UNIVERSIDADE DE SÃO PAULO

Instituto de Ciências Matemáticas e de Computação

Coprojeto hardware/software das equações de Black-Scholes para precificação de opções no mercado financeiro

Thadeu Antonio Ferreira de Melo Costa

Dissertação de Mestrado do Programa de Pós-Graduação em Ciências
de Computação e Matemática Computacional (PPG-CMC)

SERVIÇO DE PÓS-GRADUAÇÃO DO ICMC-USP

Data de Depósito:

Assinatura: _____

Thadeu Antonio Ferreira de Melo Costa

**Coprojeto hardware/software das equações de
Black-Scholes para precificação de opções no mercado
financeiro**

Dissertação apresentada ao Instituto de Ciências
Matemáticas e de Computação – ICMC-USP,
como parte dos requisitos para obtenção do título
de Mestre em Ciências – Ciências de Computação e
Matemática Computacional. *VERSÃO REVISADA*

Área de Concentração: Ciências de Computação e
Matemática Computacional

Orientador: Prof. Dr. Eduardo Marques

**USP – São Carlos
Setembro de 2018**

Ficha catalográfica elaborada pela Biblioteca Prof. Achille Bassi
e Seção Técnica de Informática, ICMC/USP,
com os dados inseridos pelo(a) autor(a)

A635c Antonio Ferreira de Melo Costa, Thadeu
 Coprojeto hardware/software das equações de Black-
Scholes para precificação de opções no mercado
financeiro / Thadeu Antonio Ferreira de Melo Costa;
orientador Eduardo Marques. -- São Carlos, 2018.
 92 p.

 Dissertação (Mestrado - Programa de Pós-Graduação
em Ciências de Computação e Matemática
Computacional) -- Instituto de Ciências Matemáticas
e de Computação, Universidade de São Paulo, 2018.

 1. FPGA. 2. OpenCL. 3. Finanças. 4. Monte Carlo.
5. Black-Scholes. I. Marques, Eduardo, orient. II.
Título.

Thadeu Antonio Ferreira de Melo Costa

**Hardware/software codesign of Black-Scholes equations
for option pricing in the financial market**

Master dissertation submitted to the Institute of
Mathematics and Computer Sciences – ICMC-USP,
in partial fulfillment of the requirements for the
degree of the Master Program in Computer Science
and Computational Mathematics. *FINAL VERSION*

Concentration Area: Computer Science and
Computational Mathematics

Advisor: Prof. Dr. Eduardo Marques

**USP – São Carlos
September 2018**

*Este trabalho é dedicado às minhas tias:
Tia Tate, Tia Ica e Tia Láza
Eu não seria o homem que sou hoje sem essas mulheres*

AGRADECIMENTOS

Gostaria de agradecer em primeiro lugar e com especial destaque a minha família que me apoiou em todos os momentos da minha vida e nas decisões que eu tomei ao longo do caminho.

Outro agradecimento especial vai para o professor Eduardo Marques que acreditou no meu potencial e compartilhou a sua experiência ao longo deste projeto.

Agradeço a todos os membros do Laboratório de Computação Reconfigurável por proporcionar um agradável ambiente de trabalho e pela troca de ideias e conhecimento.

Por fim, agradeço aos contribuintes paulistas e brasileiros que por meio de seus impostos permitiu que a infraestrutura para o desenvolvimento deste projeto pudesse ser apoiada pela USP, Fapesp e CAPES.

“Random numbers should not be generated with a method chosen at random.”

(Donald Knuth)

“The generation of random numbers is too important to be left to chance.”

(Robert R. Coveyou)

RESUMO

MELO, T. A. F. **Coprojeto hardware/software das equações de Black-Scholes para precificação de opções no mercado financeiro.** 2018. 92 p. Dissertação (Mestrado em Ciências – Ciências de Computação e Matemática Computacional) – Instituto de Ciências Matemáticas e de Computação, Universidade de São Paulo, São Carlos – SP, 2018.

Este trabalho apresenta a implementação em hardware das Equações de Black-Scholes para precificação de opções usando Método de Monte Carlo. A implementação foi feita em OpenCL compatível com FPGAs recentes da Altera/Intel. Essa implementação é modular e permite a utilização de diferentes geradores de números aleatórios em configurações diferentes de software e hardware. A proposta é que essas implementações possam aproveitar as vantagens de cada componente, resultando em uma maior quantidade de simulações e por consequência melhorando a precisão dos resultados.

Palavras-chave: FPGA, OpenCL, Black-Scholes, Monte Carlo, Finanças.

ABSTRACT

MELO, T. A. F. **Hardware/software codesign of Black-Scholes equations for option pricing in the financial market.** 2018. 92 p. Dissertação (Mestrado em Ciências – Ciências de Computação e Matemática Computacional) – Instituto de Ciências Matemáticas e de Computação, Universidade de São Paulo, São Carlos – SP, 2018.

This paper presents the hardware implementation of Black-Scholes Equations for pricing options using Monte Carlo Method. The implementation was made in OpenCL compatible with recent Altera / Intel FPGAs. This implementation is modular and allows the use of different random number generators in different software and hardware configurations. The proposal is that these implementations can take advantage of each component, resulting in a greater number of simulations and consequently improving the accuracy of the results.

Keywords: FPGA, OpenCL, Black-Scholes, Monte Carlo, Finance.

LISTA DE ILUSTRAÇÕES

Figura 1 – <i>Semper Augustus</i> , o bulbo mais famoso de tulipa, que foi vendido por preço recorde	29
Figura 2 – Retorno de opções de compra (amarelo) e venda (azul) em função do valor do ativo(vermelho)	32
Figura 3 – Distribuição do preço de uma ação ao longo do tempo, simulada por movimentos brownianos	34
Figura 4 – Curvas de Distribuição Normal	35
Figura 5 – Máquina de Estados de um PRNG	43
Figura 6 – Fluxo do algoritmo de PRNG “Middle Square”	43
Figura 7 – Gerador Linear de Deslocamento, 2D	44
Figura 8 – Gerador Linear de Deslocamento, 3D	44
Figura 9 – Mersenne Twister, 3D	45
Figura 10 – 16-bit Fibonacci LFSR	46
Figura 11 – Algoritmo de geradores do tipo WELL	49
Figura 12 – Transformada de Box-Muller para geração de distribuição normal a partir de distribuição uniforme	50
Figura 13 – Representação em coordenadas polares para transformada Box-Muller	51
Figura 14 – Compilador de BlueSpec para Verilog	53
Figura 15 – Modelo de organização de memória entre aplicação host e dispositivo computacional. Arstetchnica (2008)	54
Figura 16 – Diagrama do fluxo de dados entre kernels OpenCL por canais	58
Figura 17 – Diagrama de blocos da placa de desenvolvimento DE-150 Terasic (2018)	60
Figura 18 – Diagrama de blocos da placa de desenvolvimento S5PH-Q BittWare (2018)	60
Figura 19 – Diagrama de blocos do processador Nios II	61
Figura 20 – Estrutura de da interface de comunicação entre as memórias do PC e da FPGA com OpenCL da Intel	62
Figura 21 – Janela de usuário do Metarader 5	63
Figura 22 – Diagrama de fluxo do script de negociação no MetaTrader com interface externa	65
Figura 23 – Diagrama do fluxo para testar a qualidade do PRNG descrito em Bluespec	71
Figura 24 – Modelo de instrução customizada Multicycle para o Nios II	72
Figura 25 – Diagrama de Blocos da instrução customizada com o PRNG WELL512, Quartus II da Altera	73

Figura 26 – Estrutura da implementação dos geradores e transformada Box-Muller com comunicação do Nios II e um PC	73
Figura 27 – Fluxo de execução das regras em Bluespec para transformada Box-Muller na forma básica	75
Figura 28 – Estrutura da implementação em OpenCL e a comunicação entre Software e FPGA por PCIe	77
Figura 29 – Da esquerda para a direita: fluxo dos dados entre as memórias do programa host e a memória do dispositivo FPGA	79
Figura 30 – Estrutura do sistema completo com interação com o Metatrader 5 por Sockets	80
Figura 31 – Profile visual do programa de teste executado em um processador Nio II, na placa Cyclone IV	82
Figura 32 – Profile visual do Black-Scholes em software com o gerador PCG com otimização de compilador	84
Figura 33 – Profile visual do Black-Scholes em software com o gerador Xorshift64* com otimização de compilador	84
Figura 34 – Relatório do uso de elementos lógicos e blocos de memória dos kernels OpenCL	85
Figura 35 – Relatório de loops dos kernels OpenCL	85
Figura 36 – Diagrama do fluxo de dados entre os blocos dos kernels em OpenCL	86

LISTA DE ABREVIATURAS E SIGLAS

API	Application Programming Interface
ASICs	<i>Application Specific Integrated Circuits</i>
BSV	Bluespec System Verilog
CPU	Central Processing Unit
CUDA	Compute Unified Device Architecture
EDL	Electronic Design Language
FIFO	First-In-First-Out
FPGAs	Field Programmable Gate Arrays
HLS	High Level Sintesis
HME	Hipótese do Mercado Eficiente
HPC	High Performance Computing
IDE	<i>Integrated Development Environment</i>
IP	<i>Intellectual Property</i>
LFSR	Linear-feedback shift register
OpenCL	Open Computing Language
PRNG	Pseudo-Random Number Generator
RNG	Random Number Generator
SDK	Software Development Kit
SoC	<i>Sistem on a Chip</i>
TRNG	True Random Number Generators
VHDL	VHSIC Hardware Description Language
WELL	Well equidistributed long-period linear generator
XOR	OU EXCLUSIVO

SUMÁRIO

1	INTRODUÇÃO	23
1.1	Contextualização e Motivação	23
1.2	Objetivos	24
1.3	Organização do Trabalho	25
2	FUNDAMENTAÇÃO TEÓRICA	27
2.1	Mercado Financeiro e Bolsa de Valores	27
2.1.1	<i>Contratos Futuros e Tulipas</i>	28
2.1.2	<i>Mercado Financeiro Moderno</i>	29
2.1.3	<i>Hipótese do Mercado Eficiente</i>	30
2.2	Opções	31
2.2.1	<i>Especulando com Opções</i>	31
2.2.2	<i>Venda de opções descobertas</i>	33
2.3	Movimento Browniano do preço de ações	33
2.3.1	<i>Distribuição Normal</i>	35
2.4	Modelo de Black-Scholes-Merton	35
2.4.1	<i>Fórmula de precificação</i>	36
2.4.2	<i>Equações de Black-Scholes</i>	37
2.4.3	<i>Volatilidade</i>	38
2.5	Métodos Monte Carlo	38
2.5.1	<i>Áreas de aplicação para métodos Monte Carlo</i>	40
2.5.2	<i>Monte Carlo no Mercado Financeiro</i>	41
2.6	Geradores de Números Aleatórios	42
2.6.1	<i>Testes de qualidade para PRNG</i>	43
2.6.2	<i>Geradores LFSR e Xorshift</i>	46
2.6.3	<i>PRGN Mersenne Twister</i>	48
2.6.4	<i>PRNG WELL</i>	48
2.6.4.1	<i>Strutura do gerador WELL</i>	49
2.7	Transformada Box-Muller	49
2.8	Sistemas Computacionais Heterogêneos	50
2.8.1	<i>Verilog e VHDL</i>	52
2.8.2	<i>System Verilog</i>	52
2.8.3	<i>Bluespec System Verilog</i>	53

2.8.4	<i>OpenCL</i>	53
2.9	FPGAs	55
2.10	Coprojeto Hardware/Software	56
2.11	Trabalhos Relacionados	56
2.11.1	<i>Exemplo de implementação da Altera</i>	57
3	MATERIAL E RECURSOS	59
3.1	Placas FPGA de desenvolvimento	59
3.2	IDE Quartus II e Nios II	60
3.3	Intel FPGA SDK para OpenCL	61
3.4	Ferramentas de <i>profile</i> e desempenho	62
3.5	Biblioteca TestU01 para Testes Empíricos de PRNGs	62
3.6	MetaTrader e MQL5	63
3.6.1	<i>Script de Negociação Automatizada</i>	64
4	IMPLEMENTAÇÕES	67
4.1	Implementações - Fase 1	67
4.1.1	<i>Algoritmo Black-Scholes com Monte Carlo</i>	67
4.1.2	<i>Passo a passo do algoritmo</i>	68
4.1.3	<i>PRNGs em Hardware com Bluespec e Nios II</i>	69
4.1.3.1	<i>Implementação em Bluespec System Verilog</i>	69
4.1.4	<i>Testes de validação dos PRNGs</i>	71
4.1.5	<i>Nios II e instrução customizada</i>	72
4.1.6	<i>Componente Box-Muller e Funções de Ponto Fixo</i>	74
4.2	Implementações - Fase 2	75
4.2.1	<i>PRNGs em Software</i>	75
4.2.2	<i>Implementação com OpenCL</i>	76
4.2.2.1	<i>Kernel OpenCL do PRNG</i>	76
4.2.2.2	<i>Kernel OpenCL da Transformada Box-Muller</i>	78
4.2.2.3	<i>Programa Host</i>	79
4.3	Implementações - Fase 3	79
4.3.1	<i>Interface com MetaTrader</i>	79
5	RESULTADOS	81
5.1	Geradores em Hardware	81
5.2	Instrução Customizada para Nios II	82
5.3	Black-Scholes em Software	83
5.4	Black-Scholes em OpenCL	83
6	CONCLUSÃO E TRABALHOS FUTUROS	87
6.1	Fase 1	87

6.2 Fase 2 88

6.3 Fase 3 88

REFERÊNCIAS 89

INTRODUÇÃO

1.1 Contextualização e Motivação

O mercado financeiro é tradicionalmente considerado uma das atividades mais competitivas exercidas pela humanidade. A imagem popular de homens implacáveis, negociações arriscadas, fortunas ganhas e perdidas em um piscar de olhos, etc. não é infundada. Nesse ambiente, não é de se espantar que cada um dos participantes esteja sempre em busca de uma vantagem extra sobre seus colegas de mercado financeiro.

Essa busca incessante para estar sempre na frente faz com que o mercado financeiro busque o melhor que as ciências e a tecnologia podem oferecer. É nesse cenário que as equações desenvolvidas por Fischer Black e Myron Scholes nos anos 70 abrem a oportunidade para computação de alto desempenho - High Performance Computing (HPC) - e aceleração por hardware.

As equações de Black-Scholes foram um marco para os estudos sobre mercado financeiro, pois se tratava do primeiro trabalho com sustentação matemática rigorosa para a solução do problema da precificação de opções e, posteriormente outros derivativos. Esse era um problema que vinha eludindo matemáticos e economista no mundo todo por mais de 200 anos. Antes de sua publicação as opções eram negociadas base da oferta e procura e o preço determinado, em grande parte, no instinto de cada negociante ([SZPIRO, 2011](#)). Mais detalhes sobre o mercado de opções e as equações de Black-Scholes na próxima seção desta dissertação.

O trabalho de Fisher Black e Myron Scholes teve um grande impacto nos mercados financeiros do mundo todo. Nas décadas seguintes grandes corretores e bancos de Wall Street passaram a contratar cada vez mais especialistas do mundo acadêmicos como matemáticos, estatísticos e até físicos nucleares. Enquanto isso, a adoção de negociações eletrônicas e informatização das bolsas de negociação seguiu em paralelo. Com o passar dos anos, a capacidade de processamento e a velocidade com a qual esses sistemas eletrônicos conseguiam negociar tornou

o *trader* "de carne e osso" obsoleto. (PATTERSON, 2010)

Além dos matemáticos e físicos nucleares Wall Street, passou também a depender de um esquadrão de programadores para codificar os modelos matemáticos em computadores de última geração, cada vez mais poderosos. Apesar de ganharem mais do que programadores em outros setores, em Wall Street os programadores estão na parte de baixo da hierarquia e muitas vezes os detalhes dos sistemas e modelos matemáticos por trás dos códigos lhes é desconhecido. O foco do investimento costuma estar em computadores mais poderosos e com ênfase em código facilmente escalável. (WEATHERALL, 2013)

Chegando na segunda década do século 21, o cenário para computação de alto desempenho passou a ser dominado por arquiteturas híbridas com Unidades de Processamento Gráfico (GPU), que oferecem grande capacidade de processamento vetorial, ideais para simulações estocásticas e Monte Carlo, como será detalhado nas próximas seções. Para conseguir o melhor desempenho possível desses sistemas os programadores precisam estruturar os códigos para alimentar as GPUs com a maior quantidade possível de dados. Atualmente as APIs mais populares para programação com sistemas híbridos são *Compute Unified Device Architecture (CUDA)* e *Open Computing Language (OpenCL)*.

CUDA é uma *Application Programming Interface (API)* proprietária e só pode ser utilizada com GPUs fabricadas pela Nvidia. OpenCL, por outro lado, é uma API aberta suportada por diferentes fabricantes (incluindo NVidia) e pode ser utilizada em diferentes famílias de dispositivos, incluindo *Central Processing Unit (CPU)* e *Field Programable Gate Arrays (FPGAs)*.

O nicho de mercado das FPGAs nas últimas décadas ficou restrito a sistemas embarcados e sistemas de telecomunicação como roteadores. Tais sistemas exploravam as vantagens das FPGAs em baixo consumo elétrico e grande capacidade de processamento paralelo (throughput) operando em frequências baixas. Nos últimos anos, entretanto, fabricantes tem olhado para as possibilidades de FPGAs em sistemas híbridos para HPC. Vários artigos citam condições em que algoritmos têm melhor performance ou melhor eficiência energética executando em FPGAs ao invés de CPUs ou GPUs.

1.2 Objetivos

O objetivo deste projeto é desenvolver e avaliar um coprojeto hardware/software, em FPGA, para solução das equações Black-Scholes pelo método de Monte Carlo. Essas equações podem ser utilizadas para precificação de "opções" em diferentes mercados financeiros. O sistema desenvolvido deve ser capaz de realizar uma grande quantidade de simulações de forma eficiente e confiável.

1.3 Organização do Trabalho

Capítulo 2 - Fundamentação Teórica: Nesse capítulo será primeiro apresentada a teoria base das equações de Black-Scholes e o funcionamento do mercado financeiro. Serão descritos os tipos de negociações com opções e derivativos. Em seguida serão detalhados o funcionamento dos métodos Monte Carlo e seus componentes, em especial os geradores de números aleatórios. Finalizando o capítulo, são apresentadas as principais tecnologias necessárias para a implementação das equações Black-Scholes com o máximo de desempenho e eficiência oferecidos pelas FPGAs disponíveis para este projeto.

Capítulo 3 – Material e Recursos: Nesse capítulo são apresentadas as ferramentas e tecnologias utilizadas na implementação deste projeto.

Capítulo 4 – Implementação: Nesse capítulo temos os detalhes de como o sistema foi implementado em suas diversas fases.

Capítulo 5 – Resultados:

Capítulo 6 – Conclusões e Trabalhos Futuros:

FUNDAMENTAÇÃO TEÓRICA

2.1 Mercado Financeiro e Bolsa de Valores

O mercado financeiro é um termo geral que descreve qualquer mercado em que ocorra a negociação de títulos, incluindo ações, títulos, moedas e derivativos. Embora alguns mercados financeiros sejam muito pequenos e com pouca atividade, outros, incluindo a Bolsa de Valores de Nova York (NYSE) e os mercados cambiais (Forex), negociam trilhões de dólares de títulos diariamente.

Dependendo da interpretação dos registros arqueológicos é possível considerar que os primeiros mercados financeiros surgiram logo depois da formação das primeiras cidades e mercados agrícolas da antiga mesopotâmia. A própria escrita cuneiforme dessa época surgiu como forma de registrar a compra, venda e doação de jarros de trigo e cevada para templos em oferta aos deuses da região. Entretanto a visão mais convencional costuma colocar o ponto de partida dos mercados financeiros, como conhecemos hoje em dia, com as grandes navegações europeias dos séculos 16 e 17 e a proliferação do mercantilismo nos estados europeus.

Entre as bolsas europeias dos séculos 16 e 17, um especial destaque pode ser dado à bolsa de valores de Amsterdã. Fundada em 1602 pela Companhia das Índias Orientais (*Verenigde Oostindische Compagnie*, ou "**VOC**"), inicialmente para negociar apenas ações da empresa. Como os Países Baixos, na época eram um aglomerado de pequenos condados e baronatos, os comerciantes holandeses e flamencos não contavam com um estado forte o suficiente para arcar com os custos das arriscadas travessias do Atlântico. A venda direta dessas ações servia para financiar as expedições marítimas e o comércio de especiarias.

Toda a riqueza trazida pelo comércio de especiarias fez de Amsterdã uma das cidades mais prósperas e dinâmicas de toda Europa. Em meados do século 17, comerciantes, fazendeiros, funcionários públicos, artistas e até padres frequentavam os diversos “clubes de negociação”. Nesses era possível negociar ações da **VOC** em um mercado secundário, além de ações de outras

companhias e contratos futuros de especiarias e outros produtos exóticos - entre esses produtos, as tulipas.

2.1.1 *Contratos Futuros e Tulipas*

Contratos futuros são contratos firmados por duas partes para realizar uma transação no futuro por um valor pré definido. Por exemplo, suponha que um padeiro tenha sido contratado para realizar uma grande festa de casamento da filha do Barão local. O casamento vai ser realizado só daqui a três meses. Como o padeiro não pode arriscar ficar sem farinha para o bolo e brioques na data da festa e ele não possui o local ideal para armazenar toda essa farinha por tanto tempo, ele tem que negociar com o fornecedor de farinha a entrega da quantidade desejada. O problema é que o Barão ainda não pagou o padeiro, e este não tem o dinheiro para pagar adiantado pela farinha. Então, o padeiro e fornecedor devem firmar um contrato futuro em que um se compromete a pagar um valor fechado por uma quantidade fechada de farinha que o fornecedor se compromete entregar numa data próxima do casamento da filha do Barão.

Contratos futuros possibilitam a proteção de comerciantes e produtores contra variações bruscas nos preços. Mas também oferecem uma oportunidade para especuladores lucrarem sem terem que produzir nada de valor. No exemplo anterior, se algum especulador souber do casamento antes do padeiro e antever a demanda por farinha, ele pode atuar no mercado assegurando para si contratos futuros a um preço de comprar menor do que padeiro iria conseguir, já que a notícia da festa teria espalhado e o preço futuro subido. O especulador pode então vender esses contratos de volta para o padeiro por um bônus.

Especular com contratos futuros por si só apresenta um risco muito grande, já que uma das partes pode simplesmente não ter condições de honrar o contrato na data acertada (a festa de casamento pode ser cancelada ou o armazém de farinha pegar fogo). Por essa razão, um contrato futuro costuma exigir que as partes deixem algo de valor como garantia para o caso de uma delas deixar de cumprir sua obrigação. Nos Países Baixos do século 17, além de casas, lojas e navios, as ações da **VOC** também eram usadas como garantias nesses contratos.

Por volta da terceira década daquele século, a economia holandesa estava fortemente entrelaçada ao mercado financeiro e as negociações nas bolsas e clubes afetavam diretamente a vida de milhares de seus habitantes. Toda essa prosperidade foi brevemente interrompida quando a especulação com contratos futuros de bulbos de tulipas fez seus preços explodirem em menos de um ano. Entretanto, quando a data de vencimento daquele ano chegou, praticamente nenhum comprador tinha o dinheiro suficiente para honrar os contratos. O efeito cascata na economia foi devastador: propriedades foram abandonadas e lojas fechadas, desvalorização da moeda e hiper-inflação se seguiram. As ações da **VOC** se desvalorizaram tanto que a companhia não tinha mais dinheiro no caixa para pagar funcionários.

Enquanto a crise de desenrolava, os fornecedores de bulbos de tulipas entraram na justiça

Figura 1 – *Semper Augustus*, o bulbo mais famoso de tulipa, que foi vendido por preço recorde

de Amsterdã, no Tribunal dos Anciões, exigindo que eles fossem compensados pelo prejuízo e até que os especuladores fossem presos. Os anciões atenderam partes dessas demandas ao impor uma multa referente a uma fração do preço total do contrato para quem deixasse de honrar um contrato futuro de compra. Essa medida aliviou a crise das tulipas, mas não acabou com as especulações nos mercados futuros, pois, sem saber, os anciões de Amsterdã criaram os primeiros contratos de **opções**.

2.1.2 Mercado Financeiro Moderno

A bolsa de valores de Amsterdã e seus clubes de negociação serviram de base para as bolsas europeias que surgiram nos séculos 17 ao 19. Na segunda metade do século 19 as maiores e mais importantes bolsas de valores no mundo eram a de Paris e a de Londres. A supremacia europeia se manteve até as primeiras décadas do século 20, quando o centro financeiro mundial se deslocou para os Estados Unidos da América, em especial para Nova York¹.

Ainda no século 19, a "mecânica" por trás da flutuação de preços começa a despertar o interesse de matemáticos e cientistas europeus. Entretanto o interesse do mundo acadêmico não estava focado em assuntos "mundanos", como ajudar negociantes a ficarem ainda mais ricos. Nesse período as estratégias de investimento eram difundidas oralmente ou por livretos com dicas e artimanhas para os investidores nas bolsas. Um dos primeiros a examinar o mercado financeiro de modo científico foi Louis Bachelier. Entre as suas conclusões sobre o mercado financeiro, com foco na bolsa de Paris está que: "*A expectativa de lucro de um especulador na bolsa de valores é em média zero*". Essa conclusão deriva da observação de que os preços dos ativos negociados variavam aleatoriamente sem uma tendência clara de alta ou de queda. O trabalho de Bachelier permaneceu pouco apreciado até por volta da década de 60 quando passou a ser uma das bases da doutrina da "Hipótese do Mercado Eficiente" entre os economistas dos EUA.

Do começo do século 20 até hoje em dia, o mercado financeiro dos EUA é indiscutível-

¹ Nova York foi fundada por holandeses com o nome de Nova Amsterdã

mente o maior e mais importante no mundo ². Os EUA também lideram o mundo na adoção de novas tecnologias para facilitar cada vez mais as transações financeiras. O processo de informatização das bolsas de valores por negociações eletrônicas teve início na década de 80 em Nova York e hoje é predominante nas principais bolsas do mundo. Negociações eletrônicas permitem que investidores ou especuladores possam negociar de sua casa ou escritório com o *click* do mouse ou toque de tela do celular.

O advento de negociações eletrônicas possibilitou o desenvolvimento de uma indústria paralela de negociações automatizadas por meio de programas de computador conectados à interface das corretoras. Esses programas são chamados de "robôs" e suas operações são chamadas "negociação algorítmica". Esses robôs operam em um ambiente altamente competitivo em que cada microssegundo de vantagem sobre um robô rival pode significar ganhos vultuosos para seus donos. As corretoras e bancos de investimento tem feito grandes investimentos em hardware de última geração, além de buscar nas universidades os matemáticos e programadores mais talentosos.

2.1.3 Hipótese do Mercado Eficiente

A Hipótese do Mercado Eficiente (HME) originou-se na década de 1960 e graças ao trabalho do economista Eugene Fama. Essa hipótese sustenta que é impossível vencer o mercado, já que os preços no mercado já incorporam e refletem todas as informações relevantes que podem impactar o preço de um ativo ou ação. Ou seja, a expectativa de ganho de um investidor é, na média, igual à média dos ganhos do mercado como um todo. Essa teoria levantou muitas controvérsias desde seu surgimento, assim como apoiadores e entusiastas, com discussões até hoje (DELCEY, 2017). Aqueles que acreditam na HME afirmam que é essencialmente inútil procurar ações subvalorizadas para tentar fazer previsões de tendências de mercado usando ferramentas como análise fundamental ou análise técnica.

Por essa hipótese, toda vez que um investidor compra ou vende um título, ele ou ela participa de um jogo de azar, em vez de um de habilidade. Um mercado que seja eficiente e atualizado sempre refletirá o preço mais preciso, de modo que nunca se pode comprar uma ação a preço menor que o seu valor real, ou vender ao um preço maior.

Analistas técnicos contestam amplamente essa teoria. Eles argumentam que muitos investidores baseiam suas expectativas de desempenho de ações em preços passados, lucros, *track records* e outros indicadores históricos. Os preços das ações são em grande parte baseados na expectativa do investidor, o que significa que os preços das ações e a atividade no passado indiretamente impactam tanto os preços atuais e quanto a perspectiva de preços futuros. Os analistas técnicos acreditam essencialmente no oposto daqueles que apoiam a hipótese do mercado eficiente: que as ações são frequentemente desvalorizadas no mercado, e que pesquisa

² <<http://money.visualcapitalist.com/all-of-the-worlds-stock-exchanges-by-size/>>

suficiente e tomadas de decisão sábias podem fornecer aos investidores uma vantagem sobre a concorrência.

Uma outra crítica é que grande parte dos trabalhos acadêmicos que serviram de base à HME ignoram os efeitos de eventos imprevisíveis no comportamento do mercado, como a falência de uma empresa, novas tecnologias, resultados de eleições ou mesmo guerras e ataques terroristas. O argumento é que os dados empíricos mostram que os mercados financeiros estão muito mais sujeitos a variações bruscas, tanto em crises quanto em bolhas, do que a teoria do HME estabelece. (TALEB, 2013; TALEB, 2014)

2.2 Opções

Em mercados financeiros o termo opção - plural opções – refere-se a um contrato de transação em uma data futura em que uma das partes recebe o direito, mas não a obrigação, de comprar ou vender um ativo para a outra parte.

As opções são classificadas em dois grandes grupos de acordo com a possibilidade ou não de serem exercidos os direitos do contrato antes da data de exercício. Se a opção só pode ser exercida na data de vencimento, é uma opção europeia. Se pode ser exercida a qualquer momento antes da data de vencimento, é uma opção americana. No Brasil, na BM&F (Bolsa de Mercadorias e Futuro) e Bovespa, a maioria dos contratos de opções de compra enquadram-se no tipo americano, enquanto as opções de venda são do tipo europeu ³.

Uma opção de compra dá ao seu titular o direito de comprar o objeto da opção, ao preço de exercício, a qualquer instante, até a data de vencimento. Após essa data, se não for exercida, a opção deixa de existir (o que no linguajar dos operadores de mercado é “virar pó”).

O lançador de uma opção de compra é aquele que a vende, assumindo a obrigação de vender o objeto a que se refere a opção, caso ela seja exercida. Ele entregará a totalidade do objeto mediante o recebimento do preço do exercício. Uma opção é exercida por um preço previamente acertado, ou seja, o preço de exercício. Uma opção que se torna lucrativa, se for exercida ao preço corrente do ativo, é chamada no-dinheiro ou dentro-do-dinheiro (*in-the-money*). Inversamente, se não for lucrativa, é chamada fora-do-dinheiro (*out-of-the-money*).

2.2.1 Especulando com Opções

No mercado de ações, os papéis são vendidos em lotes com quantidade mínima de ações - geralmente 100 ações por negócio. Se um especulador pretende ganhar com a valorização da ação, ele tem que fazer um investimento mínimo de $100 * P$ - onde P é o preço da ação na compra. O retorno do investimento - ignorando custos de corretagem - na venda do lote vai ser totalmente atrelado ao valor final. Por exemplo:

³ <http://www.bmfbovespa.com.br/pt_br/servicos/market-data/consultas/mercado-a-vista/opcoes-1>

Supondo que a ação esteja valendo \$20.00 no momento da compra, o investidor precisa desembolsar \$2000.00. Se no momento da venda o título tiver se valorizado 10.00% ele vai ter um ganho de \$200.00, exatamente 10% do valor investido.

Uma das vantagens oferecidas ao se negociar opções é a possibilidade de se especular com a movimentação dos preços do ativo de forma alavancada. Usando opções um especular pode movimentar a mesma quantidade de ações do exemplo anterior por uma fração do valor, e ainda ter uma chance de retorno muito maior.

Seguindo o exemplo anterior uma ação que esteja valendo \$20.00 pode ter uma opção de compra com preço alvo de \$21.00 e vencimento em 30 dias valendo \$0.20. Nesse caso, o investimento inicial mínimo não passa de \$20.00. Caso a ação esteja valendo \$22.00 na data do vencimento da opção, o especulador pode executar o contrato pagando \$2100.00 e imediatamente vendendo o lote por \$2200.00, deste modo, ele obteve um ganho de \$100.00 depois de um investimento inicial de \$20.00, ou seja um retorno de 500.00%.

A [Figura 2](#) mostra as curvas de ganho para opções de compra e venda em função do valor do ativo na data de vencimento. É possível ver que em opções de compra o investidor não recebe nada enquanto o valor do ativo estiver abaixo do valor alvo. Se o valor estiver acima o investidor ganha a diferença entre o valor alvo e o preço atual do ativo. O limite de perdas é para o caso (pouco provável) de o ativo subjacente chegar a zero, que acontece para ações no caso da companhia ir a falência, ser removida da bolsa de valores e não poder recomprar as suas ações.

Figura 2 – Retorno de opções de compra (amarelo) e venda (azul) em função do valor do ativo(vermelho)

A especulação com opções tem um risco associado maior do que a especulação com ações, pois no vencimento das opções a possibilidade do preço alvo não ser atingido é consideravelmente maior do que a probabilidade de uma ação valer zero no curto prazo.

2.2.2 Venda de opções descobertas

Na sessão anterior o conceito de opções foi discutido do ponto de vista do comprador de uma opção - a pessoa que recebe o direito. Mas é lógico que para se ter uma compra é preciso ter um vendedor. O vendedor é também chamado de emissor, pois ele é o “criador” do papel e recebe um prêmio pela venda. Ao vender uma opção de compra o emissor tem a obrigação de vender o ativo pelo preço do contrato para o portador da opção. E vice-versa para uma opção de venda.

O emissor de uma opção geralmente precisa de privilégio para realizar esse tipo de operação em uma corretora, e está sujeito às regras da corretora e da bolsa em que os papéis vão ser negociados. Uma das regras costuma ser que o emissor precisa possuir uma quantidade mínima do ativo em carteira ao lançar uma opção de compra e possuir uma quantidade mínima de dinheiro ao emitir uma opção de venda. Essas regras existem para proteger as corretoras pois o emissor de opções está exposto a perdas ilimitadas, no caso de opções de compra. Caso seja permitido um especulador emitir opções de compra sem ter o ativo subjacente, essas opções são chamadas de opções descobertas (*naked calls*). Como a variação do valor de um ativo segue uma distribuição probabilística a questão é como estimar o prêmio que o emissor deveria pedir para compensar os riscos da operação.

A resposta foi dada por Black e Scholes, e também por Merton em seus artigos sobre a teoria de precificação de opções publicada em 1973. A ideia é que um emissor de uma opção de compra descoberta pode proteger sua posição vendida comprando uma certa quantia do ativo de modo que a perda na posição vendida pode ser exatamente compensada pela posição longa. Isso é padrão na cobertura. A questão é: "quantas ações ele deve comprar para minimizar o risco?" Ao ajustar a proporção entre ações e opções continuamente na carteira durante a vida da opção, Black e Scholes demonstraram que os investidores podem criar uma carteira de *hedge*, ou seja: sem risco, onde todos os riscos de mercado são eliminados. Em um mercado eficiente, sem oportunidades de arbitragem sem risco, qualquer carteira com risco zero de mercado deve ter uma taxa de retorno esperada igual à taxa de juros livre de risco. A formulação de Black-Scholes estabelece a condição de equilíbrio entre o retorno esperado da opção, o retorno esperado da ação e a taxa de juros livre de risco (BLACK; SCHOLES, 1973).

2.3 Movimento Browniano do preço de ações

O Movimento Browniano foi descrito pela primeira vez em 1827 por Robert Brown - e nomeado posteriormente em sua homenagem - como o movimento aleatório de partículas suspensas em um líquido. A razão do movimento das partículas em suspensão permaneceu um mistério até 1905, quando Albert Einstein aplicou ferramentas estatísticas para descrever o Movimento Browniano, esse trabalho ajudou a comprovar a teoria de que toda a matéria seria composta de átomos discretos, formalizada um século antes pelo químico inglês John Dalton.

Em 1900, Louis Bachelier, já tinha indicado em seu doutorado que os preços das ações seguiam movimentos aleatórios em passos discretos. As equações usadas para descrever essas variações eram semelhantes as usadas posteriormente por Einstein. Entretanto, como o seu trabalho foi pouco divulgado na época, e é mais provável que Einstein nunca o tenha lido. (SZPIRO, 2011)

Na Equação 2.1, a probabilidade do preço atual S_n depende do preço anterior S_{n-1} multiplicado pela amplitude de variação. Por esse modelo, o preço de uma ação varia no tempo em Movimento Browniano discreto com amplitude seguindo distribuição log-normal. A distribuição log-normal é consequência da variação ter distribuição normal e pelo fato de que o preço de uma ação não pode ter valores negativos.

$$S_n = S_{n-1} e^{((r-\mu^2/2)+\mu\sqrt{(\delta)})} \quad (2.1)$$

Na Equação 2.1 δ é uma distribuição normal com média zero e μ é a volatilidade da ação. Com essa função é possível gerar um conjunto de curvas no gráfico de preços, em que cada uma delas representa um caminho possível na variação dos preços da ação.

Na Figura 3 o preço da ação (representado no eixo Y) é US\$34.81 no dia 0. Sendo o tempo em dias representado no eixo X. Após o dia 0, todos os caminhos simulados são igualmente prováveis, com o conjunto final de preços futuros agrupados em uma distribuição log-normal.

Figura 3 – Distribuição do preço de uma ação ao longo do tempo, simulada por movimentos brownianos

2.3.1 Distribuição Normal

Uma das principais suposições utilizadas para se derivar as equações de Black-Scholes é que os preços de um ativo variam de forma aleatória seguindo uma distribuição gaussiana normal. Tal comportamento é chamado de caminhada aleatória ou “Andar do Bêbado”, e é uma propriedade emergente em muitos sistemas naturais como o movimento de partículas suspensas em um líquido (Movimento Browniano).

Figura 4 – Curvas de Distribuição Normal

A Figura 4 mostra quatro distribuições normais, três delas com média μ zero. A variância σ^2 descreve o quão concentrados próximo da média os valores tendem a ser encontrados.

Uma distribuição normal na variação de preços indica que quanto maior a amplitude da variação - para mais ou para menos - menos provável é a sua ocorrência, resultando em uma concentração em torno do zero ($\mu = 0$). σ^2 é considerado a volatilidade do ativo. Esse comportamento foi descrito pela primeira vez por Louis Bachelier em 1900 na sua tese de doutorado na Universidade de Paris.

2.4 Modelo de Black-Scholes-Merton

O modelo de Black-Scholes-Merton, também conhecido como Black-Scholes, é um modelo matemático de mercados de derivativos financeiros dos quais a fórmula de Black-Scholes pode ser derivada. Esta fórmula estima os preços das opções de compra e venda do tipo europeia. Os investidores e especuladores do mercado financeiro norte americano foram rápidos em adotar

o modelo para a precificação de opções e operações de cobertura (*hedge*). O modelo foi um dos responsáveis pelo aumento significativo na negociação de opções nos financeiros modernos, a partir dos anos 70. Antes da invenção dessa fórmula e modelo, os operadores de opções não usavam todos uma maneira matemática consistente para avaliar as opções, e a análise empírica mostrou que as estimativas de preço produzidas por essa fórmula estão próximas dos preços observados.

Inicialmente, Fischer Black e Myron Scholes (os economistas que originalmente formularam o modelo) apresentaram uma equação diferencial parcial conhecida como a equação de Black-Scholes. Eles se basearam na teoria do portfólio, que descreve a relação de preços entre um ativo e os seus derivativos.(BLACK; SCHOLES, 1973) Mais tarde Robert Merton comprovou a validade das equações usando cálculo estocástico e dados empíricos. A fórmula completa da precificação também é conhecida como "equações Black-Scholes-Merton". Myron Scholes e Robert Merton dividiram o Prêmio Nobel de Economia em 1997. Fischer Black falecera antes de ser premiado.

De modo simplificado, seu modelo determina o preço de uma opção, calculando o retorno obtido pelo investidor, menos o valor que o investidor deve pagar, usando probabilidades de distribuição log-normal para contabilizar a volatilidade no ativo subjacente. A distribuição log-normal dos retornos usada no modelo é baseada nas teorias do movimento browniano, com os preços dos ativos exibindo um comportamento similar ao movimento natural no movimento browniano.

A fórmula ajudou a legitimar o comércio de opções, tornando-o menos parecido com um jogo de azar e mais com uma ciência. Hoje, a fórmula Black-Scholes-Merton é amplamente utilizada, com algumas modificações dependendo do mercado, por *traders* e investidores. Já que esses *traders* utilizam a compra e emissão de opções como forma fundamental de *hedge* para controlar melhor os riscos associados à volatilidade do mercado de ações em operações mais sofisticadas.

2.4.1 Fórmula de precificação

Na publicação original em que Black Fisher e Myrion Scholes apresentaram o seu modelo são consideradas as "condições ideais" para a precificação das opções de um ativo em particular:

- A taxa de juros de curto prazo é conhecida e constante ao longo do tempo.
- O preço do ativo segue uma caminhada aleatória contínua no tempo com uma variação proporcional ao quadrado do preço do ativo. Assim, a distribuição possível dos preços do ativo no final de qualquer intervalo finito segue uma distribuição log-normal. A variância no retorno do ativo é constante.
- O ativo não paga dividendos ou outros bônus.

- A opção é do tipo Europeu.
- Não existem custos de transações para compra ou venda do ativo ou da opção.
- É possível comprar, vender ou manter qualquer subdivisão de uma garantia de empréstimo na taxa de juros de curto prazo.

Sob essas suposições, o valor da opção vai depender apenas no preço do ativo e das variáveis que são assumidas constantes. Dessa forma, é possível montar uma posição de *hedge* composta de comprado no ativo e vendido na opção, cujo valor já é derivado da próprio ativo. Como ultimo parâmetro para a expectativa de retorno nessa posição é considerado o custo de oportunidade em investimentos considerados "livres de risco", como os títulos do tesouro americano (BLACK; SCHOLES, 1973).

A partir desse portfólio "livre de risco", foi possível derivar as equações em que os parâmetros corretos poderiam facilmente precificar opções nos mais diferentes mercados. Com as contribuições posteriores de Robert C. Merton, além de refinar as bases matemáticas, mostrou que as opções de fato eram precificadas aproximadamente como o modelo de Black-Scholes previa. (MERTON, 1973).

2.4.2 Equações de Black-Scholes

Para opções do tipo europeu a fórmula de Black-Scholes-Merton é representada pelo seguinte sistema de equações diferenciais:

$$\frac{\delta V}{\delta t} + \frac{1}{2}\sigma^2 S^2 \frac{\delta^2 V}{\delta^2 S^2} + rS \frac{\delta V}{\delta S} - rV = 0 \quad (2.2)$$

Esse sistema pode ser resolvido numericamente, considerando as condições de contorno, pelas equações a seguir.

$$C(S, t) = N(d_1)S - N(d_2)Ke^{-rt} \quad (2.3)$$

$$d_1 = \frac{1}{\sigma\sqrt{t}} \left[\ln\left(\frac{S}{K}\right) + t\left(r + \frac{\sigma^2}{2}\right) \right] \quad (2.4)$$

$$d_2 = \frac{1}{\sigma\sqrt{t}} \left[\ln\left(\frac{S}{K}\right) + t\left(r - \frac{\sigma^2}{2}\right) \right] \quad (2.5)$$

$$N(x) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^x e^{-\frac{1}{2}z^2} dz \quad (2.6)$$

Sendo:

C = Preço alvo da opção

S = Preço atual do ativo

K = Preço de vencimento do ativo

r = taxa de juros livre de risco (entre 0 e 1)

σ = volatilidade implícita do ativo (entre 0 e 1)

t = tempo para o vencimento da opções - em anos

N = funções de distribuição cumulativa para uma distribuição aleatória normal

2.4.3 Volatilidade

Entre os parâmetros das equações de Black-Scholes a Volatilidade σ (sigma), tem especial destaque pois é o único que não está disponível imediatamente no momento do calculo das equações, trata-se de uma medida futura, do momento da compra da opção até o seu vencimento.

A volatilidade é a medida da amplitude e velocidade da variação do preço de um ativo financeiro. Ela é um parâmetro de instabilidade e incerteza no mercado. Quanto maior a volatilidade menor é a certeza dos operadores de mercado quanto ao valor “justo” do ativo. Isso faz com que a volatilidade também esteja associada ao risco de uma operação futura ser bem sucedida ou não. O valor de uma opção é maior o quão maior seja a volatilidade do ativo subjacente, pois a possibilidade de a opção vencer “dentro do dinheiro ” é maior.

Antes da fórmula Black-Scholes-Merton, os investidores tinham suas próprias estratégias de estimar o preço das opções. Esses métodos eram, geralmente, intuitivos ou de oferta e demanda, mesmo assim os resultados empíricos se aproximavam do esperado pelo modelo. Ou seja, eles usavam a expectativa de volatilidade futura baseando em dados históricos e eventos sazonais. A lógica é de que a volatilidade de um ativo costuma ser relativamente e previsível, sendo afetada eventualmente por sazonais como o clima ao longo do ano, divulgação de relatórios financeiros ou datas de calendário como o Natal, Eleições ou estação de colheita agrícola.

2.5 Métodos Monte Carlo

Métodos Monte Carlos são formas de solucionar equações com componentes estocásticos ou o uso de simulações com variáveis aleatórias para a resolução numérica. Essas equações podem surgir “naturalmente” como parte da modelagem de um sistema físico, como uma complexa rede de estradas, o fluxo de nêutrons em uma explosão nuclear ou a evolução dos preços das ações ao longo de um período de tempo. Em muitos casos, entretanto, os objetos aleatórios em métodos Monte Carlo são introduzidos “artificialmente” para resolver problemas

puramente determinísticos. Nesse caso, o método Monte Carlo envolve diretamente a amostragem aleatória das distribuições de probabilidade de ocorrência de cada um dos eventos relacionados a uma variável.

No cenário natural ou artificial das técnicas de Monte Carlo, o conceito central é repetir o experimento muitas vezes (ou executar uma simulação suficientemente longo) para obter muitas quantidades de dados úteis (KROESE *et al.*, 2014). refere-se ao tratamento de um problema real através da reprodução em um ambiente controlado pelo experimentador – ou seja, uma simulação. A simulação é frequentemente fornecida por equipamentos computacionais, embora possa ser uma reprodução em menor escala, ou simplificada, do sistema sob estudo.

Existem sistemas que alguns ou todos os seus componentes estão sujeitos a flutuação aleatória. Eles não podem ser descritos por uma regra matemática exata, mas apenas através do uso de afirmações probabilísticas (GAMERMAN; LOPES, 2006)). Já que o desenvolvedor escolhe simular esses sistemas, haverá um componente de incerteza. Imagine que um fosse jogar uma moeda 100 vezes, esperaria acabar com 50 caras e 50 coroas. Esse resultado é senso comum e o próprio fundamento da Teoria das Probabilidades. No entanto, não se pode prever como os processos se desdobrarão ou qual a distribuição exata depois de um número determinado de jogadas. Sequências de 2 ou 10 repetições também tem uma distribuição aleatória independente.

Do ponto de vista estatístico, esses sistemas podem ser considerados como uma função (possivelmente complicada) de variáveis aleatórias. O objetivo da simulação é reproduzir essas variáveis aleatórias em um ambiente sob nosso controle, independentemente da estrutura complexa que relaciona essas variáveis. Normalmente, um computador é usado para este exercício de reprodução (GAMERMAN; LOPES, 2006). A categoria de problemas que os métodos de Monte Carlo podem ser empregados são:

- Amostragem: Aqui, o objetivo é coletar informações sobre um objeto aleatório, observando muitas realizações dele. Um exemplo é a modelagem de simulação, em que um processo aleatório imita o comportamento de algum sistema da vida real, como uma linha de produção ou uma rede de telecomunicações. Outro exemplo é encontrado na estatística Bayesiana, onde Monte Carlo com Cadeia de Markov (MCMC) é frequentemente usado para amostragem de uma distribuição posterior.
- Estimativa: Neste caso, a ênfase está na estimativa de certas grandezas numéricas relacionadas a um modelo de simulação. Um exemplo no cenário natural das técnicas de Monte Carlo é a estimativa do rendimento esperado em uma linha de produção. Por exemplo, a avaliação de integrais multidimensionais via técnicas de Monte Carlo, escrevendo a integral como a expectativa de uma variável aleatória.
- Otimização: Monte Carlo é uma ferramenta poderosa para a otimização de funções objetivas complexas. Em muitas aplicações, essas funções são determinísticas, e a aleatoriedade

é introduzida artificialmente para buscar com mais eficiência o domínio da função objetivo. As técnicas de Monte Carlo também são usadas para otimizar funções ruidosas, onde a própria função é aleatória - por exemplo, o resultado de uma simulação.

2.5.1 Áreas de aplicação para métodos Monte Carlo

Muitos problemas quantitativos em ciência, engenharia e finanças são hoje resolvidos através de técnicas de Monte Carlo.(KROESE *et al.*, 2014)

- Engenharia Industrial e Pesquisa Operacional: Esta é uma das principais áreas de aplicação da modelagem de simulação. Aplicações típicas envolvem a simulação de processos de inventário, agendamento de tarefas, roteamento de veículos, redes de filas e sistemas de confiabilidade.
- Processos Físicos e Estruturas: A simulação direta do processo de transporte de nêutrons foi a primeira aplicação do MCM na era moderna, e técnicas de Monte Carlo continuam sendo necessárias para a simulação de processos físicos. As técnicas de Monte Carlo agora desempenham um papel significativo na ciência dos materiais, onde são usadas no desenvolvimento e análise de novos materiais e estruturas.
- Gráficos Aleatórios e Estruturas Combinatórias: De um ponto de vista mais matemático e probabilístico, as técnicas de Monte Carlo provaram ser muito eficazes no estudo das propriedades de estruturas aleatórias e gráficos que surgem em física estatística, teoria da probabilidade e ciência da computação.
- Estatísticas computacionais: o MCM mudou drasticamente a maneira como as estatísticas são usadas na análise de dados atual. A complexidade cada vez maior de dados (“big data”) requer modelos estatísticos e técnicas de análise radicalmente diferentes daqueles usados há 20–100 anos.
- Economia e Finanças: À medida que os produtos financeiros continuam a crescer em complexidade, as técnicas de Monte Carlo se tornaram ferramentas cada vez mais importantes para analisá-las. O MCM não é usado apenas para precificar instrumentos financeiros, mas também desempenha um papel crítico na análise de risco. O uso das técnicas de Monte Carlo na precificação de opções financeiras foi popularizado no final dos anos 90. Essas técnicas são particularmente eficazes na solução de problemas envolvendo algumas fontes diferentes de incerteza (por exemplo, opções de cesta de preços, que são baseadas em uma carteira de ações).

Os métodos e simulações de Monte Carlo são profundamente dependentes das sequências de números aleatórios. No entanto, não é necessário que eles passem em todos os critérios de aleatoriedade, como os necessários para uma aplicação de criptografia. Os geradores precisam

cumprir alguns critérios de qualidade para serem considerados em uma simulação confiável de Monte Carlo. Os critérios de qualidade para os PRNGs desejados para as simulações de Monte Carlo estão listados abaixo.

- Uniformidade: A distribuição dos números aleatórios deve ser distribuída uniformemente, mas não muito regular. Uma vez que seqüências aleatórias são esperadas para produzir algumas estruturas de curto período.(KORN; KORN; KROISANDT, 2010)
- Velocidade: Simulações de Monte Carlo exigem uma grande quantidade de números aleatórios. Um bom gerador deve ser capaz de produzi-los muito rapidamente e com uso eficiente da memória.
- Longo Período: Um gerador irá eventualmente começar a repetir uma seqüência de números após alguma quantidade de iterações, isto é chamado de “Período” do gerador aleatório. Espera-se que um gerador de números aleatórios deva ter um período de pelo menos o quadrado da quantidade de números necessária .
- Independência: Para os métodos de Monte Carlo, é importante que uma seqüência aleatória pareça aleatória em vez de imprevisível. Ele precisa passar por testes estatísticos para ser independente, identicamente distribuído (i.i.d.).
- Reprodutibilidade: Primeiro, devido a razões de depuração, espera-se que uma seqüência de RN possa ser revertida se um resultado estranho aparecer. Também é comum que alguns aplicativos usem a mesma seqüência mais de uma vez.
- Portabilidade: Dada uma semente, um gerador aleatório deve dar a mesma seqüência independente do computador que está sendo executado.
- Paralelização: os computadores com vários núcleos agora são padrão, e uma implementação de gerador de números aleatórios deve aproveitar o máximo possível de núcleos de processamento com um alto nível de escalabilidade.

2.5.2 Monte Carlo no Mercado Financeiro

No mercado financeiro, a precificação de derivativos é um processo que envolve múltiplas variáveis com um volume gigantesco de possíveis caminhos a serem simulados. É suficientemente complexo que um poder computacional significativo, combinado com uma implementação correta do algoritmo, seja crítico para a exatidão e integridade dos resultados. Além disso, aproveitar ao máximo a maioria dos modelos de precificação depende da capacidade de produzir excelentes números aleatórios, algo em que os FPGAs são melhores do que os processadores convencionais (LOCKWOOD *et al.*, 2012) (ECHEVERRIA; LOPEZ-VALLEJO, 2013).

Métodos Monte-Carlo para precificação de opções podem ser eficientemente implementados em FPGAs, já que eles podem ser colocados em paralelo (CHATZIPARASKEVAS;

[BROKALAKIS; PAPAESTATHIOU, 2012](#)). Um método importante que pode aproveitar as vantagens de uma arquitetura subjacente do FPGA é o modelo Black-Scholes para a precificação de opções.

2.6 Geradores de Números Aleatórios

Os números aleatórios são úteis para uma variedade de finalidades, como gerar chaves de criptografia de dados, simular e modelar fenômenos complexos e para selecionar amostras aleatórias de conjuntos de dados maiores. Eles também têm sido usados esteticamente, por exemplo, na literatura e na música, e são úteis para jogos de inteligência artificial e jogos de azar.

Quando se discute números únicos, um número aleatório é aquele que é retirado de um conjunto de valores possíveis, cada um dos quais é igualmente provável, isto é, uma distribuição uniforme. Já em se tratando de uma sequência de números aleatórios, cada número sorteado deve ser estatisticamente independente dos outros. Um gerador de números aleatórios (Random Number Generator (RNG)) é um algoritmo computacional ou um dispositivo físico projetado para gerar uma sequência de números ou símbolos que não podem ser razoavelmente previstos. do que por uma chance aleatória.

Um RNG físico poderia ser tão simples quanto um conjunto de jogadas de dados ou uma roleta de cassino, outra solução mais sofisticada poderia utilizar a radiação eletromagnética de fundo, com raios cósmicos ou decaimento radioativo. Embora as implementações mais sofisticadas dos RNGs possam fornecer aleatoriedade real, elas não são rápidas o suficiente para qualquer modelo computacional eficiente.

Com o advento dos computadores, os cientistas reconheceram a possibilidade de simulações maiores e mais rápidas. No entanto, por mais surpreendente que pareça, é difícil fazer um computador fazer algo por acaso. Um computador segue suas instruções cegamente e, portanto, é completamente previsível. - Um computador que não segue suas instruções desta maneira está quebrado - Como os True Random Number Generators (TRNG) não são adequados para saída de dados de alta velocidade e alto volume, basicamente todos os modelos de computação usam algum tipo de gerador de números pseudoaleatórios - Pseudo-Random Number Generator (PRNG).

Em computação um PRNG pode ser modelado como uma máquina de estados. Essa máquina de estados idealizado na [Figura 5](#). Os passos são:

- Inicialização: A máquina de estados recebe como uma entrada um valor – também chamado de semente - que funciona como base para os seus estados internos da máquina.
- Transformação: Os estados internos da máquina passam por um tipo de transformação linear e um valor retornado como saída.

- Atualização: Os estados internos são atualizados e o gerador está pronto para retornar um a etapa de transformação e gerar um novo numero aleatório.

Figura 5 – Máquina de Estados de um PRNG

Uma das primeiras funções de transformação sugeridas para geração de números pseudo-aleatórios por computadores foi proposta por Von Neumann em 1946. Chamado de raiz média o método consistia em tirar a raiz quadrada de uma semente suficientemente grande e usar as casas decimais do resultado como o estado seguinte para repetir o processo [Figura 6](#). Além de exigir um processamento pesado para época e pensado para computadores decimais, esse gerador não satisfaz os parâmetros de qualidade para um PRNG([KNUTH, 1997](#)).

Figura 6 – Fluxo do algoritmo de PRNG “Middle Square”

2.6.1 Testes de qualidade para PRNG

Há uma série de testes estatísticos concebidos para distinguir os geradores de números pseudo-aleatórios dos verdadeiros; quanto mais um PRNG passa, mais próximo ele é considerado de uma verdadeira fonte aleatória, entre eles:

- Equidistribuição:
- Teste de sequências contínuas:
- Rank de matrix binária:
- Transformada Discreta de Fourier:
- Distância máxima do zero:

Esses métodos, entre outros, fazem parte das suítes de teste **TestU01** e **Diehard**. Ambas oferecem ferramentas automatizadas para testar a qualidade da sequência de números aleatórios gerada baseada na quantidade de testes que o gerador passa e nos parâmetros utilizados para cada teste. Algumas das baterias de testes como o **BigCrunch** do **TestU01** necessitam de trilhões de números aleatórios, o que faz com que o teste possa demorar dias para ser executado.

Uma forma simples de visualizar a qualidade de um PRNG é utilizar os pares de números gerados em uma distribuição unitária ($[x_1, x_2, \dots, x_{n-1}, x_n]$) como coordenadas em um espaço 2D ($[(x_1, y_1), (x_{n-1}, y_n)]$), ou trios de números em espaço 3D. Como pode ser visto nas [Figura 7](#) e [Figura 8](#) o mesmo gerador que em um espaço 2D parece mostrar uma qualidade aleatória razoável, em 3D mostra uma clara formação de estrutura. O que indica que esse gerador não é dos melhores para aplicações que exigem maior aleatoriedade.

Figura 7 – Gerador Linear de Deslocamento, 2D

Figura 8 – Gerador Linear de Deslocamento, 3D

Essas regiões que são visíveis em 2 ou 3 dimensões são características de todos os geradores de deslocamento linear. Cada deslocamento no espaço está associado com bits independentes, representando

um acréscimo de dimensão geométrica no espaço cartesiano. Os pacotes de testes citados anteriormente utilizam funções matemáticas para calcular o grau máximo de independência. Geradores com mais estados e graus de independência são considerados melhores. O PRNG Mersenne Twister possui 617 espaços de estados e como pode ser visto na [Figura 9](#) tem boa distribuição sem estruturas no espaço 3D.

Figura 9 – Mersenne Twister, 3D

Como os PRNGs são comumente usados para propósitos criptográficos, às vezes é solicitado que as funções de transformação e saída satisfaçam duas propriedades adicionais:

- Não previsibilidade: dada uma sequência de bits de saída, os bits precedentes e seguintes não devem ser previsíveis
- Não reversibilidade: dado o estado atual do PRNG, os estados anteriores não devem ser computáveis.

A regra 1 garante que a interceptação da saída de um PRNG não permita que um invasor aprenda mais do que os bits que ouviu. A regra 2 garante que as comunicações anteriores não sejam comprometidas caso um invasor consiga obter conhecimento do estado atual do PRNG (comprometendo, assim, todas as comunicações futuras com base na mesma execução). Quando um PRNG satisfaz estas duas propriedades, diz-se que é criptograficamente seguro.

No contexto deste trabalho os PRNG não precisam ser necessariamente criptograficamente seguros. Os gerados mostrados a seguir e os utilizados nos experimentos são aqueles que possuem uma quantidade significativamente grande de espaço de estados, período longo e equidistribuição bons o suficiente para Métodos Monte Carlo com grande quantidade de simulações([KORN; KORN; KROISANDT, 2010](#)). Bons PRNGs são projetados com base em análises matemáticas de suas propriedades teóricas, só então implementados e por fim submetidos à baterias de testes estatísticos empíricos.([GHERSI; PARAKH; MEZEI, 2017](#))

2.6.2 Geradores LFSR e Xorshift

Um registrador de deslocamento com feedback linear - Linear-feedback shift register (LFSR) - possui como entrada um de seus estados anteriores operados por uma função linear. Essa função linear em sistemas computacionais é geralmente a função OU EXCLUSIVO (XOR). Aplicações de LFSRs incluem geração de números pseudo-aleatórios, seqüências de pseudo-ruído, contadores digitais rápidos e seqüências de embaralhamento. Ambas as implementações de hardware e software de LFSRs são comuns.

A Figura 10 mostra a representação de um LFSR do tipo Fibonacci. As posições de bit que afetam o próximo estado são chamadas de toques (*taps*). No diagrama, os toques são [16,14,13,11]. O bit mais à direita do LFSR é chamado de bit de saída. Os toques são sequencialmente XOR'eados' com o bit de saída e depois são enviados de volta para o bit mais à esquerda. A seqüência de bits na posição mais à direita é chamada de fluxo de saída.

Figura 10 – 16-bit Fibonacci LFSR

A seqüência de números gerados por um LFSR ou seu equivalente XNOR pode ser considerada um sistema numérico binário tão válido quanto o código Gray ou o código binário natural.

Os geradores de números aleatórios Xorshift são uma classe de geradores de números pseudo-aleatórios que foram descobertos por George Marsaglia (MARSAGLIA, 1972). Eles são um subconjunto dos LFSRs que permitem uma implementação particularmente eficiente sem usar polinômios excessivamente esparsos. (BRENT, 2004) Eles geram o próximo número em sua seqüência, repetidamente, tirando o número XOR de um número com uma versão deslocada de si mesmo. Isso os torna extremamente rápidos em arquiteturas de computadores modernos. Como todos os LFSRs, os parâmetros devem ser escolhidos com muito cuidado para se obter um longo período.

O trecho de Código-fonte 1 a seguir mostra a implementação em C de uma função Xorshift com palavra de 32 bits.

Código-fonte 1 – Função Xorshift32

```
1 /* "state[0] deve ser inicializado com algum valor diferente de zero" */
2 uint32_t xorshift32(uint32_t state[static 1])
3 {
4     /* Algorithm "xor" from p. 4 of Marsaglia, "Xorshift RNGs" */
```

```

5  uint32_t x = state[0];
6  x ^= x << 13; x ^= x >> 17; x ^= x << 5;
7  state[0] = x;
8  return x;
9  }

```

Os geradores **xorshift** estão entre os geradores de números aleatórios mais rápidos, não criptograficamente seguros, possuindo código e estado muito pequenos. Embora eles não passem em todos os testes estatísticos sem mais refinamento, esta fraqueza é bem conhecida e facilmente corrigida combinando-os com uma função não linear (como apontado por Marsaglia no artigo original ([MARSAGLIA, 1972](#))). Implementações básicas desses geradores, conhecidos como **xorshift*** e **xorshift+** que passa na maioria dos testes do pacote **BigCrush** (com uma quantidade de falhas de magnitude maior do que **Mersenne Twister** ou **WELL**) normalmente levam menos de 10 ciclos de clock na arquitetura x86 para gerar um número aleatório, graças ao pipeline de instrução na CPU. ([VIGNA, 2017](#)) Entretanto, todos os geradores xorshift falham em alguns testes do pacote de testes **BigCrush** do **TestU01**. Isto é verdade para todos os geradores baseados em recorrências lineares, como o **Mersenne Twister** ou **WELL**. No entanto, é fácil embaralhar a produção desses geradores para melhorar sua qualidade.

Marsaglia sugeriu aumentar o período combinando-o com um simples modulo aditivo 232 (que ele chama de “sequência de Weyl” depois do teorema de equidistribuição de Weyl), produzindo um período de $2^{160} - 2^{32}$. O [Código-fonte 2](#) a seguir mostra a implementação de uma função com esse embaralhamento chamada de **xorwow**:

Código-fonte 2 – Função Xorwow

```

1  /* The state array must be initialized to not be all zero in the first four
   words */
2  uint32_t xorwow(uint32_t state[static 5])
3  {
4      /* Algorithm "xorwow" from p. 5 of Marsaglia, "Xorshift RNGs" */
5      uint32_t s, t = state[3];
6      t ^= t >> 2;      t ^= t << 1;
7      state[3] = state[2]; state[2] = state[1]; state[1] = s = state[0];
8      t ^= s;          t ^= s << 4;
9      state[0] = t;
10     return t + (state[4] += 362437);
11 }

```

O gerador Xorwow falha alguns testes no **BigCrush**. Mesmo assim o bom desempenho em termos de velocidade o torna bom o suficiente para algumas aplicações científicas. O gerador Xorwow é o padrão no kit de desenvolvimento CUDA da Nvidia. [Nvidia \(2018\)](#)

2.6.3 PRGN Mersenne Twister

Mersenne Twister (MT) é considerado um dos mais populares geradores de números pseudo-aleatórios. Está disponível em muitas linguagens de programação, bibliotecas e APIs. O MATLAB usa o PRNG padrão desde que foi incluído pela primeira vez há quase uma década. O MT é uma variação do algoritmo conhecido como GFSR (LEWIS; PAYNE, 1973). Os GFSRs são geradores rápidos e suas implementações independem do tamanho da palavra da máquina alvo.

O algoritmo foi descrito pela primeira vez em um artigo de Makoto Matsumoto e Takuji Nishimura publicado em 1998 e disponível no site <<http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/emt.html>> de um dos autores. A definição original de Mersenne Twister tem um período de $2^{19937} - 1$, e sua saída é livre de correlações de longo prazo considerando um espaço de estados com 623 dimensões.

O algoritmo Mersenne Twister tem uma forte relação com os primos de Mersenne - daí o nome. Os primos de Mersenne são primos da forma $2^p - 1$ onde p em si é primo. Eles são nomeados após um frade francês que os estudou no início do século XVII. Até o momento desta escrita, o maior número primo conhecido é o primo de Mersenne com p igual a 57,885,161. O Mersenne Twister tem p igual a 19937. Isso é minúsculo até onde os primos de Mersenne vão, mas é enorme no que diz respeito aos geradores de números aleatórios.

2.6.4 PRNG WELL

Well equidistributed long-period linear generator (WELL) é uma família de geradores de números pseudo-aleatórios desenvolvidos em 2006 por François Panneton, Pierre L'Ecuyer e Makoto Matsumoto.(PANNETON; L'ECUYER; MATSUMOTO, 2006) É uma forma de registro de deslocamento de feedback linear otimizado para implementação de software em uma máquina de 32 bits.

A estrutura é semelhante ao do Mersenne Twister: um estado grande composto de palavras de saída anteriores (32 bits cada), a partir das quais uma nova palavra de saída é gerada usando o módulo de recorrências lineares 2 sobre um campo binário finito. No entanto, uma recorrência mais complexa produz um polinômio gerador mais denso, produzindo melhores propriedades estatísticas.

Cada etapa do gerador lê cinco palavras de estado: os 32 bits mais antigos (que podem ultrapassar um limite de palavra se o tamanho de estado não for um múltiplo de 32), os 32 bits mais novos e três outras palavras intermediárias. Em seguida, uma série de oito transformações de palavra única (principalmente da forma $x \leftarrow x \oplus (x \gg k)$) e seis operações exclusivas combinando-as em duas palavras, que se tornam as mais novas duas palavras de estado, uma das quais será a saída.

O nome dos geradores WELL devem seguir uma estrutura onde os números indicam

o tamanho do estado em bits; sufixos de letras denotam variantes do mesmo tamanho. Por exemplos: WELL512a é uma implementação com 512 estados.

2.6.4.1 Estrutura do gerador WELL

A recorrência geral para os geradores WELL é definida pelo algoritmo dado na [Figura 11](#), que usa a seguinte notação. k é decomposto como $k = rw - p$ onde r e p são inteiros únicos tais que $r > 0$ e $0 \leq p < w$. O vetor de estado x_i é decomposto em blocos de w bits como $x_i = (v_{i,0}, \dots, v_{i,r-1})$ onde os últimos p bits de v_i , $r - 1$ são sempre zero, e $T_0, \dots, T_7$ são matrizes binárias $w \times w$ que aplicam transformações lineares a esses blocos de w bits. Os inteiros k, p, m_1, m_2, m_3 , onde $0 < m_1, m_3 < r$, e as matrizes $T_0, \dots, T_7$, são escolhidos de modo que $P(z)$, o polinômio característico de A com o grau $k = r * w - p$ é primitivo sobre F_2 . Denotando por rot_p uma rotação à esquerda por p bits, mp é uma máscara de bits que mantém os primeiros $w-p$ bits e define todos os outros bits como zero e “ $\oplus$ ” e “ $\leftarrow$ ” representam o XOR bit a bit e a instrução de atribuição, respectivamente. As variáveis temporárias $z_0, \dots, z_4$ são vetores w -bit.

Figura 11 – Algoritmo de geradores do tipo WELL

$$\begin{array}{ll}
 z_0 & \leftarrow rot_p(v_{i,r-2}^T, v_{i,r-1}^T)^T; \\
 z_1 & \leftarrow T_0 v_{i,0} \oplus T_1 v_{i,m_1}; \\
 z_2 & \leftarrow T_2 v_{i,m_2} \oplus T_3 v_{i,m_3}; \\
 z_3 & \leftarrow z_1 \oplus z_2; \\
 z_4 & \leftarrow T_4 z_0 \oplus T_5 z_1 \oplus T_6 z_2 \oplus T_7 z_3; \\
 v_{i+1,r-1} & \leftarrow v_{i,r-2} \& m_p; \\
 \text{for } j = r-2, \dots, 2, \text{ do} & \\
 \quad v_{i+1,j} & \leftarrow v_{i,j-1}; \\
 v_{i+1,1} & \leftarrow z_3; \\
 v_{i+1,0} & \leftarrow z_4; \\
 \text{output } y_i & = v_{i,1} \text{ or } y_i = v_{i,0}.
 \end{array}$$

A estrutura dos geradores WELL oferece vantagens em relação aos geradores Mersenne Twister pois necessita de menos registradores para armazenar os estados, é de implementação mais simples e melhor performance. Como o algoritmo mantém uma boa distribuição ele deve ser considerado compatível com métodos que utilizam simulação Monte Carlo. ([PANNETON; L'ECUYER; MATSUMOTO, 2006](#))

2.7 Transformada Box-Muller

Todos os geradores citados anteriormente geram sequências de números aleatórios com distribuição uniforme. Entretanto, para o escopo, deste projeto distribuições normais ou gaussianas são necessárias. A transformada Box-Muller permite que uma distribuição normal bivariada bidimensional (dois valores por iteração) seja produzida a partir de uma distribuição uniforme. A [Figura 12](#) ilustra a transformada.

Figura 12 – Transformada de Box-Muller para geração de distribuição normal a partir de distribuição uniforme

A definição formal da transformada Box-Muller é: se U_1 e U_2 forem pares de números de uma série com distribuição uniforme e independente entre 0 e 1, então z_1 e z_2 , definidos por [Equação 2.7](#), terão uma distribuição normal com média $\mu = 0$ e variância $\sigma^2 = 1$.

$$\begin{aligned} z_1 &= R \cos(\Phi) = \sqrt{-2 \ln(U_1)} \cos(2\pi U_2) \\ z_2 &= R \sin(\Phi) = \sqrt{-2 \ln(U_1)} \sin(2\pi U_2) \\ R^2 &= -2 \ln U_1 \\ \Phi &= 2\pi U_2 \end{aligned} \tag{2.7}$$

A forma polar dessa transformada é mais interessante para o uso em GPUs modernas e implementações de hardware, pois não precisa utilizar as funções seno e cosseno. Com z_1 e z_2 definidos pela [Equação 2.8](#)

$$\begin{aligned} z_1 &= \sqrt{-2 \ln(U_1)} \cos(2\pi U_2) = \sqrt{-2 \ln(s)} (u/\sqrt{s}) = u \sqrt{-2 \ln(s)/s} \\ z_2 &= \sqrt{-2 \ln(U_1)} \sin(2\pi U_2) = \sqrt{-2 \ln(s)} (v/\sqrt{s}) = v \sqrt{-2 \ln(s)/s} \end{aligned} \tag{2.8}$$

A forma polar da transformada pode ser obtida definindo que: u e v de uma série de números aleatórios uniformemente distribuídos independentes no intervalo entre $[-1, +1]$ tendo $s = R^2 = u^2 + v^2$, descartando-se u e v caso $s \geq 1$ verifica-se com outro par (u, v) que cumpra a condição. Já que u e v são valores uniformemente distribuídos e apenas pontos dentro do círculo unitário são admitidos, os valores de s também serão uniformemente distribuídos entre $[0, 1]$. Sendo o valor de s com o de U_1 e $\cos \theta / (2\pi)$ com o de U_2 na forma básica. Como mostrado na [Figura 13](#), os valores de $\cos \theta = \cos(2\pi U_2)$ e $\sin \theta = \sin(2\pi U_2)$ na forma básica podem ser substituídos pelas proporções $\cos \theta = u/R = u/\sqrt{s}$ e $\sin \theta = v/R = v/\sqrt{s}$, respectivamente.

2.8 Sistemas Computacionais Heterogêneos

Os dispositivos eletrônicos modernos, como telefones inteligentes, são capazes de executar uma variedade de tarefas, desde mensagens de texto mais simples até operações gráficas multimídia que requerem intensidade computacional. Até a metade da primeira década do século

Figura 13 – Representação em coordenadas polares para transformada Box-Muller

21, tais tarefas eram restritas a supercomputadores especializados, caros e consumidores de grande quantidade de energia elétrica.

Para atender essa demanda computacional e atingir um grau aceitável de portabilidade, os fabricantes de sistemas computacionais passaram a desenvolver cada vez mais sistemas heterogêneos em projetos modernos de sistema em chip - *System on a Chip* (SoC). Tais sistemas mitigam os problemas encontrados por escalonamento de núcleos (usando vários núcleos homogêneos), originando-se principalmente da chamada “memory wall” (parede da memória) e gargalo de Von Neumann (TEUBNER; WOODS, 2013; PEREIRA *et al.*, 2011). Designs com tais arquiteturas heterogêneas consistem essencialmente em uma combinação de processadores multi-core e uma variedade de aceleradores de hardware para acelerar a execução de tarefas computacionalmente intensivas (HOROWITZ, 2014).

Unidades gráficas de processamento (GPUs) oferecem maior taxa de transferência de ponto flutuante, uma arquitetura favorável para paralelismo de dados e maior largura de banda de memória que os processadores. Essas propriedades fazem deles bons candidatos para serem usados como aceleradores em sistemas modernos de computação de alto desempenho (HPC) (WEBER *et al.*, 2011). Os sistemas HPC que usam aceleradores baseados em GPU, no entanto, são ineficientes em termos de consumo de energia (SCHRYVER *et al.*, 2011a).

Os dispositivos FPGA modernos em comparação com as GPUs podem fornecer uma velocidade de processamento razoável, consumindo apenas uma fração da potência de uma GPU (OVTCHAROV *et al.*, 2015). Além disso, em comparação com CPUs multi-core, há uma lacuna de desempenho que aumenta continuamente, favorecendo FPGAs de uma geração para a próxima, especialmente no que diz respeito a HPC ou aplicações de *data center*. O desempenho aprimorado, combinado com uma eficiência de energia superior, resulta em um aumento da eficiência do desempenho em potência dos FPGAs em comparação às GPUs e às CPUs (SUNDARARAJAN, 20110). Esses recursos dos FPGAs foram reconhecidos por várias

empresas de *big data* como Microsoft e Baidu, como fica evidente pela decisão de usar FPGAs em vez de GPUs como aceleradores em seus servidores de dados (OUYANG *et al.*, 2014).

Um dos principais obstáculos na utilização de FPGAs para aceleração, no entanto, é a complexidade de programá-los. Os FPGAs são geralmente programados usando uma das linguagens de descrição de hardware (HDL), como Verilog ou VHDL, usadas por projetistas de hardware. Essa limitação, no entanto, pode ser resolvida por uma técnica chamada síntese de alto nível - High Level Synthesis (HLS). O HLS permite que os projetistas programem um FPGA usando linguagens de alto nível, por exemplo, C, C++, SystemC ou OpenCL. Isso, por sua vez, reduz a verificação e o tempo de projeto em comparação ao design de HDL.

2.8.1 Verilog e VHDL

VHSIC Hardware Description Language (VHDL) é um HDL desenvolvido nos anos 80, sob as instruções do Departamento de Defesa dos EUA (DoD). Essa linguagem foi encomendada para documentar o comportamento dos *Application Specific Integrated Circuits* (ASICs) incluídos em equipamentos do portfólio do DoD. Se uma empresa quisesse ser um fornecedor de DoD, teria que cumprir o novo padrão VHDL. O VHDL usa muitos conceitos da linguagem de programação Ada nas estruturas de dados na sintaxe.

Uma especificação VHDL pode ser executada para atingir um alto nível de confiança em sua exatidão antes de iniciar o projeto, e pode simular de uma a duas ordens de grandeza mais rápido do que uma descrição no nível da porta. Uma especificação de VHDL para uma peça pode formar a base para um modelo de simulação para verificar a operação da peça no contexto mais amplo do sistema (por exemplo, simulação de placa de circuito impresso). A qualidade da simulação depende da precisão com que a especificação lida com aspectos como tempo e inicialização.

Em outra alternativa, a empresa Gateway Design Automation, nos anos 80, desenvolveu primeiro um simulador lógico, Verilog-XL, e com ele uma linguagem de descrição de hardware. A Cadence Design Systems adquiriu a Gateway em 1989 e, com ela, os direitos da linguagem e do simulador. Em 1990, a Cadence colocou a linguagem (mas não o simulador) no domínio público, com a intenção de que se tornasse uma linguagem padrão e não proprietária.

Agora Verilog HDL é um padrão IEEE - número 1364. A primeira versão do padrão IEEE para Verilog foi publicada em 1995. Uma versão revisada foi publicada em 2001: Esta é a versão usada pela maioria dos usuários do Verilog.

2.8.2 System Verilog

O System Verilog combina os recursos das linguagens de descrição de hardware, como Verilog e VHDL, com recursos de linguagens de verificação de hardware especializadas, juntamente com recursos de C e C++. O System Verilog se tornou um padrão IEEE oficial (IEEE

1800TM) em 2005, foi atualizado com o IEEE 1800TM 2009 e agora está sendo aperfeiçoado sob a orientação da Accellera, assim como fornecedores de ferramentas e os usuários ganham experiência com a implementação prática e aplicação da linguagem.

À medida que amadurece, o System Verilog tem encontrando aplicação prática nas áreas de codificação RTL concisa e produtiva, Verificação Baseada em Asserção, e construindo ambientes de verificação orientados a cobertura usando técnicas aleatórias restritas.

2.8.3 Bluespec System Verilog

O Bluespec System Verilog (BSV) é um conjunto de ferramentas Electronic Design Language (EDL) para o projeto em ASIC e em FPGA. Ele foi introduzido em 2004 como uma alternativa aos HDLs tradicionais, como VHDL e Verilog, e ao mesmo tempo reduz a lacuna com uma linguagem de programação de alto nível como C/C++. Ele promete 100% de hardware sintetizável a partir de abstrações e códigos mais alto nível, próximos da linguagem C, enquanto dando controle a algumas das implementações de hardware para os desenvolvedores. O modelo semântico subjacente ao BSV é o de ações guardadas e atômicas. Ou seja, uma implementação correta do BSV tem que garantir que todo comportamento observável esteja de acordo com alguma execução sequencial dessas regras atômicas. Na verdade, no coração do compilador BSV está a síntese de um gerenciador de regras que seleciona um subconjunto de regras ativadas que não interagem entre si. Aproveitamos esses atributos BSV para adaptar implementações de PRNG em C/C++ e integra-los em CPU customizável.

Figura 14 – Compilador de BlueSpec para Verilog

A Figura 14 ilustra a função do **Bluespec Compiler (BSC)** para Verilog. O compilador recebe como entrada os módulos descritos em BSV e gera módulos Verilog. Embora tenha sido descrito em alto nível, o Verilog gerado produz hardware 100% sintetizável. O BSC também pode gerar executáveis de simulação - **Bluesim**. O pacote de desenvolvimento do BSV contém conjuntos de ferramentas e módulos úteis.

2.8.4 OpenCL

OpenCL é um *framework* para escrever programas que executam através de plataformas heterogêneas consistindo de unidades centrais de processamento (CPUs), unidades de processamento gráfico (GPUs), processadores digitais de sinais (DSPs), FPGAs e outros processadores ou aceleradores de hardware. O OpenCL especifica linguagens de programação (baseadas em C99 e C++ 11) para programar esses dispositivos e interfaces de programação de aplicativos

(APIs) para controlar a plataforma e executar programas nos dispositivos de computação. O OpenCL fornece uma interface padrão para computação paralela usando paralelismo baseado em tarefas e dados.

O OpenCL é um padrão aberto mantido pelo consórcio de tecnologia sem fins lucrativos Khronos Group. As implementações em conformidade são seguidas pelas seguintes empresas em seus produtos: Altera (Intel FPGA), AMD, Apple, ARM, Creative, IBM, Imagination, Intel, Nvidia, Qualcomm, Samsung, Vivant, Xilinx e ZiiLABS.

O OpenCL considera um sistema de computação como consistindo em vários dispositivos de computação, que podem ser unidades de processamento central (CPUs) ou "aceleradores", como unidades de processamento gráfico (GPUs), anexados a um processador host (uma CPU). Define uma linguagem semelhante a C para escrever suas funções. As funções executadas em um dispositivo OpenCL são chamadas de "kernels" ou "núcleos". Um único dispositivo de computação consiste tipicamente em várias unidades de computação, que por sua vez compreendem múltiplos elementos de processamento (PEs). Uma única execução do kernel pode ser executada em todos ou em múltiplos PEs em paralelo. Como um dispositivo de computação é subdividido em unidades de computação e os PEs são definidos pelo fornecedor; uma unidade de computação pode ser considerada como um "núcleo", mas a noção de núcleo é difícil de definir em todos os tipos de dispositivos suportados pelo OpenCL (ou mesmo dentro da categoria de "CPUs"), e o número de unidades computacionais pode não corresponder ao número de núcleos reivindicados na literatura de marketing dos fornecedores (que pode, na verdade, estar contando as faixas SIMD).

Figura 15 – Modelo de organização de memória entre aplicação host e dispositivo computacional. [Arstechinica \(2008\)](#)

Além de sua linguagem de programação semelhante ao C, o OpenCL define uma interface

de programação de aplicativos (API) que permite que programas rodando no sistema hospedeiro (*host*) iniciem kernels nos dispositivos de computação e gerenciem a memória do dispositivo, que é (pelo menos conceitualmente) separada da memória do host, como indicado na [Figura 15](#). Os programas na linguagem OpenCL podem ser compilados em tempo de execução, para que os aplicativos que utilizam o OpenCL sejam portáteis entre as implementações de vários dispositivos e host diferentes. O padrão OpenCL define APIs host para C e C++; existem APIs de terceiros para outras linguagens de programação e plataformas, como Python, Java e .NET. Uma implementação do padrão OpenCL consiste em uma biblioteca que implementa a API para C e C++ e um pré-compilador OpenCL para o dispositivo de computação alvo (CPU, GPU ou FPGA).

Para abrir o modelo de programação OpenCL para outras linguagens ou para proteger a fonte do kernel da inspeção, a Representação Intermediária Portátil Padrão (SPIR - Standard Portable Intermedian Representation) pode ser usada como uma forma independente de destino para enviar kernels entre um compilador front-end e o back-end OpenCL.

2.9 FPGAs

Um FPGA é um dispositivo semicondutor cuja função pode ser definida após a fabricação. Um FPGA permite que o desenvolvedor programe recursos e funções do produto, adaptar-se a novos padrões e reconfigurar o hardware para aplicações específicas, mesmo depois de o produto ter sido instalado no campo - daí o termo campo programável. E os arranjos de portas lógicas (*gate arrays*) são organizados bidimensionalmente. Com uma quantidade significativamente grande desses elementos é possível construir sistemas complexos capazes de realizar operações sofisticadas e com grande velocidade ([PELLERIN; TAYLOR, 1997](#)).

Em termos menos técnicos, um FPGA permite flexibilidade em projetos de hardware e é uma forma de alterar a maneira como as partes de um sistema funcionam sem introduzir grandes custos e riscos de atrasos no cronograma de projeto.

Um exemplo simples é uma câmera de visão traseira projetada para um carro. Se o sistema da câmera levar 250 milissegundos a partir do momento em que o sensor de imagem vê a imagem até que o quadro da imagem realmente apareça no mostrador, e uma alteração nas regulamentações governamentais exigir que esse atraso ou latência não seja superior a 100 milissegundos, você poderá encontrar maneiras de ajustar o pipeline de processamento de sinal de imagem em um FPGA para atender aos novos requisitos de latência. Isso seria quase impossível de fazer com um sistema baseado em microprocessador. Neste exemplo, uma empresa pode obter uma grande vantagem usando um FPGA porque não precisa reprojetar peças ou comprar todos os novos processadores. ([NAVABI, 2006](#))

Os FPGAs modernos consistem em uma mistura de memória de acesso aleatório estática configurável (SRAM), pinos de entrada/saída de alta velocidade (I/Os), blocos lógicos e rote-

amento. Mais especificamente, um FPGA contém elementos lógicos programáveis chamados, naturalmente, de elementos lógicos (LEs), bem como uma hierarquia de interconexões reconfiguráveis que permitem que os LEs sejam fisicamente conectados uns aos outros. Você pode configurar LEs para realizar funções complexas ou simplesmente executar portas lógicas básicas, como AND e OR. A maioria dos FPGAs também contém blocos de memória.

Hoje, componentes e módulos de propriedade intelectual (*Intellectual Property* (IP)) proprietárias pode ser incorporada na malha FPGA para fornecer funcionalidade avançada, enquanto reduz o consumo energético e diminui o custo no desenvolvimento. Alguns exemplos do IP proprietárias incluído nos FPGAs de hoje são blocos de memória, circuitos de cálculo, transceptores, controladores de protocolo e até mesmo unidades centrais de processamento (CPUs). É importante lembrar, no entanto, que esses IPs físicos não são personalizáveis como o restante do FPGA. A integração do IP - por exemplo, um processador de sinal digital (DSP) - libera os desenvolvedores de reinventar a roda toda vez que precisarem adicionar essas partes comuns aos seus sistemas. Fabricantes de FPGA podem tem sucesso com a introdução de componentes IP proprietárias em FPGAs, porque essas funcionalidades se tornaram commodities e, portanto, bastante uniformes na maioria dos sistemas eletrônicos.

2.10 Coprojeto Hardware/Software

Em um coprojeto hardware/software é feita uma avaliação de como as características e os requisitos do sistema a ser implementado podem ser distribuídas em módulos separados entre hardware e software. Com essa organização, busca-se aproveitar as melhores vantagens em cada implementação: velocidade e baixo consumo para hardware especializado; e flexibilidade com software.

No escopo de um coprojeto o uso de FPGAs proporciona a possibilidade de desenvolvimento mais rápido e iterativo. Como esses dispositivos são reprogramáveis várias hardwares podem ser estudados e a interface com o software melhorada, assim como o reaproveitamento de componentes já utilizados em outros projetos.

2.11 Trabalhos Relacionados

A aceleração de simulações Monte Carlo com FPGA tem sido estudada em vários setores de pesquisa especialmente em Física Quântica (ORTEGA-ZAMORANO *et al.*, 2016) e Química (GOTHANDARAMAN *et al.*, 2008). Em pesquisa sobre mercado financeiro é possível encontrar exemplos para a aceleração do modelo de precificação Heston (SCHRYVER *et al.*, 2011b) e um framework para implementação genérica desse modelos de precificação (PHAM; AUNG; KUMAR, 2016).

Para implementações que utilizam OpenCL o artigo (PATEL *et al.*, 2015) descreve uma

estrutura muito semelhante a apresentada neste projeto. No trabalho de (MUSLIM *et al.*, 2017) a performance de diferentes aplicações, incluindo Black-Scholes por Monte Carlo, é avaliada.

2.11.1 Exemplo de implementação da Altera

A Intel FPGA (antiga Altera) disponibiliza em seu site⁴ um exemplo de implementação de Black-Scholes com Método Monte Carlo com uma performance de até 12 bilhões de simulações por segundo. Essa implementação utiliza o Mersenne Twister 19937 como o gerador de números aleatórios.

A placa BittWare S5PH-Q disponível no laboratório⁵ não é grande o suficiente para sintetizar todo o hardware descrito no exemplo. A solução foi remover todos os “#pragma unroll <n>” presentes no código para diminuir o uso de recursos da FPGA com o desenrolar dos loops. Esse pragma pode ser colocado antes de loops para indicar ao compilar que o loop deve ser desenrolado se possível. No exemplo do Código-fonte 3 o valor “2” indica quantas vezes o loop deve ser desenrolado. Se não for indicado a quantidade de vezes o compilador vai tentar desenrolar o máximo possível. A performance obtida pela implementação da Altera sem os pragmas foi de aproximadamente 1 bilhão de simulações por segundo.

Código-fonte 3 – Exemplo do uso de “#pragma unroll” em um kernel OpenCL

```
1 #pragma unroll 2
2 for(size_t k = 0; k < 4; k++)
3 {
4     mac += data_in[(gid * 4) + k] * coeff[k];
5 }
```

O projeto de exemplo faz uso de extensão de canais (*channel extension*) para permitir a comunicação direta entre cada um dos kernels via um buffer First-In-First-Out (FIFO). A implementação de canais desacopla a execução do kernel do processador host. Ao contrário do típico modelo de execução OpenCL, o host não precisa coordenar a movimentação de dados entre os kernels; como pode ser visto na Figura 16 (ALTERA, 2017a). Dados escritos em um canal permanece nesse canal enquanto o programa do kernel permanece carregado na FPGA.

⁴ <<https://www.altera.com/support/support-resources/design-examples/design-software/opencl/black-scholes.html>>

⁵ LCR - Laboratório de Computação Reconfigurável - ICMC/USP

Figura 16 – Diagrama do fluxo de dados entre kernels OpenCL por canais

MATERIAL E RECURSOS

Esta Seção descreve o material, as plataformas de desenvolvimento e as ferramentas utilizadas para a realização do projeto de mestrado.

Todos os recursos do Laboratório de Computação Reconfigurável estiveram disponíveis para realização do projeto. O laboratório possui os equipamentos essenciais para o desenvolvimento de arquiteturas baseadas em hardware reconfigurável e as ferramentas utilizadas representam o estado da arte dentro do mercado internacional de softwares EDA. Entretanto, os recursos hardware (CPU, GPU, FPGAs) disponíveis para testes não são os mais avançados das suas categorias e já representam modelos de gerações passadas na época de conclusão desse trabalho. Esta Seção descreve sucintamente as ferramentas de software e hardware necessárias para realização do mestrado. O objetivo desta Seção é destacar o papel que cada uma destas tecnologias desempenhou no desenvolvimento deste projeto.

3.1 Placas FPGA de desenvolvimento

A primeira fase deste projeto utilizou a placa de desenvolvimento **DEi2-150** fabricada pela empresa Terasic. Esse sistema contém um chip FPGA **Cyclone IV GX** fornecido pela Altera (Intel FPGA). A placa também conta com um processador Atom N2600 com interface otimizada com o chip Cyclone. O chip FPGA possui aproximadamente 150 mil elementos lógicos. A placa contém diversos conectores e interfaces de comunicação com dispositivos externos, incluindo saída de vídeo VGA e porta Ethernet, que podem ser vistos na [Figura 17](#).

Na segunda fase a placa utilizada foi a **S5PH-Q** fabricada pela BittWare. A FPGA contida nessa placa é um chip **Stratix V GS**, com aproximadamente 950 mil elementos lógicos. O chip FPGA é otimizado para aplicações de alta performance e que necessitam grande largura de banda de memória. Uma conexão PCI Express permite a ligação direta com a placa-mãe do computador. A [Figura 18](#) mostra as interfaces disponíveis na placa.

Figura 17 – Diagrama de blocos da placa de desenvolvimento DE-150 Terasic (2018)

Figura 18 – Diagrama de blocos da placa de desenvolvimento S5PH-Q BittWare (2018)

3.2 IDE Quartus II e Nios II

A primeira fase deste projeto consistiu no desenvolvimento de hardware especializado para geradores de números pseudo-aleatórios. Esse hardware foi integrado à instruções do processador Nios II instanciado em FPGA.

O processador Nios II é um processador *soft-core*, RISC, de propósito geral. Ele foi projetado pela Altera (agora parte da Intel FPGA) para atender principalmente aplicações de embarcados, com um foco especial em eficiência energética. O processador tem 94 instruções de propósito geral. Além dessas instruções, ele permite que projetistas criem até 256 instruções customizadas que podem ser integradas ao processador, oferecendo possíveis ganho de desempenho

em operações específicas como redes neurais ou geradores de números aleatórios (BONATO *et al.*, 2013). A Figura 19 representa o diagrama de blocos do núcleo do processador Nios II.

Figura 19 – Diagrama de blocos do processador Nios II

O pacote de desenvolvimento do Nios II é integrado ao ambiente de desenvolvimento (*Integrated Development Environment* (IDE)) Quartus II. O desenvolvedor pode escolher duas configurações básicas de núcleo do processador: "Fast" ou "Economy". A configuração "Fast" oferece funcionalidades extras e melhor performance por ciclo de clock em comparação ao núcleo "Economy". Altera (2017b)

O Quartus II foi desenvolvido pela Altera, sendo a IDE padrão para projetos com as FPGAs produzidas pela companhia. Ele possibilita a programação e análise de projetos com módulos desenvolvidos em Verilog, VHDL ou blocos de hardware pré contruídos em uma interface visual. A versão mais atual no momento dessa escrita é a 17.1, a versão utilizada nos experimentos foi a 16.1.

3.3 Intel FPGA SDK para OpenCL

Na segunda fase deste projeto foi utilizado a plataforma de desenvolvimento, Software Development Kit (SDK), para OpenCL disponibilizado pela Intel FPGA. Até o momento desta escrita, o SDK tem a versão 17.1 como a mais recente e pode ser baixado do site da companhia ¹. O foco principal desse SDK são aplicações para computação de alto desempenho, tendo suporte para os últimos lançamentos de FPGAs das famílias Stratix e Aria.

¹ <<https://www.altera.com/products/design-software/embedded-software-developers/opencl/what-s-new-in-opencl.html>>

A versão do SDK utilizado neste projeto foi a 16.1 instalado em um sistema Linux com a distribuição CentOS 6.7.

Figura 20 – Estrutura de da interface de comunicação entre as memórias do PC e da FPGA com OpenCL da Intel

3.4 Ferramentas de *profile* e desempenho

A ferramenta de profile utilizada para avaliar o desempenho de programas executados em ambiente Linux ou na plataforma do Nios II foi a GNU Profiler - gprof. Ela faz parte do conjunto de ferramentas GNU Binary Utilities (binutils). É gerado um arquivo de texto com o tempo de execução de cada parte programa, assim como uma série de estatísticas respectivas a chamadas de função no código. As ferramentas **gprof2dot** e **Graphviz** foram utilizadas em conjunto para facilitar a análise do arquivo de profiler.

Também em ambiente Linux o relatório fornecido pela função de terminal “/usr/bin/time -v <executavel>” foi usado para determinar os tempos de execução de uma implementação em relação ao uso de recursos do sistema operacional.

3.5 Biblioteca TestU01 para Testes Empíricos de PRNGs

TestU01 é uma biblioteca de software desenvolvida em linguagem ANSI C. Ela oferece um grande conjunto de ferramentas para testes estatísticos empíricos em geradores de números aleatórios uniformemente distribuídos. A terminação U01 no nome da biblioteca é uma referência explícita ao espaço de dados padrão (0,1) dos PRNGs.

A biblioteca implementa diferentes geradores de números aleatórios de forma genérica e também muitos geradores específicos propostos na literatura ou encontrados em produtos de

software utilizados no mercado. Ela também traz implementações bem estabelecidas de vários testes estatísticos clássicos para PRNGs, bem como vários outros testes propostos na literatura. Os testes podem ser aplicados aos geradores pré-definidos da própria biblioteca TestU01, aos geradores definidos pelo próprio usuário ou a um conjunto de dados numéricos convenientemente formatados no intervalo (0,1), provenientes de um arquivo externo. Há ainda a possibilidade de fazer um conjunto de testes, especialmente pré-definidos, ou baterias de testes [Montreal \(2013\)](#).

A biblioteca TestU01 foi de grande ajuda no desenvolvimento deste processo pois: é portátil e veloz devido a sua implementação em ANSI C, distribuição gratuita, muito bem documentada, e fácil uso. A biblioteca vem sendo utilizada em grande parte da literatura para testes de PRNGs. Também é disponibilizado as implementações originais dos geradores MT19937, WELL512a, WELL1024 e os outros geradores em software utilizados neste trabalho, o que serve para comparação tanto em performance quanto para verificação. Desta forma, é possível que um conjunto de resultados obtidos por sua versão em hardware possa ser comparado com as implementações padrão em software.

3.6 MetaTrader e MQL5

MetaTrader 5 é uma plataforma multimercado que permite executar transações no mercado Forex, de ações e mercados futuros. A plataforma oferece recursos para análise técnica (visualização de tendências baseado no comportamento dos gráficos de preços e volume), e também permite trabalhar com programas de negociação algorítmicas, chamados de robôs. Os robôs são programas que monitoram o mercado e realizam operações automaticamente, baseando-se em algoritmos e estratégias pré-programadas.

Figura 21 – Janela de usuário do Metatrader 5

A MetaTrader 5 implementa um sistema de negociação com o livro de ofertas (*booking*), registro real de ordens e transações. Os usuários podem fazer operações manuais com mouse e teclado e registrar posições para execução imediata ou lançar ordens programadas com indicadores de *stop-loss* e *take-profit*. O acesso ao livro de ofertas para uma bolsa de valores específica depende da intermediação de uma corretora e os tipos de operações que o usuário pode realizar dependem de permissões da corretora.

As corretoras que oferecem acesso ao MetaTrader geralmente disponibilizam aos seus clientes contas de demonstração, também chamadas de *paper accounts*. Nessas contas os clientes têm a disposição uma quantidade de dinheiro virtual (*paper money*) em que eles podem testar as suas estratégias em situações de mercado em tempo real. Também é possível baixar os dados históricos dos preços dos ativos e realizar simulações com algoritmos programados em MQL5.

A MetaQuotes Language 5 (MQL5) é uma linguagem de programação para indicadores técnicos, robôs de negociação e aplicativos auxiliares para automatizar a negociação nos mercados financeiros. O MQL5 é uma linguagem moderna de alto nível e desenvolvida pela MetaQuotes Software Corp. para o Metatrader 5. A sintaxe da linguagem é semelhante à da linguagem C++ e permite que o desenvolvedor escreva programas no estilo de programação orientada a objetos (POO).

3.6.1 Script de Negociação Automatizada

Usando a linguagem MQL5 é possível desenvolver scripts para monitorar o mercado e efetuar análises de vários indicadores. A comunidade de desenvolvedores disponibiliza online² scripts que implementam os mais diferentes modelos de análise estatística ou ferramentas úteis para os usuários que preferirem análise gráfica e negociação manual.

No MetaTrader 5 os scripts que podem executar operações de compra e venda automatizadas são chamados de “*Expert Advisor*”. Vários desses “*Expert Advisor*” também são disponibilizados online, entretanto, é recomendável que o usuário adapte com cuidado os scripts para o mercado que ele está operando. Um “*Expert advisor*” em MQL5 pode implementar a função `OnTick()` que é sempre executada a cada ciclo de operação (*Tick*). Um ciclo de operação na plataforma MetaTrader é o tempo entre cada atualização no preço da ação monitorada.

Uma das opções para conseguir melhor desempenho com aceleração das simulações Monte Carlo é delegar esse processamento para programas externos ao MetaTrader. O “*Expert Advisor*” pode monitorar conjuntos de ações e opções com scripts específicos para essa função. Uma possível estratégia de negociação consiste em comprar opções que estejam subvalorizadas e se posicionar comprado em opções que estejam supervalorizadas, com o preço ideal sendo determinado por Black-Scholes.

A Figura 22 mostra o fluxo dos dados pelo MetaTrader a cada ciclo de operação. O

² <<https://www.mql5.com/pt/code>>

Figura 22 – Diagrama de fluxo do script de negociação no MetaTrader com interface externa

“Expert Advisor” recebe do Script 1 o preço atual “S” e a volatilidade “V” da ação monitorada, e o Script 2 retorna o preço alvo “K” e o preço atual da opção “Op”. O “Expert Advisor” passa essas informações para um programa externo que calcula o preço “P” com Black-Scholes, com o retorno do valor, o “Expert Advisor” pode ou não lançar ordens para a bolsa de negociação.

IMPLEMENTAÇÕES

Essa seção mostra o passo a passo a implementação do trabalho de mestrado ao longo do projeto.

Uma das abordagens seguidas foi o desenvolvimento em linguagem de descrição de hardware *Bluespec* para geração de números aleatórios com as outras etapas do Método Monte Carlo em software no processador *soft-core* Nios II. Entretanto, como a performance desse sistema mostrou-se insatisfatória, o foco do projeto se manteve em uma implementação que utilizasse OpenCL para FPGA.

Em seguida, uma implementação básica do algoritmo para a resolução das equações de Black-Scholes por Monte Carlo em uma linguagem de alto nível (*Python*) será apresentado. Esse algoritmo será utilizado como base para as próximas implementações em *C* e *OpenCL*. Todos esses programas serão executados com os mesmos parâmetros de entrada e a mesma quantidade de simulações. A máquina (PC) que executará todos os experimentos é mesma e com dedicação exclusiva durante a execução desses experimentos.

4.1 Implementações - Fase 1

4.1.1 Algoritmo Black-Scholes com Monte Carlo

O [Código-fonte 4](#) representa uma das formas mais comuns de se resolver as equações de Black-Scholes para opções de compra do tipo europeia. Variações desse código podem ser encontradas facilmente na internet em diferentes linguagens, incluindo Matlab, Excel e Python.

Código-fonte 4 – Algoritmo de Black-Scholes em Python para opções de compra

```
1 import datetime
2 from random import gauss
3 from math import exp, sqrt
```

```
4
5 def generate_asset_price(S,v,r,T):
6     return S * exp((r - 0.5 * v**2) * T + v * sqrt(T) * gauss(0,1.0))
7
8 def call_payoff(S_T,K):
9     return max(0.0,S_T-K)
10
11 def get_OptPrice(S,v,r,T,K)
12     simulations = 1000000
13     payoffs = []
14     discount_factor = exp(-r * T)
15
16     for i in range(simulations):
17         S_T = generate_asset_price(S,v,r,T)
18         payoffs.append(
19             call_payoff(S_T, K)
20         )
21
22     price = discount_factor * (sum(payoffs) / float(simulations))
23     print ('Price: %.4f' % price)
24     return price
```

4.1.2 Passo a passo do algoritmo

A função `get_Opprice(S,v,r,T,K)` retorna o preço da opção baseando-se nos dados fornecidos nos parâmetros:

- S - preço atual da ação.
- v - volatilidade em porcentagem esperada no período até o vencimento
- r - taxa de juros livre de risco
- T - tempo até o vencimento da opção
- K - preço alvo da ação

Como o interesse é saber qual é a distribuição dos valores possíveis na data do vencimento da opção a função `generate_asset_prices(S,v,r,T)` calcula cada um dos valores finais ao multiplicar pelo tempo até o vencimento. Nela está em destaque a função `gauss(0,1,0)` que é a responsável por gerar os números aleatórios com distribuição normal (média 1 e variância 1). O destaque para essa função precisa ser destacado pois é na aceleração dessa função que grande parte do projeto esteve dedicado.

Como mostra a [Figura 2](#) na [Subseção 2.2.1](#) o valor de uma opção no vencimento é zero caso este tenha um valor alvo menor que o valor do ativo no vencimento, e cresce linearmente com os valores acima. O resultado de cada um dos possíveis valores é calculado pela função `call_payoff(S_T,K)`.

O valor final da opção é calculado como a média do retorno de todas as simulações multiplicado pelo “fator de desconto”, que é o decaimento do valor corrente da opção ao longo do tempo (quanto mais próximo da data do vencimento, menor é o valor da opção, pois há menor possibilidade de grandes variações).

4.1.3 PRNGs em Hardware com Bluespec e Nios II

A primeira fase do projeto consistiu no desenvolvimento de Geradores de Números Pseudo-aleatórios em hardware utilizando Bluespec como linguagem de descrição de hardware e Bluesim como ferramenta de simulação dos núcleos gerados. Esse trabalho seguiu o modelo desenvolvido por [Bonato et al. \(2013\)](#) que são membros do Laboratório de Computação Reconfigurável do ICMC. No trabalho de 2009 foi desenvolvido um PRNG do tipo Mersenne Twister 19937 (**MT19937**). Aquele gerador mostrou boa performance e qualidades aceitáveis para utilização de Métodos Monte Carlo(MMC) para filtro de partículas em robôs móveis.

Neste projeto foram reutilizados algumas das interfaces de comunicação criada por [Bonato et al. \(2013\)](#) com novos geradores da família WELL, WELL512a e WELL1024 que apresentam características de qualidade semelhantes ao MT19937 para utilização em MMC. ([PANNETON; L'ECUYER; MATSUMOTO, 2006](#))

4.1.3.1 Implementação em Bluespec System Verilog

Como o BSV possibilita o uso de sintaxe muito semelhante ao C, assim como os conceitos de tipo de variáveis, foi possível fazer uma tradução quase de um para um entre os códigos C e o BSV - vistos nos trechos de códigos a seguir, [Código-fonte 5](#) e [Código-fonte 6](#).

Código-fonte 5 – Trecho Well512 em BlueSpec

```

1      ....
2      z0 = state[(st_i+15)&'h0000000f];
3      z1 = mat0neg(-16,st[st_i])^mat0neg(-15,st[st_i+13&'h0000000f]);
4      z2 = mat0pos(11,st[(st_i+9)&'h0000000f]);
5      st[st_i] = z1 ^ z2;
6      st[(st_i+15)&'h0000000f] = mat0neg (-2,z0)^ mat0neg(-18,z1)^ mat3neg(-28,z2)
      ^mat4neg(-5,'hda442d24,st[st_i]) ;
7      st_i = (st_i+15)&'h0000000f;
8      ....

```

Código-fonte 6 – Trecho Well512 em C

```

1      .....
2      z0      = VRm1;
3      z1      = MATONEG (-16,V0)^MATONEG (-15, VM1);
4      z2      = MATOPPOS (11, VM2) ;
5      newV1 = z1 ^ z2;
6      newV0 = MATONEG (-2,z0) ^ MATONEG(-18,z1)^ MAT3NEG(-28,z2) ^ MAT4NEG(-5,0
xda442d24U,newV1) ;
7      state_i = (state_i + 15) & 0x0000000fU;
8      .....

```

A diferença ao se portar um código C para BSV é traduzir os conceitos de variáveis do C para registradores e bancos de registradores do BSV. Outro ponto importante é garantir que “variáveis” (no caso registradores) não sejam acessados concorrentemente dentro de uma função atômica. O compilador BSV vai indicar possíveis conflitos desse tipo e alertas quando não for possível definir uma função em um único ciclo de *clock*.

Os hardwares descritos para os geradores **WELL512** e **WELL1024a** têm ambos a mesma interface de entrada e saída de 32 bits e compartilham a mesma interface descrita no [Código-fonte 7](#):

Código-fonte 7 – Interface do componente dos PRNGs

```

1
2 interface IfcRandomNumberGenerator#(
3     type seed_type, // Seed type. 32 bits Word
4     type out_type    // Output word type. 32 bits Word
5 );
6 // Initialize the RNG.
7 method Action initialize (seed_type s);
8 // Get a number from RNG.
9 method ActionValue#(out_type) get ();
10 endinterface: IfcRandomNumberGenerator

```

Essa interface declara dois métodos: método `initialize(s)` do tipo `Action`, que indica que esse método vai alterar os estados internos do hardware, mas não retorna valor; e o método `get()` do tipo `ActionValue`, que além de alterar os estados internos retorna um valor.

- `initialize(seed)` : inicializa os estados do gerador a partir de um valor não zero de (`seed`). Mesmo que a semente for zero ou esse método não for chamado, os estados já tem valores diferentes de zero “hardcoded” e a sequência gerada será aleatória.
- `get()` : a cada execução retorna um valor aleatório entre 0 e o máximo valor possível para o tamanho da palavra escolhida (no caso aqui 32 bit). E também atualiza os estados

internos.

4.1.4 Testes de validação dos PRNGs

Uma das grandes vantagens do ambiente BSV é a possibilidade de prototipação e testes rápidos e iterativos por meio dos seus simuladores, isso tudo, tendo a garantia que o hardware testado será 100% sintetizado.

Para fazer os testes é possível criar um *"Test Bench"* em BSV que instancia os geradores e executa os seus métodos com as regras definidas pelo usuário. No ambiente de simulação é possível imprimir os valores retornados pelos geradores na saída padrão do programa de simulação. Um programa em C consegue executar um outro programa externo e ler sua saída de modo análogo à um arquivo. Para esse teste o *"Test Bench"* é configurado para gerar continuamente um novo número a cada ciclo de *clock*.

A saída padrão do simulador é um número inteiro sem sinal (*uint32_t*). Para convertê-lo em ponto flutuante com distribuição entre [0, 1] basta multiplica-lo por 1/MAXINT (onde MAXINT é o maior valor de inteiro 32 bits em C) que nesse sistema é $2.32830643653869628906e^{-10}$. Encapsulando esse processo em uma função que retorna um novo valor toda vez que é chamada, é possível usar a biblioteca de testes TestU01 para verificar a qualidade dos geradores. A Figura 23 mostra a estrutura dessa composição.

Figura 23 – Diagrama do fluxo para testar a qualidade do PRNG descrito em Bluespec

Cada um dos geradores passou em todos os testes do **Crush** do pacote **TestU01**, falhando apenas para os testes de equivalência linear para matrizes com *rank* (ordem) 1200, o que já era esperado de qualquer gerador WELL que atendesse os critérios de implementação (PANNETON; L'ECUYER; MATSUMOTO, 2006). O teste **Crush** necessita de uma quantidade na ordem de 2^{41} de números gerados e cada um dos experimentos levou por volta de 10 horas de execução. O teste **BigCrush**, considerado o mais completo, precisaria de uma quantidade até 10 ordens de magnitude maior.

Optou-se por não realizar o teste **BigCrush** pois o tempo necessário para completá-lo iria inviabilizar o uso da máquina para o resto da equipe do laboratório. Além disso, é possível ter confiança que os geradores implementados são bons ao se comparar as sequências de valores dos

geradores em hardware com as sequências geradas pelas implementações em software com as mesmas sementes. Para ambos os geradores as sequências foram idênticas em até 10 ou 12 casas decimais. Essas pequenas diferenças podem ser explicadas pela forma que valores em ponto flutuante são exibidos em terminais de sistemas diferentes e a aproximação durante a conversão de inteiro para ponto flutuante.

4.1.5 Nios II e instrução customizada

Tendo os hardwares dos geradores a fase seguinte foi integra-los a algum componente que pudesse utiliza-los em aplicações Monte Carlo. Uma das formas mais fáceis de fazer isso no ambiente disponibilizado pela Altera é com instruções customizadas para o processador *softcore* Nios II. Foi utilizado o ambiente de desenvolvimento Quartus II para o desenvolvimento.

O processador Nios II oferece quatro tipos de instruções customizadas:

- *Combinational*;
- *Multicycle*;
- *Extended*;
- *Internal Register File*;

Entre essas opções, a que mais se adéqua ao hardware do nosso PRNG é uma instrução do tipo *Multicycle* (multi-ciclo). Instruções customizadas desse tipo são completadas em um número fixo ou variável de ciclos de *clock*. Para uma instrução customizada que é concluída em um número fixo de ciclos de *clock*, é preciso especificar o número de ciclos na geração do sistema. A [Figura 24](#) do manual do usuário do Nios II mostra quais são os tipos de sinais que a instrução deve possuir para ser interpretada corretamente pelo compilador do processador Nios II.

Figura 24 – Modelo de instrução customizada Multicycle para o Nios II

Dos sinais indicados na [Figura 24](#) apenas os *clk*, *clk_en* e *reset* são obrigatórios. Como nosso gerador possui a opção de ser semeado com diferentes valores, os sinais *dataa[31..0]* e

result[31..0] foram especificados como entrada e saída, respectivamente, da instrução, como pode ser visto na [Figura 25](#).

Figura 25 – Diagrama de Blocos da instrução customizada com o PRNG WELL512, Quartus II da Altera

Com todos os sinais conectados corretamente pela interface do Quartus II, este gera toda a especificação do processador com as instruções customizadas inclusas no conjunto de instruções para o compilador gcc que faz parte do pacote de desenvolvimento do Quartus II. Como pode ser visto em [Código-fonte 8](#) a instrução customizada é acessada por meio de uma MACRO - ALT_CI_WELLPRNG_0(0) gerada pelo Quartus II e especificada no parser do compilador.

No [Código-fonte 8](#) o nosso gerador é semeado sempre que a função é chamada com algum valor diferente de zero. Se a instrução for invocada com zero como parâmetro um novo número aleatório será retornado.

Código-fonte 8 – Acesso a instrução customizada pelo código C

```

1    ...
2    int init = ALT_CI_WELLPRNG_0(2000);
3    i = 0;
4    for(;i<100000;i++){
5        int get = ALT_CI_WELLPRNG_0(0);
6        printf("get = %d\n", get);
7    }
8    ...

```

A estrutura da implementação pode ser visualizada na [Figura 26](#) a seguir. A comunicação é feita pela interface JTag disponibilizada no kit de desenvolvimento Quartus II.

Figura 26 – Estrutura da implementação dos geradores e transformada Box-Muller com comunicação do Nios II e um PC

A verificação do funcionamento desta implementação foi feita comparando-se a sequência gerada com a sequência obtida pela implementação padrão em software dos geradores com as mesmas sementes.

4.1.6 Componente Box-Muller e Funções de Ponto Fixo

Um esforço extra foi empregado na tentativa de melhorar o gargalo do sistema Nios II que se encontrava nas funções de ponto flutuante na transformada de Box-Muller. Foi implementada uma interface que encapsulava os geradores utilizando suas saídas como alimentação de um novo componente que transformaria os valores para ponto flutuante e então convertidos em uma distribuição normal pela transformada Box-Muller.

Como operações de ponto flutuantes são muito complexas e exigem uma grande quantidade de recursos da FPGA para implementação, foi escolhido fazer a transformada Box-Muller com ponto fixo e só depois transformar o resultado para ponto flutuante. A biblioteca de ponto fixo para Bluespec é disponibilizada pelos apoiadores da linguagem. Essa biblioteca implementa operações CORDIC e oferece interfaces para representação de números em ponto fixo. No desenvolvimento desse componente, ainda na fase com o simulador Bluesim foi constatado que a biblioteca de ponto fixo continha um erro (bug) na operação de raiz quadrada. Esse "bug" foi corrigido e divulgado para a comunidade Bluespec. O [Código-fonte 9](#) mostra a declaração dos componentes em ponto fixo implementados no módulo principal.

Código-fonte 9 – Interface do componente dos PRNGs

```

1 ...
2 Reg#(FixedPoint#(33,52))    uniform_rand_num1 <- mkReg(0);
3 Reg#(FixedPoint#(33,52))    uniform_rand_num2 <- mkReg(0);
4 FIFO#(FixedPoint#(33,52))    angle_q <- mkSizedFIFO(2);
5 CosAndSin#(33,52,33,52)      cos_and_sin <- mkCosAndSin_Pipe('cordicIters','
   cordicItersPerStage);
6 FIFO#(FPComplex#(33,52))
   angle_adjust_q <- mkLFIFO;
7
8 Server#(FixedPoint#(33,52),Tuple2#(FixedPoint#(33,52),Bool)) sqrt <-
   mkFixedPointSquareRooter(1);
9 ...

```

As operações necessárias para a transformada são executadas em mais de um ciclo de clock. Por esse motivo essas operações são invocadas utilizando enfileiramento a cada execução das regras. A [Figura 27](#) mostra o fluxo de disparo das regras para o modulo Box-Muller. Nesse fluxo os geradores são semeados no primeiro ciclo de clock em `rule init(count == 0)`, para cada clock seguinte até um total especificado por `n` em `rule run(count > 0 && count < n)`; , os módulos de PRNG alimentam as filas das operações da transformada Box-Muller

na forma básica Equação 2.7 Seção 2.7. A função $\text{CosAndSin}\#(m,n,m,n)$ retorna um par de valores para seno e cosseno de um ângulo de uma vez. Essa mudança tornou essa implementação mais simples, mas não necessariamente mais eficiente que a transformada na forma polar.

Figura 27 – Fluxo de execução das regras em Bluespec para transformada Box-Muller na forma básica

A verificação do funcionamento da transformada foi feito de modo semelhante ao descrito na Figura 23 com a saída do simulador Bluesim redirecionada para um arquivo de texto com os números gerados na distribuição normal linha à linha. Esse arquivo foi então importado para o Matlab e realizados os testes gráfico de normalidade.

A performance obtida pelo conjunto Nios II e instrução customizada, apesar de serem boas para o contexto de sistema embarcados, não é satisfatória para um sistema de alta performance competitivo no mercado financeiro. A implementação da instrução customizada da transformada Box-Miller em hardware foi suspensa e o foco do projeto mudado para o uso de OpenCL.

4.2 Implementações - Fase 2

4.2.1 PRNGs em Software

Considerando que a performance de uma implementação utilizando o processador Nios II não seria boa o suficiente para competir mesmo com computadores de uso doméstico, uma nova abordagem foi escolhida. Essa abordagem se mostrou possível com a introdução de novos dispositivos no laboratório e o acesso a ferramentas de desenvolvimento com OpenCL para FPGAs.

Com o OpenCL torna-se possível a comunicação direta entre um programa executado em um PC “top de linha” e uma ou várias FPGAs. A comunicação é feita pelo barramento PCIe

com uma taxa de transferência de dados na ordem de GB/s (Gigabytes por segundo) entre a memória do PC e a memória da FPGA.

Mas antes de partir para a implementação com OpenCL, foi feita a análise da performance possível de ser obtida com implementações de software. Para essas foram selecionados um conjunto de PRNGs considerados bons o suficiente para simulações Monte Carlo:

- Xorshift64*
- Xorshift128+
- Xorwow
- WELL512
- PCG
- Mersenne Twister 19937

O [Código-fonte 4](#) foi adaptado para C++ com a diferença de que os números aleatórios e a transformação Box-Muller são pré-computadas, com vetores preenchidos em lotes contínuos. Esse é um método que tende a melhorar o desempenho do algoritmo ao ser otimizado para processadores modernos. A função Box-Muller também foi implementada para verificar os pontos críticos de performance e para comparação com as implementações em OpenCL.

4.2.2 Implementação com OpenCL

A versão do programa com OpenCL foi implementada tendo como base as implementações anteriores em software. Do mesmo modo, os vetores de números aleatórios são gerados de uma vez para aproveitar otimizações do processador. Como os pontos críticos se encontravam, tanto nos PRNG, quanto na função da transformada de Box-Muller, ambas foram escolhidas para implementação em *kernels* de OpenCL.

Ao se implementar os kernels em OpenCL o desenvolvedor deve ficar atento a dois pontos no seu código: minimizar a dependência de dados dentro de loops; e evitar o acesso à memória global compartilhada com o processador do PC. Tendo essas garantias, o compilador do OpenCL vai gerar hardware maximizando o número de instruções realizadas pelo kernel a cada ciclo de clock.

4.2.2.1 Kernel OpenCL do PRNG

Primeiramente foi implementado um kernel com o gerador de números pseudo aleatórios WELL512a. Foi testado a sua conformidade com o gerador implementado em software com as mesmas sementes. As sequências geradas foram idênticas em até dez casas decimais com os números normalizados para o intervalo $[0,1]$ em ponto flutuante.

Figura 28 – Estrutura da implementação em OpenCL e a comunicação entre Software e FPGA por PCIe

Código-fonte 10 – Chamada da função do kernel WELL512a

```

1 __kernel void WELLRNG512a(__global float *restrict result,
2                           __global unsigned int *restrict STATE,
3                           __global unsigned int *restrict state_i,
4                           const uint seed,
5                           const uint init,
6                           const int size)

```

O [Código-fonte 10](#) mostra a assinatura do kernel WELL512a em OpenCL onde:

- `__global float *restrict result`: vetor com os números aleatórios gerados;
- `__global unsigned int *restrict STATE`: vetor de estados do gerador;
- `__global unsigned int *restrict state_i`: índice de estados do gerador;
- `const uint seed`: semente para o gerador;
- `const uint init`: indica se os estados vão ser semeados ou continuar a geração;
- `const int size`: indica a quantidade de números a serem gerados nessa execução;

As variáveis indicadas com `__global` são compartilhadas entre as memórias do programa principal e a memória do dispositivo FPGA. Nesse caso é recomendável, sempre que possível, copiar essas variáveis globais para variáveis locais do kernel e, no caso de vetores, com dimensões fixas.

Código-fonte 11 – Optimização do kernel com o uso de memória local

```

1 ...
2   unsigned int _STATE[16];
3 ...
4   for(int i = 0; i < 16; ++i){
5       _STATE[i] = STATE[i];
6   }

```

```

7     uint _state_i = *state_i;
8 ...
9 ...
10    *state_i = _state_i;
11    for(int i = 0; i < 16; ++i){
12        STATE[i] = _STATE[i];
13    }

```

Essa simples mudança, indicada no [Código-fonte 11](#), permitiu que o pré-compilador OpenCL maximizasse o pipeline de execução do kernel, possibilitando ainda o desenrolamento de loops com o `#pragma unroll`.

4.2.2.2 Kernel OpenCL da Transformada Box-Muller

O próximo kernel a ser implementado foi o da transformada Box-Muller.

Código-fonte 12 – Chamada da função do kernel Box-Muller

```

1 __kernel void BoxMuller(__global float *restrict result,
2                        __global int *restrict final_size,
3                        __global float *restrict input_buffer,
4                        const float m,
5                        const float s,
6                        const int size )

```

O [Código-fonte 12](#) mostra a assinatura do kernel BoxMuller em OpenCL onde:

- `__global float *restrict result`: é o vetor com os números aleatórios com distribuição normal com média `m` e desvio padrão `s`;
- `__global int *restrict final_size`: indica a quantidade de números transformados;
- `__global float *restrict input_buffer`: é o vetor de números aleatórios com distribuição uniforme `[0,1]`;
- `const float m`: indica a média da distribuição normal a ser gerada;
- `const float s`: é o desvio padrão da distribuição normal a ser gerada ;
- `const int size`: indica a quantidade de item no vetor de entrada `input_buffer`;

Como visto em [Equação 2.8](#) a implementação com coordenadas polares da transformada Box-Muller descarta pares de coordenadas com dimensão maior que a unitária, o kernel não tem definido exatamente quantos números vão ser gerados no total. Considerando esse fator e a

dependência de dados no algoritmo da transformada, o pré-compilador OpenCL não conseguiu otimizar esse kernel do mesmo modo que o kernel do WELL.

4.2.2.3 Programa Host

O programa host para essa fase foi implementado com estrutura mais próxima às implementações examinadas em [Subseção 4.2.1](#). Dessa forma a comparação entre a performance de cada função pode ser melhor avaliada. No programa host são reservados os espaços de memórias dos vetores de números aleatórios, um para a distribuição uniforme gerada pelo kernel WELL e outro vetor com a distribuição normal. O vetor que armazena os estados do gerador WELL também é compartilhado entre o programa host e o kernel WELL. Desse modo é possível ter uma geração contínua de novos números na mesma série sem precisar semear novamente o gerador.

Figura 29 – Da esquerda para a direita: fluxo dos dados entre as memórias do programa host e a memória do dispositivo FPGA

4.3 Implementações - Fase 3

A terceira e última etapa deste projeto consistiu em conectar a implementação anterior com uma plataforma de negociação online. A plataforma escolhida foi a do **MetaTrader 5** na linguagem de programação **MQL5**. O objetivo era que o programa com aceleração para precificação pudesse ser invocado pelo script de negociação executado sobre o MetaTrader 5.

4.3.1 Interface com MetaTrader

A interface entre o MetaTrader e o programa host com o hardware de aceleração foi implementada usando *Sockets*. Sockets possuem um padrão compatível em várias linguagens de programação e diferentes sistemas operacionais. No MQL5 é implementado a interface para socket WinAPI. Na configuração deste trabalho nós temos o programa MetaTrader executando em uma máquina Windows 10 e o programa Black-Scholes em uma máquina Linux CentOS 6.7. Como ambas as máquinas compartilham a mesma rede, nenhum tipo de configuração de segurança é necessário, podendo usar o endereço IP de cada máquina diretamente. A [Figura 30](#) ilustra a organização desse sistema.

Nessa organização, são criados dois pares cliente/servidor para a transferência das informações. No primeiro par o servidor é criado no programa host, com o cliente. Quando

Figura 30 – Estrutura do sistema completo com interação com o Metatrader 5 por Sockets

o evento no script é ativado, os dados são mandados para o programa host. Depois disso os papéis se invertem, o script espera uma resposta enquanto o programa host executa o simulação Black-Scholes.

RESULTADOS

Neste capítulo serão apresentados os resultados dos experimentos discutidos no [Capítulo 4](#).

5.1 Geradores em Hardware

As [Tabela 1](#) e [Tabela 2](#) apresentam os resultados dos hardwares dos geradores de números pseudo-aleatório para as placas Cyclone IV e Stratix V. O relatório é gerado durante a compilação do código HDL descrevendo o hardware e na operação de *Fitter* pela IDE Quartus II. Os dados relevantes para avaliação são frequência máxima (em Mega Herz) e área da FPGA ocupada pelo hardware (em número de elementos lógicos - LE).

Na Cyclone IV os geradores WELL ocupam entre 1 e 2% e o gerador MT19937 ocupa 13%. Na Stratix V o hardware ocupado não passou de 2% no pior caso.

Os hardwares para as duas versões de geradores WELL passaram nos testes de qualidade da biblioteca TestU01 como esperado. A sequência de números inteiros também foi a mesma

Tabela 1 – Relatório de compilação para Cyclone IV

	Freq. Max (MHz)	Área (EL)
WELL512a	122,8	1.849
WELL1024	99,3	3.425
MT19937*	89,2	20.126

Tabela 2 – Relatório de compilação para Stratix V

	Freq. Max (MHz)	Área (EL)
WELL512a	237.42	1.164
WELL1024	189.0	2.490
MT19937*	190.84	5.058

esperada das implementações em software para a mesma semente inicial. O resultado manteve-se consistente para todas as sementes e intervalos da sequência testados.

5.2 Instrução Customizada para Nios II

Os geradores anteriores foram implementados como instruções customizadas para o processador Nios II com o núcleo tipo *Fast*. Os testes para averiguar a performance dos geradores foram executados em um programa em C executando no próprio Nios II pela interface de programação disponibilizada pelo Quartus II. Os valores gerados (inteiros 32bits) também foram idênticos aos exibidos no simulador Bluesim e versões de softwares para a mesma semente.

Cada uma dos hardwares PRNG produz um número do tipo inteiro a cada ciclo de *clock*. Entretanto, na execução da instrução customizada precisa de dois ciclos para executar completamente. A operação de multiplicação de ponto flutuante no Nios II é realizada em quatro ciclos, em um total de seis ciclos para normalizar a distribuição para o intervalo entre 0 e 1. O que resultaria com o melhor caso da Tabela 1 em aproximadamente 20 milhões de números gerados por segundo.

A função da transformada Box-Muller foi incluída no software de teste e os valores impressos na saída como no exemplo anterior. Mais uma vez os resultados foram equivalentes às outras implementações e em conformidade com distribuições normais com número equivalente de amostras.

Em termos de performance, como pode ser visto na Figura 31 o tempo de execução é dominado pelas operações de ponto flutuante na transformada Box-Muller.

Figura 31 – Profile visual do programa de teste executado em um processador Nio II, na placa Cyclone IV

Sem nenhuma outra otimização o sistema de teste demorou aproximadamente 30

Tabela 3 – Performance dos geradores implementados em software

Gerador	Tempo de Execução		Speed up
	-O0	-O2	
MT19937	23,32	2,14	10,89
PCG	12,08	0,73	16,54
WELL512	8,99	0,76	11,82
Xorshift64*	10,73	0,47	22,82
Xorwow	9,74	0,5	19,48
Xorshift128+	9,1	0,75	12,13

segundos para produzir 100mil números pseudo-aleatórios em ponto flutuante. Uma performance muito inferior à implementações em software executado em computadores modernos.

5.3 Black-Scholes em Software

Todos os experimentos foram realizados em um computador com processador Xeon ES-1603 v3 com frequência de *clock* de 2.80GHz com 80GB de memória RAM, com sistema operacional CentOS 6.7. O uso da máquina foi exclusivo durante os experimentos, com nenhum outro processo em execução que não fosse do sistema operacional.

A [Tabela 3](#) indica o tempo de execução médio (em segundos) para que 1 milhão de simulações sejam executadas. A coluna “Speed up” indica a relação entre os tempos obtidos pelos sistemas sem otimização (-O0) e com otimização do compilador (-O2).

Outro fator a ser considerado na avaliação de um gerador de números pseudo-aleatórios é se os resultados das simulações convergem para um valor limite conforme o número de simulações aumenta. Essa medida pode ser obtida como o desvio de cada conjunto de simulações com o valor médio de todas as simulações. Utilizando esses geradores o desvio médio foi da ordem de 10^{-2} com 100 mil simulações até 10^{-4} com 10 milhões de simulações.

Como pode ser visto na [Figura 32](#) as otimizações de compilador conseguem reduzir significativamente o tempo de execução das operações de ponto flutuante da função Box-Muller. A implementação Xorshift64* obteve a melhor performance com o gerador demandando 76% do tempo total. Para todos os geradores o tempo de uso da CPU foi de 99%.

5.4 Black-Scholes em OpenCL

O pacote de desenvolvimento Intel FPGA para OpenCL oferece uma interface HTML para o relatório de compilação dos kernels. [Figura 34](#), [Figura 35](#), [Figura 36](#) são capturas desse relatório.

Como indicado na [Figura 34](#) os kernels da transformada Box-Muller e gerador WELL ocupam 4% e 1% do total de elementos lógicos da placa Stratix V. O kernel Box-Muller ainda

Figura 32 – Profile visual do Black-Scholes em software com o gerador PCG com otimização de compilador

Figura 33 – Profile visual do Black-Scholes em software com o gerador Xorshift64* com otimização de compilador

utiliza 31 blocos de DSP; esses blocos são requisitados para as operações de ponto flutuante da transformada. As linhas de "Board interface" e "Global interconet" indicam a quantidade de elementos utilizada pela interface do OpenCL. Essa interface é gerada pela ferramenta de desenvolvimento e é constante para qualquer implementação que utilize essa placa.

A Figura 35 é o relatório da análise dos *loops* presentes nos kernels compilados. O compilador OpenCL transforma cada loop (*for* ou *while*) do código em um bloco de instruções em pipeline. Nesse relatório os blocos Block1 e Block2 fazem parte do kernel Box-Muller e o bloco Block7 faz parte do kernel WELL; os blocos indicados por "*Fully unrolled loop*" são blocos que são executados em um único ciclo e são resultados das otimizações da cópia de

Figura 34 – Relatório do uso de elementos lógicos e blocos de memória dos kernels OpenCL

	ALUTs	FFs	RAMs	DSPs	Details
▼ Kernel System (Logic: 27%)	76673 (16%)	111089 (12%)	580 (23%)	31 (12%)	
Board interface	38262	44528	257	0	• Platform i...
Global interconnect	14572	17570	104	0	• Global int...
► BoxMuller	18718 (4%)	34323 (4%)	169 (7%)	31 (12%)	• Number of ...
► WELLRNG512a_generate	5121 (1%)	14668 (2%)	50 (2%)	0 (0%)	• Number of ...

dados da memória global para memória local comentado em [Código-fonte 11](#) da [Subseção 4.2.2](#). Nenhum dos loops teve indicação para serem “desenrolados”, entretanto esse relatório indica que o *loop* principal do *kernel* WELL pode ser desenrolado com eficiência.

Figura 35 – Relatório de loops dos kernels OpenCL

Loops analysis	☒ Show fully unrolled loops			
	Pipelined	II	Bottleneck	Details
Block1 (well_kernel.cl:51)	Yes	2	n/a	
Block2 (Unknown location)	Yes	217	II	Data dependency
Fully unrolled loop (well_kernel.cl:110)	n/a	n/a	n/a	Auto-unrolled
Block7 (well_kernel.cl:120)	Yes	1	n/a	
Fully unrolled loop (well_kernel.cl:132)	n/a	n/a	n/a	Auto-unrolled

A [Figura 36](#) mostra o fluxo de dados entre blocos dos kernels e a memória principal. O acesso à memória principal pelo bloco Block2 faz dele o ponto crítico para a performance geral para as simulações Monte Carlo.

No programa host foram testados diferentes tamanhos para os vetores de números aleatório inicial. A performance geral foi melhor para grupos de vetores maiores até aproximadamente 2^{10} elementos. Nessa configuração é possível realizar aproximadamente 70 milhões de simulações por segundo; com uso de CPU em torno de 45%. Vetores maiores poderiam causar o esgotamento da memória disponível na FPGA.

Os valores obtidos por Black-Scholes estiveram em conformidade com o obtido em outras implementações com os mesmos parâmetros. A variação média nos valores ficou na ordem de 10^{-5} para os testes com mais de 500 milhões de simulações.

Figura 36 – Diagrama do fluxo de dados entre os blocos dos kernels em OpenCL

CONCLUSÃO E TRABALHOS FUTUROS

O trabalho desenvolvido explorou as possibilidades de aceleração do Método Monte Carlo para a resolução da fórmula de precificação de opções Black-Scholes. A busca por uma implementação competitiva em termos de velocidade e precisão fez com que o projeto fosse dividido em etapas. Cada etapa resultou em contribuições significativas para o estado-da-arte em seus respectivos ambientes. Nesse capítulo será discutido brevemente as conclusões de cada fase, bem como as possibilidades de trabalhos futuros em seus respectivos setores.

6.1 Fase 1

Nessa fase foi implementado com sucesso dois hardwares para geração de números pseudo-aleatórios da família de geradores WELL. Esses hardwares utilizam menos recursos e operam com maior frequência de clock que o hardware usado como comparação e descrito na mesma linguagem (Bluespec). O desenvolvimento de outros geradores dessa mesma família pode utilizar a mesma estrutura deste projeto. Esses componentes podem melhorar o desempenho de aplicações embarcadas com maiores limitação de espaço e energia.

As instruções customizadas para o processador Nios II também foram bem sucedidas. Entretanto, a performance desse sistema com pontos flutuantes e a velocidade da interface de dados com um sistema externo foi uma limitação deste projeto. Trabalhos futuros devem explorar interfaces de dados mais robustas com maiores taxas de transmissão. Na parte de software estratégias mais sofisticadas no uso de memória e no pipeline de instruções devem ser exploradas.

6.2 Fase 2

Nessa fase as implementações em software serviram como parâmetro para o desempenho a ser batido por uma implementação híbrida. Os resultados obtidos foram consistentes com o esperado. Um julgamento da qualidade estatística dos geradores não foi testado, já que o volume de simulações não foi suficiente para gerar distorções significativas. A análise estatística ou empírica do efeito de diferentes geradores em métodos Monte Carlo para o mercado financeiro é um assunto que merece atenção.

6.3 Fase 3

Nessa fase foi desenvolvido com sucesso um coprojeto hardware/software para solução das equações de Black-Scholes com um desempenho competitivo. Esse resultado foi possível graças às ferramentas para o uso de OpenCL para FPGA disponibilizado pela Intel. Assim como na Fase 1 a implementação de outros geradores da família WELL pode ser objeto de estudos futuros. Deve ser considerado ainda uma implementação que faça melhor uso da comunicação entre as memórias do kernels, assim como no projeto de exemplo disponibilizado pela Intel. O uso de aceleração com FPGA e OpenCL também deve ser considerado para outras aplicações que utilizam métodos Monte Carlo, como em Física e Química.

Por fim a interface com a plataforma de negociação MetaTrader foi implementada com sucesso por Sockets. Essa interface é a mais simples possível, mas que serve como prova de conceito para implementações futuras de sistemas de operações automatizadas que utilizem Black-Scholes, ou qualquer outro método Monte Carlo, em suas estratégias de negociação.

REFERÊNCIAS

ALTERA. 2017. Disponível em: <https://www.altera.com/support/support-resources/design-examples/design-software/opencl/black-scholes.html>. Acesso em: 02/03/2018. Citado na página 57.

_____. 2017. Disponível em: <https://www.altera.com/products/processors/overview.html>. Acesso em: 03/04/2018. Citado na página 61.

ARSTECHINICA. 2008. Disponível em: <https://arstechnica.com/gadgets/2008/12/gpgpu-opens-up-with-opencl-1-0-spec-release/>. Acesso em: 03/04/2018. Citado nas páginas 15 e 54.

BITTWARE. 2018. Disponível em: <https://www.bittware.com/fpga/intel/boards/s5phq/>. Acesso em: 03/03/2018. Citado nas páginas 15 e 60.

BLACK, F.; SCHOLLES, M. The pricing of options and corporate liabilities. **Journal of Political Economy**, v. 81, n. 3, p. 637–54, 1973. Disponível em: <https://EconPapers.repec.org/RePEc:ucp:jpollec:v:81:y:1973:i:3:p:637-54>. Citado nas páginas 33, 36 e 37.

BONATO, V.; MAZZOTTI, B. F.; FERNANDES, M. M.; MARQUES, E. A mersenne twister hardware implementation for the monte carlo localization algorithm. **Journal of Signal Processing Systems**, v. 70, n. 1, p. 75–85, Jan 2013. ISSN 1939-8115. Disponível em: <https://doi.org/10.1007/s11265-012-0661-y>. Citado nas páginas 61 e 69.

BRENT, R. Note on marsaglia's xorshift random number generators. **Journal of Statistical Software, Articles**, v. 11, n. 5, p. 1–5, 2004. ISSN 1548-7660. Disponível em: <https://www.jstatsoft.org/v011/i05>. Citado na página 46.

CHATZIPARASKEVAS, G.; BROKALAKIS, A.; PAPAEFSTATHIOU, I. An fpga-based parallel processor for black-scholes option pricing using finite differences schemes. In: **Proceedings of the Conference on Design, Automation and Test in Europe**. San Jose, CA, USA: EDA Consortium, 2012. (DATE '12), p. 709–714. ISBN 978-3-9810801-8-6. Disponível em: <http://dl.acm.org/citation.cfm?id=2492708.2492887>. Citado na página 42.

DELCEY, T. Efficient Market Hypothesis, Eugene Fama and Paul Samuelson: A reevaluation. Working paper or preprint. 2017. Disponível em: <https://hal.archives-ouvertes.fr/hal-01618347>. Citado na página 30.

ECHEVERRIA, P.; LOPEZ-VALLEJO, M. High performance fpga-oriented mersenne twister uniform random number generator. **J. Signal Process. Syst.**, Kluwer Academic Publishers, Hingham, MA, USA, v. 71, n. 2, p. 105–109, maio 2013. ISSN 1939-8018. Disponível em: <https://dx.doi.org/10.1007/s11265-012-0661-y>. Citado na página 41.

GAMERMAN, D.; LOPES, H. **Markov Chain Monte Carlo: Stochastic Simulation for Bayesian Inference, Second Edition**. Taylor & Francis, 2006. (Chapman & Hall/CRC Texts in Statistical Science). ISBN 9781584885870. Disponível em: https://books.google.com.br/books?id=yPvECi_L3bwC. Citado na página 39.

- GHERSI, D.; PARAKH, A.; MEZEI, M. Comparison of a quantum random number generator with pseudorandom number generators for their use in molecular monte carlo simulations. v. 38, 09 2017. Citado na página 45.
- GOTHANDARAMAN, A.; PETERSON, G.; WARREN, L.; HINDE, R. J.; HARRISON, R. Fpga acceleration of a quantum monte carlo application. v. 34, p. 278–291, 05 2008. Citado na página 56.
- HOROWITZ, M. 1.1 computing's energy problem (and what we can do about it). In: **2014 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)**. [S.l.: s.n.], 2014. p. 10–14. ISSN 0193-6530. Citado na página 51.
- KNUTH, D. E. **The Art of Computer Programming, Volume 2 (3rd Ed.): Seminumerical Algorithms**. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1997. ISBN 0-201-89684-2. Citado na página 43.
- KORN, R.; KORN, E.; KROISANDT, G. **Monte Carlo Methods and Models in Finance and Insurance**. CRC Press, 2010. (Chapman and Hall/CRC Financial Mathematics Series). ISBN 9781420076196. Disponível em: <<https://books.google.com.br/books?id=TkTkgxsAoT4C>>. Citado nas páginas 41 e 45.
- KROESE, D. P.; BRERETON, T.; TAIMRE, T.; BOTEV, Z. I. Why the monte carlo method is so important today. **Wiley Interdisciplinary Reviews: Computational Statistics**, John Wiley & Sons, Inc., v. 6, n. 6, p. 386–392, 2014. ISSN 1939-0068. Disponível em: <<http://dx.doi.org/10.1002/wics.1314>>. Citado nas páginas 39 e 40.
- LEWIS, T. G.; PAYNE, W. H. Generalized feedback shift register pseudorandom number algorithm. **J. ACM**, ACM, New York, NY, USA, v. 20, n. 3, p. 456–468, jul. 1973. ISSN 0004-5411. Disponível em: <<http://doi.acm.org/10.1145/321765.321777>>. Citado na página 48.
- LOCKWOOD, J. W.; GUPTE, A.; MEHTA, N.; BLOTT, M.; ENGLISH, T.; VISSERS, K. A low-latency library in fpga hardware for high-frequency trading (hft). In: **Proceedings of the 2012 IEEE 20th Annual Symposium on High-Performance Interconnects**. Washington, DC, USA: IEEE Computer Society, 2012. (HOTI '12), p. 9–16. ISBN 978-0-7695-4831-9. Disponível em: <<http://dx.doi.org/10.1109/HOTI.2012.15>>. Citado na página 41.
- MARSAGLIA, G. Choosing a point from the surface of a sphere. **Ann. Math. Statist.**, The Institute of Mathematical Statistics, v. 43, n. 2, p. 645–646, 04 1972. Disponível em: <<https://doi.org/10.1214/aoms/1177692644>>. Citado nas páginas 46 e 47.
- MERTON, R. C. Theory of rational option pricing. **The Bell Journal of Economics and Management Science**, [Wiley, RAND Corporation], v. 4, n. 1, p. 141–183, 1973. ISSN 00058556. Disponível em: <<http://www.jstor.org/stable/3003143>>. Citado na página 37.
- MONTREAL. 2013. Disponível em: <<http://simul.iro.umontreal.ca/testu01/guideshorttestu01.pdf>>. Acesso em: 02/04/2018. Citado na página 63.
- MUSLIM, F. B.; MA, L.; ROOZMEH, M.; LAVAGNO, L. Efficient fpga implementation of opencl high-performance computing applications via high-level synthesis. **IEEE Access**, v. 5, p. 2747–2762, 2017. Citado na página 57.
- NAVABI, Z. **Embedded Core Design with FPGAs**. McGraw-Hill, 2006. (Embedded Core Design with FPGAs, v. 1). ISBN 9780071484718. Disponível em: <<https://books.google.com.br/books?id=5NEmAQAAAJ>>. Citado na página 55.

NVIDIA. 2018. Disponível em: <<https://developer.nvidia.com/curand>>. Acesso em: 02/03/2018. Citado na página 47.

ORTEGA-ZAMORANO, F.; MONTEMURRO, M. A.; CANNAS, S. A.; JEREZ, J. M.; FRANCO, L. Fpga hardware acceleration of monte carlo simulations for the ising model. **IEEE Transactions on Parallel and Distributed Systems**, v. 27, n. 9, p. 2618–2627, Sept 2016. ISSN 1045-9219. Citado na página 56.

OUYANG, J.; LIN, S.; QI, W.; WANG, Y.; YU, B.; JIANG, S. Sda: Software-defined accelerator for large-scale dnn systems. In: **2014 IEEE Hot Chips 26 Symposium (HCS)**. [S.l.: s.n.], 2014. p. 1–23. Citado na página 52.

OVTCHAROV, K.; RUWASE, O.; KIM, J.-Y.; FOWERS, J.; STRAUSS, K.; CHUNG, E. **Accelerating Deep Convolutional Neural Networks Using Specialized Hardware**. Microsoft Research, 2015. Disponível em: <<https://www.microsoft.com/en-us/research/publication/accelerating-deep-convolutional-neural-networks-using-specialized-hardware/>>. Citado na página 51.

PANNETON, F.; L'ECUYER, P.; MATSUMOTO, M. Improved long-period generators based on linear recurrences modulo 2. v. 32, p. 1–16, 03 2006. Citado nas páginas 48, 49, 69 e 71.

PATEL, C.; SRIKANTH, M.; KUMAR, K. C.; S, S. Monte-carlo black-scholes implementation using opencpl standard. v. 8, 12 2015. Citado na página 56.

PATTERSON, S. **The Quants: How a New Breed of Math Whizzes Conquered Wall Street and Nearly Destroyed It**. Crown Publishing Group, 2010. ISBN 9780307453396. Disponível em: <<https://books.google.com.br/books?id=-ydNYWGIussC>>. Citado na página 24.

PELLERIN, D.; TAYLOR, D. **VHDL Made Easy!** Prentice Hall, 1997. ISBN 9780136507635. Disponível em: <<https://books.google.com.br/books?id=No5wQgAACAAJ>>. Citado na página 55.

PEREIRA, K.; ATHANAS, P.; LIN, H.; FENG, W. Spectral method characterization on fpga and gpu accelerators. In: **2011 International Conference on Reconfigurable Computing and FPGAs**. [S.l.: s.n.], 2011. p. 487–492. ISSN 2325-6532. Citado na página 51.

PHAM, N. K.; AUNG, K. M. M.; KUMAR, A. Automatic framework to generate reconfigurable accelerators for option pricing applications. In: **2016 International Conference on ReConfigurable Computing and FPGAs (ReConFig)**. [S.l.: s.n.], 2016. p. 1–8. Citado na página 56.

SCHRYVER, C. d.; SHCHERBAKOV, I.; KIENLE, F.; WEHN, N.; MARXEN, H.; KOSTIUK, A.; KORN, R. An energy efficient fpga accelerator for monte carlo option pricing with the heston model. In: **2011 International Conference on Reconfigurable Computing and FPGAs**. [S.l.: s.n.], 2011. p. 468–474. ISSN 2325-6532. Citado na página 51.

_____. An energy efficient fpga accelerator for monte carlo option pricing with the heston model. In: **2011 International Conference on Reconfigurable Computing and FPGAs**. [S.l.: s.n.], 2011. p. 468–474. ISSN 2325-6532. Citado na página 56.

SUNDARARAJAN, P. **High Performance Computing Using FPGAs**. Bambuí, 20110. Citado na página 51.

- SZPIRO, G. G. In: _____. **Pricing the Future: Finance, Physics, and the 300-year Journey to the Black-Scholes Equation**. [S.l.: s.n.], 2011. ISBN 9780465022489. Citado nas páginas 23 e 34.
- TALEB, N. N. **Where Do Thin Tails Come From?** [S.l.], 2013. Disponível em: <<https://ideas.repec.org/p/arx/papers/1307.6695.html>>. Citado na página 31.
- TALEB, N. N. Risk neutral option pricing with neither dynamic hedging nor complete markets, a measure-theoretic proof. 04 2014. Citado na página 31.
- TERASIC. 2018. Disponível em: <<http://www.terasic.com.tw/cgi-bin/page/archive.pl?Language=English&No=529>>. Acesso em: 02/03/2018. Citado nas páginas 15 e 60.
- TEUBNER, J.; WOODS, L. Data processing on fpgas. **Synthesis Lectures on Data Management**, v. 5, n. 2, p. 1–118, 2013. Disponível em: <<https://doi.org/10.2200/S00514ED1V01Y201306DTM035>>. Citado na página 51.
- VIGNA, S. Further scramblings of marsaglia's xorshift generators. **Journal of Computational and Applied Mathematics**, v. 315, p. 175 – 181, 2017. ISSN 0377-0427. Disponível em: <<http://www.sciencedirect.com/science/article/pii/S0377042716305301>>. Citado na página 47.
- WEATHERALL, J. **The Physics of Wall Street: A Brief History of Predicting the Unpredictable**. Houghton Mifflin Harcourt, 2013. ISBN 9780547618296. Disponível em: <https://books.google.com.br/books?id=FVV0tper_0UC>. Citado na página 24.
- WEBER, R.; GOTHANDARAMAN, A.; HINDE, R. J.; PETERSON, G. D. Comparing hardware accelerators in scientific applications: A case study. **IEEE Transactions on Parallel and Distributed Systems**, v. 22, n. 1, p. 58–68, Jan 2011. ISSN 1045-9219. Citado na página 51.