N}\sigma_X^2 = \frac{1}{4N}\sigma_X^2.$$

Como,

$$var(\hat{I}_f) = \frac{1}{2N}\sigma_X^2$$

então,

$$var(\hat{I}_f^a) = \frac{var(\hat{I}_f)}{2}$$

□

Assim, pela Proposição 1, podemos concluir que é possível reduzir a variância do estimador em até duas vezes utilizando a técnica de variáveis antitéticas.

Variáveis de Controle

Como visto, a utilização de variáveis antitéticas pode reduzir a variância do estimador utilizado de forma considerável. Essa técnica explora a propriedade de correlação negativa das variáveis. Portanto, se utilizarmos outras variáveis, não necessariamente antitéticas, mas negativamente correlacionadas com as variáveis geradas, é possível reduzir a variância do estimador.

Considere uma variável aleatória X e outra variável aleatória Y correlacionada com X. Chamamos de Y, variável de controle[1] de X. Assumindo que a média de Y é conhecida, podemos construir a seguinte nova variável:

$$X^* = X + \alpha(Y - \mathbb{E}[Y]).$$

Podemos observar que a média de X^* é idêntica à de X: $m_X = m_X^*$. Assim, podemos estimar a média de X pela estimativa de média de X^*.

Proposição 2. *Se Y é uma variável de controle de X,*

$$X^* = X + \alpha(Y - \mathbb{E}[Y]),$$

então, $\exists\ \alpha \in \mathbb{R} : Var(X^*) \leq Var(X)$.

Demonstração. Temos que a variância de X^* é

$$\begin{aligned} Var(X^*) &= \mathbb{E}[(X^* - m_{X^*})^2] = \mathbb{E}[(X + \alpha(Y - \mathbb{E}[Y]) - m_{X^*})^2] \\ &= Var(X) + \alpha^2 Var(Y) + 2\alpha Cov(X, Y). \end{aligned}$$

Deseja-se que a variância seja mínima, então devemos escolher $\alpha = \alpha^*$ tal que

$$\frac{dVar(X^*)}{d\alpha} = 0.$$

Isso implica que

$$\alpha^* = -\frac{Cov(X, Y)}{Var(Y)}$$

Com essa escolha de α temos que a variância de X^* se torna

$$Var(X^*) = Var(X) - \frac{|Cov(X, Y)|^2}{Var(Y)}$$

Isso mostra que $Var(X^*) \leq Var(X)$ □

Amostragem Estratificada

A técnica de Amostragem Estratificada[2] baseia-se na observação de que a variância de uma função $f(x)$ sobre um sub-intervalo de seu domínio $x \in [0, 1]$ é comumente menor do que sobre o intervalo inteiro. A partir dessa divisão, é possível realizar mais amostragens em regiões de maior densidade. Assim, um dos objetivos dessa técnica é maximizar o número de regiões da amostra cobertas na estimativa.

[1] Denominada *Control Variate* em inglês.
[2] Denominada *Stratified Sampling* em inglês.

Por exemplo, suponha que dividamos o intervalo $[0,1]$ em $[0,\alpha]$ e $[\alpha,1]$. Então, se tivermos N_1 e N_2 pontos de amostragem no primeiro e segundo intervalos, respectivamente, temos o seguinte estimador

$$\hat{I}_f = \frac{\alpha}{N_1}\sum_{i=1}^{N_1} f(x_{1i}) + \frac{1-\alpha}{N_2}\sum_{i=1}^{N_2} f(x_{2i}),$$

onde $x_{1i} \in [0,\alpha]$, $i = 1,\ldots,N_1$ e $x_{2i} \in [\alpha,1]$, $i = 1,\ldots,N_2$. Assim, sua variância é

$$\frac{\alpha}{N_1}\int_0^\alpha f(x)^2dx - \frac{\alpha}{N_1}\left(\int_0^\alpha f(x)dx\right)^2 + \frac{1-\alpha}{N_2}\int_0^\alpha f(x)^2dx - \frac{1-\alpha}{N_2}\left(\int_0^\alpha f(x)dx\right)^2.$$

A efetividade da técnica depende da escolha do parâmetro α. Uma boa escolha para o mesmo é aquela que iguala a variância sobre $[0,\alpha]$ com aquele sobre $[\alpha,1]$. Assim, a ideia central nessa técnica é evitar a acumulação de pontos em uma região e, assim, reduzir a variância do estimador.

6.2 Definições de Cálculo Estocástico

6.2.1 Preliminares

Um espaço amostral ou espaço de eventos Ω é um conjunto com os possíveis resultados de experimentos ou estados de um sistema. Nesse conjunto, consideramos uma família $\mathcal{F}$ de subconjuntos de Ω e uma medida de probabilidade $\mathbb{P} : \mathcal{F} \to [0,1]$, conforme as definições a seguir.

Definição 2. *A família de subconjuntos $\mathcal{F} \subset 2^\Omega$ é uma σ-álgebra se*

i) $\emptyset \in \mathcal{F}$

ii) $\mathcal{F}$ *é fechada por complementos, i.e.,* $F \in \mathcal{F} \implies \bar{F} \in \mathcal{F}$.

iii) $\mathcal{F}$ *é fechada por uniões enumeráveis, i.e,* $\{F_i\}_{i=1}^\infty \subset \mathcal{F} \implies \bigcup_{i=1}^\infty F_i \in \mathcal{F}$.

Definição 3. *A medida de probabilidade $\mathbb{P}$ é uma função $\mathbb{P} : \mathcal{F} \to [0,1]$ tal que*

i) $P(\emptyset) = 0$, $P(\Omega) = 1$

ii) Se $A_1, A2, \ldots \in \mathcal{F}$ *e* $\{A_i\}_{i=1}^\infty$ *é disjunto, então*

$$\mathbb{P}\left(\bigcup_{i=1}^\infty A_i\right) = \sum_{i=1}^\infty \mathbb{P}(A_i).$$

Definição 4. *Uma variável aleatória X é uma função $X : \Omega \to \mathbb{R}^n$.*

Definição 5. *Seja X uma variável aleatória com espaço amostral Ω e medida de probabilidade $\mathbb{P}$. Então, o espaço de probabilidade gerado por X é definido por*

$$\{\Omega, \mathcal{F}, \mathbb{P}\}$$

Definição 6. *Seja $\{X_k\}_{k=1}^\infty$ uma sequência de variáveis aleatórias discretas e identicamente distribuídas. Para cada inteiro positivo n, denotamos S_n como a soma $X_1 + X_2 + \ldots + X_n$. A sequência $\{S_n\}_{n=1}^\infty$ é chamada de Passeio Aleatório.*

Propriedade 1. *Incrementos em um Passeio Aleatório são independentes e identicamente distribuídos.*

6.2.2 Processos Estocásticos

Suponha que a cotação do Dólar para cada instante fixo $t \in [a, b]$ seja aleatória. É razoável observarmos a evolução do preço da moeda em todo o intervalo $[a, b]$ para estimar um preço futuro. Além disso, seria interessante analisar o comportamento da trajetória aleatória para diferentes valores de partida da cotação do Dólar. Um modelo matemático para descrever tal fenômeno é chamado de processo estocástico.

Definição 7. *Um Processo Estocástico X é uma coleção de variáveis aleatórias*

$$(X_t, t \in T) = (X_t(\omega), t \in T, \omega \in \Omega),$$

definido em um espaço amostral Ω.

O parâmetro T é usualmente uma semi-reta $[0, \infty)$, mas também pode se referir a um intervalo $[a, b]$, o conjunto dos números naturais ou até mesmo sub-conjuntos de $\mathbb{R}^n$ para $n \geq 1$. O índice t da variável aleatória X_t é frequentemente referida como *tempo*. Podemos notar que, para um valor fixo de t, X é uma variável aleatória:

$$X_t = X_t(\omega), \omega \in \Omega.$$

Para uma amostra aleatória $\omega \in \Omega$, X é uma função do tempo:

$$X_t = X_t(\omega), t \in T.$$

Essa última função é chamada de *realização*, *trajetória* ou *caminho* do processo X.

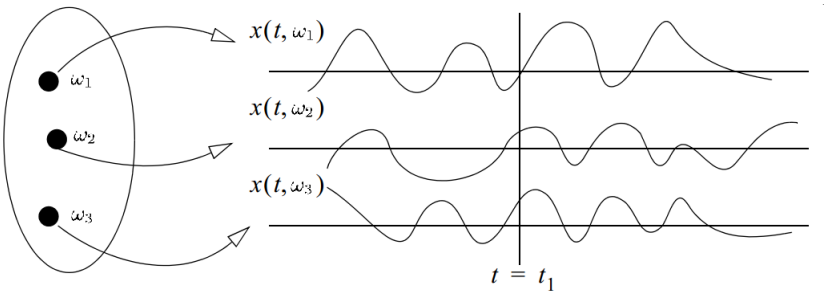

Figura 6.1: *Representação de um processo estocástico. Note que para $t = t_1$ temos que $X(t_1)$ é uma variável aleatória e que para cada evento ω_i é gerada uma trajetória correspondente.*

A esperança de um vetor de variáveis aleatórias $X = (X_1, \ldots, X_n)$ é definido como $\mu_X = (\mathbb{E}[X_1], \ldots, \mathbb{E}[X_n])$ e sua matriz de covariância como $\sum_X = (cov(X_i, X_j), i, j = 1, \ldots, n)$. Um processo estocástico $X = (X_t, t \in T)$ pode ser considerado uma coleção de vetores aleatórios $(X_t, \ldots, X_{t_n})$ para $t_1, \ldots, t_n \in T$ e $n \geq 1$. Assim, para cada vetor de variáveis aleatórias, podemos determinar suas respectivas esperanças e matrizes de covariância. Alternativamente, podemos considerar essas grandezas como funções de $t \in T$ como na definição a seguir.

Definição 8. *Seja $X = (X_t, t \in T)$ um processo estocástico.*

A função de esperança de X é dada por

$$\mu_X(t) = \mathbb{E}[X_t], \quad t \in T.$$

A função de covariância de X é dado por

$$c_X(t, s) = cov(X_t, X_s) = \mathbb{E}[(X_t - \mu_X(t))(X_s - \mu_X(s))], \quad t, s \in T.$$

A função de variância de X é dado por

$$\sigma_X^2(t) = c_X(t, t) = var(X_t), \quad t \in T.$$

Como para um vetor de variáveis aleatórias, a função de esperança $\mu_X(t)$ é uma quantidade determinística que representa a média X. A função de covariância $c_X(t,s)$ é uma medida de dependência no processo X. Por sua vez, a função de variância $\sigma_X^2(t)$ pode ser considerada uma medida de dispersão dos caminhos aleatórios em relação a $\mu_X(t)$.

6.2.3 Movimento Browniano

O Movimento Browniano tem um papel fundamental em teoria de probabilidade, teoria de processos estocásticos e ciências em geral. Em finanças, trata-se do processo estocástico mais comumente utilizado para simulação de preços de ativos e será base para modelagem dos estudos de casos em tempo contínuo desse trabalho. Para defini-lo, é importante enunciar o conceito de incrementos estacionários e independentes para processos estocásticos em geral.

Definição 9. *Seja $X = (X_t, t \in T)$ um processo estocástico e $T \subset \mathbb{R}$ um intervalo. X é dito ter incrementos estacionários se*

$$X_t - X_s \overset{d}{=} X_{t+h} - X_{s+h}, \text{ para todo } t, s \in T \text{ e } t+h, s+h \in T.$$

X tem incrementos independentes se para cada escolha de $t_i \in T$ com $t_1 < \ldots < t_n$ e $n \geq 1$,

$$X_{t_2} - X_{t_1}, \ldots, X_{t_n} - X_{t_{n-1}}$$

são variáveis aleatórias independentes.

Definição 10. *Um processo estocástico $B = (B_t, t \in [0, \infty))$ é chamado Processo de Wiener ou Movimento Browniano Padrão se:*

i) O processo tem seu início em zero: $B_0 = 0$

ii) Possui incrementos estacionários e independentes.

iii) Para todo $t > 0$, B_t tem uma distribuição normal $\mathcal{N}(0, t)$.

iv) O processo tem caminhos contínuos.

Segue da definição que um Movimento Browniano tem uma função de esperança

$$\mu_B(t) = \mathbb{E}[B_t] = 0, \quad t \geq 0,$$

e como os incrementos $B_s - B_0 = B_s$ e $B_t - B_s$ são independentes para $t > s$, sua função de covariância é

$$\begin{aligned} c_B(t,s) &= \mathbb{E}[[(B_t - B_s) + B_s]B_s] = \mathbb{E}[(B_t - B_s)B_s] + \mathbb{E}[B_s^2] \\ &= E(B_t - B_s)\mathbb{E}[B_s] + s = 0 + s = s, \quad 0 \leq s < t. \end{aligned}$$

Definição 11. *Considere o processo*

$$X_t = \mu t + \sigma B_t, \quad t \geq 0,$$

com constantes $\sigma > 0$ e $\mu \in \mathbb{R}$. X é chamado de Movimento Browniano com drift (linear) e possui as seguintes funções de esperança e covariância, respectivamente

$$\mu_X(t) = \mu t \text{ e } c_X(t,s) = \sigma^2 s, \quad s, t \geq 0 \text{ com } s < t.$$

O movimento browniano, como um processo Gaussiano, pode assumir valores negativos, o que pode ser uma propriedade não desejável para um preço de ativo financeiro. No célebre trabalho de (Black e Scholes, 1973), os autores sugeriram outro processo estocástico para o modelo de preços. Esse modelo é o Movimento Browniano Geométrico.

Definição 12. *Considere o processo*

$$X_t = e^{\mu t + \sigma B_t}, \quad t \geq 0,$$

com constantes $\sigma > 0$ *e* $\mu \in \mathbb{R}$. X *é chamado de Movimento Browniano Geométrico.*

Assim, o Movimento Browniano Geométrico é a exponencial do movimento Browniano com *drift*.

Proposição 3. *O Movimento Browniano Geométrico possui as seguintes funções de esperança e covariância, respectivamente*

$$\begin{aligned} \mu_X(t) &= e^{(\mu+0.5\sigma^2)t} \quad e \\ c_X(t,s) &= e^{(\mu+0.5\sigma^2)(t+s)}(e^{\sigma^2 s-1}), \quad s,t \geq 0 \text{ com } s < t \end{aligned}$$

Demonstração. Temos que,

$$B_t = B_t - B_0 \sim N(0, \Delta t) \sim \sqrt{\Delta t} N(0,1).$$

Seja Z uma variável aleatória tal que $Z \sim N(0,1)$, então

$$\mu_X(t) = e^{\mu t} E[e^{\sigma B_t}] = e^{\mu t} E[e^{\sigma t^{1/2} Z}].$$

Sabemos que,

$$E[e^{\lambda Z}] = e^{\lambda^2/2}, \quad \lambda \in \mathbb{R}.$$

Então,

$$E[e^{\sigma t^{1/2} B_1}] = e^{0.5\sigma^2 t}$$

Portanto,

$$\mu_X(t) = e^{\mu t} e^{0.5\sigma^2 t} = e^{(\mu+0.5\sigma^2)t}$$

Para $s \leq t$, $B_t - B_s$ e B_s são independentes e $B_t - B_s \stackrel{d}{=} B_{t-s}$. Então,

$$\begin{aligned} c_X(t,s) &= E[X_t X_s] - E[X_t]E[X_s] \\ &= e^{\mu(t+s)} E[e^{\sigma(B_t+B_s)}] - e^{(\mu+0.5\sigma^2)(t+s)} \\ &= e^{\mu(t+s)} E[e^{\sigma[(B_t-B_s)+2B_s]}] - e^{(\mu+0.5\sigma^2)(t+s)} \\ &= e^{\mu(t+s)} E[e^{\sigma(B_t-B_s)}] E[e^{2\sigma B_s}] - e^{(\mu+0.5\sigma^2)(t+s)} \\ &= e^{(\mu+0.5\sigma^2)(t+s)}(e^{\sigma^2 s-1}) \end{aligned}$$

□

6.2.4 Simulação do Movimento Browniano

A visualização de processos estocásticos serve como uma boa ferramenta para entendimento dos mesmos. Em muitos casos, não é possível determinar a distribuição do processo estocástico e suas propriedades. Assim, simulações e técnicas numéricas oferecem uma alternativa para o cálculo dessas distribuições. Nessa seção apresentamos uma forma de simulação do movimento Browniano.

Algoritmo Geral

Dado um incremento fixo $\Delta t > 0$, é possível simular a trajetória do processo de Wiener no intervalo $[0, T]$. De fato, para $W_{\Delta t}$ é verdade que

$$W(\Delta t) = W(\Delta t) - W(0) \sim N(0, \Delta t) \sim \sqrt{\Delta t} N(0, 1),$$

e o mesmo também é verdade para um incremento em tempo arbitrário $W(t + \Delta t) - W(t)$:

$$W(t + \Delta t) - W(t) \sim N(0, \Delta t) \sim \sqrt{\Delta t} N(0, 1).$$

Assim, podemos simular a trajetória do processo estocástico conforme Algoritmo 2.

Algoritmo 2 Simulação do Movimento Browniano

Input: Intervalo [0,T]
Output: $W = (W_t, t \in [0, T))$, Movimento Browniano Padrão

Divida o intervalo $[0, T]$, tal que $0 = t_1 < t_2 < \ldots < t_{N-1} < t_N = T$ com $t_{i+1} - t_i = \Delta t$.
$W(0) \leftarrow W(t_1) \leftarrow 0$
for $i = 2 \rightarrow N$ **do**
  Gere z, tal que $z \sim \mathcal{N}(0, 1)$
  $W(t_i) \leftarrow W(t_{i-1}) + z\sqrt{\Delta t}$
end for

Representação por Séries

Como sabemos, uma função periódica contínua em $\mathbb{R}$ pode ser representada como uma série de Fourier, isto é, pode ser escrita como uma série infinita de funções trigonométricas. Como caminhos de um movimento Browniano são funções contínuas, é natural pensar em sua expansão em séries de Fourier. Entretanto, esses caminhos são funções aleatórias: para cada ω obtemos uma função diferente. Isso significa que os coeficientes das séries de Fourier devem ser variáveis aleatórias. A seguinte representação de um movimento Browniano no intervalo $[0, 2\pi]$ é chamado de *Representação de Paley-Wiener*:

$$B_t(\omega) = Z_0(\omega)\frac{t}{(2\pi)^{1/2}} + \frac{2}{\pi^{1/2}}\sum_{n=1}^{\infty} Z_n(\omega)\frac{\sin(nt/2)}{n}, \quad t \in [0, 2\pi],$$

onde $(Z_n, n \geq 0)$ corresponde a uma sequência de variáveis aleatórias gaussianas independentes.

Para uma aplicação dessa fórmula, é necessário decidir sobre o número M de funções seno e o número N de pontos de discretização nos quais essas funções serão realizadas:

$$Z_0(\omega)\frac{t}{(2\pi)^{1/2}} + \frac{2}{\pi^{1/2}}\sum_{n=1}^{M} Z_n(\omega)\frac{\sin(nt_j/2)}{n}, \quad t_j = \frac{2\pi j}{N}, \quad j = 0, 1, \ldots, N.$$

Na verdade, a Representação de Paley-Wiener é uma de infinitas séries possíveis de representação de um movimento Browniano. Ela é um caso especial da chamada Representação de Lévy-Ciesielski (Hida, 1980). Ciesielski mostrou que um movimento Browniano em $[0, 1]$ pode ser representado na forma

$$B_t(\omega) = \sum_{n=1}^{\infty} Z_n(\omega)\int_0^t \phi_n(x)dx,$$

onde Z_n corresponde a uma sequência de variáveis aleatórias gaussianas independentes no intervalo

$[0,1]$ e $\phi_n(x)$ deve possuir algumas propriedades especiais, como a formação de um sistema completo de funções ortonormais (Kaplan, 1991).

6.3 Equações Diferenciais Estocásticas

6.3.1 Integral de Itô

Definição 13. *A integral de Itô é definida como*

$$\int_0^t B(\tau)dW(\tau) = \lim_{n\to\infty} \sum_{k=0}^{n-1} B(t_k)[W(t_{k+1}) - W(t_k)], \text{ onde } t_k = k\frac{t}{n} \tag{6.1}$$

No caso particular onde $B(t)$ é uma função determinística, essa integral é chamada de Integral de Wiener.

Considere a seguinte equação

$$X(t) = X(0) + \int_0^t a(X(\tau),\tau)d\tau + \int_0^t b(X(\tau),\tau)dW(\tau). \tag{6.2}$$

Em um intervalo infinitesimal, podemos re-escrever essa equação em sua forma diferencial:

$$dX(t) = a(X(t),t)dt + b(X(t),t)dW(t), \tag{6.3}$$

onde $W(t)$ representa um processo de Wiener e $a(X(t),t)$ e $b(X(t),t)$ são, respectivamente, as variações instantâneas da média e desvio padrão.

Genericamente, podemos escrever

$$X(t) = X(0) + \int_0^t A(\tau)d\tau + \int_0^t B(\tau)dW(\tau), \tag{6.4}$$

onde $A(\tau)$ e $B(\tau)$ são funções de $X(t)$ para $0 \leq \tau \leq t$.

Estamos interessados no caso em que

$$\int_0^t \mathbb{E}[|A(\tau)|]d\tau + \int_0^t \mathbb{E}[|B(\tau)|^2]d\tau < \infty. \tag{6.5}$$

Processos que são soluções dessa equação são chamados de Processos de Itô.

Propriedades de um Processo de Itô

Propriedade 2. *Se X é um processo de Itô, então*

$$\mathbb{E}[\int_0^t X(\tau)dW(\tau)] = 0 \tag{6.6}$$

e

$$Var\left(\int_0^t X(\tau)dW(\tau)\right) = \int_0^t E[X^2(\tau)d\tau] \tag{6.7}$$

Propriedade 3. *Linearidade*
Se X e Y são dois processos de Itô e a e b duas constantes, então

$$\int_0^t (aX(\tau) + bY(\tau))\, dW(\tau) = a\int_0^t X(\tau)dW(\tau) + b\int_0^t Y(\tau)dW(\tau). \tag{6.8}$$

Propriedade 4. *Segue da propriedade de linearidade anterior que*

$$\int_0^t aW(\tau) = a\int_0^t dW(\tau) = aW(t). \tag{6.9}$$

6.3.2 Lema de Itô

Considere X um processo uni-dimensional definido como:

$$dX(t) = a(X(t),t)dt + b(X(t),t)dW(t). \tag{6.10}$$

Seja $Y(t) = g(t, X(t))$, onde g é duplamente diferenciável e contínua. Então, o Lema de Itô diz que

$$\begin{aligned} dY = &\left(\frac{\partial g}{\partial t} + a(X(t),t)\frac{\partial g}{\partial x} + \frac{1}{2}b^2(X(t),t)\frac{\partial^2 g}{\partial x^2}\right)dt \\ &+ \left(b(X(t),t)\frac{\partial g}{\partial x}\right)dW. \end{aligned}$$

Similarmente ao caso uni-dimensional, a mesma regra aplica-se para o caso multi-dimensional. Seja $\bar{X} \in \mathbb{R}^n$ um vetor de variáveis aleatórias com processo definido como

$$d\bar{X}(t) = A(\bar{X}(t),t)dt + B(\bar{X}(t),t)d\bar{W}(t). \tag{6.11}$$

onde $A(\bar{X}(t),t) \in \mathbb{R}^n, \bar{W} \in \mathbb{R}^n$ e $B(\bar{X}(t),t) \in \mathbb{R}^{n\times m}$.

Fazendo

$$\bar{Y}(t) = \bar{g}(t, \bar{X}(t)) = (Y_1(t,\ldots,Y_d(t)))^T \tag{6.12}$$

com $\bar{g} : \mathbb{R}\times\mathbb{R}^n \to \mathbb{R}^d$, então o Lema de Itô generalizado é

$$dY_k(t) = \frac{\partial g_k}{\partial t}dt + \sum_i \frac{\partial g_k}{\partial x_i}dX_i(t) + \frac{1}{2}\sum_i \frac{\partial^2 g_k}{\partial x_i \partial x_j}dX_i(t)dX_j(t) \tag{6.13}$$

A seguir, demonstramos a proposição 4, como exemplo de aplicação do Lema de Itô enunciado.

Proposição 4.

$$\int_0^t W(\tau)dW(\tau) = \frac{1}{2}W^2(t) - \frac{1}{2}t.$$

Demonstração. Seja $X(t) = W(t)$, escolha g tal que

$$\begin{aligned} Y(t) =& g(t, X(t)) \\ =& \frac{1}{2}W^2(t). \end{aligned}$$

Aplicando o Lema de Itô, temos

$$dY(t) = W(t)dW(t) + \frac{1}{2}dt.$$

Assim,

$$\begin{aligned} \int_0^t W(\tau)dW(\tau) =& Y(t) - Y(0) - \frac{1}{2}t \\ =& \frac{1}{2}W^2(t) - \frac{1}{2}t. \end{aligned}$$

□

Processo de Ornstein-Uhlenbeck

Em finanças, o processo de Ornstein-Uhlenbeck é um processo de reversão à média[3] muito comum para descrever a dinâmica de taxas de juros e volatilidades estocásticas de retornos de ativos. Sua definição é dada pela equação a seguir:

$$dX(t) = \theta(\mu - X(t))dt + \sigma dW(t), \tag{6.14}$$

onde $W(t)$ é um processo padrão de Wiener, σ é o parâmetro de volatilidade do processo e representa o nível de intensidade das perturbações estocásticas, μ é o valor ao qual o processo possui reversão e θ representa a velocidade dessa reversão.

A seguir, descrevemos a solução do processo de Ornstein-Uhlenbeck, cujo desenvolvimento pode ser encontrado em (Alves, 2008).

Proposição 5. *A solução do processo de Ornstein-Uhlenbeck definido pela equação 6.14 é dado por*

$$X(t) = \mu + (X(0) - \mu)e^{-\theta t} + \sigma \int_0^t e^{-\theta(t-s)} dW(s)$$

Demonstração. Temos que,

$$\begin{aligned} dX(t) &= \theta(\mu - X(t))dt + \sigma dW(t), \\ dX(t) + \theta X(t)dt &= \theta\mu dt + \sigma dW(t), \\ d(e^{\theta t}X(t)) &= \theta\mu e^{\theta t} + \sigma e^{\theta t} dW(t). \end{aligned}$$

Integrando em t, temos

$$\int_0^t d(e^{\theta s}X(s))ds = \int_0^t \theta\mu e^{\theta s} ds + \sigma \int_0^t e^{\theta s} dW(s).$$

Assim,

$$X(t)e^{\theta t} - X(0) = \mu e^{\theta t} - \mu + \sigma \int_0^t e^{\theta s} dW(s).$$

Portanto,

$$X(t) = \mu + (X(0) - \mu)e^{-\theta t} + \sigma \int_0^t e^{-\theta(t-s)} dW(s).$$

□

6.4 Soluções Numéricas de Equações Diferenciais Estocásticas

Antes de introduzir soluções numéricas de equações diferenciais estocásticas, apresentaremos discretizações numéricas de equação diferenciais ordinárias, como um ponto de partida. Isso facilitará o entendimento das discretizações de Euler-Maruyama e Milstein que serão cobertas nas seções posteriores.

[3] Reversão à média é um conceito matemático utilizado em investimentos financeiros para designar que um fator de risco tende a um valor médio.

6.4.1 Equações Diferenciais Ordinárias

Considere a seguinte equação diferencial ordinária

$$dx = a(x,t)dt, \quad x(t_0) = x_0.$$

Definimos,

$$t_{n+1} = t_n + \Delta t, \qquad x(t_{n+1}) = x_n + \delta x = x_{n+1}$$

e $y_{n+1} = y(t_{n+1})$ como a aproximação linear de x_{n+1}, ou seja,

$$y(t_n + \Delta t) \approx y(t_n) + \frac{dy}{dt}\Delta t.$$

Método de Euler

A aproximação de Euler considera a seguinte discretização

$$y(t_n + \Delta t) \approx y(t_n) + \frac{dy}{dt}\Delta t,$$

que corresponde a uma aproximação de primeira ordem da série de Taylor.

Método de Runge-Kutta

O método de Runge-Kutta é baseado na seguinte aproximação

$$y_{n+1} = y_n + \frac{1}{6}(k_n^1 + 2k_n^2 + 2k_n^3 + k_n^4)\Delta t,$$

onde

$$\begin{aligned}
k_n^1 &= a(y_n, t_n), \\
k_n^2 &= a(y_n + \frac{1}{2}k_n^1\Delta t, t_n + \frac{1}{2}\Delta t), \\
k_n^3 &= a(y_n + \frac{1}{2}k_n^2\Delta t, t_n + \frac{1}{2}\Delta t), \\
k_n^4 &= a(y_n + k_n^3\Delta t, t_{n+1}).
\end{aligned}$$

Esse é conhecido como algoritmo de quarta-ordem de Runge-Kutta. O mesmo possui erro de $O(\Delta t^4)$.

Método Implícito

Nesse método, consideramos a média

$$\frac{a(y_n, t_n) + a(y_{n+1}, t_{n+1})}{2},$$

que leva a aproximação numérica

$$y_{n+1} = y_n + \frac{a(y_n, t_n) + a(y_{n+1}, t_{n+1})}{2}\Delta t.$$

Enquanto que Euler e Runge-Kutta são métodos de aproximação explícita, esse método é implícito pois a solução é obtida de modo iterativo.

6.4.2 Equações Diferenciais Estocásticas

Considere a seguinte equação diferencial estocástica

$$\begin{cases} dS(t) = a(S(t),t)dt + b(S(t),t)W(t), t \in (0,T] \\ S(0) = S_0 \end{cases} \tag{6.15}$$

onde $b : \mathbb{R}^d \times [0,T] \to \mathbb{R}^d$ é conhecido e $W(\cdot)$ representa um processo de Wiener. Um método padrão para mostrar a existência de solução para o sistema de equações de 6.15 é fornecido em (Narayana *et al.*, 2012) e apresentado a seguir.

Seja

$$S^{i+1}(t) = S_0 + \int_0^t a(S^i(u))du + \int_0^t b(S^i(u))dW(u) \tag{6.16}$$

para $i \geq 1$ e $S^0 = S_0$ para todo i. Então, $S^i \to S$ quando $i \to \infty$, onde S é a solução da equação 6.15.

Assim, a equação 6.16 fornece uma forma de obtenção de uma solução numérica das equações em 6.15. Há dois métodos principais na literatura de como aproximar as integrais enunciadas: a Discretização de Euler e a Discretização de Milstein.

Discretização de Euler-Maruyama

Para $N \in \mathbb{N}$, seja $h = \frac{T}{N}$. Denotemos S_i, W_i, a_i para o i-ésimo componente de S, W, a e b_{ij} para a ij-ésima entrada de b.

Seja

$$\begin{aligned} S_i((k+1)h) = S_i(kh) &+ \int_{kh}^{(k+1)h} a_i(S(u))du \\ &+ \sum_j^m \int_{kh}^{(k+1)h} b_{ij}(S(u))dW_j(u) \end{aligned}$$

Na discretização de Euler, a integral é aproximada como

$$\int_{kh}^{(k+1)h} a_i(S(u))du \approx a_i(S(kh))h \tag{6.17}$$

e a Integral de Itô como

$$\int_{kh}^{(k+1)h} b_{ij}(S(u))dW_j(u) \approx b_{ij}(S(kh))[W_j((k+1)h) - W_j(kh)] \tag{6.18}$$

Discretização de Milstein

A diferença da Discretização de Milstein para Discretização de Euler está na forma como a Integral de Itô é aproximada. Na Discretização de Euler, não são considerados os termos de maior ordem na série de expansão de Taylor da Integral de Itô, enquanto a Discretização de Milstein os consideram. Isso faz com que o esquema de Milstein seja mais preciso e tenha maior taxa de convergência.

Para $N \in \mathbb{N}$, seja $h = \frac{T}{N}$. Denotemos S_i, W_i, a_i para o i-ésimo componente de S, W, a e b_{ij} para a ij-ésima entrada de b.

Seja

$$\begin{aligned} S_i((k+1)h) =& S_i(kh) + \int_{kh}^{(k+1)h} a_i(S(u))du \\ &+ \sum_{j}^{m} \int_{kh}^{(k+1)h} b_{ij}(S(u))dW_j(u). \end{aligned}$$

Na discretização de Milstein, a integral é aproximada como

$$\int_{kh}^{(k+1)h} a_i(S(u))du \approx a_i(S(kh))h \tag{6.19}$$

e a Integral de Itô como

$$\begin{aligned} \int_{kh}^{(k+1)h} b_{ij}(S(u))dW_j(u) &\approx b_{ij}(S(kh))[W_j((k+1)h) - W_j(kh)] \\ &+ \sum_{l=1}^{d} \sum_{m=1}^{d} \frac{\partial b_{ij}}{\partial x_l}(S(kh))b_{lm} \int_{kh}^{(k+1)h} [W_m(u) - W_m(kh)]dW_j(u) \end{aligned}$$

Uma dedução da aproximação de Milstein pode ser obtida em (Mikosch, 1999).

Cálculo da Esperança

Se tomarmos $a(S(t), t) = \mu S_t$ e $b(S(t), t) = \sigma S_t$ na Equação 6.15, então chegamos ao seguinte modelo

$$\begin{cases} dS_t = \mu S_t dt + \sigma S_t W(t), t \in (t_0, T] \\ S(t_0) = S_0 \end{cases} \tag{6.20}$$

Os possíveis valores assumidos por S_t são, então, gerados ao sub-dividir o tempo $T - t_0$ em M passos temporais discretos de tamanho $\delta t = \frac{T-t_0}{M}$.

Assim, pela discretização de Euler chegamos a

$$S_{i+1} = S_i + \mu S_i \delta t + \sigma S_i \sqrt{\delta t} x \tag{6.21}$$

Utilizando a discretização de Milstein temos

$$S_{i+1} = S_i + \mu S_i \delta t + \sigma S_i \sqrt{\delta t} x + \frac{1}{2}\sigma^2 S_i(\delta t x^2 - \delta t) \tag{6.22}$$

onde $x \sim \mathcal{N}(0, 1)$ e $i = 1, \ldots, M$.

Com isso, podemos construir o Algoritmo 3 para estimar $\theta = E[f(S_t)]$, ou seja, a esperança de uma função sobre a variável aleatória S_t. O algoritmo supõe um número de caminhos n fixo e um intervalo de discretização h.

Por meio da equação 6.22, podemos construir algoritmo análogo para discretização de Milstein.

Algoritmo 3 Esperança de uma função de variável aleatória via discretização de Euler

Input: f, S_0, n, h
Output: $\theta = E[f(S_t)]$

$t \leftarrow 0$
$\hat{S} \leftarrow S_0$
for $j = 1 \rightarrow n$ **do**
  for $k = 1 \rightarrow T/h$ **do**
    $Z \sim N(0,1)$
    $\hat{S} \leftarrow \hat{S} + \mu(t, \hat{S})h + \sigma(t, \hat{S})\sqrt{h}Z$
    $t \leftarrow t + h$
  end for
  $f_j \leftarrow f(\hat{S})$
end for
$\theta \leftarrow \sum_{j=1}^{n} f_j/n$

Parte IV

Estudo de Casos

Capítulo 7

Método

Este capítulo descreve o ambiente computacional e a configuração de execução da análise experimental dos estudos de casos das próximas seções. A configuração aqui enunciada foi utilizada em todos os experimentos das próximas seções a menos que seja dito o contrário.

7.1 Ambiente Computacional

7.1.1 GPU

O código em GPU foi desenvolvido em CUDA C utilizando o compilador nvcc (CUDA Toolkit v.3.1) com a opção *-use_fast_math* para otimização de funções matemáticas. Foi utilizado uma placa gráfica NVIDIA GeForce GT 525M de arquitetura Fermi, que possui 96 núcleos CUDA. A tabela 7.1 sumariza as principais características da mesma.

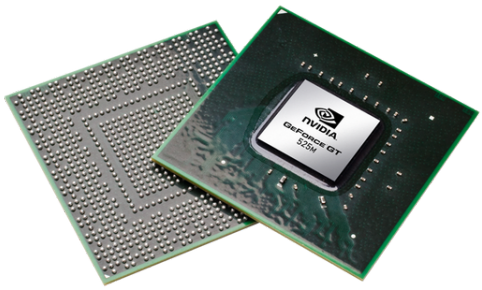

Figura 7.1: *NVIDIA GeForce GT 525M*

Placa Gráfica	GeForce GT 525M
Núcleos CUDA	96
Compute Capability	2.1
Streaming Multiprocessors	2
Dimensões máximas de um bloco	1024 x 1024 x 64
Número máximo de *threads* por bloco	1024
Número máximo de registradores por bloco	32768
Memória Global	961 MB
Memória Compartilhada	48 KB
Memória Constante	64 KB

Tabela 7.1: *Características da placa gráfica NVIDIA GeForce GT 525M.*

7.1.2 CPU

O código em CPU foi desenvolvido em C/C++ utilizando o compilador *Microsoft (R) C/C++ Optimizing Compiler Version 16.00.30319.01 for x64* em um processador Intel Core i5-2430M. Os programas em CPU foram construídos utilizando uma única *thread*.

7.2 Tempo Computacional e Medidas

O tempo computacional foi medido com temporizadores de CPU, conforme elucidado na seção 4.1. Todos os tempos foram medidos considerando todo o tempo gasto no programa, ou seja, também foi considerado o tempo de transferência de dados via PCIe, no caso de GPU. A métrica de desempenho utilizada foi a de *speedup*.

O erro padrão calculado nas simulações considerou o intervalo a um nível de confiança de 95%.

7.3 Dados Disponíveis

Os preços das ações utilizadas nos experimentos foram retirados do site da BM&FBovespa (http://www.bmfbovespa.com.br/). Foram considerados os preços de fechamento do ativo.

7.4 Geração de Números Aleatórios

Em CPU, o gerador padrão utilizado foi o Mersenne Twister, com implementação retirada de (Matsumoto e Nishimura, 2009). Em GPU foi utilizada a biblioteca padrão NVIDIA CURAND. Nos experimentos que utilizaram o gerador Sobol, a implementação foi obtida em (Joe e Kuo, 2008). Para transformada gaussiana foi utilizado o método de Box-Muller.

7.5 Configuração de Execução

Seguindo as boas práticas enunciadas no capítulo 4, as funções de GPU foram lançadas com um número de *threads* por bloco como um múltiplo de 64. Empiricamente, o valor que resultou em maior ocupação, em geral, foi o de 128 *threads* por blocos. Esse foi o valor utilizado em todos os estudos de caso.

A figura 7.2 ilustra a configuração básica de execução dos experimentos realizados. Primeiramente, medimos o tempo do experimento em CPU. Em seguida, iniciamos a contagem de tempo de cálculos em GPU. De modo geral, a paralelização é feita na geração dos números aleatórios em GPU e também na dimensão das simulações realizadas.

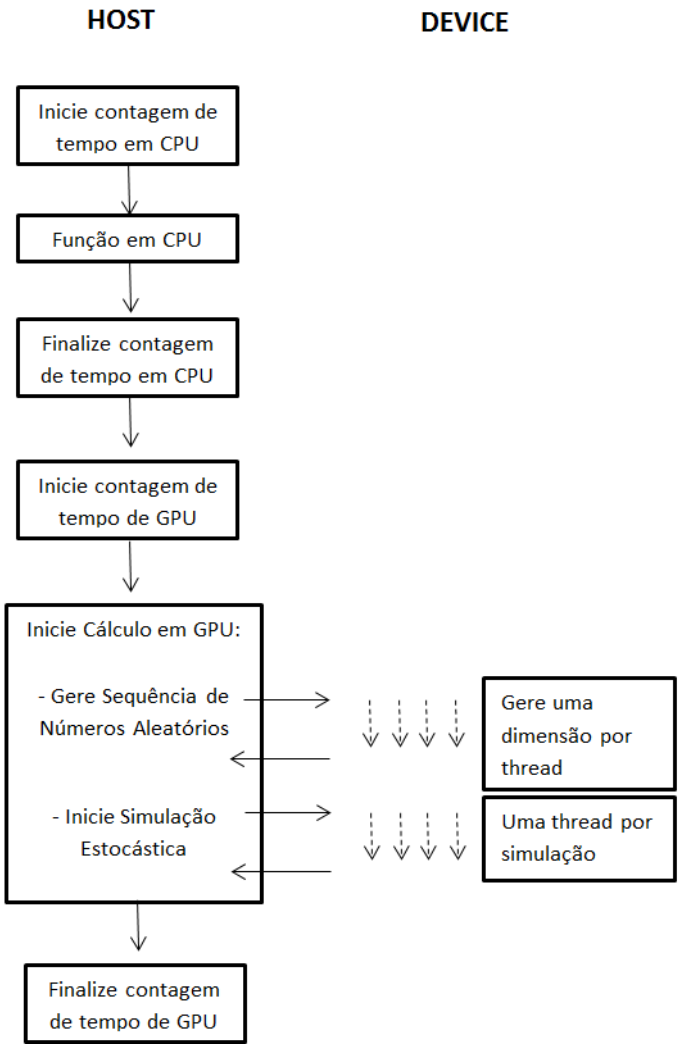

Figura 7.2: *Arquitetura geral de execução de um estudo de caso.*

Capítulo 8

Stops Ótimos

A decisão de venda de um ativo é crucial para o sucesso em investimentos no mercado financeiro. A estratégia de venda pode ser determinada, por exemplo, tanto por um ganho alvo ou um limite de perda. Em negociação de ações, uma ordem de *Stop Loss* tenta vender o ativo quando seu preço da ação cai a um certo valor pré-determinado. Assim, a estratégia de *Stop Loss* é modelada para limitar as perdas a uma região segura. De modo análogo, na estratégia de *Stop Gain*, o negociador define um limite superior para o valor do ativo de negociação. Ao ser atingido esse nível, o investidor deixa sua posição e tem seu ganho conforme pré-determinado. Há dois tipos de *Stop*: fixo ou móvel. No primeiro caso, um valor alvo é pré-determinado na ordem realizada. No caso de um *Stop* móvel, o preço para executar a ordem pode variar com o tempo.

A utilização de uma estratégia financeira de *Stop* pode ser justificada por diferentes razões como:

- Redução da frequência de negociação e, consequentemente, dos custos totais de operação
- Fornece uma maneira simples de controle de perdas ou ganhos
- Em caso de negociação automatizada, permite níveis de recalibração do modelo de estratégia empregado

Funções de ganho similares aos resultantes de uma estratégia de *Stop* podem ser obtidas por simulação correspondente em Opções com Barreiras. Entretanto, essas Opções não são comumente negociadas em Bolsa de Valores. Assim, negociadores de varejo, geralmente, devem recorrer a técnicas de *Stop* como proteção alternativa. Veja (Hull, 2012) para uma introdução a Opções com Barreiras.

(Zhang, 2001) determina uma regra de venda ótima para um ativo cujo retorno segue um modelo estocástico correlacionado ao retorno de um portfólio de mercado. (Shen e Wang, 2001) consideram uma ordem de *Stop Loss* móvel que incrementa com o avanço do tempo. Nesse trabalho, não são considerados efeitos de portfólio em caso de configuração de *Stop* por todos os participantes do mercado.

(Imkeller e Rogers, 2010) realizam uma modelagem estocástica no qual o retorno da posição segue um movimento Browniano. São analisadas as seguintes situações: (i) *Stops* Fixos; (ii) *Stop Gain* Móvel; (iii) *Stop Gain* Móvel com *Stop Loss* Fixo; (iv) *Stops* Convergentes: onde a diferença entre os limites superior e inferior tendem a um valor fixo. Ao analisar cada um desses cenários, os autores apresentaram as seguintes conclusões: (i) a incerteza sobre a taxa de retorno do ativo é determinante na escolha dos *Stops*; (ii) somente há necessidade de *Stop Loss* caso uma taxa de retorno prevista seja negativa; (iii) ao utilizar um *Stop* superior fixo, não houve diferenças significativas entre *Stops* inferiores fixos ou de subida. Contudo, o tempo médio em negociação é significativamente menor em uma estratégia de *Stop Loss* de subida. Assim, a estratégia recomendada pelos autores é a de *Stop* superior fixo com *Stop* inferior de subida.

Em (Warburtona e Zhang, 2006), os autores definem um modelo computacional em tempo discreto para análise probabilística de *Stops* fixos. Além de analisar as estratégias de *Stop Loss* e *Stop*

Gain, os autores estudam a definição de pontos de entrada de negociação, ou seja, o usuário inicia ou termina negociação dependendo de limites definidos para o valor do ativo.

Na seção 8.1 formalizamos o problema de *Stops*. Na seção 8.2 discutimos o modelo em tempo discreto proposto por (Warburtona e Zhang, 2006) e apresentamos um algoritmo paralelo em CUDA baseado nesse trabalho. Na seção 8.3 descrevemos um modelo estocástico simplificado para o problema, bem como um algoritmo em CUDA para sua resolução. Finalmente, na seção 8.5, realizamos uma análise experimental das modelagens propostas.

8.1 Formulação do Problema

Suponha que uma partícula se move aleatoriamente em um sub-conjunto $D \subset \mathbb{R}^n$. No instante t sua posição é x_t. Desejamos escolher o *Tempo de Parada* t do movimento de modo a maximizar o ganho esperado $\mathbb{E}[g(x_t)]$, onde $g(x)$ representa o ganho correspondente a $x \in D$ (vide figura 8.1).

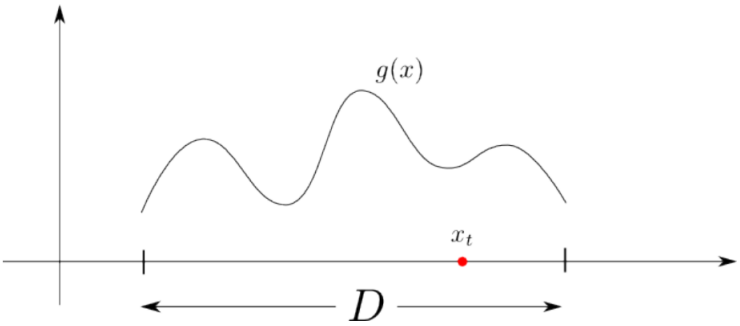

Figura 8.1: *O problema de Stop*

O termo Tempo de Parada tem sua origem na teoria de apostas. A variável x_t pode representar o ganho acumulado (ou perda) de um apostador no tempo t. Um apostador pragmático pode desejar uma regra para decidir no tempo t se é hora de sair do jogo ou não. Como jogadores, geralmente, não têm a capacidade de prever o futuro, essa regra deve se basear somente no histórico do jogo até o momento.

Para formular o problema rigorosamente, necessitamos de uma definição formal para o conceito de Tempo de Parada.

Definição 14. *Tempo de Parada (Oksendal, 1992)*
Seja $\{N_t\}$ uma família crescente de σ-álgebras (de sub-conjuntos de Ω). A função $\tau : \Omega \to [0, \infty]$ é chamado de tempo de parada se

$$\{\omega; \tau(\omega) \leq t\} \in N_t, \quad \forall t \geq 0.$$

Assim, intuitivamente, a decisão de parar no instante t deve depender apenas da informação (σ-álgebra) disponível até t. A figura 8.2 ilustra exemplos de Tempo de Parada.

Dessa forma, nosso problema pode ser formalizado como:

$$\begin{gathered} \max \mathbb{E}^{x_0}[g(x_\tau)] \\ \text{sujeito a } \tau \text{ tempo de parada} \end{gathered} \tag{8.1}$$

Com isso, $g(x_\tau)$ é o ganho que obteremos ao parar o processo no tempo τ. Desejamos maximizar o valor esperado do ganho para as trajetórias que tem inicio em x_0. A variável do problema é τ, um tempo de parada, ou seja, temos um problema de otimização no espaço abstrato dos tempos de parada.

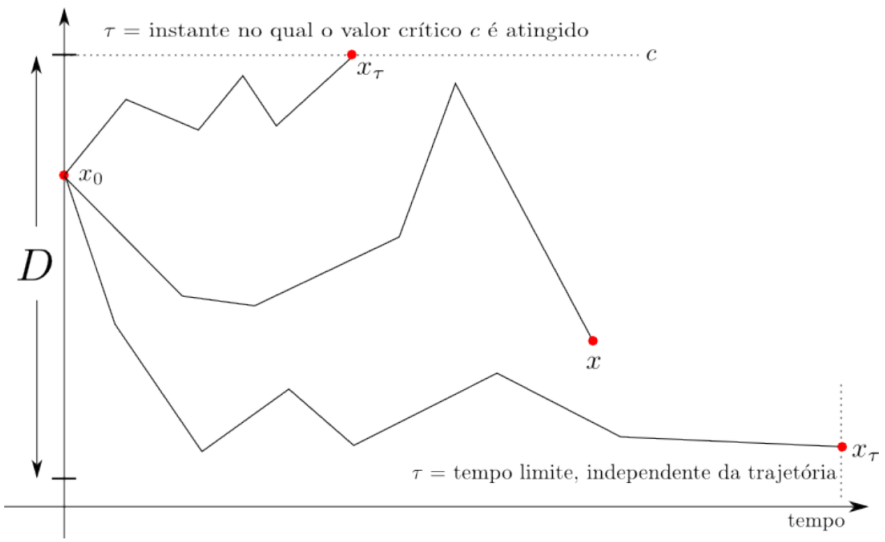

Figura 8.2: *Exemplos de Tempo de Parada*

8.2 Modelagem em Tempo Discreto

8.2.1 O Modelo Recursivo de (Warburtona e Zhang, 2006)

Em (Warburtona e Zhang, 2006), os autores propõem um modelo baseado em uma árvore trinomial para o passeio aleatório dos preços. Dessa forma, o valor do ativo pode subir um nível, continuar constante ou descer um nível. O processo para assim que uma barreira é atingida ou ao final do horizonte de tempo. O modelo apresenta as seguintes premissas:

(i) Preço do ativo segue um passeio aleatório

(ii) Horizonte de tempo é finito

(iii) *Stops* são fixos

Sejam,

- T: o número total de intervalos de tempo no horizonte de tempo indexados por $t = 0, 1, \ldots, T$
- Δ: o tamanho de cada intervalo de tempo
- H: o tamanho do horizonte de tempo, $H = T\Delta$
- L: o valor de *Stop Loss* (barreira inferior)
- K: o valor de *Stop Gain* (barreira superior)
- Ω: o espaço de probabilidade de eventos (nível de preço x a cada instante t)
- $P(t, x)$: probabilidade do processo estar no estado (t, x)
- $S(t, x)$: o preço do ativo se o processo está no estado (t, x)
- $p(t, x), q(t, x), r(t, x)$: as probabilidades do preço subir um nível, continuar inalterado, e descer um nível, respectivamente, no estado (t, x). $p(t, x) + q(t, x) + r(t, x) = 1$

Considere, $B(t, x)$ como os predecessores diretos de (t, x):

$$B(t, x) = \{(t-1, y) : (t-1, y) \in \Omega, y \in \{x-1, x, x+1\}\}$$

Seja $I_{B(t,x)}(t-1,y)$ a função booleana definida por

$$I_{B(t,x)}(t-1,y) = \begin{cases} 1, & \text{se } (t-1,y) \in B(t,x) \\ 0, & \text{caso contrário} \end{cases}$$

Então, a seguinte função de probabilidade é definida para computar $P(t,x), (t,x) \in \Omega$:

$$\begin{aligned} P(0,0) &= 1 \\ P(t,x) &= P(t-1,x-1)I_{B(t,x)}(t-1,x-1)p(t-1,x-1) \\ &\quad + P(t-1,x)I_{B(t,x)}(t-1,x)r(t-1,x) \\ &\quad + P(t-1,x+1)I_{B(t,x)}(t-1,x+1)q(t-1,x+1), \\ &\quad (t,x) \in \Omega \backslash (0,0). \end{aligned} \tag{8.2}$$

A figura 8.3 exemplifica movimentos possíveis para os preços do ativo.

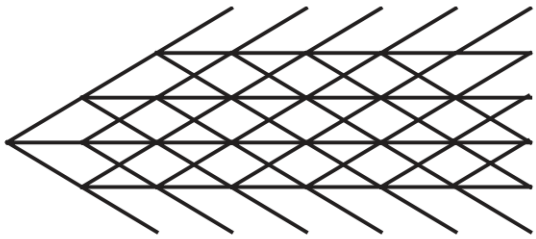

Figura 8.3: *Movimentos de preço possíveis para* $T = 7, K = 3, L = -2$

Um investimento termina assim que um nó sorvedouro é atingido. Seja, Ω_a o conjunto de estados sem sucessores (nós sorvedouros). Então, define-se a Probabilidade Terminal como $P(t,x), (t,x) \in \Omega_A$. A Distribuição de Probabilidade Terminal fornece uma medida conveniente de critério de investimento em uma estratégia de *Stop*.

Seja τ o tempo em um estado terminal (Tempo de Parada), então

$$\mathbb{E}[\tau] = \Delta \sum_{(t,x)\in\Omega_A} tP(t,x) \tag{8.3}$$

e

$$Var[\tau] = \Delta \sum_{(t,x)\in\Omega_A} (t - \mathbb{E}[\tau])^2 P(t,x) \tag{8.4}$$

Também é possível calcular a probabilidade de se atingir a barreira superior (*Stop Gain*) ou inferior (*Stop Loss*):

$$P(StopGain) = \sum_{(t,K_t)\in\Omega_A} P(t,K_t) \tag{8.5}$$

$$P(StopLoss) = \sum_{(t,L_t)\in\Omega_A} P(t,L_t) \tag{8.6}$$

Para implementação computacional da recursão 8.2 é necessário definir uma distribuição para probabilidade de transição do preço do ativo. Uma premissa comum em modelagem de preços de ativos é que os mesmos seguem um Processo de Wiener e que μ e σ são constantes. Neste caso, para $\Delta \to 0$, um Passeio Aleatório converge para um Processo de Wiener, como visto em (Huynh *et al.*, 2011). Assim, uma possibilidade seria a utilização de uma Gaussiana como Distribuição de Probabilidade de Transição.

Por sua vez, (Cox *et al.*, 1979) apresenta um modelo de transição de preços em um modelo binomial. Esse foi o modelo utilizado por (Warburtona e Zhang, 2006). A seguir, detalhamos melhor essa maneira de precificação discreta.

8.2.2 Aproximação Binomial de (Cox *et al.*, 1979)

Premissas do modelo de precificação:

(i) O preço do ativo aumenta a uma taxa u com probabilidade p e decresce a uma taxa d com probabilidade $1-p$, onde $u > 1$ e $d < 1$ em um período Δ

(ii) O ativo não paga dividendos

(iii) A taxa livre de risco[1] r_f é positiva e constante com $d < r_f < u$.

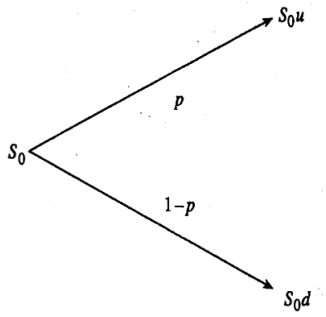

Figura 8.4: *Transição no preço do ativo (Cox et al., 1979)*

(Cox *et al.*, 1979) mostram que, em uma hipótese de não-arbitragem[2], temos que:

$$u = e^{\sigma\sqrt{\Delta}} \tag{8.7}$$

$$d = e^{-\sigma\sqrt{\Delta}} \tag{8.8}$$

$$p = \frac{e^{\mu\Delta} - d}{u - d} \tag{8.9}$$

Com isso, em um modelo binomial com períodos de tamanho $\Delta/2$ utilizamos as probabilidades de transição p_b e q_b dadas por $p_b = (e^{\mu\Delta/2} - d_{\Delta/2})/(u_{\Delta/2} - d_{\Delta/2})$, $q_b = 1 - p_b$, onde $u_{\Delta/2} = e^{\sigma\sqrt{\Delta/2}}$, $d_{\Delta/2} = 1/u_{\Delta/2}$. Assim, notando que uma árvore trinomial pode ser construída com dois passos em uma árvore binomial, as probabilidades de transição do modelo trinomial são $p = p_b^2$, $q = q_b^2$, $r = 2p_bq_b$. Dessa forma, a cada passo o preço do ativo cresce por um fator u ou decresce por um fator d, onde $ud = 1$ e $u = u_{\Delta/2}^2$, $d = 1/u$.

Passeios aleatórios para precificação de ativos têm sido estudados extensivamente. Veja (Cox *et al.*, 1979), (Hull, 2012), (Huynh *et al.*, 2011). Como visto, (Cox *et al.*, 1979) fornece uma modelagem com probabilidades de transição constantes: assumindo que K e L são constantes, são derivadas expressões analíticas para $P(t, K)$ e $P(t, L)$. Contudo, o esforço computacional gasto para calcular essas expressões é no mínimo tão custoso quanto na implementação da recursão de (Zhang, 2001). Recursão essa também presente no modelo de (Warburtona e Zhang, 2006), como vimos.

A modelagem proposta por (Warburtona e Zhang, 2006) é de difícil implementação em uma plataforma de GPU devido à recursão sugerida em sua função de probabilidade proposta. Na verdade, a funcionalidade de recursão inexiste em placas gráficas de *Compute Capability* 1.x. A seguir, apresentamos um algoritmo alternativo para o cálculo de probabilidade de *Stop* com modelo trinomial

[1] Uma taxa de juro livre de risco é aquela que pode ser auferida sem que se assuma qualquer risco. Isso que dizer que se tivermos uma quantia Q de reais, é garantido que teremos Qe^{r_fT} daqui a T anos se aplicarmos a uma taxa livre de risco r_f. No Brasil, essa taxa é geralmente relacionada aos títulos soberanos do Governo Federal.

[2] Grosso modo, arbitragem refere-se a uma situação onde se pode obter ganho sem risco.

via Simulação Estocástica. A solução proposta tem paralelismo inerente e é de fácil portabilidade em GPU.

8.2.3 Simulação Estocástica do Modelo Trinomial

Como vimos, em um modelo trinomial temos três estados possíveis para transição de preços: subida, descida ou o ativo tem seu preço inalterado. Assim, considere o espaço de probabilidade $\Omega = \{Subida, Descida, Constante\}$. Seja X uma variável aleatória, tal que $X(\omega) \in \{u, 1, d\}$, onde ω pertence ao espaço Ω. Então, pelo modelo de transição de preços de (Cox *et al.*, 1979), definimos uma função de probabilidade $U_3(\omega)$ como:

$$U_3(\omega) = \left\{ \begin{array}{l} P(\omega : X(\omega) = u) = p, \\ P(\omega : X(\omega) = 1) = q, \\ P(\omega : X(\omega) = d) = r \end{array} \right\}$$

Assim, podemos construir o Algoritmo 4 utilizando a técnica de aceitação e rejeição para geração da distribuição de probabilidade $U_3(x)$ de transição de preços. Esse algoritmo é baseado no caso geral do método de aceitação e rejeição, cuja corretude pode ser obtida em (Glasserman, 2004).

Algoritmo 4 Gera $X \sim U_3(x)$, distribuição de probabilidade trinomial de preços

Input: $U_3(x), u, d$
Output: $X \sim U_3(x)$
1: Defina $g(x)$ como uma distribuição uniforme sobre o conjunto $\{u, 1, d\}$
2: $c \leftarrow max \left\{ \frac{U_3(x)}{g(x)} \right\}$
3: Gere $y \sim g(x)$
4: Gere $U \sim U[0, 1]$
5: **if** $U \leq \frac{U_3(y)}{cg(y)}$ **then**
6: $\quad X \leftarrow y$
7: **else**
8: $\quad$ Rejeite y e volte para o passo 3
9: **end if**

De posse do gerador $U_3(x)$, podemos simular um caminho pseudo-aleatório para o preço do ativo até que o horizonte de tempo limite seja alcançado ou a barreira de *Stop* superior seja atingida. Podemos repetir esse processo um número muito grande de vezes e, ao final, contar quantas vezes o *Stop* superior foi atingido e, assim, calcular a probabilidade de *Stop Gain*. O Algoritmo 5 apresenta esse algoritmo de Simulação Estocástica proposto em CUDA.

É fácil notar que o Algoritmo 5 proposto pode ser adaptado para cálculo de *Stop Loss* ao alterar a condição da linha 7 para comparação de barreira inferior (i.e. $S_0 \leq L$).

8.2.4 Notas sobre *Stops* Móveis

Em contraste com *Stops* Fixos, ao usarmos *Stops* móveis, quando o preço de um ativo aumenta, o valor de *Stop Loss* configurado é atualizado. Nesta seção, apresentamos uma breve análise teórica do problema de *Stops* Móveis em um horizonte de tempo ilimitado.

Considere um ativo com preços movendo-se de acordo com o modelo trinomial visto, exceto pelo T que deverá ser ilimitado. Sejam K e $M = -L$ inteiros positivos que definem os valores de *Stop Gain* e *Stop Loss*, respectivamente. Se o preço cai a um valor $-M$ antes que K seja atingido, a operação tem um *Stop*. Caso contrário, se o preço alcança o valor K atualizamos o valor de *Stop Loss* para $K - M$. Assim, se o preço atinge a barreira superior $(j-1)K$, $j > 1$, atualizamos o valor de *Stop Loss* para $(j-1)K - M$.

A probabilidade do ativo atingir uma barreira superior K antes que atinja seu valor de *Stop* M é dada pelo resultado clássico da Ruína do Apostador (Cover, 1987), no qual um apostador tem a

Algoritmo 5 *Kernel*: Calcula Probabilidade de *Stop Gain*

Input: $S_0, \mu, \sigma, r, K, \Delta, T$
Output: $P[K]$, probabilidade de *Stop Gain*

```
1:
2: id ← blockDim.x * blockIdx.x + threadIdx.x
3: S[id] = 0
4: for i = 1 → T do
5:    u3 ~ U3
6:    S0 ← S0 * u3
7:    if S0 ≥ K then
8:       S[id] = 1
9:       break
10:   end if
11: end for
12:
13: Após execução do Kernel, faça em CPU
14: P[K] = (Σ_{i=0}^{NSimulacoes-1} S[i]) / NSimulacoes
```

probabilidade P de ganhar K antes de perder M:

$$P = \frac{1-(q/p)^M}{1-(q/p)^{K+M}} \tag{8.10}$$

Para que o valor de barreira $(j-1)K - M$ seja atingido, é necessário que o valor de *Stop Gain* tenha sido atingido $(j-1)$ vezes antes que o valor de *Stop Loss* seja alcançado. Assim, a probabilidade desse evento ocorrer é dada pela distribuição geométrica

$$U(j) = P^{j-1}(1-P) \tag{8.11}$$

Dessa forma, a esperança e variância da variável aleatória X que conta o número de saltos que ocorrem antes que a estratégia sofra um *Stop* são, respectivamente,

$$\mathbb{E}[X] = 1/(1-P) \tag{8.12}$$

$$Var[X] = P/(1-P)^2 \tag{8.13}$$

De posse da distribuição de probabilidade do número de saltos, chegamos a um valor esperado para o preço em caso de *Stop*

$$KP/(1-P) - M \tag{8.14}$$

onde sua variância é dada por

$$K^2P/(1-P)^2. \tag{8.15}$$

8.3 Modelagem Estocástica

Nas seções anteriores, calculamos a probabilidade do evento de *Stop* utilizando uma técnica de precificação baseada em não-arbitragem para descrição do passeio aleatório dos preços. Uma alternativa de modelagem muito comum em finanças é a suposição de um processo estocástico seguido pelos preços do ativo. Nesta seção, apresentaremos os passos gerais de uma modelagem desse tipo ao resolver, de modo alternativo, o problema de *Stop* visto até aqui com um modelo estocástico. Com esse modelo, apresentaremos um método para obtenção da esperança de ganho em uma estratégia de *Stop Gain* e, ao final, resolveremos o problema original de obtenção do *Stop* ótimo.

Considere o processo

$$dS = \mu S dt + \sigma S dW, \tag{8.16}$$

onde μ e σ são constantes e dW representa um processo de Wiener.

Seja

$$Y = \ln S,$$

pelo Lema de Itô, temos que

$$dY = \left(\frac{\partial Y}{\partial t} + \mu S \frac{\partial Y}{\partial S} + \frac{1}{2}\sigma^2 S^2 \frac{\partial^2 Y}{\partial S^2} \right) dt + \sigma S \frac{\partial Y}{\partial S} dW.$$

Como

$$\frac{\partial Y}{\partial S} = \frac{1}{S}, \qquad \frac{\partial^2 Y}{\partial S^2} = -\frac{1}{S^2}, \qquad \frac{\partial Y}{\partial t} = 0,$$

então,

$$dY = \left(\mu - \frac{\sigma^2}{2} \right) dt + \sigma dW. \tag{8.17}$$

Como μ e σ são constantes, essa equação indica que $Y = \ln S$ segue um processo de Wiener. O mesmo possui, então, média $\mu - \sigma^2/2$ e variância com taxa constante σ^2.

O incremento em $\ln S$ de um tempo 0 a um tempo futuro T segue, portanto, uma distribuição normal com média $(\mu - \sigma^2/2)T$ e variância $\sigma^2 T$. Isso significa que

$$\ln S_T - \ln S_0 \sim \mathcal{N}(\left(\mu - \sigma^2/2\right) T, \sigma^2 T).$$

ou

$$\ln S_T \sim \mathcal{N}(\ln S_0 + \left(\mu - \sigma^2/2\right) T, \sigma^2 T).$$

Processos com essas propriedades são de comum uso para descrever a dinâmica de ativos financeiros e podem ser caracterizados como Processos Log-Normais.

Assim, podemos re-escrever a equação 8.17 em função de S como:

$$\ln S_T = \ln S_0 + \left(\mu - \sigma^2/2\right) \Delta t + \sigma \Delta W$$

e podemos chegar na seguinte recorrência geral para o valor do ativo objeto no tempo

$$S_{i+1} = S_i e^{\left(\mu - \sigma^2/2\right)\delta t + \sigma\sqrt{\delta t} z}, \quad z \sim \mathcal{N}(0,1). \tag{8.18}$$

Note que, se $S_0 > 0$, temos um processo no qual o valor do ativo objeto sempre assume valores positivos, o que é uma característica desejável em muitas situações reais.

Dessa forma, podemos estimar o valor esperado ($\mathbb{E}[S(K)]$) do ganho de um *Stop Gain* conforme *kernel* proposto no Algoritmo 6. Nesse algoritmo, o valor obtido na venda do ativo é descontada de uma taxa de juros r, relativo ao tempo de negociação. Com isso, como critério de parada, além do valor de *Stop*, também é considerado um valor ϵ limite para desconto de taxa de juros, a partir do qual o ganho seria, aproximadamente, nulo.

Algoritmo 6 *Kernel*: Valor Esperado do Ganho de um *Stop Gain K*

Input: $S_0, \mu, \sigma, r, K, \epsilon, \delta t, NSimulacoes$
Output: $\mathbb{E}[S(K)]$, ganho esperado de *Stop Gain K*.

$id \leftarrow blockDim.x * blockIdx.x + threadIdx.x$
$ES_{acumulado} \leftarrow 0$
$S \leftarrow S_0$
$t \leftarrow 0$
while $S < K$ AND $e^{-rt\cdot\delta t} \geq \epsilon$ **do**
  $z \sim N(0,1)$
  $S \leftarrow Se^{(\mu-\sigma^2/2)\delta t+\sigma\sqrt{\delta t}z}$
  $t \leftarrow t+1$
end while
if S > K **then**
  $S \leftarrow K$
end if
$ES[id] = Se^{-rt\cdot\delta t}$

Após execução do *Kernel*, faça em CPU
$\mathbb{E}[S(K)] = \frac{\sum_{NSimulacoes-1}^{i=0} ES[i]}{NSimulacoes}$

8.3.1 Considerações Numéricas da Discretização de Euler

Na seção anterior, encontramos a solução exata para o processo descrito na equação 8.16. Alternativamente, pela discretização de Euler, podemos chegar na seguinte recorrência:

$$\begin{aligned} S(t+\delta t) &= S(t) + \mu S(t)\delta t + \sigma S(t)\sqrt{\delta t}z \\ &= S(t)(1+\mu\delta t + \sigma\sqrt{\delta t}z), \quad z \sim \mathcal{N}(0,1). \end{aligned}$$

Contudo, é importante notar que o número pseudo-aleatório z pode ser gerado de uma distribuição Gaussiana em uma ou mais simulações de tal forma que

$$(1+\mu\delta t + \sigma\sqrt{\delta t}z) < 0$$

ou seja,

$$z < -\frac{1+\mu t}{\sigma\sqrt{\delta t}}$$

e, assim, $S(t+\delta t)$ pode assumir valores negativos. Portanto, apesar da garantia de convergência para um processo log-normal, não há garantia que esse resultado seja obtido em todos os passos da simulação.

8.4 Cálculo do *Stop* Ótimo

Como vimos, de posse do Algoritmo 6, conseguimos obter um valor esperado de ganho ($\mathbb{E}[S(K)]$) dado um valor de *Stop Gain* (K). Assim, podemos retomar nosso problema original de obtenção de *Stop* ótimo:

$$\max \mathbb{E}^{x_0}[g(x_\tau)]$$
$$\text{sujeito a } \tau \text{ tempo de parada}$$

Seja $f(K) = \mathbb{E}[S(K)]$, então o valor $K = K^*$ tal que $f'(K) = 0$ é um ponto de extremo de *Stop*. Assim, a obtenção de *Stop* ótimo se reduz a um problema de obtenção de raiz de função, que pode ser resolvido, por exemplo, pelo método de otimização de bisecção áurea de Brent (W. H. Press e Flannery, 2007).

8.5 Análise Experimental e Discussão

Nesta seção realizamos uma análise experimental dos algoritmos de cálculo de probabilidade de *Stop* propostos no capítulo. Serão analisados os algoritmos propostos na seção 8.2.3 para o modelo trinominal em tempo discreto, o modelo estocástico da seção 8.3 e as implicações financeiras dos mesmos.

8.5.1 Experimentos do Modelo Trinomial

Os exemplos computacionais aqui apresentados possuem a seguinte configuração:

- Ativo objeto: Ação ITUB4[3]
- Período de amostragem: 15/06/2011 a 31/05/2012
- μ: -0.1%
- σ: 19,98%
- r_f: 11%
- $S(0,0)$: R$28,3
- Δ: 2
- T: 20

Antes de configurar um valor de *Stop Loss*, é de interesse do investidor saber qual é a probabilidade desse *Stop* acontecer. Utilizando o Algoritmo 5, a análise foi realizada variando a média e desvio padrão do ativo objeto em relação a diferentes configurações de preço de *Stop Loss*. A figura 8.5 exibe o comportamento da probabilidade de *Stop Loss* com o aumento da barreira inferior (L). Nesse caso, o desvio padrão foi mantido fixo e houve variação na média. Como podemos perceber, com o aumento da média, a curva de probabilidade tem um aumento aproximadamente linear.

Na figura 8.6, o valor da média foi mantido fixo e variamos o desvio padrão. Também podemos concluir que quanto maior o desvio padrão maior a probabilidade de *Stop Loss*. Contudo, essa relação mostrou ser mais acentuada nesse último caso, em relação ao caso de variação da média.

Nas figuras 8.7 e 8.8, podemos observar os resultados obtidos em análise correspondente ao cálculo de Probabilidade de *Stop Gain*. Ao final, podemos concluir que quanto maior a média, menor a probabilidade de *Stop Loss* e maior a probabilidade de *Stop Gain*, mantida a variância fixa. Além disso, quanto maior o desvio padrão, maiores são as probabilidades de *Stop Gain* ou de *Stop Loss*, mantida a média fixa.

[3] Ação Preferencial Itaú Unibanco Holding S.A.

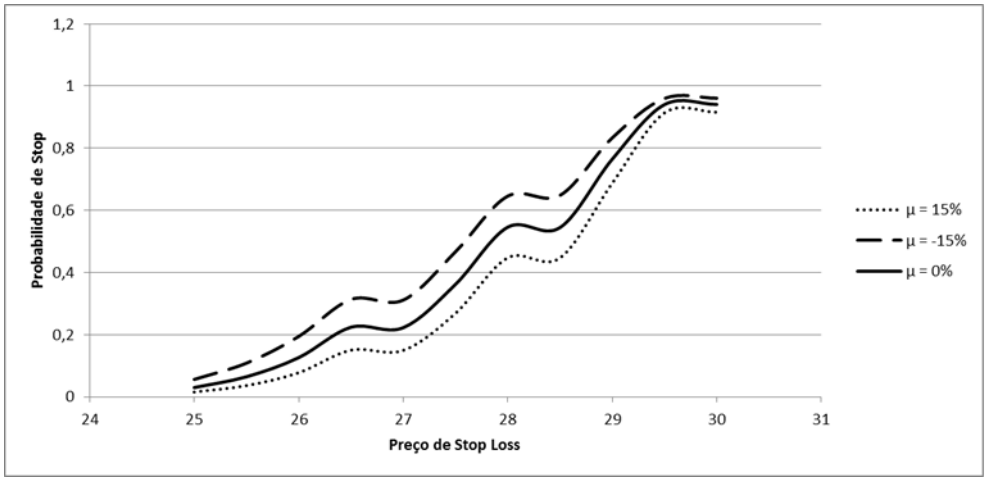

Figura 8.5: *Probabilidade de Stop em função do preço de Stop Loss configurado. Valor do desvio padrão foi mantido fixo.*

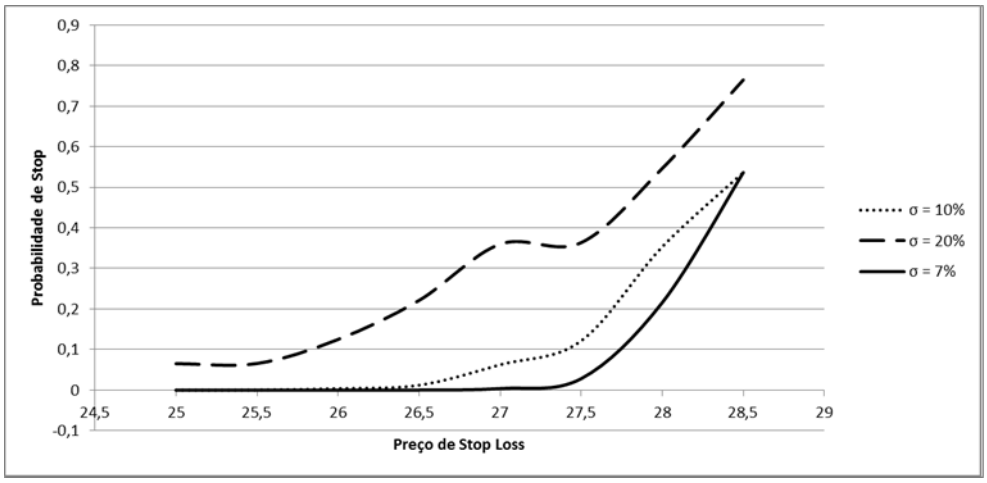

Figura 8.6: *Probabilidade de Stop em função do preço de Stop Loss configurado. Valor da média foi mantido fixo.*

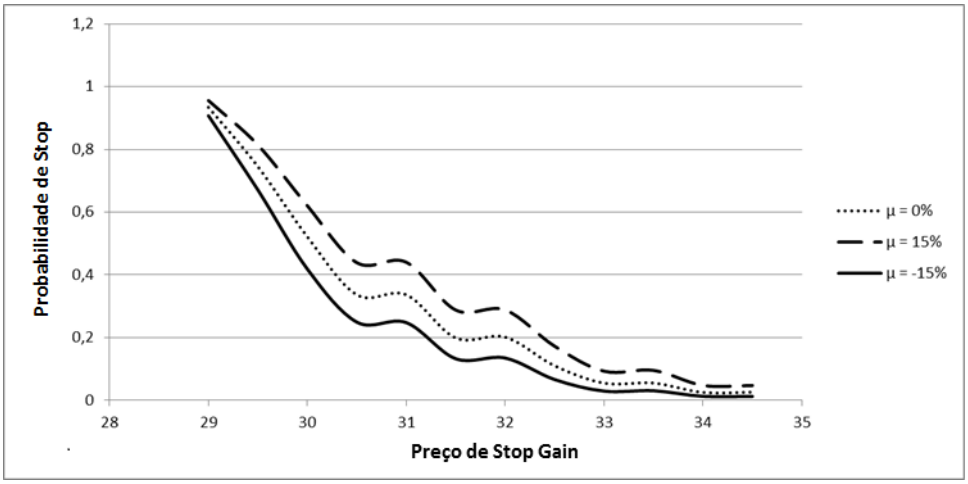

Figura 8.7: *Probabilidade de Stop em função do preço de Stop Gain configurado. Valor do desvio padrão foi mantido fixo.*

A modelagem proposta por (Warburtona e Zhang, 2006) é de difícil implementação em uma plataforma de GPU devido à recursão sugerida em sua função de probabilidade proposta. Enquanto

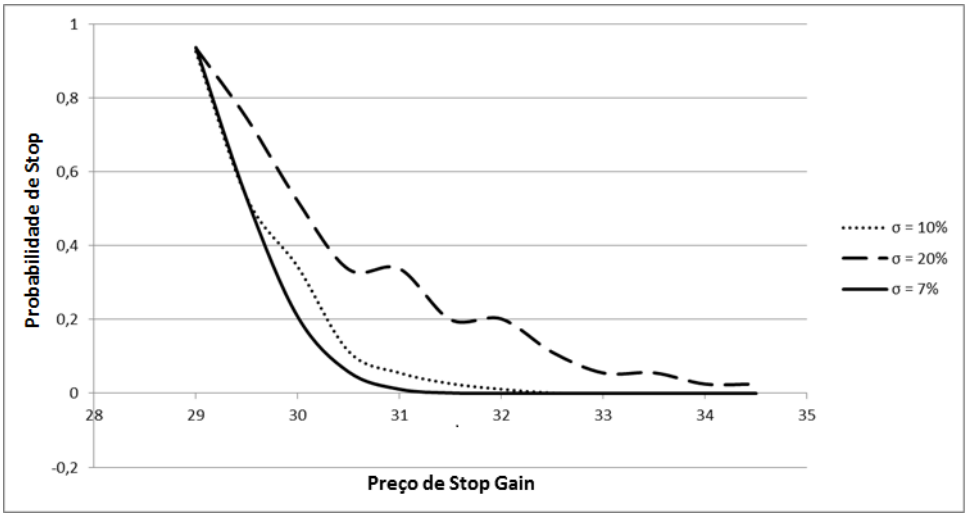

Figura 8.8: *Probabilidade de Stop em função do preço de Stop Gain configurado. Valor da média foi mantido fixo.*

isso, o algoritmo 5 proposto que resolve o problema via simulação estocástica pode ser facilmente portado em GPGPU. A figura 8.9 demonstra o tempo gasto nas implementações em CPU e GPU do algoritmo para cálculo de probabilidade de *Stop Gain* versus o valor da barreira configurado. O horizonte de tempo limite para simulação (ΔT) foi configurado em um valor de 2.000 dias. Como podemos ver, o tempo gasto na GPU mostrou ser estável ao longo de todas as simulações, enquanto que na CPU o tempo teve um acréscimo aproximadamente linear nas primeiras simulações. Isso mostra que o algoritmo paralelo é mais escalável. Para valores grandes de barreira o tempo gasto em CPU tendeu a se estabilizar. Isso é justificado pois para valores grandes de *Stop Gain* configurados, o fator limitante na simulação se torna o horizonte de tempo definido.

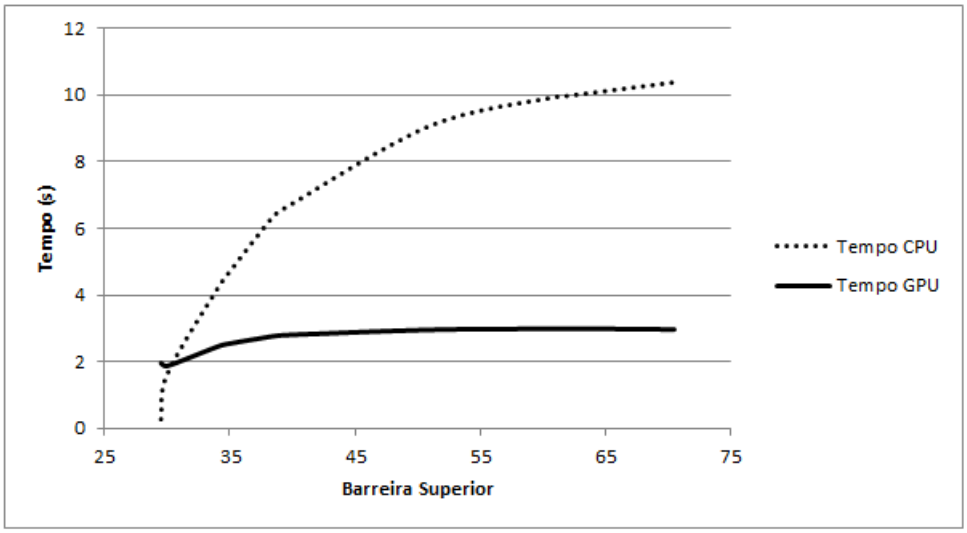

Figura 8.9: *Tempo gasto para cálculo de probabilidade de Stop Gain em função do preço de barreira configurado.*

8.5.2 Experimentos do Modelo Estocástico

Os exemplos computacionais aqui apresentados possuem a seguinte configuração:

- μ: 0,5%

- σ: 2,4432633%
- r_f: 13%
- S_0: R$33,99
- Total de dias do passeio aleatório dos preços: 2520 (10 anos)
- Tamanho do passo temporal (δt): 1 dia
- Número de Simulações: 10.000

Nesta seção, apresentamos experimentos realizados com a modelagem estocástica e algoritmo propostos na seção 8.3. Assim, serão apresentados os resultados obtidos na obtenção da esperança de ganho em uma estratégia de *Stop Gain*.

Nesse modelo, há uma premissa que o preço do ativo analisado segue um processo log-normal e, assim, seu passeio aleatório pode ser iterado conforme equação 8.18. A figura 8.10 ilustra a simulação de diferentes passeios aleatórios para esse processo.

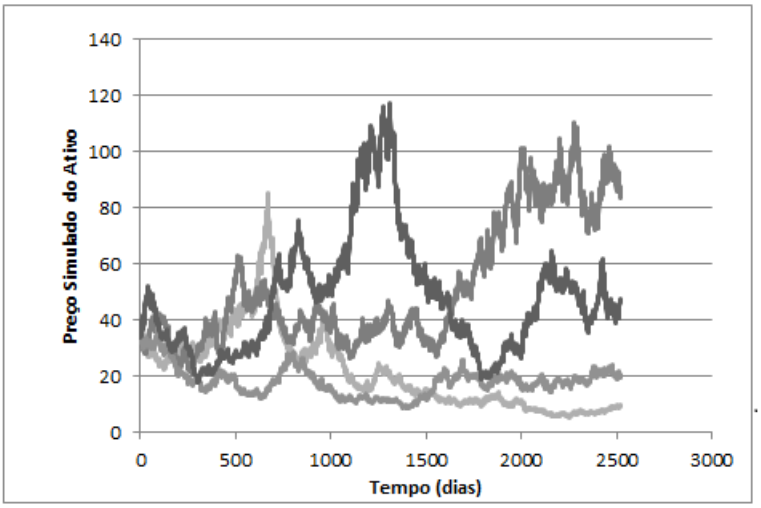

Figura 8.10: *Simulação de passeios aleatórios do processo log-normal.*

Por meio da simulação dos passeios aleatórios dos preços, é possível obter o ganho esperado de um *Stop Gain* configurado com a implementação do Algoritmo 6. As figuras 8.11 e 8.12 apresentam os resultados obtidos. Como podemos perceber, com o aumento da média (μ) temos uma esperança maior para o valor da barreira configurado. Em contraposição, um valor de volatilidade maior sugere a necessidade de configuração de um valor mais alto para valor de barreira.

De posse da esperança do retorno de um *Stop Gain*, é possível calcular o valor de retorno ótimo conforme discutido na seção 8.4. A figura 8.13 exibe os valores de retornos ótimos obtidos para diferentes valores de taxa de juros configurados utilizando o método de biseccção áurea de Brent. Podemos observar uma estimativa de ganho ótimo de 9,91% sobre uma taxa de juros de 11%, na configuração analisada. Além disso, é possível concluir que existe um valor crítico de taxa de juros a partir da qual não faz mais sentido a utilização de uma estratégia de *Stop Gain*, pois o retorno ótimo é negativo. No exemplo considerado, esse valor foi de aproximadamente 13%.

Com implementação do Algoritmo 6, também foi analisado o tempo computacional gasto na obtenção da esperança do ganho de um *Stop Gain* em GPU e CPU. Na figura 8.14. podemos ver o tempo gasto nesse cálculo em função do número de simulações realizadas. O tempo gasto em CPU apresentou um crescimento linear em relação ao número de simulações, contudo foi muito mais

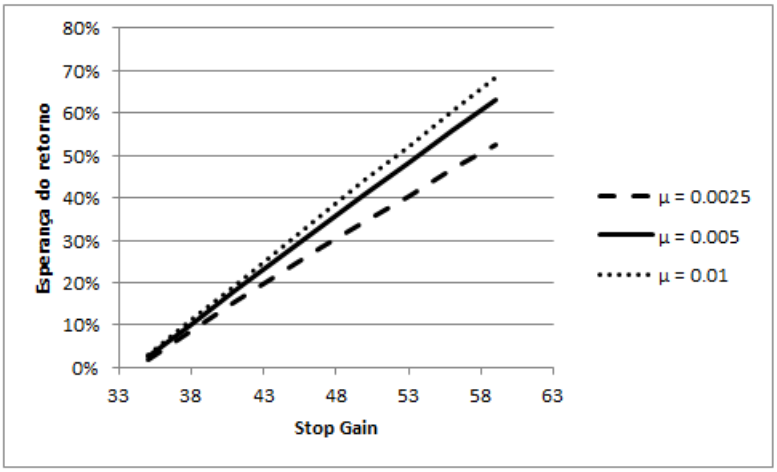

Figura 8.11: *Esperança de retorno de Stop Gain. Análise da variação da média* μ.

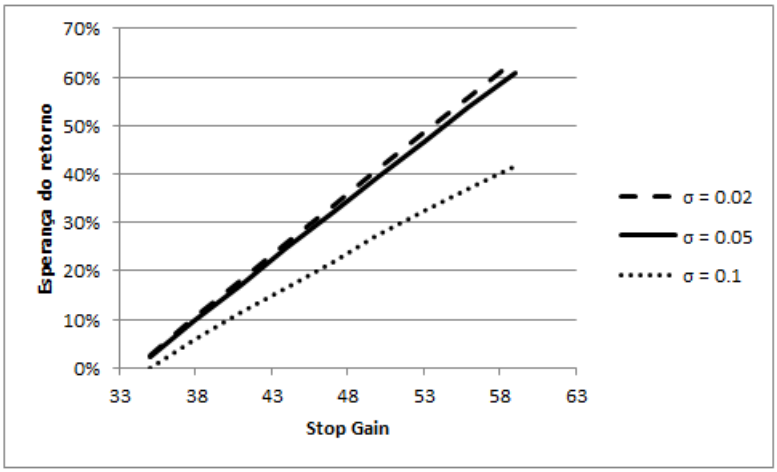

Figura 8.12: *Esperança de retorno de Stop Gain. Análise da variação da volatilidade* σ.

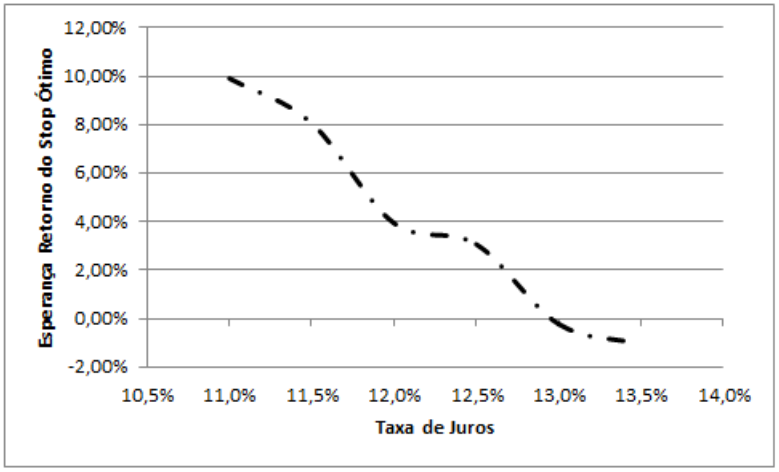

Figura 8.13: *Esperança de retorno ótimo em função da taxa de juros. Configuração* $\mu = 0,0521139\%$ *e* $\sigma = 2,4432633\%$.

acentuado do que o crescimento no tempo gasto em GPU. Para um número de simulações igual a 51.200 foi obtido um *speedup* de mais de 12 vezes.

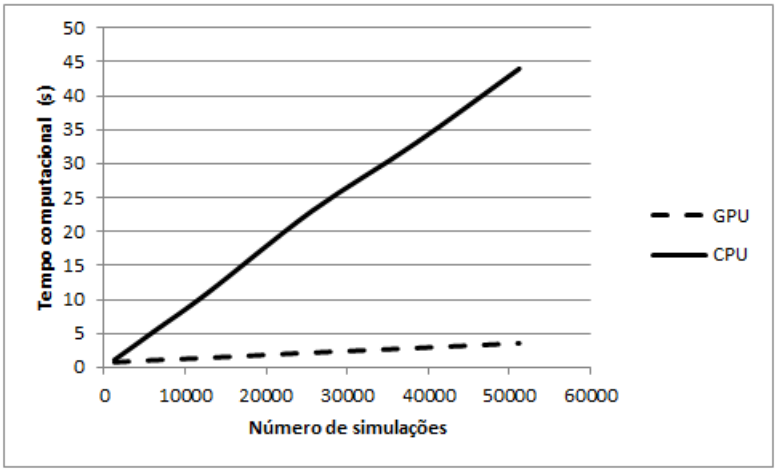

Figura 8.14: *Tempo computacional gasto em cálculo da esperança de ganho de um Stop Gain em função do número de simulações. Configuração: Stop = 65,* $\mu = -0,00055\%$, $\sigma = 2\%$.

A figura 8.15 apresenta o resultado obtido no tempo computacional variando o valor de *Stop Gain* configurado para um número fixo de simulações realizadas. Mais uma vez, o tempo computacional em GPU mostrou seu muito mais estável do que aquele medido em CPU. Para valores altos de barreira superior o tempo de cálculo gasto em CPU tendeu a se estabilizar. Isso é justificado pois para valores grandes de *Stop Gain* configurado, o fator limitante na simulação se torna o horizonte de tempo definido.

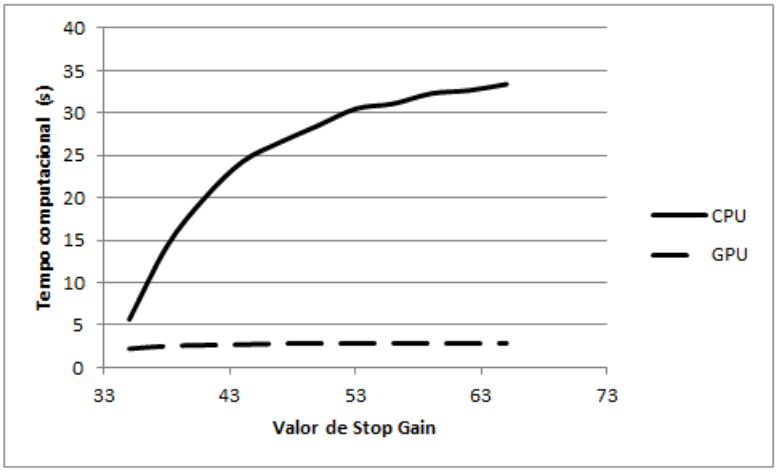

Figura 8.15: *Tempo computacional gasto em cálculo da esperança de ganho de um Stop Gain. Configuração: número de simulações = 38.400,* $\mu = -0,00055\%$, $\sigma = 2\%$.

8.6 Conclusão

Neste capítulo realizamos o estudo de caso do problema de *Stops* Ótimos em finanças. Após definição formal do problema, revisamos o modelo trinomial proposto por (Warburtona e Zhang, 2006) e propusemos um algoritmo alternativo para o mesmo via simulação utilizando a técnica de aceitação e rejeição para probabilidade de transição dos preços. O algoritmo sugerido é, comparativamente, de fácil implementação e naturalmente paralelizável em GPU.

Em seguida, apresentamos uma modelagem estocástica para o problema e propomos um algoritmo para cálculo da esperança de ganho de um *Stop Gain*. De posse desse algoritmo, resolvemos o problema de otimização de *Stop* Ótimo.

Os algoritmos propostos foram implementados tanto em CPU quanto em GPU. Após análise financeira dos resultados, comparamos o tempo computacional gasto nas duas plataformas. Comparado com CPU, as implementações em GPU mostraram um gasto computacional muito mais escalável ao longo dos experimentos. Para o algoritmo de obtenção de esperança de retorno de um *Stop Gain*, foi alcançado um *speedup* de mais de 12 vezes.

8.7 Notas e Leituras Complementares

As origens do estudo da teoria de *Stops* remontam ao trabalho pioneiro de (Wald, 1945) sobre a natureza de experimentos estatísticos. Para uma introdução no assunto, veja (Karlin e Taylor, 1975), (Chow *et al.*, 1971), (Oksendal, 1992). Uma abordagem em processos de Markov pode ser obtida em (Shiryaev e Aries, 2008). Um trabalho recente e aprofundado pode ser visto em (Gut, 2009).

Capítulo 9

Risco de Mercado

Risco de Mercado é aquele relacionado com perdas no fluxo de caixa da organização causadas por flutuações nos preços de ativos e passivos da empresa (Kimura *et al.*, 2010). Para gestão desse risco, uma métrica particularmente difundida é o *Value-at-Risk* (VaR), que visa a obter uma medida de perda máxima da instituição. Para o cálculo do mesmo, o gestor de risco deve ser capaz de calcular o valor de mercado de seus produtos, para assim conseguir avaliar as possíveis variações na carteira da empresa.

Com isso, neste capítulo abordaremos o cálculo de risco de mercado pela métrica de VaR. Com esse fim, estudaremos a precificação de um produto comum em negociações financeiras: Opções.

9.1 Precificação de Opções

Uma Opção é um produto financeiro que corresponde a um acordo entre duas partes, o comprador da opção e o vendedor da opção[1]. Nesse contrato, o primeiro tem o direito (mas não a obrigação) de exercer a opção acordada em um momento futuro, enquanto que o vendedor tem a obrigação de realizar a operação em caso de exercício pelo comprador. O valor pago pelo comprador é comumente chamado de prêmio da opção e corresponde ao seu preço de mercado. Há dois tipos básicos de opções. A opção de compra (*call*) dá ao seu detentor o direito de comprar um ativo a certo preço em uma determinada data. A opção de venda (*put*) dá ao seu detentor o direito de vender um ativo em uma certa data por determinado preço. A data especificada no contrato é conhecida como data de vencimento[2]. O preço especificado no contrato é denominado preço de exercício (*strike*). As opções também podem ser classificadas conforme a data de exercício. Opções Europeias são aquelas que podem ser exercidas somente na data de vencimento. Já as Opções Americanas são mais flexíveis. Elas permitem o exercício em qualquer data até o vencimento.

A seguir, fornecemos um exemplo simples para entendimento da nomenclatura até aqui exposta.

Exemplo 1. ***Opção de Compra de Dólar***
Suponha que uma Opção Europeia com vencimento em 24/12/2013 de Compra (Call) de Dólar à R$2,00 em 24/12/2013 esteja sendo vendida a R$0,5 hoje. Assim, um comprador deve pagar um prêmio de R$0,5 para ter o direito de comprar a quantidade especificada de Dólar à R$2,00 na data futura de 24/12/2013. Como se trata de uma opção europeia, o comprador pode ser exercer esse direito somente no vencimento, dia 24/12/2013.

Caso em 24/12/2013 o preço à vista (Spot) do dólar seja maior do que R$2,00, o comprador pode lucrar ao exercer a opção de compra e vendê-la em seguida[3]. Caso em 24/12/2013 o preço à vista (Spot) do dólar seja menor do que R$2,00, o exercício da opção não trará lucro ao comprador.

Para ilustração, exibimos na figura 9.1 um trecho de um contrato de Opção de Compra de Taxa de Câmbio de Reais por Dólar Comercial disponível na BM&FBovespa.

[1] também chamado de lançador da opção
[2] também conhecida como: data de expiração, data de exercício ou data de maturidade.
[3] operação chamada de *day-trade*

Especificações do Contrato de Opção de Compra sobre Taxa de Câmbio de Reais por Dólar Comercial

1. **Definições**
   Contrato (especificações): termos e regras sob os quais as operações serão realizadas e liquidadas.
   Contrato negociado: um lote (unidade de negociação) negociado sob os termos e as regras destas especificações.
   Série: conjunto de características do contrato de opção, que determinam a data de vencimento e seu preço de exercício, sendo identificada por código específico estabelecido pela BM&F.
   Compra de um contrato: operação na qual o participante é titular, ou seja, tem o direito de comprar o objeto de negociação pelo preço de exercício.
   Venda de um contrato: operação na qual o participante é lançador, ou seja, se exercido pelo titular tem a obrigação de vender o objeto de negociação pelo preço de exercício.
2. **Objeto da opção**
   A taxa de câmbio de reais por dólar dos Estados Unidos da América, para entrega pronta, contratada nos termos da Resolução 1690/1990, do Conselho Monetário Nacional (CMN).
3. **Cotação**
   Prêmio da opção, em reais por US$1.000,00, com até três casas decimais.
4. **Variação mínima de apregoação**
   R$0,001 por US$1.000,00.
5. **Oscilação máxima diária**
   Não há limites de oscilação diária, podendo a Bolsa, excepcionalmente e a seu critério, estabelecê-los.
6. **Unidade de negociação**
   Cada opção refere-se a um lote padrão de US$50.000,00.
7. **Preços de exercício**
   Os preços de exercício serão estabelecidos e divulgados pela BM&F, expressos em reais por US$1.000,00.

Figura 9.1: *Contrato de Opção de Compra de Taxa de Câmbio de Reais por Dólar Comercial fornecido pela BM&FBovespa*

Assim, o problema de precificação de uma Opção se resume a encontrar seu preço justo (prêmio) a partir das variáveis de seu contrato de negociação como: Tipo da Opção, Data de Vencimento, *Strike*, *Spot*, taxa de juros livre de risco, volatilidade do ativo objeto etc. Nas próximas seções apresentamos os passos para precificação de uma opção via o método tradicional de Black & Scholes em GPU, bem como, uma alternativa de precificação via simulação. Ao final, analisamos a possibilidade de melhoria na qualidade da simulação com utilização de técnicas de redução de variância.

9.1.1 O Método Tradicional de Black-Scholes

O método mais tradicional para precificação de opções é dado pelo modelo de Black & Scholes (Black e Scholes, 1973). A partir da premissa de que os ativos seguem um movimento Browniano Geométrico, Black & Scholes chegaram a uma expressão analítica de fácil utilização para obter o prêmio V de uma opção Europeia dada pelas equações:

$$\begin{aligned} V &= S \times CND(d_1) - Xe^{-rT} \times CND(d_2) \\ d_1 &= \frac{\log\left(\frac{S}{X}\right) + T\left(r + \frac{v^2}{2}\right)}{v\sqrt{T}} \\ d_2 &= d_1 - v\sqrt{T} \end{aligned} \tag{9.1}$$

onde, S é o valor *Spot* do ativo objeto, X o *Strike*, T o prazo para o vencimento, r a taxa de juros livre de risco, v representa a volatilidade do ativo objeto e CND é a distribuição normal padrão acumulada.

Para precificação de uma opção via equação 9.1, primeiramente necessitamos de um método numérico para cálculo da distribuição normal padrão acumulada:

$$CND(x) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{x} e^{\frac{-u^2}{2}} du. \tag{9.2}$$

Uma possibilidade é a aproximação por um polinômio de quinta ordem conforme sugere (Hull,

2012). O código 9.1 implementa esse método em GPU.

```
1 __host__ __device__ float CND(float d)
2 {
3   float K = 1.0f / (1.0f + 0.2316419f * fabsf(d));
4   float CND = rsqrt(2PI) * expf(- 0.5f * d * d) *
5   (K * (A1 + K * (A2 + K * (A3 + K * (A4 + K*A5)))));
6   if(d > 0)
7     return 1.0f - CND;
8   else
9     return CND:
10 }
```

Código 9.1: *Aproximação da distribuição normal acumulada padrão*

Na Código 9.2, fornecemos os passos gerais para precificação paralela de um conjunto de N opções em GPU. Esse passos podem ser sumarizados em:

1. Alocar vetores do *host* : hOptSpot(N), hOptStrike(N). Linhas 2 até 5.
2. Alocar vetores no *device*: dOptSpot(N), dOptStrike(N). Linhas 8 até 12.
3. Inicializar vetores com variáveis dos contratos e mercado. Linha 14.
4. Transferir vetores da memória *host* para memória da GPU. Linhas 17 até 19.
5. Precificar opção em GPU via Black&Scholes (Equação 9.1). Linha 22.
6. Transferir resultado da GPU para *host*. Linha 25.
7. Desalocar memória. Linhas 28 e 32.

```
1   /* Aloca vetores no Host */
2   float *hOptSpot, *hOptStrike, *hOptT;
3   hOptSpot = (float *) malloc(sizeof(float)*N);
4   hOptStrike = (float *) malloc(sizeof(float)*N);
5   hOptT = (float *)malloc(sizeof(float)*N);
6
7   /* Aloca vetores em GPU */
8   float *dOptSpot, *dOptStrike, *dOptT;
9   cudaMalloc( (void **) &dOptSpot, sizeof(float)*N);
10  cudaMalloc( (coid **) &dOptStrike, sizeof(float)*N);
11  cudaMalloc( (coid **) &dOptT, sizeof(float)*N);
12  // alocacao de possiveis outras variaves de contrato para o modelo
13
14  /* Inicialize hOptSpot, hOptStrike, hOptT ho Host conforme leitura de dados de
        entrada */
15
16  /* Transfere dados do host para device com cudaMemcpy */
17  cudaMemcpy (dOptSpot, hOptSpot, sizeof(float)*N, cudaMemcpyHostToDevice);
18  cudaMemcpy (dOptStrike, hOptStrike, sizeof(float)*N, cudaMemcpyHostToDevice);
19  cudaMemcpy (dOptT, hOptT, sizeof(float)*N, cudaMemcpyHostToDevice);
20
21  /* Precificacao Opcao na GPU via Black|&Scholes */
22  BlackScholes<<<128, 256>>>(dPremio, dOptStrike, dOptSpot, dOptT, r, v, N);
23
24  /* Transfere dados do device para host com cudaMemcpy */
25  cudaMemcpy (hPremio, dPremio, sizeof(float)*N, cudaMemcpyDeviceToHost);
26
27  /* Libera memoria do host e de device */
28  free(hOptSpot);
29  free(hOptStrike);
30  free(hOptT);
```

```
31
32    cudaFree(dOptSpot);
33    cudaFree(dOptStrike);
34    cudaFree(dOptT);
35 }
```

Código 9.2: *Passos gerais para precificação de N opções em GPU via Método de Black&Scholes*

9.1.2 Precificação via Simulação

Como vimos, o método de Black & Scholes fornece uma expressão de fácil uso para precificação de uma opção. Contudo, muitas vezes, não é possível obter uma expressão analítica para precificar esse produto financeiro. Assim, nesta seção, ilustramos uma forma de precificar uma opção via Simulação. Esse método pode ser utilizado independente da distribuição de probabilidade seguida pelo ativo objeto. O método aqui descrito supõe uma opção europeia, para precificação de outros tipos de opções via simulação de Monte Carlo veja (Jasra e Del Moral, 2011).

Antes de descrevermos o método de precificação via simulação, vamos analisar melhor a função de ganho (*payoff*) em uma negociação de contrato de opção europeia. Como sabemos, há quatro possibilidades de posições em opções:

1. Posição comprada em uma opção de *Call*
2. Posição comprada em uma opção de *Put*
3. Posição vendida em uma opção de *Call*
4. Posição vendida em uma opção de *Put*

É possível caracterizar essas posições em opções europeias em termos de *payoff* do investidor na data de vencimento. Seja X o *Strike* e S_T, o preço final do ativo objeto, o resultado de uma posição comprada em uma opção de *Call* será:

$$\max(S_T - X, 0). \tag{9.3}$$

Já que a opção será exercida somente se $S_T > X$. Analogamente, temos que o *payoff* de uma posição vendida de uma opção de *Call* será:

$$-\max(S_T - X, 0). \tag{9.4}$$

O *payoff* de uma posição comprada de uma opção de Put é:

$$\max(X - S_T, 0). \tag{9.5}$$

e o *payoff* de uma posição vendida de uma opção de Put é:

$$-\max(X - S_T, 0). \tag{9.6}$$

A figura 9.2 ilustra essas quatro possibilidades de *payoff*.

Assim, o prêmio (c_T) de uma Opção Europeia de *Call* pode ser calculado a partir de um valor esperado de seu retorno, conforme:

$$c_T = e^{-rT}\mathbb{E}[max(S_T - X, 0)]. \tag{9.7}$$

Ao simular aleatoriamente o preço do instrumento S_T, podemos precificar a opção via simulação

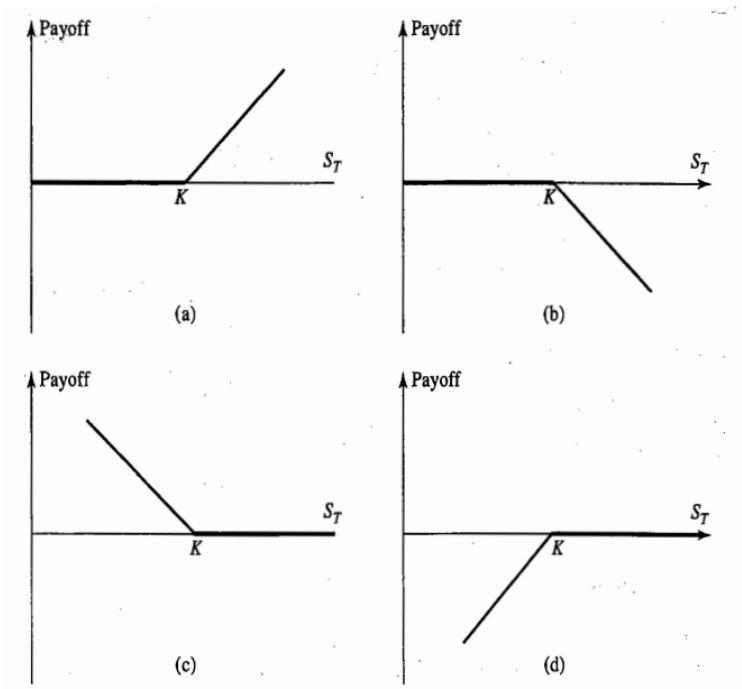

Figura 9.2: *Payoffs das posições em opções europeias: (a) compra de Call; (b) venda de Call; (c) compra de Put; (d) venda de Put. Retirado de (Hull, 2012).*

de monte carlo como

$$\hat{c}_T = e^{-rT} \sum_{i=1}^{n} max(S_T - X, 0). \tag{9.8}$$

A precificação de diferentes tipos de opções foi extensivamente estudada com aceleração em GPU (Solomon *et al.*, 2010), (Pages e Wilbertz, 2010), (Pages e Wilbertz, 2012). Assim, de modo complementar, a seguir, iremos propor técnicas que visam à melhoria da qualidade do resultado obtido.

Redução da Variância

As equações vistas na seção anterior para precificação de opções *call* e *put* possuem uma aparente simetria. De fato, é possível provar que o valor de uma opção de *call* (c) pode ser obtida diretamente pelo valor da opção correspondente de *put* (p) (Hull, 2012):

$$c + Xe^{-rT} = p + S_0. \tag{9.9}$$

Essa relação é conhecida como paridade *put-call*. Ela mostra que o prêmio c de uma opção europeia de *call* com *strike* X e data de vencimento T pode ser deduzido do valor da opção europeia de *put* com o mesmo preço de exercício e mesma data de exercício.

De posse da equação 9.9, podemos estimar a esperança do prêmio de uma opção de *call* $\mathbb{E}[c]$ como:

$$\mathbb{E}[c] = \mathbb{E}[p] + S - Xe^{-rT} \tag{9.10}$$

Em geral, a variância de uma opção de *put* é menor do que uma opção de *call* já que no primeiro caso o *payoff* é limitado, como podemos ver na figura 9.2. Dessa forma, o preço da opção de *put* se torna uma variável de controle para o cálculo do prêmio da opção de *call* e, assim, obtemos um estimador de c com variância potencialmente reduzida.

A seguir, propomos o Algoritmo 7 para cálculo do valor esperado do prêmio de uma opção

europeia de *call* baseado no procedimento de simulação sugerido na seção 9.1.2 e na técnica de variável de controle proposta nesta seção. Além disso, também foi aplicada a técnica de variáveis antitéticas para melhoria da qualidade da simulação. Como podemos ver, nas linhas 6 e 7 as variáveis antitéticas são geradas, na linha 17 o valor esperado da opção europeia de *put* é calculado e, finalmente, o valor da opção de *call* é calculado na linha 18 pela relação de paridade *put-call*.

Algoritmo 7 Cálculo do valor esperado do prêmio de uma opção europeia de *call* via simulação utilizando técnicas de redução de variância.

Input: $S, X, r, T, \delta t, NSimulacoes$
Output: c, Prêmio de uma opção de *call* europeia

1: $E[i] \leftarrow 0,\ \forall 1 \leq i \leq NSimulacoes$
2: **for** $n = 1 \rightarrow NSimulacoes$ **do**
3: $\quad S_T \leftarrow SA_T \leftarrow S$
4: $\quad$ **for** $i = 1 \rightarrow T$ **do**
5: $\quad\quad Z \sim \mathbb{N}(0,1)$
6: $\quad\quad S_T \leftarrow S_T e^{(\mu-\sigma^2/2)\delta t+\sigma\sqrt{\delta t}Z}$
7: $\quad\quad SA_T \leftarrow SA_T e^{(\mu-\sigma^2/2)\delta t+\sigma\sqrt{\delta t}(-Z)}$
8: $\quad$ **end for**
9: $\quad$ **if** $X - S > 0$ **then**
10: $\quad\quad E[n] \leftarrow E[n] + X - S_T$
11: $\quad$ **end if**
12: $\quad$ **if** $X - SA_T > 0$ **then**
13: $\quad\quad E[n + NSimulacoes] \leftarrow E[n + NSimulacoes] + X - SA_T$
14: $\quad$ **end if**
15: **end for**
16:
17: $p \leftarrow \frac{\sum_{i=1}^{2NSimulacoes} E[i]}{2NSimulacoes}$
18: $c \leftarrow p + S - Xe^{rT}$
19:
20: **return** c

9.2 *Value-at-Risk*

(Choudhry e Tanna, 2007) define o $VaR_{1-\alpha}$ como a perda máxima que pode ocorrer em um grau de confiança $(1-\alpha)$ e período de tempo (T) determinados. Um dos primeiros bancos a utilizar o conceito de VaR foi o J. P. Morgan. Em um de seus relatórios, o banco anunciou para carteira de sua tesouraria, um VaR de US$ 15 milhões, para o horizonte de tempo de 1 dia e grau de confiança de 95% (Kimura *et al.*, 2010). Assim, podemos entender que o Banco forneceu uma previsão de que sua carteira não sofreria uma perda diária maior do que US$ 15 milhões com uma probabilidade de 95%. Uma grande vantagem do VaR é sua capacidade de resumir o risco de uma instituição financeira devido a múltiplas variáveis do mercado em uma única medida de fácil interpretação.

Se assumirmos que F é a função de distribuição acumulada do retorno da carteira da instituição, podemos definir o VaR como

$$F(VaR) = \alpha \tag{9.11}$$

onde é α a probabilidade correspondente ao nível de confiança estabelecido. Ao utilizar a função de densidade dos retornos f, podemos definir o VaR de modo equivalente como

$$\alpha = \int_{-\infty}^{VaR} f(x)dx. \tag{9.12}$$

Assim, a questão principal no problema de análise de VaR está na obtenção da função f de distribuição dos retornos para um horizonte de tempo definido. Para isso, existem três principais metodologias: Modelo Paramétrico, Simulação Histórica e Simulação de Monte Carlo.

Também chamado de Modelo de Variâncias-Covariâncias ou Delta-Normal, o modelo Paramétrico tem como hipótese a normalidade dos retornos. Dessa forma, a partir da média e do desvio-padrão da variável aleatória associada ao retorno, pode-se construir facilmente os intervalos de confiança desejados.

O método de Simulação Histórica não necessita de estimação de parâmetros que reflitam uma distribuição de probabilidades, por isso, pode ser considerado um método não-paramétrico de estimação de VaR. Nesse tipo de simulação, a distribuição histórica dos retornos dos ativos da carteira é utilizada para simular os retornos futuros e, assim, o VaR desejado.

A ideia essencial do método via simulação de Monte Carlo é simular várias vezes um processo aleatório para as variáveis financeiras de interesse. Para isso, assume-se que as distribuições de probabilidade destas variáveis sejam conhecidas. Assim, variações hipotéticas de preços para os fatores de risco são criadas a partir de ocorrências aleatórias de um processo estocástico pré-determinado. De posse dessa amostragem, o cálculo do VaR é análogo ao método de simulação histórica. A figura 9.3 apresenta um resumo comparativo dos métodos enunciados.

Itens	Delta-normal	Simulação Histórica	Simulação Monte Carlo
Posições			
Avaliação	Linear	Plena	Plena
Distribuição			
Formato	Normal	Corrente	Geral
Varia no tempo	Sim	Possível	Sim
Dados implícitos	Possível	Não	Possível
Eventos extremos	Probabilidade baixa	Nos dados recentes	Possível
Utiliza correlações	Sim	Sim	Sim
Precisão do VaR	Excelente	Pobre quando a janela é pequena	Boa com muitas iterações
Implementação			
Facilidade de cálculo	Sim	Intermediária	Não
Precisão	Depende da carteira	Sim	Sim
Divulgação	Fácil	Fácil	Complicada
Análise do VaR	Fácil, analítica	Mais difícil	Mais difícil
Desvantagens	Não-linearidades e caudas pesadas	Variação do risco ao longo do tempo; Eventos extraordinários	Risco de Modelo

Figura 9.3: *Resumo comparativo de metodologias de cálculo de VaR. Retirado de (Jorion, 2003)*

.

A simulação de Monte Carlo pode amenizar alguns problemas da simulação histórica, como a não consideração da variação do risco no tempo e a janela temporal restrita utilizada. Além disso, trata-se de método de maior flexibilidade por possibilitar a modelagem do comportamento dos preços por qualquer distribuição definida, ao contrário do método paramétrico, que supõe comportamento normal dos retornos. Contudo, o preço a pagar pela maior flexibilidade do método de Monte Carlo pode ser alto computacionalmente.

(Dixon *et al.*, 2009) realizou experimentos de cálculo de VaR utilizando o método Delta-Normal em GPU e obteve *speedup* máximo de aproximadamente 8 vezes em uma implementação tradicional desse método. Alternativamente, nas próximas seções iremos explorar o método de Monte Carlo para cálculo de VaR e avaliar, experimentalmente, sua viabilidade em GPU.

9.2.1 Cálculo do VaR via Simulação de Monte Carlo

Seja $\Pi_0 = (S_0^1, S_0^2, \ldots, S_0^n)$ o valor de um portfólio composto por n ativos em t_0. Desejamos calcular o $VaR_{1-\alpha}$ de Π_0 para um período T. A seguir, apresentamos os passos para resolução desse problema via Simulação de Monte Carlo:

1. Escolha e configure um modelo estocástico para projeção dos preços dos ativos do portfólio
2. Simule os preços dos ativos para projetar seus valores em T: $S_T^1, S_T^2, \ldots, S_T^n$
3. Calcule o valor do portfólio $\Pi_T = (S_T^1, S_T^2, \ldots, S_T^n)$
4. Repita os passos 2 e 3 um número (N) de vezes tão grande quanto se queira
5. Ordene de modo crescente os valores dos portfólios obtidos e obtenha a distribuição: $(\Pi_T^1, \Pi_T^2, \ldots, \Pi_T^N)$
6. Compute o VaR a partir do quantil $(1-\alpha)$ de interesse: $VaR_{1-\alpha} = \Pi_T^{\lfloor (1-\alpha)N \rfloor} - \Pi_0$

O cálculo do valor de um portfólio pode ser de alto custo computacional (e.g. portfólio de opções). É fácil notar que as estimativas dos valores dos N portfólios são independentes entre si e, portanto, podem ser facilmente calculadas de modo paralelo. Essas características sugerem que o cálculo de VaR seja suscetível em GPU. Dessa forma, no Algoritmo 8, propomos uma implementação desse método em CUDA.

Algoritmo 8 *Kernel*: Cálculo do VaR pelo Método de Simulação de Monte Carlo

Input: $\Pi_0 : \{S_0^1, S_0^2, \ldots, S_0^n\}$, δt, α, n
Output: $VaR_{1-\alpha}$ do portfólio Π_0 em δt

$id \leftarrow blockDim.x * blockIdx.x + threadIdx.x$
for $i = 1 \rightarrow n$ **do**
  $x \sim N(0,1)$
  $S_{\delta t}^i \leftarrow S_0^i e^{(\mu - \sigma^2/2)\delta t + \sigma\sqrt{\delta t}x}$
end for
$\Pi_{\delta t}[id] \leftarrow (S_{\delta t}^1, S_{\delta t}^2, \ldots, S_{\delta t}^n)$

Após execução do *Kernel*, faça em CPU
Ordene de modo crescente $\Pi_{\delta t}$
$VaR_{1-\alpha} = \Pi_{\delta t}^{\lfloor (1-\alpha)N \rfloor} - \Pi_0$

9.2.2 Notas sobre Cálculo do VaR para Múltiplos Portfólios

Considere n portfólios $(\Pi^1, \Pi^2, \ldots, \Pi^n)$ com respectivos valores de VaR dados pelo vetor $x = (VaR^1, VaR^2, \ldots, VaR^n)$. Então, o VaR_Π do portfólio composto $\Pi = (\Pi^1, \Pi^2, \ldots, \Pi^n)$ é dado por

$$VaR = \sqrt{xRx'} \tag{9.13}$$

onde, R é uma matriz com as correlações dos portfólios:

$$R = \begin{pmatrix} 1 & p_{12} & \cdots & p_{1n} \\ p_{21} & 1 & \cdots & p_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ p_{n1} & p_{n2} & \cdots & 1 \end{pmatrix} \tag{9.14}$$

Em geral, a matriz R pode ter grande dimensão. Assim, é comum a utilização de técnica de Análise de Componentes Principais para redução de dimensionalidade. Para uma implementação desse método em GPU veja (Jung e O'Leary, 2006).

É importante notar que cada portfólio pode ser precificado de uma maneira diferente. Por exemplo, Π_0 pode representar um portfólio de ações, Π_1 uma carteira de opções etc. Dessa forma, a execução de um único *kernel* em GPU para cálculo do VaR para múltiplos portfólios não é uma boa estratégia, na prática. Assim, de modo geral, é necessário construir um *kernel* para cada precificação realizada. Em placas de *compute capability* 2.x é possível o lançamento de *kernels* paralelos. Com isso, uma possível solução para esse problema seria a precificação de portfólios distintos em *kernels* de diferentes *streams*, conforme discutido na seção 4.2.2. Contudo, essa estratégia é limitada a um total de 16 *kernels* em paralelo em uma arquitetura Fermi, conforme visto no Apêndice A.1.

9.3 Análise Experimental e Discussão

9.3.1 Experimentos da Precificação de Opções

Para o cálculo das opções, foram utilizados as seguintes configurações:

- σ: 25%
- r_f: 10%
- Dias úteis até o vencimento: 126
- $S = K = 25$

Visando à análise da qualidade da simulação proposta no Algoritmo 7 para cálculo do prêmio de uma opção de *call* europeia, o mesmo foi implementado em CPU com diferentes configurações.

Em primeira análise, o algoritmo foi testado sem aplicação de técnicas de redução de variância. Nessa configuração, a figura 9.4 apresenta o erro padrão obtido em simulações realizadas com três RNGs distintos: Mersenne Twister, Sobol e *rand()*, função padrão da biblioteca de C. Como podemos notar, de modo geral, o gerador Sobol apresentou uma qualidade maior. Esse resultado vai ao encontro do trabalho de (Giles *et al.*, 2008), que sugere a utilização de QRNGs para simulações de Monte Carlo em aplicações financeiras.

Tendo em vista as técnicas de redução de variância sugeridas no capítulo, a figura 9.5 apresenta o erro padrão obtido utilizando as técnicas de variáveis antitéticas e variável de controle em Sobol. Podemos ver uma melhora na qualidade da simulação em ambos os casos, havendo um destaque para a última técnica. Ao utilizar as duas técnicas combinadas, o erro padrão da simulação apresentou uma melhora de mais de 4 vezes.

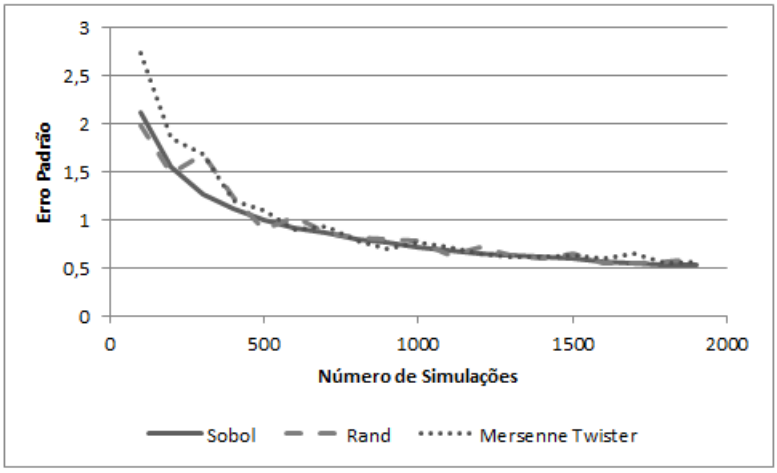

Figura 9.4: *Simulação do prêmio de uma opção de call europeia em diferentes RNGs.*

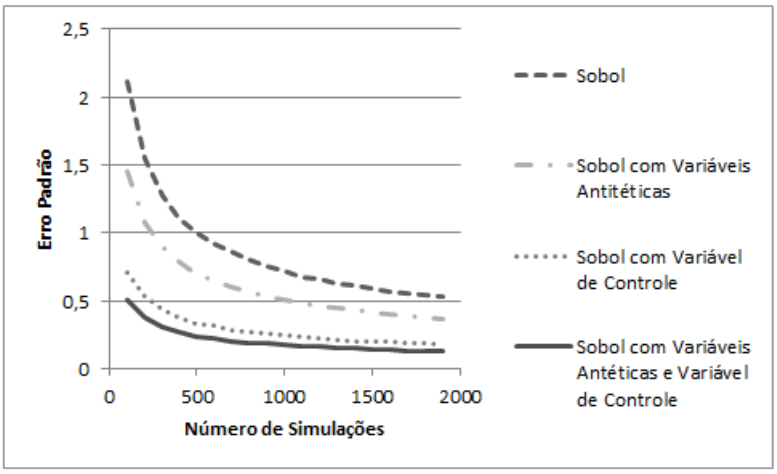

Figura 9.5: *Aplicação de técnicas de redução de variância na simulação do prêmio de uma opção de call europeia utilizando Sobol como RNG.*

9.3.2 Experimentos do Cálculo de VaR

Nesta seção, realizamos uma análise experimental do algoritmo de cálculo de VaR estudado nesse capítulo. Em vista à exploração de capacidade computacional, todos os experimentos foram realizados supondo o cálculo de VaR em um portfólio Π_0 de Opções Europeias. As mesmas foram precificadas conforme método de Black & Scholes visto na seção 9.1.1. O VaR foi estimado a um nível de confiança de 95% para um período de 1 dia.

Para simulação dos preços dos ativos foram utilizados os seguintes parâmetros:

- μ: -0,05%
- σ: 2%

Para o cálculo das opções foram utilizados as seguintes configurações:

- σ: 25%
- r_f: 10%
- Dias úteis até o vencimento: 126

- A i-ésima opção do portfólio possui valores de *Spot* e *Strike* iguais a i.

O Algoritmo 8 foi implementado tanto em CPU quanto em GPU em vista à obtenção do $VaR_{95\%}$ de 1 dia da carteira Π_0 gerada. O portfólio Π_0 foi gerado inicialmente com um total de 2 mil opções, o que resultou em um valor total de R$191.740. Nessa configuração, o valor obtido para o $VaR_{95\%}$ de 1 dia foi de aproximadamente -R$896. Isso quer dizer que a carteira gerada tem uma previsão de perda máxima de R$896 em 1 dia a um nível de confiança de 95%.

Nas figuras 9.6, 9.7 e 9.8 podemos observar a distribuição da variação de 1 dia no valor das carteiras simuladas. Na figura 9.6 apresentamos o resultado para um total de 128 simulações, enquanto que na figura 9.7 há um total de 1280 simulações. Como podemos notar, quanto maior o número de simulações, mais próximo o resultado de uma distribuição normal, o que fica evidente na figura 9.8 que corresponde a um número grande de simulações de 51.200.

Analise de tempo computacional foi feita variando o número de simulações e o número total de opções no portfólio. Nas tabelas 9.1 e 9.2 podemos observar o tempo computacional gasto em CPU e GPU, enquanto que nas figuras 9.9 e 9.10 são exibidos os *speedups* obtidos. Para um total de 2.500 opções com 51.200 simulações, foi obtido um *speedup* de mais de 50 vezes, como podemos observar na figura 9.10.

9.4 Conclusão

Na primeira parte do capitulo foi apresentado um algoritmo via simulação para cálculo do prêmio de uma opção europeia. De posse do mesmo, foram propostas técnicas para melhoria da qualidade do resultado obtido. Em análise experimental, ficou comprovado a assertividade das técnicas apresentadas, com uma melhoria de mais de 4 vezes no erro padrão da simulação.

Após essa análise de precificação, estudamos o cálculo de risco de mercado do ponto de vista da métrica *Value-at-Risk*. Após conceituação do problema e de seus diferentes métodos de cálculo, apresentamos um algoritmo via Simulação de Monte Carlo proposto em GPU. Em análise experimental, o cálculo de VaR mostrou ser muito suscetível para GPGPU com um *speedup* de mais de 50 vezes.

9.5 Notas e Leituras Complementares

Para uma introdução simplificada ao cálculo de risco pelo VaR veja (Kimura *et al.*, 2010). Para um aprofundamento no assunto, veja (Gregoriou, 2009), (Choudhry e Tanna, 2007) e (Jorion, 2003). Adicionalmente, (Cherubini e Lunga, 2007) fornece uma abordagem em gerenciamento de risco com foco em engenharia de software e em uma modelagem orientada a objetos.

(Hull, 2012) apresenta a precificação de outros tipos de opções, além dos vistos aqui, dentre outros produtos financeiros. Um guia completo de gestão de portfólios e precificação em finanças pode ser obtido em (Lee *et al.*, 2010b). Para uma abordagem objetiva e simplificada do modelo de Black & Scholes e apreçamento via não-arbitragem em língua portuguesa, veja (Zubelli e Souza, 2007).

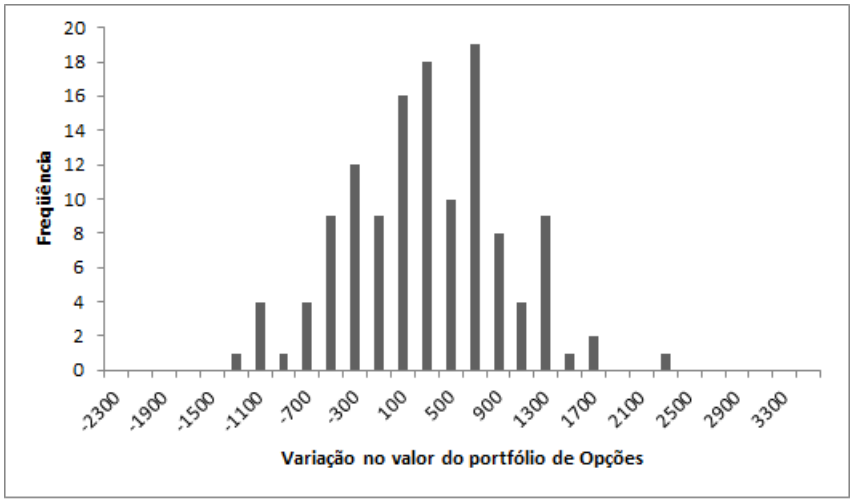

Figura 9.6: *Variação no valor do Portfólio de Opções. Número de Simulações fixo em 128.*

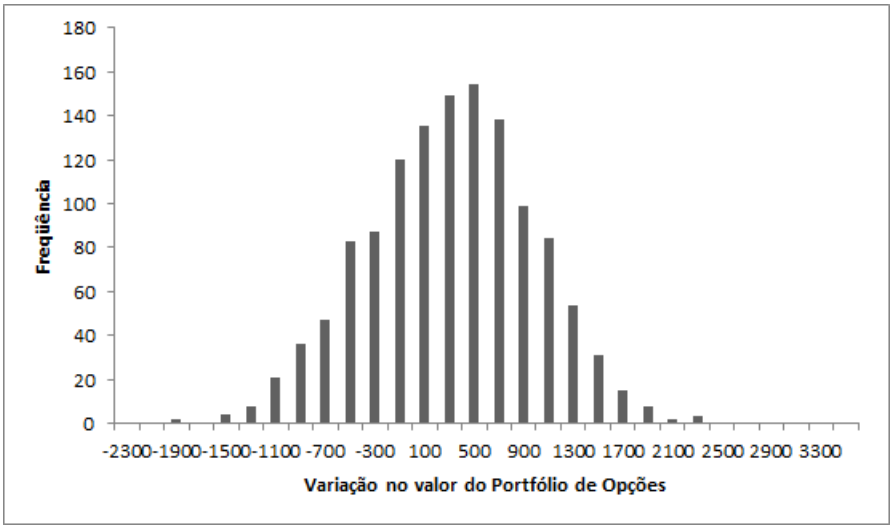

Figura 9.7: *Variação no valor do Portfólio de Opções. Número de Simulações fixo em 1.280.*

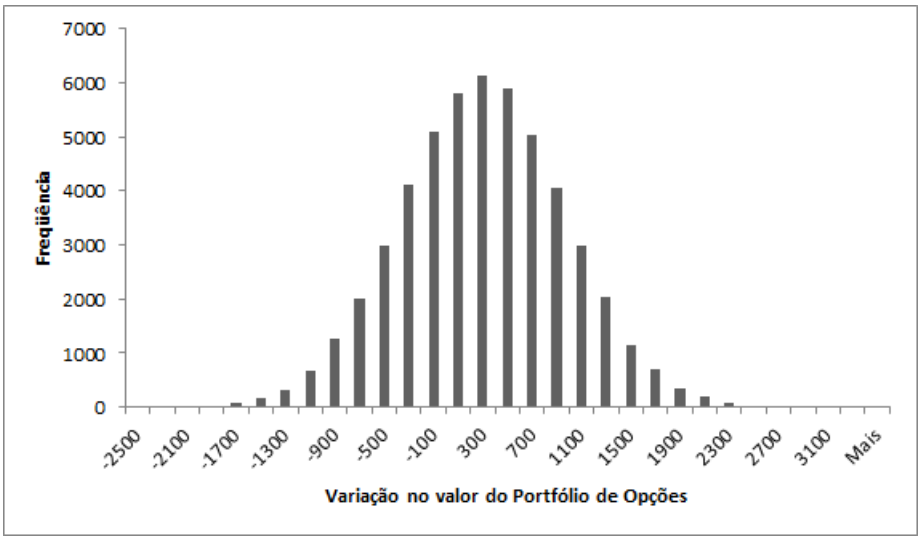

Figura 9.8: *Variação no valor do Portfólio de Opções. Número de Simulações fixo em 51.200.*

Número de Simulações	Tempo CPU (s)	Tempo GPU (s)
1280	2,91	0,07
3200	7,178	0,16
6400	14,79	0,31
12800	28,59	0,60
25600	56,58	1,19
38400	84,03	1,81
51200	112,02	2,43

Tabela 9.1: *Tempo Computacional gasto em CPU e GPU no cálculo de VaR em função do número de simulações. Número de opções fixo em 2.000.*

Número de Opções	Tempo CPU (s)	Tempo GPU (s)
500	28,03	0,99
750	42,54	1,23
1000	56,31	1,47
1250	70,41	1,70
1500	84,95	1,95
1750	99,17	2,18
2000	112,02	2,43
2500	146,76	2,90

Tabela 9.2: *Tempo Computacional gasto em CPU e GPU no cálculo de VaR em função do número de opções no portfólio. Número de simulações fixo em 51200.*

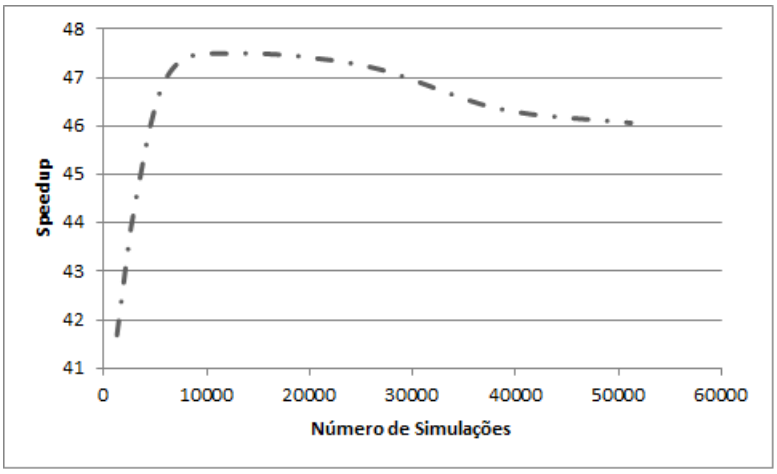

Figura 9.9: *Speedup no cálculo de VaR de uma carteira com 2000 opções em função do número de simulações.*

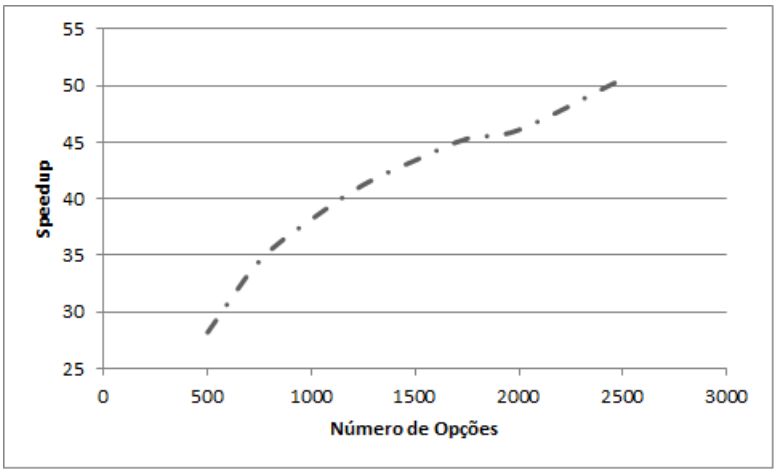

Figura 9.10: *Speedup no cálculo de VaR em função do número de opções no portfólio. Número de simulações fixo em 51200.*

Parte V

Conclusão

Capítulo 10

Conclusão

Neste trabalho estudamos o ferramental matemático e computacional necessário para modelagem estocástica em finanças com a utilização de GPUs como plataforma de aceleração.

Para isso, apresentamos a GPU como uma plataforma de computação de propósito geral com a introdução a CUDA em uma arquitetura NVIDIA Fermi. Nesse sentido, apresentamos um resumo das principais técnicas de otimização utilizadas nessa plataforma.

Tendo em vista o papel base em simulações estocásticas de qualquer tipo, apresentamos um estudo dos mais diferentes tipos de PRNGs e QRNGs. Além de abordar técnicas de paralelização dos mesmos e estudar a portabilidade dos geradores Sobol e MRG32k3a em GPU. Em seguida, discutimos questões relativas a simulação de Monte Carlo e introduzimos a teoria de soluções de equações diferenciais estocásticas como base fundamental em modelagem estocástica em finanças.

Em um primeiro estudo de caso, resolvemos o problema de *Stops* Ótimos em finanças. Após definição formal do problema, revisamos o modelo trinomial proposto por (Warburtona e Zhang, 2006) e propomos um algoritmo alternativo para o mesmo via simulação utilizando uma técnica de aceitação e rejeição para probabilidade de transição dos preços de (Cox *et al.*, 1979). O algoritmo sugerido é, comparativamente, de fácil implementação e naturalmente portável para GPU. Em seguida, apresentamos uma modelagem estocástica genérica para o problema e propomos um algoritmo para cálculo da esperança de ganho de um estratégia de *Stop Gain*. De posse desse algoritmo, resolvemos o problema de otimização de *Stop* Ótimo. Em análise experimental, as implementações em GPU mostraram ser mais escaláveis. Para o algoritmo de obtenção de esperança de retorno de um *Stop Gain*, foi alcançado um *speedup* de mais de 12 vezes.

No segundo estudo de caso, sobre cálculo de risco de mercado, inicialmente foi apresentado um algoritmo via simulação para precificação de opções. De posse do mesmo, foram propostas técnicas para melhoria da qualidade do resultado obtido. Em análise experimental, ficou comprovada a assertividade das técnicas apresentadas, com uma melhoria de mais de 4 vezes no erro padrão da simulação. Para o cálculo de risco pela métrica *Value-at-Risk*, com a proposição de um algoritmo via Simulação de Monte Carlo em GPU, foi obtido um *speedup* de mais de 50 vezes.

Parte VI

Apêndice

Apêndice A

NVIDIA CUDA

A.1 Compute Capability

	Compute Capability				
Especificação Técnica	**1.0**	**1.1**	**1.2**	**1.3**	**2.x**
Máxima dimensão da grade de blocos de threads	2				3
Máxima x-, y- ou z-dimensão da grade de blocos	65535				
Máxima dimensão de um bloco de threads	3				
Máxima x- ou y-dimensão de blocos em eixos de threads	512				1024
Máxima z-dimensão de blocos em eixos de threads	64				
Tamanho de um Warp	32				
Máximo número de blocos residentes por multiprocessador	8				
Máximo número de warps residentes por multiprocessador	24		32		48
Máximo número de threads residentes por multiprocessador	768		1024		1536
Número de registradores de 32 bits por multiprocessador	8k		16k		32k
Máxima quantidade de Memória Compartilhada por multiprocessador	16 KB				48 KB
Quantidade de memória local por thread	16 KB				512 KB
Tamanho de Memória Constante	64 KB				

Figura A.1: *Especificação técnica por Compute Capability*

A.2 Funções Otimizadas pela Opção *-use_fast_math*

Operação	Função
x/y	__fdividef(x,y)
sinf(x)	__sinf(x)
cosf(x)	__cosf(x)
tanf(x)	__tanf(x)
sincosf(x,sptr,cptr)	__sincosf(x,sptr,cptr)
logf(x)	__logf(x)
log2f(x)	__log2f(x)
log10f(x)	__log10f(x)
expf(x)	__expf(x)
exp10f(x)	__exp10f(x)
powf(x,y)	__powf(x,y)

Figura A.2: *Funções afetadas por -use_fast_math*

Referências Bibliográficas

Almasi e Gottlieb(1989) G. S. Almasi e A. Gottlieb. *Highly Parallel Computing*. Benjamin/-Cummings, Redwood CA. Citado na pág. 9

Alves(2008) C.A.M. Alves. Modelos de volatilidade estocástica para o índice ibovespa: Reversão rápida à média e análise assintótica. Citado na pág. 60

Amdahl(1967) G. M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. Em *Proceedings of the April 18-20, 1967, spring joint computer conference*, AFIPS '67 (Spring), páginas 483–485, New York, NY, USA. ACM. doi: 10.1145/1465482.1465560. URL http://doi.acm.org/10.1145/1465482.1465560. Citado na pág. xv, 13

Antonov e Saleev(1979) I. A. Antonov e V. M. Saleev. An economic method of computing lpt-sequences. Citado na pág. 44

Black e Scholes(1973) F. Black e M. Scholes. The pricing of options and corporate liabilities. *Journal of Political Economy*. Citado na pág. 55, 88

Box e Muller(1958) G. E. P. Box e M. E Muller. A note on the generation of random normal deviates. *Ann. Math. Stat.* Citado na pág. 40

Brodtkorb *et al.*(2012) A. R. Brodtkorb, T. R. Hagen e Ma. L. Saetra. Graphics processing unit (gpu) programming strategies and trends in gpu computing. *Journal of Parallel and Distributed Computing*. Citado na pág. xv, 17

Buluc *et al.*(2010) A. Buluc, J. R. Gilbert e C. Budak. Solving path problems on the gpu. *Parallel Comput.*, 36(5-6):241–253. ISSN 0167-8191. doi: 10.1016/j.parco.2009.12.002. URL http://dx.doi.org/10.1016/j.parco.2009.12.002. Citado na pág. 3

C. Mazel e Hill(2011) B. Bachelet C. Mazel e D.R.C Hill. Shoverand: A model-driven framework to easily generate random numbers on gp-gpu. *High Performance Computing and Simulation (HPCS)*. Citado na pág. xv, 45, 46

Che *et al.*(2008) S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer e K. Skadron. A performance study of general-purpose applications on graphics processors using cuda. *J. Parallel Distrib. Comput.*, 68(10):1370–1380. ISSN 0743-7315. doi: 10.1016/j.jpdc.2008.05.014. URL http://dx.doi.org/10.1016/j.jpdc.2008.05.014. Citado na pág. xv, 29, 30, 31

Cherubini e Lunga(2007) U. Cherubini e G. D. Lunga. *Structured Finance: The Object Oriented Approach*. The Wiley Finance Series. Wiley. ISBN 9780470512722. Citado na pág. 97

Choudhry e Tanna(2007) M. Choudhry e K. Tanna. *An Introduction to Value-at-Risk*. Securities Institute. Wiley. ISBN 9780470033777. Citado na pág. 92, 97

Chow *et al.*(1971) Y.S. Chow, H. Robbins e D. Siegmund. *Great expectations: the theory of optimal stopping*. Houghton Mifflin. Citado na pág. 86

Cover(1987) T. M. Cover. Gambler's ruin: A random walk on the simplex. *Open Problems in Communications and Computation*. Citado na pág. 76

Cox *et al.***(1979)** J. Cox, S. Ross e M. Rubenstein. Option pricing: a simplified approach. *Journal of Financial Economics*. Citado na pág. ix, xvi, 75, 76, 103

Dally e Nickolls(2010) W. J. Dally e J. Nickolls. The gpu computing era. *IEEE Computer Society*. Citado na pág. xv, 17, 18, 19

Dehne e Yogaratnam(2010) F. Dehne e K. Yogaratnam. Exploring the limits of gpus with parallel graph algorithms. *CoRR*, abs/1002.4482. Citado na pág. 3

Dixon *et al.***(2009)** M. Dixon, J. Chong e K. Keutzer. Acceleration of market value-at-risk estimation. Em *Proceedings of the 2nd Workshop on High Performance Computational Finance*, WHPCF '09, páginas 5:1–5:8, New York, NY, USA. ACM. ISBN 978-1-60558-716-5. doi: 10.1145/1645413.1645418. URL http://doi.acm.org/10.1145/1645413.1645418. Citado na pág. 3, 93

Egloff(2010) D. Egloff. High performance finite difference PDE solvers on GPUs, 2010. Citado na pág. 3

Fernando(2003) M. J. Fernando, R. e Kilgard. *The Cg Tutorial: The Definitive Guide to Programmable Real-Time Graphics*. Addison-Wesley Professional. Citado na pág. 16

Fischer *et al.***(1999)** G. W. Fischer, Z. Carmon, D. Ariely, G. Zauberman e P. L'Ecuyer. Good parameters and implementations for combined multiple recursive random number generators. *Oper. Res.*, 47(1):159–164. ISSN 0030-364X. doi: 10.1287/opre.47.1.159. URL http://dx.doi.org/10.1287/opre.47.1.159. Citado na pág. 42

Flynn(1972) M. J. Flynn. *Some Computer Organizations and Their Effectiveness*. IEEE Trans. on Computing. Citado na pág. 9

Giles *et al.***(2008)** M. B. Giles, F. Y. Kuo, I. H. Sloan e B. J. Waterhouse. Quasi-monte carlo for finance applications. Em Geoffry N. Mercer e A. J. Roberts, editors, *Proceedings of the 14th Biennial Computational Techniques and Applications Conference, CTAC-2008*, volume 50 of *ANZIAM J.*, páginas C308–C323. Citado na pág. 95

Glasserman(2004) P. Glasserman. *Monte Carlo Methods in Financial Engineering*. Applications of Mathematics Series. Springer. ISBN 9780387004518. Citado na pág. 76

Govindaraju e Manocha(2007) N. K. Govindaraju e D. Manocha. Cache-efficient numerical algorithms using graphics hardware. *Parallel Comput.*, 33(10-11):663–684. ISSN 0167-8191. doi: 10.1016/j.parco.2007.09.006. URL http://dx.doi.org/10.1016/j.parco.2007.09.006. Citado na pág. 20

Gregoriou(2009) G. N. Gregoriou. *The VAR Implementation Handbook*. McGraw-Hill Finance & Investing. McGraw-Hill Companies,Incorporated. ISBN 9780071615136. Citado na pág. 97

Gut(2009) A. Gut. *Stopped Random Walks: Limit Theorems and Applications*. Springer series in operations research. Springer. ISBN 9780387878355. Citado na pág. 86

Harish e Narayanan(2007) P. Harish e P. J. Narayanan. Accelerating large graph algorithms on the gpu using cuda. Em *Proceedings of the 14th international conference on High performance computing*, HiPC'07, páginas 197–208, Berlin, Heidelberg. Springer-Verlag. ISBN 3-540-77219-7, 978-3-540-77219-4. URL http://dl.acm.org/citation.cfm?id=1782174.1782200. Citado na pág. 3

Hida(1980) T. Hida. *Brownian Motion*. Springer. Citado na pág. 57

Hong e Kim(2009) S. Hong e H. Kim. An analytical model for a gpu architecture with memory-level and thread-level parallelism awareness. Em *Proceedings of the 36th annual international symposium on Computer architecture*, ISCA '09, páginas 152–163. ACM. ISBN 978-1-60558-526-0. doi: 10.1145/1555754.1555775. Citado na pág. 31

Hull(2012) J. Hull. *Options, Futures, and Other Derivatives and DerivaGem CD Package*. Prentice Hall. ISBN 9780132777421. Citado na pág. xvi, 71, 75, 88, 91, 97

Huynh *et al.***(2011)** H. T. Huynh, V. S. Lai e I. Soumare. *Stochastic Simulation and Applications in Finance with MATLAB Programs*. The Wiley Finance Series. Wiley. ISBN 9780470722138. Citado na pág. xv, 39, 74, 75

IEEE Task P754(1985) IEEE Task P754. *ANSI/IEEE 754-1985, Standard for Binary Floating-Point Arithmetic*. IEEE, New York, Agosto 12 1985. Citado na pág. 18

IEEE Task P754(2008) IEEE Task P754. *IEEE 754-2008, Standard for Floating-Point Arithmetic*. Citado na pág. 18

Imkeller e Rogers(2010) N. Imkeller e L. C. G. Rogers. Trading to stops. URL http://www.statslab.cam.ac.uk/~chris/papers.html. Citado na pág. 71

Jasra e Del Moral(2011) Ajay Jasra e Pierre Del Moral. Sequential monte carlo methods for option pricing. doi: 10.1080/07362994.2011.548993. Citado na pág. 90

Joe e Kuo(2008) S. Joe e F. Y. Kuo. Código fonte: Sobol sequence generator, 2008. URL http://web.maths.unsw.edu.au/~fkuo/sobol/. Último acesso em 29/01/2013. Citado na pág. 68

Jorion(2003) P. Jorion. *Value at risk: a nova fonte de referência para a gestão do risco financeiro*. Bolsa de Mercadorias & Futuros. ISBN 9788574380070. Citado na pág. xvi, 93, 97

Jung e O'Leary(2006) J. H. Jung e D. P. O'Leary. Cholesky decomposition and linear programming on a gpu. Dissertação de Mestrado, University of Maryland. Citado na pág. 95

Jung e O'Leary(2009) J. H. Jung e D. P. O'Leary. Implementing an interior point method for linear programs on a CPU-GPU system. *Electronic Transactions on Numerical Analysis*. Citado na pág. 3

Kaplan(1991) W. Kaplan. *Advanced Calculus*. Addison-Wesley, Advanced Book Program. ISBN 9780201578881. Citado na pág. 58

Karlin e Taylor(1975) S. Karlin e H. M. Taylor. *A First Course in Stochastic Processes*. Number vol. 1. Academic Press. ISBN 9780123985521. Citado na pág. 86

Kimura *et al.***(2010)** H. Kimura, A. S. Suen, L. C. J. Perera e L. F. C. Basso. *Value at Risk - Como Entender e Calcular o Risco pelo VaR*. INSIDE BOOKS. ISBN 9788560550074. Citado na pág. 87, 92, 97

Kirk e Hwu(2010) D. B. Kirk e W. W. Hwu. *Programming Massively Parallel Processors: A Hands-on Approach*. Applications of GPU Computing Series. Elsevier Science. ISBN 9780123814739. Citado na pág. xv, 15, 16

Knuth(1997) D. E. Knuth. *The art of computer programming, volume 2 (3rd ed.): seminumerical algorithms*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA. ISBN 0-201-89684-2. Citado na pág. 35, 36, 42

L'Ecuyer(2006) P. L'Ecuyer. Testu01: A c library for empirical testing of random number generators. *ACM Transactions on Mathematical Software*. Citado na pág. 36

L'Ecuyer(2007) P. L'Ecuyer. *Random Number Generation*, páginas 93–137. John Wiley & Sons, Inc. ISBN 9780470172445. doi: 10.1002/9780470172445.ch4. URL http://dx.doi.org/10.1002/9780470172445.ch4. Citado na pág. 35

Lee *et al.***(2010a)** A. Lee, C. Yau, M. B. Giles, A. Doucet e C. C. Holmes. On the utility of graphics cards to perform massively parallel simulation of advanced monte carlo methods. *Journal of Computational and Graphical Statistics*, páginas 769–789. Citado na pág. 3

Lee *et al.***(2010b)** C. F. Lee, A. C. Lee e J. Lee. *Handbook of Quantitative Finance and Risk Management*. Springer. ISBN 9780387771175. Citado na pág. 97

Lee *et al.***(2009)** M. Lee, C. H. Chun e S. Hong. Financial derivatives modeling using gpu's. Em *Proceedings of the 2009 International Conference on Scalable Computing and Communications; Eighth International Conference on Embedded Computing*, SCALCOM-EMBEDDEDCOM '09, páginas 440–445. IEEE Computer Society. ISBN 978-0-7695-3825-9. doi: 10.1109/EmbeddedCom-ScalCom.2009.85. Citado na pág. 40

Marsaglia(1995) G. Marsaglia. The diehard battery of tests of randomness. Citado na pág. 36

Marsaglia(2003) G. Marsaglia. Xorshift rngs. *Journal of Statistical Software*, 8(14):1–6. ISSN 1548-7660. URL http://www.jstatsoft.org/v08/i14. Citado na pág. 45

Marsaglia e Tsang(2000) G. Marsaglia e W. W. Tsang. The ziggurat method for generating random variables. *Journal of Statistical Software*, 5(8):1–7. Citado na pág. 40

Matsumoto e Nishimura(1998) M. Matsumoto e T. Nishimura. Mersenne twister: A 623-dimensionally equidistributed uniform pseudorandom number generator. *ACM Transactions on Modeling and Computer Simulation*. Citado na pág. 37

Matsumoto e Nishimura(2009) M. Matsumoto e T. Nishimura. A c-program for mt19937, 2009. URL http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/emt.html. Último acesso em 29/01/2013. Citado na pág. 68

Matteis e Pagnutti(1990) A. De Matteis e S. Pagnutti. Long-range correlations in linear and non-linear random number generators. *Parallel Computing*. Citado na pág. 41

Mattson *et al.***(2004)** T. G. Mattson, B. A. Sanders e B. L. Massingil. *Patterns for Parallel Programming*. Addison-Wesley Professional. Citado na pág. xv, 9, 10, 11

Mikosch(1999) T. Mikosch. *Elementary Stochastic Calculus With Finance in View*. World Scientific Publishing Company. Citado na pág. 63

Miller *et al.***(2010)** F. P. Miller, A. F. Vandome e M. B. John. *Marsaglia Polar Method*. VDM Publishing. ISBN 9786132746498. Citado na pág. 40

Muller e Frauendiener(2013) T. Muller e J. Frauendiener. Charged particles constrained to a curved surface. *European Journal of Physics*, 34(1):147. URL http://stacks.iop.org/0143-0807/34/i=1/a=147. Citado na pág. 3

Narayana *et al.***(2012)** D. Narayana, P. Somawanshi e M. Joshi. Stochastic differential equations simulation using gpu. Citado na pág. 62

Nguyen(2007) H. Nguyen. *Gpu gems 3*. Addison-Wesley Professional. ISBN 9780321545428. Citado na pág. 35

NVIDIA(2009a) NVIDIA. *White paper NVIDIA's Next Generation CUDA Compute Architecture: Fermi*. Citado na pág. xv, 19, 20

NVIDIA(2009b) NVIDIA. Site de computação em finanças da nvidia, 2009b. URL http://www.nvidia.com/object/computational_finance.html. Último acesso em 29/01/2013. Citado na pág. xv, 4

NVIDIA(2010a) NVIDIA. *Cuda Ocuppancy Calculator*. URL http://developer.download.nvidia.com/compute/cuda/3_1/sdk/docs/CUDA_Occupancy_calculator.xls. Citado na pág. 31

NVIDIA(2010b) NVIDIA. *Cuda C Best Practices Guide 3.2*. Citado na pág. 29

NVIDIA(2011) NVIDIA. Cuda 4.0 math libraries performance. *NVDIA Corporation*. Citado na pág. 45

NVIDIA(2011a) NVIDIA. *Cuda C Programming Guide 4.0*. Citado na pág. xv, 21, 22, 29, 32

Oksendal(1992) B. Oksendal. *Stochastic Differential Equations*. Springer-Verlag. Citado na pág. 72, 86

Pages e Wilbertz(2010) G. Pages e B. Wilbertz. Parallel implementation of quantization methods for the valuation of swing options on gpgpu. Em *High Performance Computational Finance (WHPCF), 2010 IEEE Workshop on*, páginas 1 –5. doi: 10.1109/WHPCF.2010.5671811. Citado na pág. 3, 91

Pages e Wilbertz(2012) G. Pages e B. Wilbertz. Gpgpus in computational finance: massive parallel computing for american style options. *Concurr. Comput. : Pract. Exper.*, 24(8):837–848. ISSN 1532-0626. doi: 10.1002/cpe.1774. Citado na pág. 3, 91

Preis(2011) T. Preis. GPU-computing in econophysics and statistical physics. *The European Physical Journal - Special Topics*, 194(1):87–119. ISSN 1951-6355. doi: 10.1140/epjst/e2011-01398-x. URL http://dx.doi.org/10.1140/epjst/e2011-01398-x. Citado na pág. 3

Reuillon *et al.***(2008)** R. Reuillon, D. Hill, Z. El Bitar e V. Breton. Rigorous distribution of stochastic simulations using the distme toolkit. *IEEE Transactions On Nuclear Science*. Citado na pág. 41

Roy(2002) R. Roy. Comparison of different techniques to generate normal random variables. Citado na pág. 40

Sadiq *et al.***(2012)** S. K. Sadiq, F. Noé e G. De Fabritiis. Kinetic characterization of the critical step in hiv-1 protease maturation. *Proceedings of the National Academy of Sciences*. doi: 10.1073/pnas.1210983109. Citado na pág. 3

Saito *et al.***(2012)** K. Saito, E. Koizumi e H. Koizumi. Application of parallel hybrid algorithm in massively parallel gpgpu-the improved effective and efficient method for calculating coulombic interactions in simulations of many ions with simion. *Journal of The American Society for Mass Spectrometry*, 23:1609–1615. ISSN 1044-0305. doi: 10.1007/s13361-012-0435-6. URL http://dx.doi.org/10.1007/s13361-012-0435-6. Citado na pág. 3

Satish *et al.***(2009)** Na. Satish, M. Harris e M. Garland. Designing efficient sorting algorithms for manycore gpus. Em *Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing*, IPDPS '09, páginas 1–10, Washington, DC, USA. IEEE Computer Society. ISBN 978-1-4244-3751-1. doi: 10.1109/IPDPS.2009.5161005. URL http://dx.doi.org/10.1109/IPDPS.2009.5161005. Citado na pág. 3

Shen e Wang(2001) S. Shen e A. Wang. On stop-loss strategies for stock investments. *Applied Mathematics and Computation*. Citado na pág. 71

Shiryaev e Aries(2008) A. N. Shiryaev e A. B. Aries. *Optimal Stopping Rules*. Applications of mathematics. Springer-Verlag Berlin Heidelberg. ISBN 9783540740117. Citado na pág. 86

Sobol(1967) I. M. Sobol. On the distribution of points in a cube and the approximate evaluation of integrals. *Computational Mathematics and Mathematical Physics*, 7(4):86+. Citado na pág. 43

Solomon *et al.*(2010) S. Solomon, R. K. Thulasiram e P. Thulasiraman. Option pricing on the gpu. Em *Proceedings of the 2010 IEEE 12th International Conference on High Performance Computing and Communications*, HPCC '10, páginas 289–296, Washington, DC, USA. IEEE Computer Society. ISBN 978-0-7695-4214-0. doi: 10.1109/HPCC.2010.54. Citado na pág. 3, 91

Spampinato e Elstery(2009) D. G. Spampinato e A. C. Elstery. Linear optimization on modern gpus. Em *Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing*, IPDPS '09, páginas 1–8, Washington, DC, USA. IEEE Computer Society. ISBN 978-1-4244-3751-1. doi: 10.1109/IPDPS.2009.5161106. URL http://dx.doi.org/10.1109/IPDPS.2009.5161106. Citado na pág. 3

Stojanovski *et al.*(2012) M. Stojanovski, D. Gjorgjevikj e G. Madjarov. Parallelization of dynamic programming in nussinov rna folding algorithm on the cuda gpu. Em Ljupco Kocarev, editor, *ICT Innovations 2011*, volume 150 of *Advances in Intelligent and Soft Computing*, páginas 279–289. Springer Berlin Heidelberg. ISBN 978-3-642-28663-6. Citado na pág. 3

W. H. Press e Flannery(2007) W. T. Vetterling W. H. Press, S. A. Teukolsky e B. P. Flannery. *Numerical Recipes 3rd Edition: The Art of Scientific Computing*. Cambridge University Press, New York, NY, USA, 3 edição. ISBN 0521880688, 9780521880688. Citado na pág. 80

Wald(1945) A. Wald. Sequential tests of statistical hypotheses. *The Annals of Mathematical Statistics*, 16(2):117–186. ISSN 00034851. doi: 10.2307/2235829. Citado na pág. 86

Warburtona e Zhang(2006) A. Warburtona e Z. G. Zhang. A simple computational model for analyzing the properties of stop-loss, take-profit, and price breakout trading strategies. *Computers and Operations Research*. Citado na pág. ix, 71, 72, 73, 75, 81, 86, 103

Zhang(2001) Q. Zhang. Stock trading: an optimal selling rule. *SIAM Journal on Control and Optimization*. Citado na pág. 71, 75

Zubelli e Souza(2007) J. Zubelli e M. O. Souza. *Modelagem Matemática em Finanças Quantitativas em Tempo Discreto*. Notas em Matemática Aplicada. SBMAC. ISBN 9788586883347. Citado na pág. 97

Índice Remissivo