Iris Ribeiro Dos Santos

Otimização de um robô de mercado ForEx usando programação paralela em arquitetura CUDA

Salvador

2017

Iris Ribeiro Dos Santos

Otimização de um robô de mercado ForEx usando programação paralela em arquitetura CUDA

Monografia apresentada ao Colegiado de Sistemas de de Informação da Universidade do Estado da Bahia como parte integrante dos requisitos necessários para obtenção do título de Bacharel em Sistemas da Informação.

Universidade do Estado da Bahia

Departamento de Ciências Exatas e da Terra I

Colegiado de Sistemas de Informação

Orientador: Prof. Dr. Cláudio Alves de Amorim

Coorientador: Prof. Dr. Diego Gervasio Frías Suárez

Salvador

2017

Iris Ribeiro Dos Santos

Otimização de um robô de mercado ForEx usando programação paralela em arquitetura CUDA

Monografia apresentada ao Colegiado de Sistemas de de Informação da Universidade do Estado da Bahia como parte integrante dos requisitos necessários para obtenção do título de Bacharel em Sistemas da Informação.

Trabalho aprovado. Salvador, 06 de Junho de 2017:

Prof. Dr. Cláudio Alves de Amorim
Orientador

Prof. Dr. Diego Gervasio Frías Suárez
Coorientador

Prof. Dr. Leandro Coelho
Convidado

Salvador
2017

Dedico este trabalho ao meu pai que não pôde me acompanhar até o fim desta vitória.

Agradecimentos

A minha mãe, cujo amor incondicional, esforço, resignação e paciência eu jamais poderei agradecer o suficiente. Ao meu pai que se faz vivo em minha memória e estará sempre presente na minha saudade. A minha família que é a minha base.

Aos meus amigos que além do apoio, cuidado, carinho e preocupação, têm sido extremamente compreensivos com a minha ausência para dedicar-me a esta pesquisa.

Aos meus colegas de trabalho que um dia já percorreram o caminho que estou fazendo agora e que entenderam o quanto essa conquista é importante para mim.

A Universidade do Estado da Bahia que me proporcionou esta oportunidade de aprendizado, aos professores, técnicos e demais colaboradores que mantêm esta organização funcionando dando sempre o melhor de si para que alunos como eu tornem-se pessoas melhores. Em especial ao meu orientador que tornou-se amigo e conselheiro dentro e fora do meio acadêmico.

Resumo

Este trabalho apresenta uma solução paralelizada na plataforma CUDA para otimização de um *robô-trader* que opera no mercado *ForEx*. Para isto foi elaborada a criação de dois ambientes simulados utilizando séries temporais reais de mercado onde o robô pode transacionar livremente, o primeiro ambiente foi desenvolvido utilizando técnicas de computação paralela na plataforma CUDA e o segundo aplicando os conceitos de programação serial convencionais na linguagem C. Para validação do modelo foram realizados testes submetendo os ambientes a variações nos parâmetros de entrada para coleta dos tempos de execução nos diferentes cenários. A métrica utilizada na avaliação dos resultados foi a de speedup, que mede o aumento na velocidade de execução do código obtido com o paralelismo em comparação com a velocidade obtida na execução serial. Os resultados apontam um *speedup* de 22 vezes, o que significa dizer que o algoritmo paralelizado conseguiu ser 22 vezes mais rápido que seu equivalente em serial, os números apresentam ganhos expressivos e servem de motivação para aprimoramento da ferramenta.

Palavras-chaves: CUDA. Computação Paralela. ForEx. GPU. Robô.

Abstract

This work presents a parallel solution in the CUDA platform for optimizing a robot-trader operating in the *ForEx* market.For this, the creation of two simulated environments using real time series of market where the robot is able to transact freely, the first environment was developed using parallel computing techniques in the CUDA platform and the second one applying conventional serial programming concepts in the C language.For model validation, tests were performed subjecting the environments to variations in the input parameters to collect execution times in the different scenarios. The metric used for evaluating the results was the speedup, which measures the increase in code execution speed obtained with the parallelism in comparison with the speed obtained in the serial execution.The results reached a speedup of up to 22 times of the environment paralleled as to the serial, which present expressive gains and serve as motivation for the improvement of the tool.

Key-words: CUDA. Parallel Computing. ForEx. GPU. Robot

Lista de ilustrações

Figura 1 – Taxonomia de Flynn. 19
Figura 2 – Sistema de Memória Compartilhada. 21
Figura 3 – Sistema Multicore UMA 22
Figura 4 – Sistema Multicore NUMA 22
Figura 5 – Sistema de Memória Distribuída 23
Figura 6 – Pipeline gráfico com estágios programáveis mapeados a matriz de processadores 26
Figura 7 – GPU NVIDIA GeForce8800 27
Figura 8 – Comparativo organização interna CPU X GPU 28
Figura 9 – Hierarquia virtual de Threads 30
Figura 10 – Comparação entre execução de kernel serial e paralelo 34
Figura 11 – Fermi Streaming Multiprocessor (SM) 36
Figura 12 – Percurso metodológico 40
Figura 13 – Fluxo de execução do robô 42
Figura 14 – Spawner 53
Figura 15 – Tempos de execução do experimento 1 - Threads fixas 55
Figura 16 – Tempos de execução do experimento 2 - Threads variáveis 56

Lista de códigos

1 Chamada para função Kernel 32
2 Obtenção do índice de uma Thread 32
3 Método princial de execução do robô - Algoritmo Serial Original 46
4 Método princial de execução do robô - Algoritmo Serial Reescrito 47
5 Método princial de execução do robô - Algoritmo Paralelo 48
6 Registro de eventos em CUDA 51
7 Obtenção das propriedades da plataforma 64
8 Alocação de memória e transferência de dados entre host e device 65
9 Validação de erros e transferência de dados do device para o host 65

Lista de tabelas

Tabela 1 – Características da GPU 50
Tabela 2 – Características dos arquivos de séries temporais 52
Tabela 3 – Speedup do experimento 1 55
Tabela 4 – Speedup do experimento 2 57
Tabela 5 – Tempos de execução do experimento 1: Threads Fixas 66
Tabela 6 – Tempos de execução do experimento 2: Threads Variáveis 66

Lista de abreviaturas e siglas

ALU	Arithmetic Logic Unit
CPU	Central Processing Unit
CUDA	Compute Unified Device Architecture
DRAM	Dynamic Random Access Memory
ECC	Error Correcting Code
ForEx	Foreign Exchange
GPGPU	General Purpose Computing on Graphics Processing Units
GPU	Graphics Processing Unit
ISO	International Standards Organization
MIMD	Multiple Instruction Multiple Data
MISD	Multiple Instruction Single Data
MMU	Memory Management Unit
NUMA	Non Uniform Memory Access
RAM	Random Access Memory
SIMD	Single Instruction Multiple Data
SISD	Single Instruction Single Data
SM	Streaming Multiprocessor
SP	Streaming Processor
UC	Unidade de controle
UGP	Unidade Gráfica de Processamento
ULA	Unidade Lógica Aritmética
UMA	Uniform Memory Access
UPF	Unidade de Ponto Flutuante

Sumário

1 Introdução 13

2 ForEx 15

2.1 O mercado 15

2.2 Séries Temporais 17

3 Computação Paralela 19

3.1 Arquitetura Paralela 19

3.2 Tipos de Paralelismo 20

3.3 Hierarquia de Memória 21

3.4 Métricas de Desempenho 23

4 GPU - Graphics Processing Unit 26

4.1 Arquitetura das GPUs 26

4.2 Comparativo CPU x GPU 27

4.3 Hierarquia de Memória da GPU 28

5 CUDA: Compute Unified Device Architecture 30

5.1 CUDA Threads 30

5.2 Hierarquia de Memória 31

5.3 Modelo de Programação 31

5.4 Arquitetura FERMI 33

6 Projeto e Implementação 37

6.1 Trabalhos Correlatos 37

6.2 Metodologia 39

6.3 Robô 40

6.4 Natureza dos Dados 42

6.5 Estratégia Serial 44

6.6 Estratégia de paralelização 44

7 Plano Experimental 50

7.1 Método 50

7.1.1 Ambiente computacional 50

7.1.2 Tempo e medidas 50

7.1.3 Dados disponíveis 52

7.2 Experimentos 54

7.2.1 Experimento 1: Threads fixas 54
7.2.2 Experimento 2: Threads variáveis 56

8 Conclusões 58

Referências 60

APÊNDICE A Códigos auxiliares 64

APÊNDICE B Detalhamento dos tempos dos experimentos 66

1 Introdução

O maior mercado financeiro do mundo em número de participantes e volume de transações é o mercado de câmbio, também chamado de *Foreign Exchange Market (ForEx)* (INVESTOPEDIA, 2016). De acordo com BANK FOR INTERNATIONAL SETTLEMENTS (2016), no ano de 2016 as operações no mercado *ForEx* movimentaram em média $5.1 trilhões de dólares ao dia somente no mês de abril. Este valor é reflexo de transações envolvendo bancos, empresas e investidores particulares que negociam moedas no mercado de balcão.

Com foco na competitividade do mercado financeiro o uso de agentes virtuais inteligentes para operar no mercado *(robô-traders)* tem sido cada vez mais empregado. Um robô é um sistema automático, baseado em matrizes de análise matemática de comportamento do mercado que buscam prever os movimentos de subida e descida nos valores das moedas (MARINI, 2009). A agência de pesquisa A.T.Kearney (2015) estima que até 2020 os robôs irão gerenciar $2.2 trilhões de dólares em investimentos devido a rápida adesão da nova geração de investidores.

Entretanto, escrever algoritmos para criação dos *robôs-traders* exige debruçar-se sobre as inúmeras variáveis que envolvem o mercado, de forma a encontrar a combinação onde se possa obter o maior lucro. Por sua vez, determinar se um robô é eficiente ou não, implica em testá-lo sob condições tão próximas quanto possível do mundo real. Uma das maneiras de fazer isso é permitir que o robô realize suas operações em um ambiente simulado tomando como entrada séries temporais reais do mercado. Em outras palavras, fazer com que o robô realize operações de compra e venda em um ambiente virtual alimentado com dados reais passados adquiridos numa janela de tempo.

A necessidade de ajustar as parametrizações do robô a cada teste, faz com que o processo demande tempo e poder computacional. Indo na contramão da necessidade do mercado, que busca otimizar o tempo, a velocidade de processamento das *CPU's(Central Processing Unit)* tem crescido de forma menos acelerada nos últimos anos de acordo com Kirk e Hwu (2012). Visando preencher esta lacuna de exigência por maior poder computacional as *GPU's (Graphics Processing Unit)* estão se tornando cada vez mais populares entre os desenvolvedores de software. Em novembro de 2006 a fabricante de placas de vídeo NVIDIA lançou a plataforma CUDA *(Compute Unified Device Architecture)*, que estabelece um modelo de programação utilizando linguagens de alto nível como C/C++ para enviar comandos a GPU. Atraindo muitos programadores, a plataforma conta hoje com uma base instalada de mais 375 milhões de GPU's habilitadas para CUDA em notebooks, estações de trabalho, clusters e supercomputadores (NVIDIA, 2017).

O objetivo geral deste trabalho é utilizar computação paralela massiva na plataforma CUDA para otimizar o tempo de aprendizagem de um robô que opera no mercado *ForEx*. Assim, será necessário a construção de dois ambientes de treinamento de robôs: O primeiro na plataforma CUDA aplicando técnicas de paralelização de algoritmos e o segundo com as técnicas seriais convencionais. Os critérios de avaliação e a metodologia proposta são descritas em detalhes ao decorrer desta monografia.

Espera-se que a construção do ambiente paralelizado diminua o tempo gasto nos testes de funcionamento do robô permitindo que o mesmo seja empregado na elaboração da estratégia. Além disso, o *know-how* empregado na pesquisa é uma aplicação prática da união entre o conhecimento adquirido na universidade e as necessidades do ambiente de negócios. Aliado a isto, do ponto de vista comercial encontrar a melhor tática em menos tempo está intimamente ligada a possibilidade de ganhos financeiros devido à natureza do tema com o qual a pesquisa lida.

Esta monografia foi dividida em 8 capítulos, sendo eles:

Capítulos 2, 3, 4 e 5 apresentam os conceitos necessários para o entendimento do projeto e perfazem o caminho desde uma breve descrição do mercado *ForEx* a computação paralela e os seus diferentes modelos, a organização física de uma GPU e finalizando com as especificidades da plataforma CUDA.

O **capítulo** 6, possui uma seção dedicada aos principais trabalhos correlatos, apresenta em detalhes o problema, o projeto e implementação para a solução, o funcionamento do robô e as estratégias utilizadas tanto no ambiente paralelizado quanto no ambiente serial.

O **capítulo** 7 relata o ambiente computacional e a maneira como os testes do modelo foram feitos, as métricas adotadas e apresenta os resultados obtidos.

Por fim, o **capítulo** 8 contém as conclusões e os relatos para trabalhos futuros.

2 ForEx

2.1 O mercado

O mercado de câmbio internacional - *foreign exchange market* ou *Forex* - é onde são realizadas as negociações de troca entre os pares de moedas. Surgindo em 1971 com o fim do acordo de *Bretton Woods* ele é considerado hoje o maior mercado do mundo em volume de transações.(INVESTOPEDIA, 2016). Ao redor do globo, investidores têm a possibilidade de comprar ou vender uma moeda por outra objetivando lucrar com as mudanças no valor das moedas que flutuam em resposta aos eventos do mercado, notícias ou puramente com a especulação. Esse mercado tem como características importantes: predominância do mercado de balcão, alto grau de liquidez, baixo grau de regulação e alta alavancagem.(ROSSI, 2010).

Não existe exigência de uma localização física para que as transações cambiais ocorram nem mesmo horário determinado para que as ordens de compra e venda sejam enviadas, funcionando diariamente através de rede eletrônica que envolve bancos, empresas, corretoras e investidores particulares que negociam moedas, sendo então caracterizado como mercado de balcão - OTC - *Over the Counter*. O *ForEx* oferece vinte quatro horas de liquidez para as principais moedas do mercado durante os cinco dias úteis da semana, iniciando suas transações em Sydney no domingo à noite dando a volta ao mundo, até o encerramento em Nova Iorque na sexta-feira. De acordo com (MARINI, 2009) em volume de negócios, o mercado cambial é o maior mercado do mundo e quando comparado ao NYSE, maior mercado acionista, com volumes diários de aproximadamente 100 bilhões de dólares, o mercado de câmbio consegue ser 30 vezes maior.

Os valores das negociações das moedas - taxa de câmbio - são determinados pelo mercado, mas elas sempre são cotadas em pares e especificadas através de uma abreviatura padronizada pela ISO (*International Standards Organization*). A abreviatura USD/BRL por exemplo, refere-se à taxa de câmbio de dólar por real. A primeira sigla é tomada como moeda base, enquanto que a segunda é a moeda de cotação. No caso, USD/BRL, é uma taxa de câmbio que especifica quantos reais é que se tem de pagar por um dólar, ou quantos reais se obtêm ao vender um dólar.

A cotação do câmbio é disposto como um par constituído por uma moeda base, *bid*: preço do comprador, e a moeda cotada, *ask*: preço do vendedor, onde o *bid* é sempre menor ou igual ao *ask*. A diferença entre o *bid* e o *ask* dá-se o nome de margem de variação ou *spread*. Quanto menor o *spread* , maior facilidade se tem de obter lucro devido às variações da taxa de câmbio, pela margem de variação pré-fixada somada a

uma volatilidade momentânea.(MARINI, 2009).Os pares recebem valores definidos pelo mercado e podem ser atualizados a qualquer instante chamado de *tick*. Um *tick* possui um único valor de *bid*, *ask*, *spread* e o horário em que os valores foram cotados. Por exemplo, no caso do USD/JPY - *Dólar/iene* - cotados a 98.27/30, o *bid* é 98.27 e o *ask* é 98.30; uma variação de 3 pontos, ou *spread* de 3.

O agente que trata das ordens dos investidores para comprar e vender divisas é chamado de *broker* ou corretor que cobra uma comissão pelo serviço que dependendo do corretor e do montante da transação, poderá ou não ser negociado. As transações ocorrem em 3 passos: **1)** O investidor comunica ao operador através da plataforma online o par de moedas e a quantia que pretende negociar. **2)** O operador revela o *bid* e o *ask*. **3)** O *trader* dá a ordem de compra ou venda, ou recusa o *spread*.De acordo com Cheng (2017) é possível classificar as ordens do mercado *ForEx* em:

- Ordens a mercado (*Market Orders*): Ordem mais comum emitida no mercado onde as operações de compra e venda acontecem com os preços de *ask* e *bid* atuais. A ordem é executada sempre ao preço do momento.
- *Stop Order*: É utilizada para comprar ou vender um par a um preço pré-determinado. É uma ordem estratégica para gestão de risco e serve para evitar perdas monetárias abaixo do nível desejado.
- Ordens limite (*Limit Order*): São executadas apenas quando um preço pré-determinado é alcançado. Tanto para venda, quanto para compra. Permite decidir quanto quer ganhar no início do *trade*.

A forma de definição de estratégias de investimento dos participantes do *ForEx* é de importância crucial, pois essas têm potencial de impactar as trajetórias cambiais no curto prazo. Por exemplo, é possível emitir uma ordem automática de fechamento para proteger a posição atual do investidor de grandes imprevistos na movimentação dos preços caso um limite x de prejuízo seja atingido (*Stop-Loss*). Do mesmo modo que pode ser usado o *Take Profit* para garantir o lucro quando o par atinge determinado valor. Para otimizar essa tarefa de definição de estratégia, as corretoras permitem a criação de *robôs-traders* que são programas de computador habilitados a operar no mercado de forma automatizada segundo as estratégias nele definidas. Antes de permitir que o robô opere no ambiente de mercado, é prudente testar o seu comportamento com dados os mais próximos possíveis do real para encontrar as melhores parametrizações nas escolhas dos valores reduzindo as chances de grandes perdas. Usualmente, os ambientes de testes utilizam-se de séries temporais do mercado *ForEx* com valores passados de: *ask*, *bid*, *spread*.

2.2 Séries Temporais

A redução da incerteza no âmbito das previsões econômicas é de especial importância dentro do mercado *ForEx*, constantemente sujeito a distúrbios em intervalos muito curtos de tempo.Existem diversas técnicas capazes de auxiliar a tomada de decisões por parte dos agentes envolvidos em atividades que necessitam de planejamento, avaliação de políticas e redução da incerteza. De acordo com Bressan (2004) existem diversas formas de se obter previsões, desde métodos puramente subjetivos (opinião de especialistas) e modelos causais ou explanatórios (econométricos), até métodos extrapolativos (séries temporais) ou mesmo uma combinação destes.

Segundo a definição apresentada por Ehlers (2009) uma série temporal é uma coleção de observações feitas sequencialmente ao longo do tempo. Dados de séries temporais são utilizados em diferentes áreas de conhecimento como na meteorologia, medicina, agropecuária, etc. A análise das séries depende de qual objetivo o estudo espera alcançar, por exemplo nos campos da sismologia e da meteorologia o objetivo da análise é a previsão, já no contexto de data mining e aprendizado de máquinas a análise pode ser utilizada para detecção de anomalias. O objetivo de uma análise exploratória por exemplo, é o de descobrir padrões e tendências inerentes ao conjunto de dados analisados. Já uma análise preditiva como o nome sugere, tem por objetivo a análise dos dados visando a predição dos valores que a variável poderá assumir.

Quanto a sua classificação as séries podem ser determinísticas ou estocásticas. Quando os valores da série podem ser escritos através de uma função matemática y = f(tempo) diz-se que a série é determinística. Quando a série envolve, além de uma função matemática do tempo, também um termo aleatório y = f(tempo,) a série é chamada estocástica.(ALVARES, 2013). Quanto as suas características o autor descreve-as como sendo:

- Estacionárias ou Não-estacionárias: Na primeira a série flutua em torno de uma mesma média ao longo do tempo. Na segunda, as séries caracterizam-se pela presença do componente de tendência e, sendo assim, apresentam comportamento de crescimento ou queda.
- Sazonal ou Não sazonal: Sazonais apresentam padrões de comportamento semelhantes em períodos regulares de tempo, diferente das não sazonais, que não apresentam tal comportamento.
- Linear ou Não Linear: Uma série pode ser dita linear caso seja possível a regressão da mesma para uma função linear. As séries que descrevem fenômenos do mundo real e que não podem ser aproximadas por são ditas não lineares.

- Univariável ou Multivariada: Séries temporais multivariadas são geradas por observações simultâneas de dois ou mais processos e mais de uma variável é levada em consideração no contexto da análise, enquanto que o termo univariada se refere as séries temporais que são obtidas de amostras de um único padrão de observação.
- Caótica: Sua principal característica é não apresentar padrão definido das observações ao longo do tempo.

Na prática é possível as séries temporais assuam mais de uma das característica acima.

É através da análise matemática que o mercado cambial encontra-se com os estudos de séries temporais podendo utilizar-se de recursos de *software* e a computação permitindo por exemplo o uso de técnicas mais sofisticadas em sua aplicabilidade como é o uso da computação paralela a ser descrita no próximo capítulo.

3 Computação Paralela

Velocidade é a palavra chave que move os avanços tecnológicos na área computacional. O homem costuma buscar máquinas mais velozes, mais baratas e normalmente menores. Por volta dos anos 2000, a frequência dos relógios dos processadores - velocidade a que um computador realiza suas operações mais básicas como somar, ou transferir informações de um registrador a outro - aumentaram de cerca de 40 MHz[1] para mais de 2,0 GHz [2] (GRAMA et al., 2003), o que leva a expectativa de que a cada novo lançamento de processadores a performance das aplicações também aumentariam. Contudo, desde 2002 a melhoria no desempenho de um processador único desacelerou cerca de 20% ao ano. (PACHECO, 2011). Desde então, a tática dos principais fabricantes de processadores tem sido colocar múltiplos processadores num único circuito integrado. O uso de múltiplos processadores trabalhando em simultâneo para resolver um único problema é a definição de paralelismo. Tamanha mudança de arquitetura dos processadores, no entanto, trouxe consequências importantes para os desenvolvedores de software. Grande parte dos programas até hoje é escrito intrinsecamente em serial, ou seja, os softwares não conseguem executar em múltiplos processadores, apenas um, deixando os demais ociosos e recaindo no problema anterior de desempenho. Como enfatizam Kirk e Hwu (2012) ao ressaltar que um programa sequencial só será executado em um dos núcleos do processador, que não se tornará significativamente mais rápido do que aqueles em uso hoje.

3.1 Arquitetura Paralela

A Taxonomia de Flynn criada em 1972 é ainda hoje a mais utilizada para classificar arquitetura de computadores paralelos. A classificação é feita de acordo com o número de fluxo de instruções e o número de fluxos de dados que podem ser gerenciados simultaneamente. São elas: *SISD*, *SIMD*, *MISD*, e *MIMD*. A figura 1 representa um resumo da taxonomia.

FLUXO DE INSTRUÇÕES		**FLUXO DE DADOS**	
		Único	Múltiplo
	Único	Single Instruction Single Data **SISD**	Single Instruction Multiple Data **SIMD**
	Múltiplo	Multiple Instruction Single Data **MISD**	Multiple Instruction Multiple Data **MIMD**

Figura 1 – Taxonomia de Flynn.
Fonte: Adaptado de Wikipedia (2016)

[1] MIPS R3000, em 1988
[2] Pentium 4, em 2002

Sistemas *SISD* executam uma única instrução de cada vez e podem buscar ou armazenar um item de dados por vez, é conhecido como o clássico modelo de Von Neumann[3] . Sistemas *SIMD* operam em múltiplos fluxos de dados aplicando a mesma instrução para vários itens de dados, portanto um sistema *SIMD* pode ser pensado como tendo uma única unidade de controle e múltiplas ULAs *(unidade lógica aritmética)*. Uma instrução é transmitida a partir da UC *(Unidade de controle)* para as ULAs e cada uma aplica a instrução para o item de dados atual. Por exemplo, se a instrução for ADD A B, cada processador irá adicionar o seu próprio valor de B ao seu próprio valor de A. Computadores *MISD* funcionam obtendo um dado global a ser processado por instruções individuais possivelmente executando em paralelo. Este modelo de execução é restritivo e de acordo com Rauber e Rünger (2013) nenhum computador paralelo comercial deste tipo foi produzido. Por fim, na *MIMD* os elementos de processamento trabalham assincronamente uns com os outros. Cada elemento do processamento - ULA - carrega uma instrução em separado e um elemento de dados separado e aplicam-se então as instruções aos elementos. A principal distinção entre os diferentes sistemas *MIMD* está na sua hierarquia de memória. Em sistemas de memória compartilhada, cada processador ou núcleo pode acessar diretamente cada local de memória, enquanto que em sistemas de memória distribuída, cada processador tem sua própria memória privada. Este tema será explicado mais a fundo na seção 3.3. Processadores multicore[4] ou sistemas de cluster[5] são exemplos de modelos *MIMD*.

3.2 Tipos de Paralelismo

De acordo com Rauber e Rünger (2013) existem 4 principais níveis de paralelismo, a serem explicados na sequência. O paralelismo no nível de bit diz respeito ao ganho de velocidade obtido na arquitetura do computador ao aumentar o tamanho da palavra do mesmo, ou seja, a quantidade de bits que podem ser processados em conjunto numa máquina. O paralelismo de pipeline[6] ou paralelismo em nível de instrução é a superposição da execução de múltiplas instruções, o que significa dizer que a execução de cada instrução é dividida em vários passos que são realizados por hardware dedicado (estágios de pipeline) um após o outro. O terceiro nível é o paralelismo de múltiplas unidades funcionais onde as diferentes unidades funcionais do processador - como ULAs, UPFs (unidade de ponto flutuante), unidades de leitura/armazenamento, ou unidades de ramificação - executam instruções independentes em paralelo aumentando a taxa média de execução de instruções.

[3] Modelo de um computador digital de programa armazenado que utiliza uma unidade de processamento (CPU) e uma de armazenamento de memória para comportar, respectivamente, instruções e dados.

[4] Consiste em colocar dois ou mais núcleos de processamento (*cores*) no interior de um único chip

[5] Formado por um conjunto de computadores, que utiliza um tipo especial de sistema operacional classificado como sistema distribuído

[6] Técnica de hardware que permite que a CPU realize a busca de uma ou mais instruções além da próxima a ser executada

O último nível de paralelismo é de tarefa ou processo e está diretamente ligada a forma como o programador escreve o seu código de forma que o processador possa executá-lo em paralelo.

3.3 Hierarquia de Memória

Acrescentar núcleos de processamento não é a única mudança arquitetural a influenciar no desempenho dos programas, mas está também intimamente ligado a capacidade do sistema de memória de alimentar dados para o processador. Entretanto, enquanto as taxas de *clock* dos processadores aumentaram em aproximadamente 40% ao ano nas últimas duas décadas, os tempos de acesso à memória dinâmica - *DRAM*[7] *(Dynamic random access memory)* - só melhoraram na taxa de aproximadamente 10 % por ano durante este mesmo intervalo de tempo de acordo com Grama et al. (2003)

Uma classificação da hierarquia de memória pode ser feita de acordo com a organização física da memória - *hardware* ou do ponto de vista lógico - *software* -. No primeiro caso tem-se outros 3 níveis de organização: 1) Computadores com a memória física compartilhada, 2) Computadores com a memória física distribuída, e 3) Organizações híbridas onde por exemplo uma memória virtual compartilhada pode ser criada no topo de uma memória fisicamente distribuída. Do ponto de vista lógico as memórias podem ter endereçamento distribuído ou endereçamento de memória compartilhado, esta abordagem não precisa estar necessariamente em conformidade com a memória física, como ilustra o exemplo dado por Rauber e Rünger (2013) onde um computador paralelo com uma memória fisicamente distribuída pode aparecer para o programador como um computador com um espaço de endereço compartilhado quando um ambiente de programação correspondente é usado.

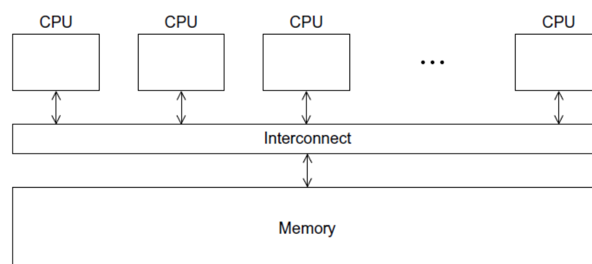


Figura 2 – Sistema de Memória Compartilhada.
Fonte: Pacheco (2011)

Nos sistemas de memória compartilhada - figura 2 - um conjunto de processadores está ligado através de redes de conexões e cada processador pode acessar um local na

[7] Tipo de memória RAM de acesso direto que armazena cada bit de dados num condensador ou capacitor

memória não no mesmo instante de tempo para garantir a integridade dos dados, o que pode tornar custoso o gerenciamento desse tipo de sistema.

Em sistemas de memória compartilhada com múltiplos processadores, *multicore*, a conexão pode ser feita ligando todos os processadores a memória principal, é o chamado acesso uniforme à memória ou *UMA (Uniform Memory Access)*. Este tipo está ilustrado na figura 3.

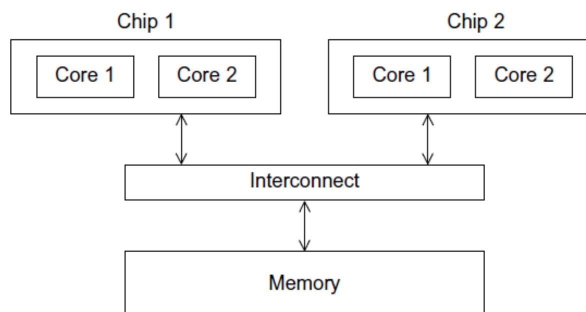


Figura 3 – Sistema Multicore UMA
Fonte: Pacheco (2011)

Cada processador pode ter uma conexão direta com um bloco de memória principal, e os processadores podem acessar blocos de memória principal uns dos outros através de *hardware* especial que compõe os processadores, este modelo por sua vez é conhecido como *NUMA* - *Non Uniform Memory Access*, ou acesso não uniforme à memória. A figura 4 ilustra este tipo de arquitetura.

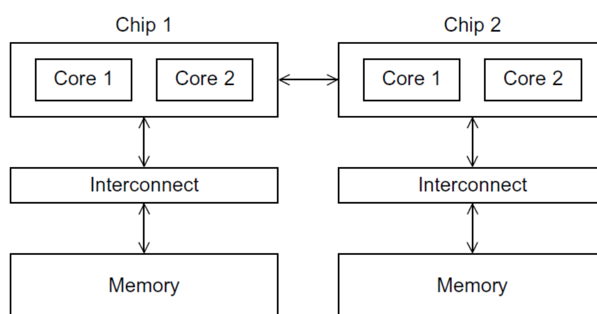


Figura 4 – Sistema Multicore NUMA
Fonte: Pacheco (2011)

Comparando estas duas últimas hierarquias - UMA e NUMA - na primeira o tempo de acesso as posições de memória será a mesma para todos os processadores enquanto que no segundo tipo o acesso a uma memória diretamente ligado a ele geralmente é mais rápido do que uma localização de memória que deve ser acessada através de outro chip.

Em sistemas de memória distribuída - figura 5 - cada processador possui uma memória privada e os pares (processador + memória) comunicam-se entre si através de

conexões de rede, assim, os processadores costumam se comunicar explicitamente enviando mensagens ou usando funções especiais que fornecem acesso a memória de outro processador. Os sistemas de memória distribuída comumente conhecidos são os chamados *clusters* , compostos por diferentes computadores através de rede *ethernet* por exemplo.

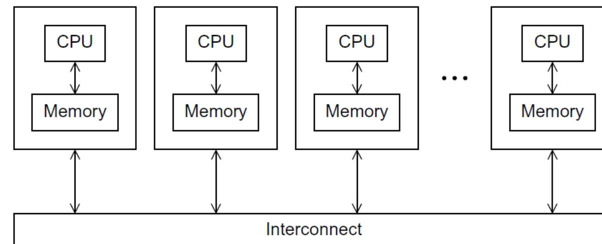


Figura 5 – Sistema de Memória Distribuída
Fonte: Pacheco (2011)

3.4 Métricas de Desempenho

Ao avaliar o desempenho de programas em paralelo foram criadas algumas métricas específicas para este tipo de paradigma. A primeira e talvez mais importante delas, seja o *speedup*, traduzindo livremente para o português, aceleração. Por definição, o número de *speedup* expressa quantas vezes um programa em paralelo é mais rápido que um programa seqüencial ao tentar resolver o mesmo problema. Na melhor das possibilidades espera-se que ao dividir o trabalho realizado pelos *cores* do processador sem adicionar nenhum trabalho extra obterem-se um *speedup* linear.

Para exercício de entendimento propõe-se que o programa irá executar com p *cores*, onde cada *thread* ou processo encontra-se também num único *core*, acredita-se que o programa em paralelo irá funcionar p vezes mais rápido que a mesma versão em serial executando a mesma tarefa, sob as mesmas condições. Denomina-se então o tempo serial de execução como T_{serial}, e o tempo de execução em paralelo como $T_{paralelo}$ e p como sendo o número de *cores* utilizados, obtendo a seguinte expressão:

$$T_{paralelo} = \frac{T_{serial}}{p}$$

De acordo com a lei de Amdahl [8] ainda que num sistema paralelo ideal não é possível obter um *speedup* linear porque cada programa em termos de tempo de execução possui uma fração y que não pode ser paralelizada e tem que ser executada sequencialmente por um único processador.

[8] É um modelo de *speedup* esperado sobre a relação entre implementação paralelizada de um algoritmo e sua implementação sequencial, sob a suposição de que o tamanho do problema continua a ser o mesmo quando paralelizado

Acrescenta-se ainda o fator de sobrecarga do sistema gerado pela inclusão de múltiplos processos *threads*.Programas executando em memórias compartilhadas devem utilizar algum mecanismo de exclusão mútua para garantir a integridade dos dados que estão sendo acessados. As chamadas para esse gerenciador é uma sobrecarga que não está presente na execução em serial. Ao utilizar programas de memória distribuída como alternativa ao uso de memória compartilhada acrescenta-se o tempo de troca de mensagens entre a rede, que também não está presente em aplicações em serial. Sendo assim, o *speedup* não linear pode ser calculado como:

$$S = \frac{T_{serial}}{T_{paralelo}}$$

Já a eficiência de um sistema em paralelo quantifica o número de operações válidas exercidas pelos processadores durante o tempo de execução em paralelo e pode ser obtido através da fórmula:

$$E = \frac{T_{serial}}{T_{paralelo} * p} * 100\%$$

Existem muitas abordagens diferentes na hora de calcular $T_{paralelo}$ e T_{serial}. Pode-se por exemplo utilizar o tempo que o programa gastou ao ser executado do começo ao fim, ou o tempo total gasto em código executado por parte do programa, o que incluiria o tempo gasto em funções de bibliotecas por exemplo.

Outra métrica importante na avaliação de um programa paralelizado é o custo de execução que representa o número de processadores totais utilizados para resolver o problema, multiplicado pelo tempo de execução em paralelo, como expresso utilizando a fórmula abaixo, lembrando que para sistemas sequenciais o custo vai ser equivalente ao tempo de execução.

$$C_{paralelo} = p * T_{paralelo}$$

Ainda a respeito das métricas, a escalabilidade de um sistema está em encontrar uma taxa de aumento no tamanho do programa de forma que ele seja sempre eficiente. Em outras palavras, se ao aumentar o tamanho do problema com a mesma velocidade que aumentamos o número de processos *threads* a eficiência será inalterada o programa é considerado escalável.

A granulosidade de um sistema paralelo consiste em encontrar o equilíbrio entre a quantidade de computação utilizada e a quantidade de comunicação associada a ela. Se as tarefas a serem executadas forem muito pequenas os custos com comunicação podem superar o ganho obtido com a computação paralela. Por outro lado, poucas tarefas muito grandes mantêm a CPU constantemente ocupada não sendo possível balancear as tarefas

do sistema adequadamente. A grande questão aqui gira em torno do nível de granulosidade que o sistema deve ter para que se obtenha ganhos reais com a paralelização.

Uma vez familiarizados com os diferentes aspectos da computação paralela, é preciso retomar a problemática que abriu este capítulo: velocidade de execução dos programas versus arquitetura dos processadores versus como os programas são escritos. Pacheco (2011) observa que transformar um programa serial, em um programa paralelo por exemplo, não implica necessariamente num ganho de performance. Os programas devem primeiramente ser paralelizáveis. Uma implementação paralela eficiente de um programa serial não pode ser obtida por encontrar paralelizações eficientes de cada um dos seus passos.

O Capítulo seguinte trata de uma técnica eficiente de paralelização que faz uso de recursos do *hardware* para resolver A técnica de paralelização que Uma técnica eficiente de paralelização é a utilização de *hardwares* específicos aplicados a computação paralela visando obter o máximo de desempenho no desenvolvimento do software para resolução de um problema. O capítulo a seguir trata desta técnica.

4 GPU - Graphics Processing Unit

A UGP - Unidade Gráfica de Processamento ou *GPU (Graphics Processing Unit)* é um processador paralelo dedicado, otimizado para acelerar computações gráficas e projetado especificamente para a execução de muitos cálculos essenciais para renderização de gráficos 3D de ponto flutuante. Nas últimas décadas no entanto, os fabricantes dessas placas gráficas têm feito com que este dispositivo tornem-se também processadores paralelos programáveis de alto desempenho, contando com largura de banda de memória, capacidade de cálculo e eficiência no consumo de energia, o que tornaram as GPUs uma alternativa a processadores comuns dando origem ao termo: *GPGPU (General Purpose Graphics Processing Unit)*. Para funcionamento de uma GPU é importante começar pelo *pipeline* de gráficos e depois discutir como este dispositivo tornou-se tão atrativo na execução de programas de propósito geral.

4.1 Arquitetura das GPUs

As GPUs utilizam *pipelines* de processamento gráfico, onde as entradas do *pipeline* são vértices num sistema de coordenadas tridimensional, e convertem a representação interna em uma matriz de *pixels* que podem ser enviadas para a tela do computador. Este processo ocorre em cinco estágios: **1)** Operações de vértice: cada vértice deve ser transformado no espaço da tela e sombreado por sua interação com as luzes da cena; **2)** Montagem de primitivas: os vértices são montados em triângulos; **3)** Rasterização: determinam-se quais *pixels* do espaço da tela são cobertos por um determinado triângulo; **4)** Operações dos fragmentos: Utiliza-se o valor de cor de cada vértice e cada fragmento é sombreado para determinar sua cor final; e por fim a **5)** Composição: produz-se a imagem final com uma cor para cada *pixels*. É importante ressaltar que nenhuma destas etapas eram programáveis, só era possível ao programador controlar as posições de vértices e suas respectivas cores e as luzes de cena. Os modelos que determinavam as interações ficavam fisicamente codificados no *hardware*. A figura 6 representa os estágios acima citados.

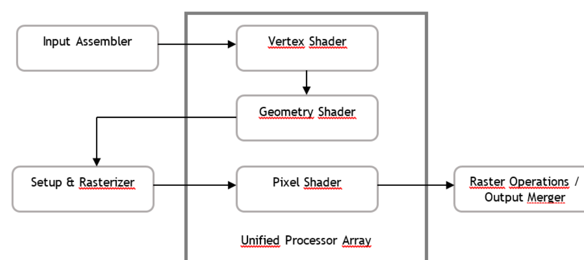


Figura 6 – Pipeline gráfico com estágios programáveis mapeados a matriz de processadores
Fonte: Adaptado de Guillen et al. (2012)

Os autores Guillen et al. (2012) mostram que a principal modificação da arquitetura nestes últimos anos foi substituir as funções fixas para os vértices e segmentos por programas que executam para cada vértice e fragmento. Atualmente as GPUs trabalham com o chamado modelo de *shaders* ou seja, várias etapas de *pipeline* são agora programáveis e o comportamentos desses estágios é especificado por funções chamadas *shader*, ou funções de sombreamento.

A arquitetura da GPU divide o *pipeline* em espaço e as saídas de cada tarefa alimentam as entradas da tarefa seguinte, assim, os recursos do processador são divididos entre as tarefas e uma parte do processador que está trabalhando em uma tarefa pode alimentar diretamente a outra parte do mesmo que está executando uma tarefa seguinte. A desvantagem desse sistema é balancear o volume das operações do *pipeline* pois a vazão depende do estágio mais lento. Para contornar esta situação as GPUs modernas compartilham uma unidade programável de *hardware* concentrada no paralelismo de dados assim é possível obter um melhor balanceamento as custas de um maior custo de *hardware*.

A *GeForce 8800* foi a primeira unidade gráfica a possuir o modelo unificado de *shaders*, em outras palavras uma unidade totalmente programável chamada de *SM (Multithreaded Streaming Multiprocessor)*. A Figura 7 mostra em detalhes a arquitetura desta placa.

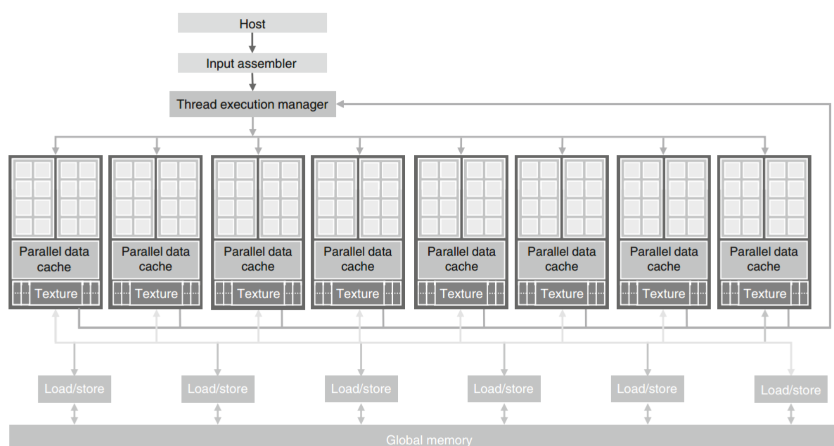


Figura 7 – GPU NVIDIA GeForce8800
Fonte: Kirk e Hwu (2012)

4.2 Comparativo CPU x GPU

A primeira grande diferença entre as placas, encontra-se na arquitetura física dos componentes, a arquitetura das GPUs é baseada numa matriz de muitos processadores programáveis enquanto que as CPUs são baseadas em múltiplos núcleos, isso porque o projeto de uma CPU é otimizado para o desempenho do código sequencial, fazendo uso de unidades de controle lógicas sofisticadas para permitir que as instruções de um

único segmento executem em paralelo ou mesmo fora da ordem sequencial, mas mantendo a aparência de execução sequencial. Mais importante ainda sobre as CPUs é que elas possuem grandes memórias cache que reduzem latências de instrução e acesso a dados de aplicações complexas.

As GPUs por outro lado estão focadas em executar várias *threads* paralelamente mapeadas em múltiplas unidades de execução. A organização deste *hardware* é uma matriz de *SPs (Streaming Processors Cores)* distribuídos ao longo de alguns SMs. Cada SP por sua vez, possui unidades lógicas aritméticas de inteiros e pontos flutuantes. É importante destacar que o conceito de *threads* é superficialmente diferente no contexto das GPUs já que são menos custosas e de fina granulosidade ao passo que o desempenho individual é mais fraco.

Deve ficar claro que os dois tipos de *hardware* foram criados para funções específicas diferentes e que uma pode não executar bem a tarefa da outra, portanto espera-se que a maioria das aplicações irão utilizar CPUs e GPUs trabalhando em conjunto onde as partes sequenciais são exploradas pela CPU e as partes numericamente intensivas executando na GPU. A figura 8 deixa explícita essas diferenças na arquitetura.

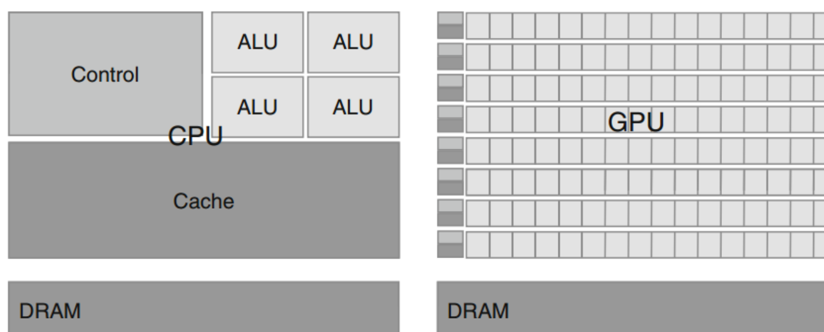


Figura 8 – Comparativo organização interna CPU X GPU
Fonte: NVIDIA (2017)

4.3 Hierarquia de Memória da GPU

Normalmente é configurado no cartão gráfico uma hierarquia completa com vários níveis de memória: memória principal fora do chip ou *DRAM (Dynamic Random Access Memory)*, memória compartilhada, memória local, e memória de constantes. Os subsistemas de memória do dispositivo são organizados em partições de memória que compreendem um controlador de memória independente e um ou dois dispositivos DRAM exclusivos a placa. Esta memória externa permite a comunicação entre os diferentes blocos de *threads* de forma que o endereçamento é repartido entre todas as partições de memória.

Visando maximizar a disponibilidade de dados para a execução das *threads* as GPUs também contam com as memórias *cache (Streaming Cache Architecture)*, que são

utilizadas para aumentar a eficiência do acesso a memória através de grandes blocos. Além disso, GPUs modernas são capazes de traduzir endereços virtuais em físicos tornando necessária a inclusão de uma unidade *MMU (Memory Management Unit)*.

Na GPU as *threads* compartilham memória que fica dentro dos processadores, o que torna essa memória visível apenas para este bloco de *threads*. Uma das vantagens deste tipo de memória compartilhada é que por estar dentro do chip, não existe contenção elevada para seu acesso, além de permitir uma alta largura de banda para sustentar as demandas de cada SP (GUILLeN et al., 2012). As memórias locais são visíveis para uma *threads* individual , já o banco de registradores é compartilhado entre as *threads*.

Um modelo específico de GPU utilizada como unidade de processamento gráfico de propósito geral é apresentada no capítulo seguinte.

5 CUDA: Compute Unified Device Architecture

Lançada no mercado em novembro de 2006 pela NVIDIA, CUDA é um modelo de plataforma de computação e programação paralela para resolver problemas computacionais complexos de forma mais eficiente que na CPU, de acordo com o fabricante. Como modelo de programação a plataforma utiliza principalmente extensões das linguagens de programação C e C++ visando diminuir a curva de aprendizado do programador. Este por sua deve preocupar-se com três níveis chaves de abstração: hierarquia de grupos de *threads* , compartilhamento de memórias e barreira de sincronização. Este modelo também permite grande escalabilidade uma vez que o sistema pode possuir uma ou mais GPUs, cada uma com quantidades de multiprocessadores diferentes e ainda assim a aplicação consegue se adaptar a arquitetura do *hardware*.

5.1 CUDA Threads

Na documentação fornecida pela (NVIDIA, 2017) existe uma clara distinção entre *threads* de CPU e CUDA Threads para lembrar os programadores que apesar de semelhantes existem diferenças profundas no funcionamento de ambas. No decorrer do texto será adotado o termo Threads para referir-se a CUDA Threads e *threads* ao fazer referência as convencionais. As GPUS são capazes de lançar e executar em paralelo milhares de Threads, porém sua performance individual é baixa e de pouca flexibilidade se comparada a *threads* comuns.

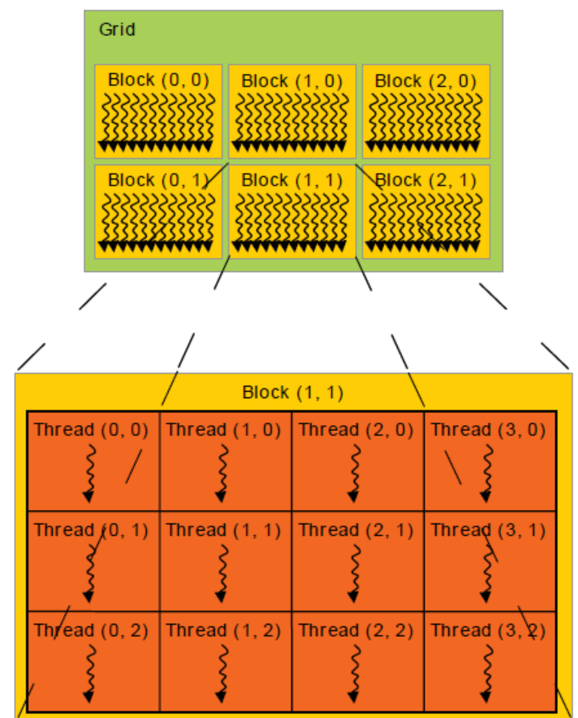


Figura 9 – Hierarquia virtual de Threads

Fonte: NVIDIA (2017)

Em CUDA, existe uma hierarquia virtual muito bem definida na criação e execução de Threads, não existe comunicação direta entre elas - o que pode ser contornado fazendo uso de alguma memória compartilhada. Cada Thread está contida em um bloco de *threads*, que podem possuir até três dimensões e consequentemente até três coordenadas geográficas

por Threads dentro do bloco. Os blocos por sua vez podem estar agrupados em uma grid de até duas dimensões, atribuindo assim duas coordenadas de identificação para os blocos. A figura 9 é uma representação desta hierarquia interna.

Todas as Threads da plataforma CUDA iniciam a sua execução através da chamada da função Kernel, padrão adotado pela NVIDIA na extensão da linguagem C. Uma função Kernel quando chamada é executada N vezes por N diferentes Threads na GPU.

Apesar de conseguir executar muito mais *threads* que uma CPU, existe um limite para o número de Threads que podem ser executadas simultaneamente dentro de um SM. O conjunto de Threads em execução dentro de um SM num dado instante de tempo é chamado de *warp* e possui valor máximo de 32. Quando a um multiprocessador é dado um ou mais blocos de Threads para executar, este as divide em *warps* e cada um é agendado para execução. Um *warps* executa uma instrução comum ao bloco por vez, assim dizemos que um *warp* é 100% eficiente se todas as Threads possuem o mesmo caminho de execução. Threads que estão fora do caminho são bloqueadas e esperam o seu momento de execução, uma vez que todos os caminhos foram percorridos todos eles voltam para a mesma ramificação de *warp*. É tarefa do programador entender o funcionamento do *hardware* e com isso modelar a solução de forma a evitar quebra de fluxo na execução dos *warps* sempre que possível.

5.2 Hierarquia de Memória

O modelo expresso pela plataforma CUDA propõe uma separação entre a memória da GPU e a memória principal da máquina e ainda que não haja uma separação física como no caso de uma GPU integrada, o programador deverá fazer uma cópia dos dados entre as memórias. Toda Thread possui três principais espaços de memória para realizar leitura e gravação de dados, sendo eles: memória local privada restrita a cada Thread, memória compartilhada dividida por todas as Threads de um bloco e uma memória global que é comum a todas as Threads inclusive aquelas que executam em diferentes funções Kernel. No caso das memórias locais e compartilhadas, o tempo de vida dos dados armazenados é igual ao tempo de vida da Thread. No caso da memória global o tempo de vida é igual ao tempo de execução do programa.

5.3 Modelo de Programação

Enquanto modelo de programação a plataforma CUDA possui algumas especificidades na escrita do código que influenciam diretamente nas decisões de implementação.

Em CUDA, o *host* refere-se à CPU e sua memória, enquanto o *device* se refere à GPU e sua memória. Existem 3 formas de declarar uma função em CUDA:1) _ _device_

__ são executadas pelo *device* e chamadas pelo próprio. 2)__ __host__ __ executadas no *host* e chamadas pelo mesmo. 3)__ __global__ __ são executadas pelo *device* e chamas pelo *host*, para esta última dá-se o nome de Kernel. O conjunto de Threads de um Kernel define um grid que por sua vez é subdividida em blocos de Threads. Todas as Threads em uma mesma grade executam o mesmo Kernel.

Um grid pode ser bidimensional possuindo valores em x e y definidos pelas variáveis de sistema *blockIdx.x* e *blockIdx.y*. As Threads por sua vez podem ser tridimensionais e são elas que determinam a dimensão do bloco. É na chamada da função Kernel que as dimensões e as quantidades são criadas através das variáveis *gridDim* e *blockDim*.Os valores máximos das dimensões dependem do modelo da placa de vídeo em que o código é executado, mas, o compilador acusa erro quando esses valores excedem os disponíveis.

O código 1 abaixo ilustra uma chamada de função Kernel.

```
1  __global__ void kernel(void){
2      int x = 0;
3  }
4
5  int main( void ) {
6      dim3 grid(100,100);
7      dim3 block(254,100,1000);
8      kernel<<<grid,block>>>();
9  }
```

Listing 1 – Chamada para função Kernel

Assim como os blocos, as Threads também possuem um identificador único que deve ser obtido calculando os valores disponíveis pelas variáveis de sistema *threadIdx.x*, *threadIdx.y* e *threadIdx.z*. Para calcular o índice de uma Thread é preciso levar em consideração a dimensão do grid e das Threads. O trecho de código 2 exemplifica este cálculo.

```
1  //1D grid of 1D blocks
2  threadIndex = blockIdx.x *blockDim.x + threadIdx.x;
3
4  //2D grid of 2D blocks
5  blockIndex = blockIdx.x + blockIdx.y * gridDim.x;
6  threadIndex = blockId * (blockDim.x * blockDim.y)
7  + (threadIdx.y * blockDim.x) + threadIdx.x
```

Listing 2 – Obtenção do índice de uma Thread

Os blocos gerados ao chamar um Kernel são entregues ao gerenciador dentro de um multiprocessador (SM), sendo distribuídos entre os multiprocessadores pelo gerenciador

da GPU que por sua vez realiza o balanceamento da carga. Internamente, os blocos ainda são separados em *warps*, que também são controlados pelo gerenciador dentro do multiprocessador.

Com CUDA, é possível criar um número de *threads* muito maior do que o suportado pelo *hardware*. Para solucionar problema de gerenciamento de *threads* acima do permitido, um controlador de *software* gerencia e cria as Threads que serão executadas. Como não se tem controle sobre o gerenciador, não é possível afirmar qual delas executará primeiro, esta informação é de suma importância quando o problema exige que uma determinada Thread processe a informação e passe para a próxima, para este tipo de situação CUDA possui uma função chamada _ _ *synchthreads()* que sincroniza todas as Threads dentro do bloco fazendo com que seja criada uma barreira até que todas cheguem ao ponto de sincronização. Este método deve ser executado com cautela pois fazer com que todas as Threads sincronizem significa fazê-las esperar até que a última alcance a barreira, incrementando o tempo total de processamento e deixando o *hardware* ocioso.

Como *host* e *device* possuem espaços de memória separados para que um obtenha dados do outro é necessário realizar transferências entre os dispositivos. Normalmente como os dados a serem processados encontram-se na memória do *host* é ele quem gerencia a alocação de memória através da função *cudaMalloc()*. Uma vez que a memória tenha sido alocada é preciso inicializar a variável com os valores contidos no *host*. O comando utilizado é o *cudaMemcpy()* que funciona exatamente igual a função padrão em C *memcpy()* exceto que ele tem um quarto argumento que especifica a direção da cópia, podendo ser *cudaMemcpyHostToDevice* ou *cudaMemcpyDeviceToHost*.

Frequentemente a quantidade de memória disponível no *device* costuma ser bastante reduzida em relação a do *host* por isso é preciso que o programador haja com cautela ao alocar memória dentro da GPU para evitar erros com endereçamento de memória mas principalmente devido ao custo computacional e ao tempo que essas transferências demandam. Num estudo realizado por Barr (2015) para cálculo de transposição de matrizes utilizando GPU 97% do tempo total de execução do programa foi gasto com transferências do *host* para o *device* e de volta para o *host*.Assim como em C é responsabilidade do programador liberar a memória que foi alocada, em CUDA não é diferente e a função correspondente é a *cudaFree()*.

5.4 Arquitetura FERMI

Para o melhor entendimento da solução a ser apresentada neste trabalho é preciso conhecer mais a fundo a plataforma CUDA com destaque para a placa de vídeo em questão sob a qual os testes serão realizados, afinal, a velocidade com que os cálculos da GPU são executados está intimamente ligado ao modelo e a arquitetura sob a qual a aplicação

paralelizada irá executar. Neste trabalho a placa de vídeo é do modelo GeForce820M na arquitetura Fermi, fabricada pela NVIDIA. Especificações técnicas em torno dos testes serão detalhadas na seção 7.1.1.

As primeiras GPU's baseadas na arquitetura FERMI foram construídas com cerca de 3 bilhões de transistores possuindo até 512 núcleos CUDA, sendo que cada um deles executa instruções de inteiro ou ponto flutuante para ciclo de *clock* por Threads. A fim de comportar os 512 núcleos em questão eles são distribuídos em 16 SM's de 32 núcleos cada. Do ponto de vista de organização de memória a GPU possui 6 partições de memória de 64 bits, para uma interface de memória de 384 bits, suportando um total de até 6 GB de memória DRAM GDDR5. Uma interface de *host* conecta a GPU a CPU via PCI-Express[1]. O agendador global *gigathread* distribui blocos de Threads para os respectivos agendadores presentes em cada SM. Cada SM tem 16 unidades de carga / armazenamento, permitindo que os endereços de origem e de destino sejam calculados para até 16 Threads por *clock*.

Toda essa configuração interna de *hardware* permite a placa suportar a execução concorrente do Kernel, onde diferentes Kernels do mesmo contexto de aplicação podem ser executados na GPU ao mesmo tempo, possibilitando assim que os programas que executam um número pequeno de Kernels utilizem toda a GPU. Para efeito comparativo, a figura 10 ilustra exemplos de execução de *kernel* em serial e em paralelo.

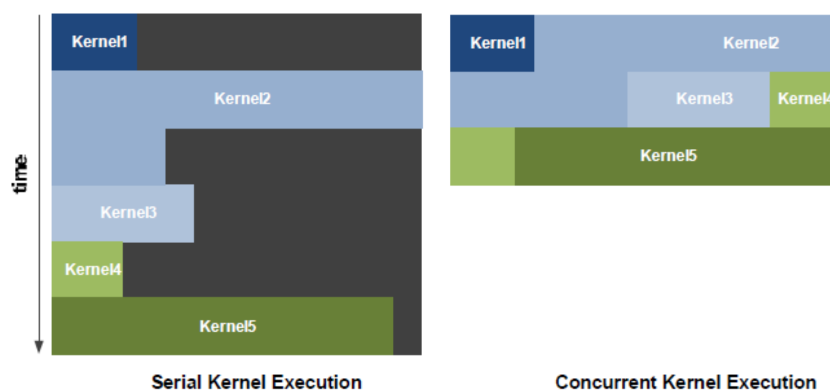


Figura 10 – Comparação entre execução de kernel serial e paralelo
Fonte: NVIDIA (2009)

Cada SM possui 32 processadores CUDA e cada um desses núcleos possui uma unidade lógica aritmética inteira (ALU) e uma unidade lógica de ponto flutuante capazes de suportar técnicas de *pipeline*. Uma característica positiva a ser destacada a respeito da placa é que esta arquitetura implementa o novo padrão de ponto flutuante IEEE 754-2008 fornecendo a instrução *Fused (Multiply-add)* fundida para aritmética de precisão simples e dupla, em outras palavras fazendo a multiplicação e adição com um único passo de

[1] Barramento ponto a ponto, onde cada periférico possui um canal exclusivo de comunicação com o chipset (MORIMOTO, 2011)

arredondamento final, sem perda de precisão na adição. Em Fermi, a ULA inteira recém concebida suporta total precisão de 32 bits para todas as instruções, consistente com os requisitos de linguagem de programação padrão. E é também otimizada para permitir com eficiência operações de precisão de 64 bits e estendidas.

Uma das inovações arquitetônicas das placas baseadas no modelo FERMI é a memória compartilhada no chip. A memória compartilhada permite que os segmentos dentro do mesmo bloco de Threads cooperem, facilitando o reuso extensivo de dados no chip e reduzindo consideravelmente o tráfego fora do chip visto que a memória compartilhada é um facilitador chave para muitas aplicações CUDA de alto desempenho.

A arquitetura Fermi implementa um único caminho de solicitação de memória unificado para cargas e armazenamentos, com um *cache* L1 por SM e um *cache* L2 unificado que fornece suporte a todas as operações (carga, armazenamento e textura). O *cache* L1 por SM é configurável para suportar tanto a memória compartilhada quanto o *cache* de operações de memória local e global. A memória de 64 KB pode ser configurada como 48 KB de memória compartilhada com 16 KB de *cache* L1 ou 16 KB de memória compartilhada com 48 KB de *cache* L1.

Fermi é a primeira GPU a suportar a proteção baseada em *ECC (Error Correcting Code)* dos dados na memória. A radiação que ocorre naturalmente pode causar um bit armazenado na memória a ser alterado, resultando em um erro suave. A tecnologia ECC detecta e corrige erros de *software* de bit único antes que eles afetem o sistema. Como a probabilidade de tais erros induzidos por radiação aumenta linearmente com o número de sistemas instalados, o ECC é um requisito essencial em instalações de grandes *clusters*. Fermi suporta *Single-Error Correct Double-Error Detect (SECDED)* Códigos ECC que corrigem qualquer erro de bit único no *hardware* à medida que os dados são acessados. Além disso, o SECDED ECC garante que todos os erros de bit duplo e muitos erros de vários bits também sejam detectados e relatados para que o programa possa ser reexecutado em vez de ser permitido continuar executando com dados incorretos.

Os arquivos de registro da Fermi, memórias compartilhadas, *cache* L1, *cache* L2 e memória DRAM são protegidos por ECC. Além disso, ela também suporta os padrões da indústria para verificação de dados durante a transmissão de chip para chip. Todas as GPUs NVIDIA incluem suporte para o padrão PCI Express para verificação CRC com repetição na camada de enlace de dados. A placa Fermi suporta o padrão GDDR5 semelhante para verificação CRC com repetição - também conhecido como EDC - durante a transmissão de dados através do barramento de memória. A figura **??** é uma ilustração da organização interna de um SM de uma placa FERMI.

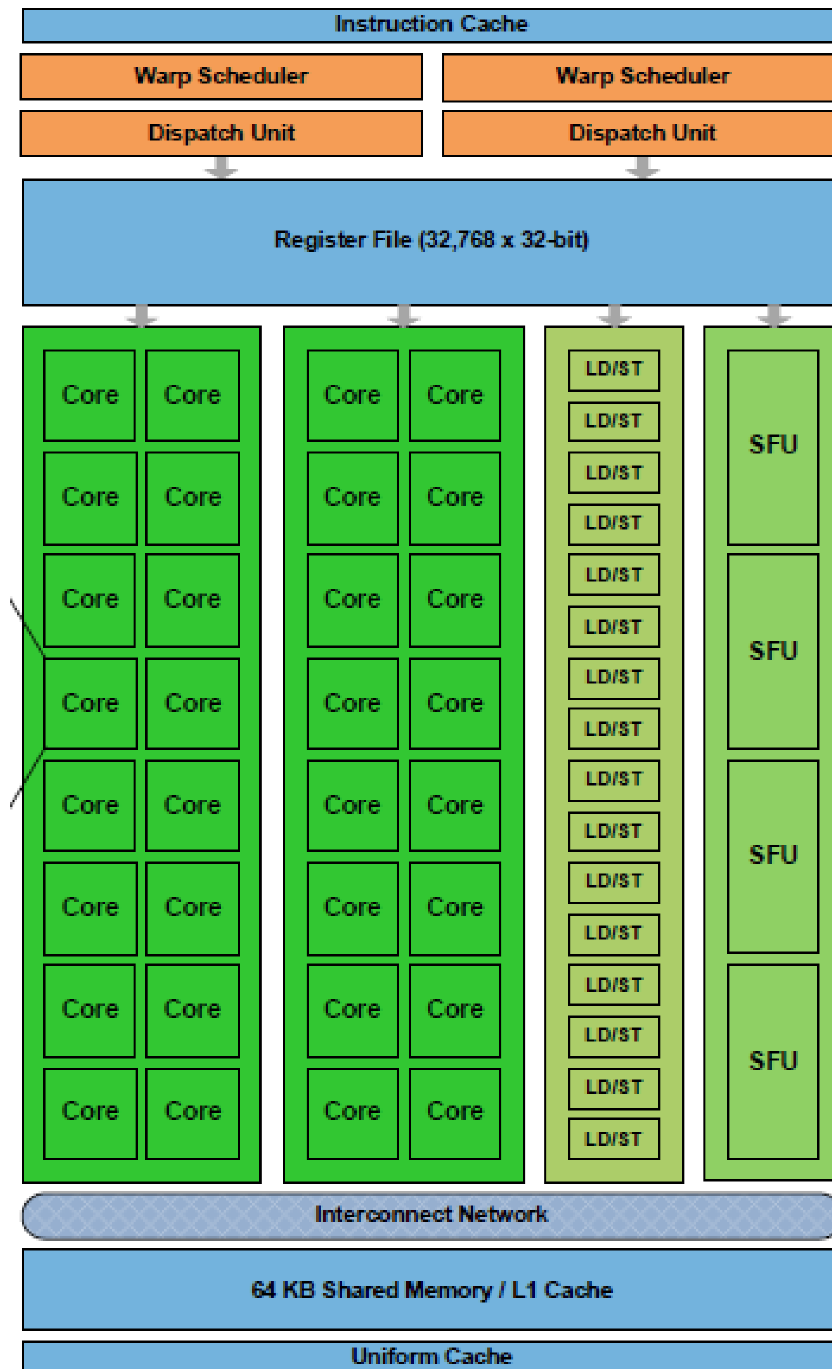


Figura 11 – Fermi Streaming Multiprocessor (SM)
Fonte: NVIDIA (2009)

6 Projeto e Implementação

6.1 Trabalhos Correlatos

No intuito de auxiliar o desenvolvimento desta monografia, foram pesquisados alguns trabalhos que possuem semelhanças com o objetivo aqui proposto avaliando questões como: a viabilidade do problema, a melhor forma de solucioná-lo, as boas práticas no desenvolvimento da arquitetura, quais falhas mais comuns, etc. Dessa forma, ainda que os artigos a serem detalhados não lidem diretamente com o problema desta pesquisa, eles foram fundamentais na construção da metodologia a ser descrita nas próximas seções.

No artigo *"Accelerating Financial Applications on the GPU"*(GRAUER-GRAY et al., 2013), os autores propõem acelerar aplicações financeiras que utilizam a biblioteca QuantLib. Para isto, 3 tipos de aplicações financeiras foram analisadas: Repo[1], Bonds[2] e Options [3], sendo esta última dividida em dois modelos matemáticos, técnicas de Black-Sholes[4] e Monte-Carlo[5]. Os 4 algoritmos foram reescritos manualmente na plataforma CUDA e OpenCL[6]. Versões em HMPP[7] e OpenACC[8] também foram testadas, porem estes códigos foram gerados de forma automática. No total 6 versões de código foram criadas para cada aplicação: CPU, CUDA, OpenCL, HMPP, OpenACC e OpenMP. Para os testes foram avaliados a quantidade de linhas de código gerada para cada umas das versões, bem como o tempo de execução dos algoritmos, tendo excluído os tempos de geração de dados de entrada e a impressão de saídas. Focando especificamente na plataforma CUDA, o tempo de execução quando comparado a versão serial, foi inferior, tornando-o mais rápido, entretanto, houve um incremento de complexidade ao código devido as transferências de dados entre o *host* e o *device*. Os autores sinalizam alguns contratempos nesta versão do código devido a necessidade de sincronização entre as Threads, alocação e

[1] Abreviação de acordo de recompra.(INVESTOPEDIA, 2017b)

[2] Investimento de dívida em que um investidor empresta dinheiro a uma entidade (normalmente corporativa ou governamental).(INVESTOPEDIA, 2017a)

[3] Derivado financeiro que representa um contrato vendido por uma parte (o autor da opção) a outra parte (o titular da opção)(INVESTOPEDIA, 2017d)

[4] Modelo matemático do mercado de um ativo, no qual o preço do ativo é um processo estocástico (WIKIPEDIA, 2017)

[5] Técnica de modelagem multivariada específica que permite aos pesquisadores executar múltiplos ensaios e definir todos os resultados potenciais de um evento ou investimento (INVESTOPEDIA, 2017c)

[6] Padrão aberto para programação em paralelo e em plataforma de diversos processadores encontrados em computadores pessoais, servidores, dispositivos móveis e plataformas embutidas. (KHRONOS, 2017)

[7] Modelo de programação projetado para lidar com aceleradores de hardware sem a complexidade associada à programação GPU (REVOLVY, 2017)

[8] É um modelo de programação paralelo com desempenho baseado em diretivas baseado no usuário (OPENACC, 2017)

liberação de memória do *device*. É possível aprender com os autores que apesar do acréscimo de tempo gerado com transferências de dados entre os dispositivos a implementação paralela consegue ser mais rápida que sua versão serial, inclusive nos experimentos todas as versões paralelizadas conseguiram ser mais rápidas que a versão serial. O artigo e este trabalho possuem como semelhança o objetivo de acelerar aplicações do mercado financeiro utilizando paralelismo, contudo, operações do mercado *ForEx* ficaram de fora do estudo dos autores.

O artigo *"Fast Computation of Stock Market Indices Using GPUs"*(INDUWARA; JAYASENA, 2013) é a descrição da tentativa dos autores em calcular o índice do mercado de ações em tempo real, utilizando computação paralela, mais especificamente na plataforma CUDA. Diferente da pesquisa aqui proposta, os autores não utilizaram uma implementação em serial para efeito comparativo de desempenho, visto que seria inviável realizar tal feito de acordo com eles, devido a necessidade de inúmeros servidores com grande poder computacional para que o índice possa ser calculado em tempo real. Por isso, as comparações foram feitas utilizando um algoritmo serial que não calcula o índice em tempo real. Na fase de testes foram usadas variações de índice de 100 a 1000 e os números de constituintes do índice variando de 100 a 10.000, sempre de um em um, o que representa uma massa de dados bem grande. Para cada variação de parâmetros e conjunto de dados um *timestamp* foi associado ao início e ao fim de cada evento. Vale destacar que a medida que a quantidade de conjunto de dados testados aumentou a latência da GPU na plataforma CUDA permaneceu praticamente inalterada, enquanto as demais cresceram, evidenciando assim uma vantagem dessa implementação em lidar com uma grande quantidade de tarefas. A semelhança entre esta monografia e o arquivo citado encontra-se na natureza dos dados a serem computados pois ambas lidam com um número grande de variações tomados como parâmetros de entrada a serem calculados na fase de teste dos experimentos. O aprendizado adquirido, portanto, está nos resultados obtidos pelos autores, visto que na implementação escolhida o desempenho da GPU conseguiu ser constante mesmo aumentando o número dos parâmetros de teste.

No artigo *"Simulações financeiras em GPU"* (SOUZA, 2013) o autor apresenta um estudo matemático e computacional de modelagem estocástica em finanças utilizando a GPU como plataforma de aceleração. São realizados dois estudos de caso de problemas em finanças: Stop ótimos, utilizando estratégia de negociação de ações de *stop gain* e cálculo de risco de mercado. Os testes foram realizados na plataforma CUDA, onde os tempos de execução do código foram obtidos de acordo com temporizadores implementados no código e que fazem parte do pacote CUDA, nesses tempos ainda foram considerados tempos gasto com transferência entre o *host* e o *device*. A métrica utilizada na validação dos testes foi o *speedup* e o autor conclui apresentando resultados onde as aplicações paralelizadas obtiveram uma aceleração de mais 50 vezes em comparação com o sequencial. A diferença aqui presente entre os trabalhos está na escolha do tipo de mercado, enquanto o autor

utilizou-se de dados do mercado de ações, a proposta desta monografia por sua vez é manipular dados do mercado ForEx. Entretanto, a metodologia escolhida na avaliação dos resultados e testes desta monografia foi retirada do artigo em questão. As funções CUDA de cálculo de tempo, a descrição do plano experimental e a apresentação em forma de gráficos foram inspirados no artigo.

Neste artigo *"Parallel Implementation of Moving Averages and Stock Market Prediction"* (JENQ, 2012) o autor utiliza a GPU para executar redes neurais artificiais no intuito de prever preços do mercado de ações. Redes do tipo *feedforward* e Backpropagation são utilizadas para o estudo, entretanto, em controversa com a expectativa o *speedup* obtido com a implementação paralela empregando os esforços da CPU e da GPU, não foi significativo em relação a implementação sequencial executada apenas pela CPU. O autor destaca que a necessidade de sincronização das Threads foi o que mais consumiu tempo na aplicação. Ao tentar diminuir o uso desta função aumentando a quantidade de chamadas ao método Kernel o *speedup* não foi significativamente maior. Aliado a isso, quando o tamanho do conjunto de dados tomados como entrada era muito pequeno, as transferências de dados entre as memórias do *host* e do *device* consumiam muito tempo, fazendo inclusive com que a implementação em serial obtivesse um *speedup* melhor. Apesar deste trabalho também ter sido montado na plataforma CUDA a principal diferença entre o artigo apresentado e esta monografia é a não utilização de redes neurais, ainda assim, através deste texto fica evidente que ainda que as GPU's consigam executar milhares de tarefas ao mesmo tempo é necessário estudar com cautela se o tipo de problema pode ser paralelizado e qual a melhor forma de fazê-lo, pois nem sempre a paralelização implica em diminuição do tempo de execução.

Através destes artigos é possível notar alguns pontos em comum que foram destacados pelos autores a respeito da plataforma CUDA, a primeira delas é o tempo gasto com as transferências de dados entre a memória do *host* e do *device*, lembrando que são feitas, na maioria dos casos, duas transferências, a primeira no sentido *host*/*device* e a segunda na direção inversa para copiar de volta a CPU os dados processados. O segundo ponto diz respeito a quantidade de informação a ser computada na GPU, pois quando a quantidade de tarefas é muito pequena a CPU pode obter um melhor desempenho. O desafio então é encontrar um equilíbrio entre o volume de dados a ser processado e o tamanho dos dados a serem transferidos entre os dispositivos a fim de justificar o uso da GPU.

6.2 Metodologia

Para desenvolver a hipótese de paralelização do robô de mercado *ForEx* na plataforma CUDA, se fez necessário o emprego de um percurso metodológico ilustrado na figura 12. As seções seguintes descrevem cada uma destas etapas.

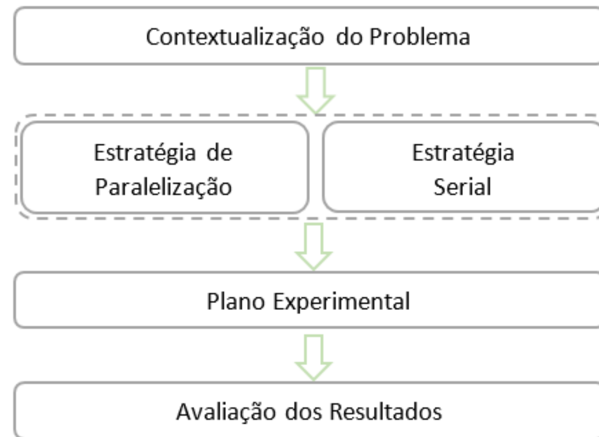


Figura 12 – Percurso metodológico
Fonte: Elaborado pela autora

A primeira etapa é descrita em duas partes compreendidas pelas seções 6.3 e 6.4 que tratam respectivamente do funcionamento do robô e da natureza dos dados utilizados pelo mesmo.

As seções 6.6 e 6.5 abordam as estratégias empregadas na escrita dos ambientes paralelo e serial

Dando continuidade ao percurso metodológico o plano experimental devido sua complexidade foi melhor organizado dentro do capítulo 7 abrangendo também a seção 7.2 que contém os resultados coletados.

6.3 Robô

Um robô é um negociador automatizado que possui autonomia para realizar operações de mercado no lugar do investidor. Em específico, o papel deste robô escrito pelo grupo de pesquisa PIMAT (Predições Inteligentes em Métodos Aplicados a Séries Temporais) da Universidade do Estado da Bahia é decidir qual o melhor momento de mercado para abrir e fechar ordens de compra/venda, ou quando apenas monitorá-las. A estratégia utilizada pelo robô bem como quais técnicas foram adotadas para a escrita do mesmo, não serão aqui discutidas pois fogem do âmbito da pesquisa que se limita a tornar os processos de teste destas estratégias mais rápidas.

Para testar a eficiência da estratégia utilizada pelo robô o PIMAT criou um ambiente simulado na linguagem de programação C que consiste na leitura de um arquivo do tipo .txt contendo series temporais com dados reais de mercado com valores de: *Ask*, *Bid*, *Spread*, Data, Hora e *Timestamp*. Estas variáveis já foram detalhadas no capítulo 2. Cada linha do arquivo - ou *tick* - pode conter dados no formato: *Ex.: 1.38763 1.38761 0.00002 2014-05-05 12:00:01 1399291201*, por exemplo.

A estratégia de execução programada no robô para decidir o melhor momento para abrir/fechar/monitorar uma ordem considera as seguintes variáveis:

- IRV: valor de referência de intervalo para suporte e resistência.[9].
- SRC: valor de suporte ao intervalo
- Target Profit: lucro almejado.
- Max Loss: valor máximo de perda aceitável.
- Trigger Value: define o valor para que a estratégia de trigger possa ser iniciada.
- Trigger Gap: define a distância do valor atual de mercado para o valor de proteção da estratégia de trigger.

Variações nas combinações destes valores resultam em diferenças nos lucros que podem ser obtidos nas negociações, é por isso que quanto maior o número de combinações testadas maior a possibilidade de encontrar aquela onde os lucros obtidos são maiores. Para os testes da estratégia um conjunto com combinações desses valores, é tomado como base, por exemplo: *1 1 999 15 7 5*.

O fluxo de execução do robô é ilustrado na figura 13 abaixo e funciona da seguinte maneira: **Etapa 1:** Um conjunto de valores das variáveis acima descritas é fornecido como input para o robô. **Etapa 2:** O arquivo contendo as séries temporais com os valores de mercado num intervalo de tempo é carregado em memória. **Etapa 3:** Para cada linha do arquivo o robô deve decidir se: A) Monitora o mercado, B) Identifica oportunidades para emitir ordens de compra ou venda, C) Monitora um ordem aberta alterando os parâmetros para decidir o melhor momento de fechá-la e D) Fecha uma ordem quando os critérios são cumpridos. **Etapa 4:** É chegado o fim do arquivo de séries temporais e o robô finaliza sua execução. Ao chegar ao fim do arquivo qualquer ordem aberta é imediatamente fechada ainda que nenhum critério tenha sido atingido, desta forma evita-se que dados inconsistentes gerados por uma ordem aberta façam parte das estatísticas de desempenho da estratégia. Para cada novo conjunto a ser testado como input o processo é reiniciado.

[9] Conceito de que o movimento do preço de um título tenderá a parar e reverter a determinados níveis de preços predeterminados. (U, 2017)

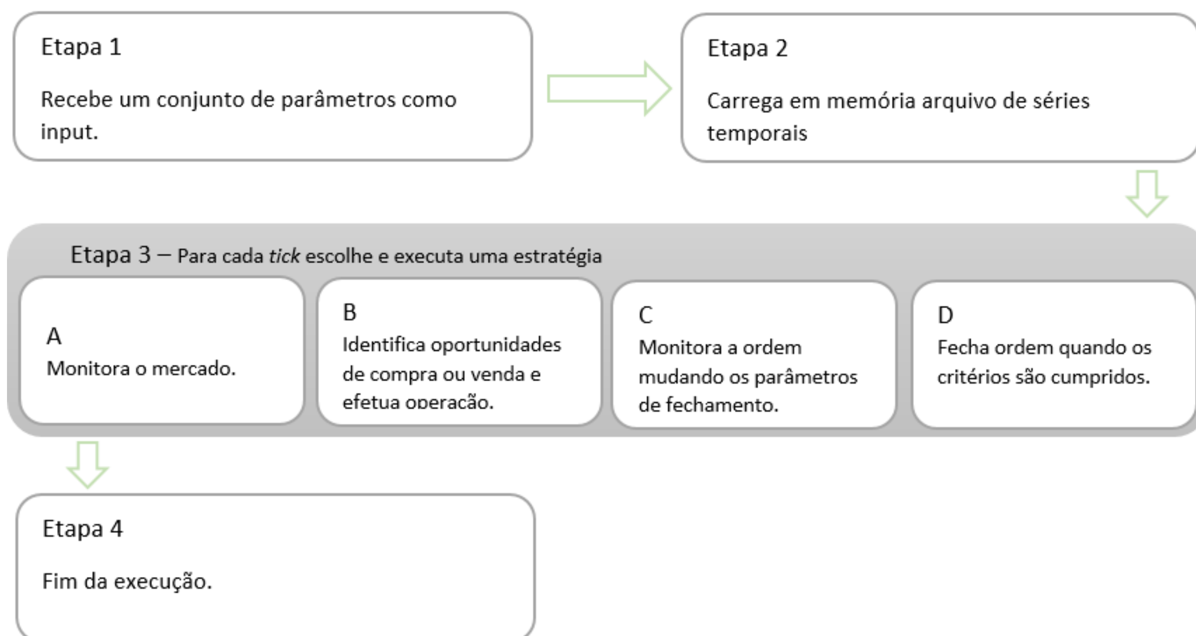


Figura 13 – Fluxo de execução do robô
Fonte: Elaborado pela autora

Para cada execução do robô 4 métricas derivadas são coletadas: **1.Lucro total:** é o valor obtido ao subtrair o montante perdido e a comissão[10], do lucro obtido. **2.Eficiência:** ou fator de compensação, ele é encontrado através da razão entre o lucro total e a quantidade de ordens que foram abertas. **3.Fator de lucro:** é a razão entre o total de ganhos e o total de perdas. **4.Taxa de sucesso:** que é a razão entre o número de ordens de sucesso sob o total de ordens executadas. Dentro do ambiente de testes da estratégia existe ainda uma outra fase que é chamada de pós-processamento cujo objetivo é determinar qual conjunto de parâmetros obteve as melhores métricas. Esta etapa de pós-processamento não é aqui discutida pois não entra na delimitação do escopo da pesquisa e não foi paralelizada.

6.4 Natureza dos Dados

Tendo entendido o comportamento do robô vamos supor um possível cenário de mercado onde para cada segundo no tempo têm-se apenas uma linha no arquivo de séries temporais, este é um cenário idealizado apenas para efeito ilustrativo pois neste intervalo de tempo é possivel que o mercado emita N valores de *Ask* , *Bid* e *Spread*. Partindo desta suposição num intervalo de 1 minuto, o arquivo teria *60* linhas. Num intervalo de uma hora seriam $60 * 60 = 3600$ linhas , se o intervalo for de um dia o cálculo seria $24 * 60 * 60 = 86400$ linhas, como o mercado funciona 5 dias na semana, isso equivale a $5 * 24 * 60 * 60 = 432000$ linhas. Levando também em consideração que os parâmetros

[10] Montante a ser pago para a corretora que intermedia as operações

utilizados como entrada podem assumir apenas um único valor por vez dentro do conjunto existem no mínimo 720 [11] possibilidades a serem testadas por vez. Vale destacar que para cada novo conjunto de parâmetros o robô reinicia o processo, assim, é possível notar ao menos dois problemas latentes: **a)** Existe uma grande demanda computacional em termos de processamento de informação visto o volume de dados e **b)** Tempo gasto na obtenção de algum resultado para um único *dataset* de testes.

Sob a perspectiva de análise do problema mencionado este é o tipo de situação que pode ser resolvido de forma mais rápida quando vários computadores são utilizados ao mesmo tempo. Remetendo ainda a um cenário idealizado, imagine que o robô iré executar os seus testes com o *dataset* x no coputador A. Ao mesmo tempo, ou seja, paralelamente, o robô executará no computador B sob o *dataset* y, e assim sucessivamente até que sejam utilizados 720 computadores ao mesmo tempo. O tempo de execução de um único *dataset* de testes não diminuirá, em compensação, com este mesmo tempo é possível obter 720 resultados. Como os processamentos são independentes é possível concluir que por definição a paralelização deste processo poderia torná-lo mais rápido.

Retomando o conhecimento sobre arquitetura paralela explicado na seção 3.1 é razoável acreditar que o problema proposto pode ser resolvido via implementação numa arquitetura do tipo SIMD, onde unidades paralelas compartilham a mesma instrução - fluxo de execução do robô - mas realizam estas operações em diferentes elementos de dados - conjuntos de parâmetros. Utilizar uma grande quantidade de computadores para realizar esta tarefa, além de financeiramente dispendioso, envolvem questões como:o local de armazenamento das máquinas, pessoas qualificadas para gerenciar estes recursos e etc. CUDA por sua vez consegue concentrar milhares de processadores dentro da GPU de forma relativamente barata, oferecendo a vantagem para o programador de utilizar linguagens estruturadas como C e C++ no desenvolvimento do código. Mas o que torna a plataforma CUDA conveniente para resolução deste problema, é o fato de sua arquitetura ser baseada no tipo SIMD, onde suas Threads individualmente fazem parte de grupos maiores chamados *warps* dentro do qual cada Thread executa exatamente a mesma seqüência de instruções. Esta arquitetura em específico recebeu o nome de *SIMT - Single Instruction Multiple Thread* onde *single instruction multiple data (SIMD)* é combinada com *multithreading*.

Relacionando os conceitos de computação paralela e CUDA, numa analogia com o cenário descrito é como se cada Thread ficasse responsável por executar um robô com um *dataset* em um dos inúmeros processadores na GPU ao mesmo tempo.

[11] $6! = 720$

6.5 Estratégia Serial

Devido as mudanças inerentes a paralelização que foi acrescentada ao ambiente de testes do robô, algumas modificações no código serial foram necessárias para evitar que os resultados obtidos na fase de coleta de dados sejam incompatíveis gerando distorções que podem favorecer ou desfavorecer uma implementação em detrimento da outra.

A versão anterior do ambiente exigia que para cada novo conjunto de parâmetros a ser testado o arquivo de séries temporais deveria ser lido e carregado a memória RAM novamente. Este procedimento não mais ocorre. Assim como na implementação paralelizada os arquivos são lidos e armazenados em *arrays* de *structs* evitando assim que ao ler uma nova linha no arquivo de parâmetros seja preciso carregar os arquivo de séries temporais em memória novamente. Ao invés disso, o uso dos vetores dentro de um laço, permite que apenas o índice do laço seja reiniciado voltando então para a primeira linha do arquivo com os valores de mercado.

Apesar da quantidade de memória disponível para uso da CPU não ser uma preocupação latente não é possível assumir que um arquivo de qualquer tamanho pode ser lido em memória e processado de uma única vez. Para garantir que o programa funcione da maneira correta sob estas condições um método foi acrescido ao código de forma a dividir o arquivo de séries temporais em partes de tamanho máximo ao disponível no *hardware*, este valor no caso é de 8GB.

A implementação sequencial fornecida pelo grupo de pesquisa PIMAT continha uma etapa de pós processamento da coleta que consistia em escrever os valores dos testes em um arquivo de texto a ser lido por um programa externo que por sua vez desenha gráficos em tela baseados nesses valores. Como dito anteriormente a paralelização da etapa de pós processamento não faz parte do escopo deste trabalho, e portanto foi removido em ambas implementações, tanto da serial quanto da paralela. Outros ajustes menores quanto a refatoração e limpeza do código também foram feitos, mas que em nada interferem no funcionamento do mesmo e por isso não foram aqui descritos.

6.6 Estratégia de paralelização

O algoritmo foi escrito de forma a adaptar-se as diferentes arquiteturas de CUDA para tornar a solução o mais flexível e desprendida do *hardware* quanto possível. Para tanto, alguns cuidados na implementação foram tomados. O mais importante deles diz respeito ao gerenciamento de memória disponível para ser utilizada pela GPU. CUDA possui uma estrutura de dados chamada *cudaDeviceProp* que contém os atributos das principais características da plataforma que podem ser obtidos através do método *cudaGetDeviceProperties()* ilustrado no código 7 localizado no apêndice A na linha 7.

Remetendo ao funcionamento do robô, duas novas etapas foram acrescentadas que é a divisão do arquivo de séries temporais do mercado em blocos de forma a caber na memória da GPU, e uma etapa de leitura do arquivo contendo os parâmetros de input sob os quais o robô será testado. O primeiro passo então é saber quanto de memória um único bloco pode utilizar por vez, e este valor é obtido em *prop.sharedMemPerBlock* - linha 19 do código 7 - no caso do *hardware* de teste este número é de 65536 Bytes, como o arquivo de séries temporais possui tamanho variável a depender da quantidade de *ticks* não é possível envia-lo de uma única vez para processamento pois a memória da GPU pode não ter espaço suficiente para tal, causando erro no processamento.

A solução encontrada foi dividir o arquivo em tantos blocos quanto forem necessários até que todo o arquivo tenha sido processado, buscando ocupar sempre o máximo de memória disponível por bloco. Outra informação que deve ser obtida é qual o espaço de memória utilizada por uma única linha do arquivo, assim é possível saber qual o máximo de linhas que podem ser processadas de uma única vez de forma a caber na memória compartilhada pelo bloco. A quantidade de memória ocupada por um *tick* é de 56bytes. O cálculo que deve ser feito então é: $\frac{65536}{56} \cong 1170$. Considera-se que o número de linhas é um valor inteiro e por isso as casas decimais são ignoradas e o valor arredondado para baixo para. Com estes valores definidos, o loop que ler o arquivo de series temporais preenche um *array* com os valores do mercado e invoca o método de processamento assim que o limite máximo de linhas definido é atingido.

Para o arquivo que contém o conjunto de parâmetros como *input*, tiveram os seus valores associados ao número de Threads lançadas por bloco numa única dimensão, significa dizer que cada Thread é responsável por percorrer o pedaço do arquivo de séries temporais enviado comportando-se cada uma como uma execução de um robô, cada Thread passa pelas etapas de 1 a 4 descritas na seção 6.3. A propriedade que define a quantidade máxima de Threads por bloco é a - linhas 22 e 23 do código- *prop.maxThreadsPerBlock*. Ainda que virtualmente seja possível executar de uma só vez mais Threads do que o permitido por bloco é necessário levar em consideração que esta escolha adiciona um custo ao gerenciador de SM consumindo memória e poder computacional que podem ser evitados. Como o valor máximo de Threads por bloco na GPU de testes é de 1024 então este é o número máximo de linhas do arquivo de *input* que pode ser enviado para processamento por vez. Apesar de ser limitado este número não é pequeno, significa dizer então que numa única passagem do robô podem ser realizados até 1024 conjuntos de teste de uma única vez.

Com os blocos e as Threads definidas a próxima etapa de execução é alocar memória no *host* para inicialização das variáveis, alocar memória no *device* que sejam correspondentes aquelas do *host* e em seguida copiar estes dados de uma variável pra outra, vale relembrar que este é considerado o maior ponto negativo da plataforma CUDA visto

que tais transferências são extremamente custosas para a aplicação e as que consomem também mais tempo. O trecho de código 8, localizado no apêndice A foi retirado da função onde é feita a transferência. Seguindo as boas práticas de programação em CUDA por padrão variáveis iniciadas por *h_* pertencem ao escopo de memória do *host* e as iniciadas por *d_* pertencem ao escopo de memória do *device*. Outra prática adotada é o uso da função *checkCudaErrors()* sempre que um método CUDA for invocado, sem ela sempre que um erro acontece no código por uso indevido das funções feita pelo programador, a execução é imediatamente interrompida e não é possível saber qual o ponto do código responsável pela falha. Com o uso desta função disponível através da biblioteca *helper_cuda.h* a linha, o código de erro e a descrição são exibidos como *output*.

Ainda devido a limitação na quantidade de memória existente na GPU, sempre que uma variável é alocada no *device* é preciso verificar se ainda há espaço na memória global do dispositivo. A propriedade que fornece este valor é *prop.totalGlobalMem* - linha 13 do código 7. A alocação da memória do bloco, somado a alocação da memória das Threads, somado a alocação de todas as variáveis utilizadas pelo programa em execução no *device* não podem ser superior ao valor total disponível na GPU que neste caso é de 2GB. Por isso, sempre que uma variável termina a sua execução é necessário liberar memória no *device* através do método *cudaFree()*. Para entender melhor o comportamento dos ambientes e as diferenças entre cada uma das implementações cabe uma análise do método principal de execução do robô. O código 3 refere-se a implementação original que foi desenvolvida pelo grupo de pesquisa PIMAT, o trecho de código 4 é também serial que foi apresentado na seção 6.5. O código 5 é por fim o código paralelizado.A execução interna do robô segue o mesmo fluxo que já foi descrito na seção 6.3 etapa 3, e por isso não cabe aqui repeti-la.

```
1  for(Tick=0 ; Tick < FileTicksCount ; Tick++){
2
3          if (OppenedOrders > 0) {
4                  OrdersControl(Tick);
5          }else{
6                  if ((Market[Tick].Spread) > SPREAD){
7                  //Não faz nada
8                  }else{
9                          if (ENABLETRADE==1){
10                                 Execution(Tick);
11                         }
12                 }
13         }
14 }
```

Listing 3 – Método princial de execução do robô - Algoritmo Serial Original

No código 3 para cada linha no arquivo de séries temporais -*tick* - uma variável global é verificada para saber se existem ordens abertas, em caso afirmativo um novo método para controle de ordens é chamado. Caso não existam orderns abertas, ele deve verificar então se o *spread* do mercado - contido no arquivo - é maior que o *spread* fornecido como input do parâmetro, em caso afirmativo nada é feito e o robô pode então lê o próximo *tick*, quando isto não ocorre ele verifica se o parâmetro ENABLETRADE está habilitado como verdadeiro, quando sim o método de execução de ordens é chamado, é dentro deste método que as ordens de compra/venda são acionadas. Note que tanto o método *OrdersControl()* quanto o *Execution()* recebem como parâmetro apenas o indice que corresponde a linha do arquivo de séries temporais, isso porque todas as demais variáveis envolvidas na estratégia possuem escopo global.

```
1  for (i = 0; i < paramCombCount; i++)
2  {
3          for(Tick= 0; Tick< amountOfLinesToProcess; Tick++)
4          {
5                  if (orders.Status == ORDER_STATUS_OPEN)
6                  {
7                          OrderMonitoring(data, params, Tick, i, statisticData, orders);
8                  }
9                  else if (((data[Tick].Spread) <= params[i].Spread)
10                 && (params[i].EnableTrade == TRUE))
11                 {
12                         Execution(data, Tick, i, statisticData,
13                         params, referenceMarketValue, newAnalisys, orders);
14                 }
15         }
16         CleanVariables(orders, statisticData);
17 }
```

Listing 4 – Método princial de execução do robô - Algoritmo Serial Reescrito

O trecho de código 4 é por sua vez possui elementos que não estavam presentes no código original. O primeiro deles é um *loop* que foi acrescentado, como dito anteriormente, o robô não estava preparado para receber mais que um conjunto de parâmetros de *input* por vez, então, com o acréscimo da etapada de leitura do arquivo de parâmetro este *loop* teve que ser acrescentado para que cada conjunto a ser testado execute a leitura das séries de mercado do inicio ao fim. A leitura deste arquivo não precisa ser feita para cada novo conjunto de parâmetros, após a primeira leitura ela já se encontra carregado em memória e para ir ao inicio do arquivo basta reiniciar o indice do *array*. A variável global contendo a quantidade de ordens abertas foi substituida pela *struct* que corresponde a ordem em si e o seu atributo *status* indica se a ordem está aberta ou não. O método *OrdersControl()* presente no algoritmo original foi removido pois ele continha apenas a chamada para o

método *OrderMonitoring()* que por sua vez recebe como parâmetros as variáveis de escopo local necessárias para monitoramento de uma ordem. Caso não exista ordem aberta é a vez de verificar se o *spread* é menor ou igual ao informado no conjunto de paramêtros na linha i, e o mesmo ocorre para a variável EnableTrade. Perceba que a construção da estrutura de decisão foi montada de acordo com o resultado oposto ao que acontece no fluxo original, para evitar que nada seja feito diante da condição da testada, como ocorria na linha 5. O método *Execution()* continua sendo invocado, entretanto como as variáveis não possuem mais o escopo global, elas são enviadas como parâmetro para a função. Um método para limpar os valores das variáveis de estatistica e ordem foi acrescentado ao fim de cada ciclo de leitura do arquivo para garantir que valores passados não interfiram no novo ciclo de execução com um novo conjunto de testes.

```
1  for (long int Tick = 0; Tick < tickCount; Tick++)
2  {
3          if (orders[threadIndex].Status == ORDER_STATUS_OPEN)
4          {
5                  OrderMonitoring(data, params, Tick, threadIndex, statisticData, orders);
6          }
7          else if (((data[Tick].Spread) < params[threadIndex].Spread)
8          && (params[threadIndex].EnableTrade == TRUE))
9          {
10                 Execution(data, Tick, threadIndex, statisticData, params,
11                 referenceMarketValue, newAnalisys, orders);
12         }
13 }
```

Listing 5 – Método princial de execução do robô - Algoritmo Paralelo

No algoritmo paralelo 5 não existe a necessidade um *loop* que corresponda as linhas do arquivo de parâmetros. Como cada Thread corresponde a um ciclo de execução do robô, ao chamar o método Kernel que contém o trecho de código paralelo as Threads executam o algoritmo como se este fosse um único robô. O *array* contendo os *datasets* de parêmtros de teste foi prenchido de forma que cada linha corresponda a um indice de Thread, assim quando as Threads executarem é possivel acessar os valores através do número do indice que é calculado como explicado na seção 5.3. É exatamente este detalhe de remoção do *loop* mais externo a principal diferença entre as implementações paralelo e serial, é ele que garante que o código do robô será executado pelas Threads ao mesmo tempo ao invés de um de cada vez.

Com a execução do robô concluída é necessário transferir os valores das estatísticas coletadas no processamento, do *device* para o *host* e dar continuidade a etapa de pós processamento. Antes de chamar a função *cudaMemcpy()* algumas outras chamadas a métodos CUDA se fazem necessárias para garantir que não houve falhas na execução

do código. A primeira delas é *cudaPeekAtLastError()* que retorna o último erro que foi produzido por qualquer uma das chamadas em tempo de execução no *host* , a principal vantagem deste método é que ele para a execução sempre que um erro é encontrado não permitindo que o programa continue executando. A segunda função utilizada é a *cudaThreadSynchronize()* função que bloqueia o dispositivo até que todas as Threads tenham concluído sua execução, a utilização deste método garante que os dados do *device* sejam copiados para o *host* apenas quando toda a execução do Kernel for finalizada, como este é um método assíncrono é necessário garantir que sua execução foi terminada. Por fim, a função *cudaDeviceSynchronize()* bloqueia o *device* até que todas as funções Kernel invocadas pelo *host* tenham sido executadas. Ao término dessas chamadas de função é possível então copiar com segurança as informações processadas pelo *device* para o *host* como ilustra o código 9 do apêndice A.

Com os resultados no *host* é a vez da CPU unir os resultados gerados pelo bloco que foi processado, afinal, as estatísticas devem corresponder a um arquivo completo, e não a uma parte dele. Para tanto, uma variável foi acrescida a *struct* que contém os parâmetros e recebe o valor da linha correspondente, é um identificador único daquele conjunto de parâmetros e ao fim de cada parte do processamento os resultados coletados são somados ao do processamento anterior cujo identificador seja correspondente. Todo o processo é repetido tantas vezes quanto necessário até que o arquivo de séries temporais chegue ao fim.

7 Plano Experimental

7.1 Método

Este capítulo busca validar a implementação da arquitetura proposta e para tanto descreve o ambiente computacional bem como a configuração e execução dos experimentos a serem apresentados nas seções seguintes. As configurações aqui enunciadas aplicam-se a todos os experimentos realizados a menos que seja dito o contrário.

7.1.1 Ambiente computacional

GPU

O código em GPU foi desenvolvido em CUDA C aplicando o compilador nvcc (CUDA Toolkit v8.0). Foi utilizada uma placa gráfica NVIDIA GeForce 820M de arquitetura FERMI (ver seção 5.4). O ambiente de desenvolvimento escolhido foi o Visual Studio 2015. A tabela 1 apresenta um resumo das principais características da GPU.

Nvidia GeForce 820M	
Compute Capability	2.1
Total Global Mem	2147483648 Bytes
Total Constant Mem	65536 Bytes
Multiprocessor Count	2
Shared Mem Per Mp	49152 Bytes
Threads in Warp	32
Max Threads per Block	1024
Max Thread Dimensions	1024, 1024, 64
Max Grid Dimensions	65536, 65536, 65536

Tabela 1 – Características da GPU

CPU

O código em CPU foi desenvolvido em C/C++ utilizando o compilador visual C++ 14.0 em um processador Intel Core I5-5200 CPU com base em x64. O dispositivo possui 8GB de memória RAM. O ambiente de desenvolvimento foi também o Visual Studio 2015.

7.1.2 Tempo e medidas

Para medição dos tempos de execução dos experimentos é possível utilizar os tradicionais métodos fornecidos pela CPU e também os métodos específicos de GPU.

Entretanto devido a natureza da paralelização é preciso ter cautela na escolha. Funções implementadas por CUDA podem ser síncronas ou assíncronas o que interfere nas medições de tempo realizadas pela CPU. A função Kernel de uma chamada em CUDA é assíncrona, ou seja, uma vez chamada pelo *host* o controle de execução é retomado ainda que o *device* não tenha concluído sua execução. Neste tipo de situação o tempo medido por funções da CPU seria incorreto. Para solucionar o problema de sincronização CUDA oferece uma alternativa aos temporizadores de CPU através da biblioteca *CUDA event API*. Esta API permite chamadas para criar, destruir, registrar e calcular o tempo decorrido entre os eventos. Eventos CUDA utilizam o conceito de *CUDA stream* onde uma sequência de operações são executadas em ordem no *device*. O código abaixo 6 exemplifica o uso de eventos CUDA.

```
1  cudaEvent_t start, stop;
2  float milliseconds = 0;
3  cudaEventCreate(&start);
4  cudaEventCreate(&stop);
5
6  cudaEventRecord(start);
7
8  checkCudaErrors(cudaMemcpy(d_Params, h_Params, paramCombCount
9                 * sizeof(Params), cudaMemcpyHostToDevice));
10 checkCudaErrors(cudaMemcpy(d_Data, h_Data, amountOfLinesToProcess
11                * sizeof(MarketData), cudaMemcpyHostToDevice));
12
13 cudaEventRecord(stop);
14
15 cudaEventSynchronize(stop);
16 cudaEventElapsedTime(&milliseconds, start, stop);
```

Listing 6 – Registro de eventos em CUDA

O trecho acima é a captura do evento de transferência do bloco de séries temporais e dos conjuntos de parâmetros que serão processaos no *device*. A linha 1 são criados dois eventos do tipo *cudaEvent_t*: **1)** O *start* para marcar o início do evento a ser registrado e o **2)** *stop* como um evento de finalização. Nas linhas 3 e 4 os eventos são instanciados e na linha 6 o evento de início é registrado inicializando o contador de tempo. Nas linhas que seguem a transferência de dados é feita e na linha 13 é registrado o evento de parada que logo em seguida é sincronizado para garantir que as operações foram executadas sequencialmente pelo *device*. Por fim na linha 16 a chamada a função *cudaEventElapsedTime* armazena na variável *milliseconds* o tempo transcorrido em milissegundos entre o evento inicial e o final.

Para obter medições mais precisas foram criados eventos temporizadores CUDA

para transferência de dados e execução do Kernel de forma separada. Assim, é possível saber quanto tempo foi gasto em execução do método principal e quanto tempo foi gasto apenas com as transferências entre o *host* e o *device* visto que este último é normalmente apontado como o maior gargalo em CUDA. Esta informação, entretanto, não será utilizada para os fins comparativos do experimento, cujo objetivo é contabilizar o tempo total de execução da solução incluindo o tempo gasto com operações realizadas pela CPU.

Para a medição dos tempos de CPU a linguagem C conta com uma biblioteca chamada *time.h* que contém a *struct timeval*. Assim como em CUDA, foi necessário criar duas variáveis deste tipo, para registrar o início e o fim dos eventos que serão medidos através da função *gettimeofday()* que deve ser chamada ao iniciar a execução do código com a variável *start* como parâmetro, e ao finalizar a execução com a variável *stop* como parâmetro. Por fim, o tempo em milissegundos é obtido através do cálculo: *stop.tv_usec - start.tv_usec*.

A métrica adotada para validar a performance da solução é o *speedup* que já foi apresentado detalhadamente na seção 3.4. Os tempos em milissegundos coletados nos experimentos servirão de base para cálculo desta métrica.

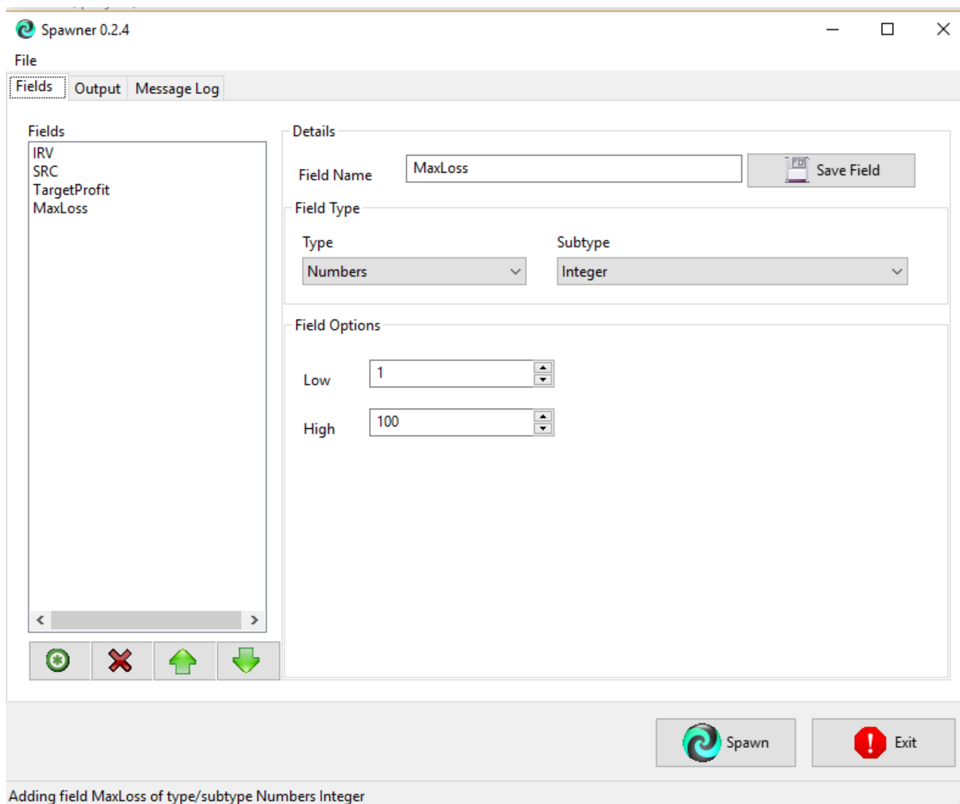
7.1.3 Dados disponíveis

O arquivo de leitura de séries temporais recebeu o nome do período de tempo em que os dados foram coletados seguido da extensão .txt. Sendo: Dia.txt, Semana.txt, Quinzena.txt e Mes.txt e contém valores reais do mercado *ForEx* como explicado nos capítulos anteriores. As características dos arquivos contendo as séries coletadas são exibidas na tabela 2. Os dados foram coletados e disponibilizados pelo grupo de pesquisa PIMAT.

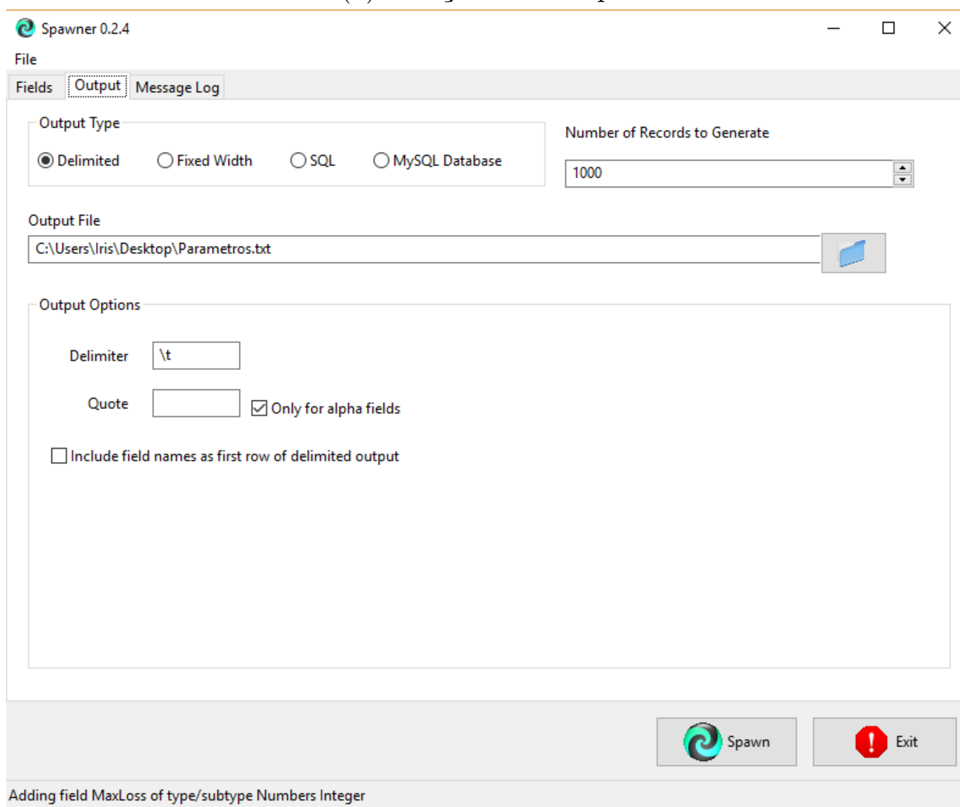
Nome	Período	Nº Linhas	Tamanho (KB)
Dia.txt	01/05/2014	40 037	2 073
Semana.txt	02/05/2014 - 07/05/2014	216 863	11 225
Quinzena.txt	02/05/2014 - 18/05/2014	657 780	34 046
Mes.txt	01/05/2014 - 30/05/2014	1 217 609	63 021

Tabela 2 – Características dos arquivos de séries temporais

O arquivo de parâmetros (detalhado na seção 6.3) foi criado através da ferramenta *Spawner*. Este programa *open source* é um gerador automático de dados na versão 0.2.4. Nele é possível configurar quantas linhas o arquivo de *output* deve ter, qual o formato de saída, os campos e o tipo deles e quais os valores máximos e mínimos para os campos. O nome do arquivo *dataset* de saída é *Parametros.txt*. A figura 14 abaixo é um *print screen* do programa.



(a) Criação dos campos



(b) Criação do arquivo

Figura 14 – Spawner

Os arquivos criados com a ferramenta receberam o nome da quantidade de li-

nhas, ou conjuntos de parâmetros contidas em cada um, seguido da extensão .txt. Sendo: 1024.txt, 512.txt, 256.txt

7.2 Experimentos

Para validar a solução desenvolvida a variável monitorada é o tempo total de execução das aplicações. Já as variáveis de suporte são o arquivo de séries temporais e o arquivo de parâmetros.

Estas duas variáveis influenciam diretamente no comportamento da plataforma CUDA uma vez que, blocos de dados muito grandes implicam em um incremento no tempo de execução devido a transferência entre os dispositivos. Por outro lado, blocos muito pequenos não fazem uso eficaz do poder computacional da plataforma. Como o arquivo de parâmetros está diretamente ligado ao número de Threads lançadas, variações no seu tamanho exigem da GPU maior ou menor poder computacional a depender da quantidade de Threads a serem gerenciadas ao mesmo tempo. O desafio maior em CUDA é sempre encontrar o nível de granularidade ideal capaz de utilizar a menor massa de dados possível e executar o maior número de operações sobre ela utilizando as Threads.

Os experimentos a serem descritos buscam manter uma das varáveis fixa enquanto a outra sofre variações monitorando sempre o tempo de execução.

7.2.1 Experimento 1: Threads fixas

Para este experimento o número de Threads lançadas foi fixado em 512 que é o valor intermediário entre o máximo e o mínimo de Threads do grupo experimental. Os experimentos irão subdividir-se em 4 para cada uma das proposições (paralelo e serial), totalizando 8 tempos. A escolha da divisão dos arquivos de séries temporais baseia-se no tamanho de cada um destes arquivos, para observar o comportamento do algoritmo a medida que a quantidade de dados para processamento também aumenta.

A figura 15 apresenta os resultados comparativos entre os tempos de execução de cada implementação. Os valores foram coletados em milissegundos em seguida convertidos para segundos e no apêndice os tempos deste experimento encontram-se detalhados na tabela 5. Os valores encontrados no experimento contendo dados do mercado no diários foi omitido do gráfico pois não apresentou valores de *speedup* maiores que 1.

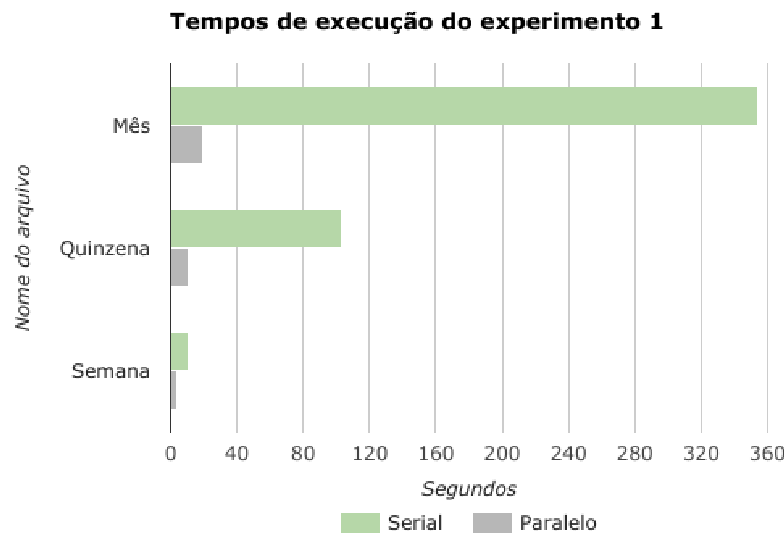


Figura 15 – Tempos de execução do experimento 1 - Threads fixas
Fonte: Elaborado pela autora

Quando o arquivo de séries temporais é pequeno, como foi o caso do teste realizado com o arquivo contendo dados de um dia de funcionamento de mercado, a implementação serial consegue executar em menos tempo que o paralelo, uma possível explicação é o tempo incrementado com as transferências entre os dispositivos e que não é compensado com um grande número de operações. Ou seja, a granularidade dos dados é muito pequena. Nos demais testes comparativos, entretanto, a implementação em paralelo apresentou os menores tempos de execução, e quanto maior o tamanho do arquivo, maior a diferença entre os tempos dos dois algoritmos. Por outro lado, ao confrontar os tempos da execução em paralelo com ele próprio nota-se que a medida que os arquivos aumentam de tamanho, a diferença entre os tempos é pequena, mesmo quando o tamanho dos arquivos dobra de tamanho, o que pode indicar que o algoritmo tende a uma estabilidade no tempo de execução ao atingir determinado volume de dados. É possível perceber também que apesar do volume de dados do arquivo mensal ser 30 vezes maior que o diário os tempos de execução não seguem esta progressão.

Arquivo	**Speedup**
Dia	0,73
Semana	2,89
Quinzena	9,80
Mês	18,65

Tabela 3 – Speedup do experimento 1

A tabela 3 resume os *Speedups* do algoritmo paralelo em relação ao serial. O maior e o menor valor de *Speedup* está diretamente relacionado ao maior e menor arquivo de

séries temporais, quanto menor o arquivo menor o *Speedup*, quanto maior o arquivo maior o *Speedup*, que para o arquivo contendo dados referentes ao intervalo de um mês chegou a ser 18 vezes mais rápido que o algoritmo serial.

7.2.2 Experimento 2: Threads variáveis

Neste experimento o arquivo contendo as séries temporais foi fixado no intervalo de um mês, pois corresponde ao arquivo com maior volume de dados. Os experimentos então são subdivididos em outros 3 onde o número das Threads varia em: 256, 512 e 1024. Para a implementação em serial, isto significa dizer que a quantidade de conjuntos de parâmetros testados corresponde ao número de Threads. Os valores escolhidos são potências de 2 propositalmente para facilitar o gerenciamento feito por CUDA. Como o número máximo de Threads suportado pela GPU é de 1024 a escolha foi testar os últimos 3 valores de 2^n até o número máximo de 1024.

A figura 16 representa os tempos de execução dos algoritmos quando o número de conjuntos de parâmetros é de 256, 512 e 1024. Ao observar os gráficos fica evidente que os tempos de execução do algoritmo serial foram superiores aos tempos do algoritmo paralelo em todos os testes, é possível acrescentar que o pior tempo de execução do algoritmo paralelo foi cerca de 9 vezes mais rápido que o melhor tempo de execução do serial. Outra análise que pode ser feita é sobre a pouca diferença entre os tempos de execução do algoritmo paralelo, as barras cinzas do gráfico mantêm-se quase que no mesmo patamar, mas o número de conjunto de parâmetros dobra entre os testes. Quanto ao algoritmo paralelo, como era esperado, o menor conjunto de parâmetros obteve também o menor tempo de processamento.

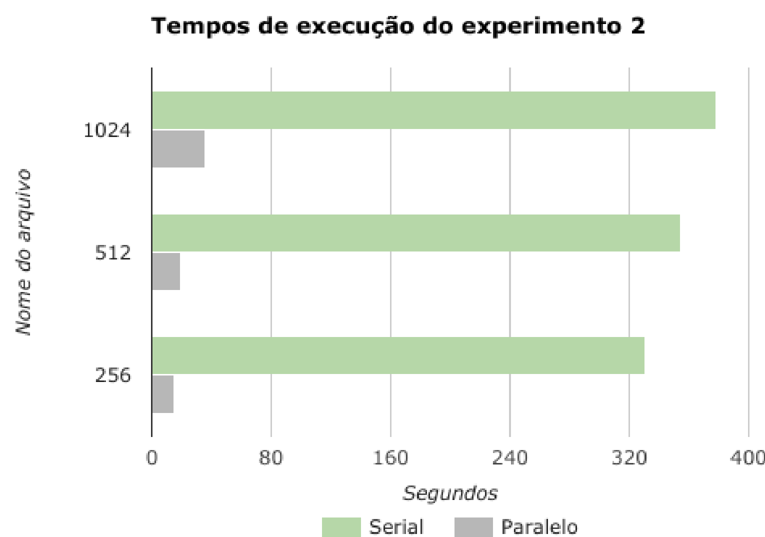


Figura 16 – Tempos de execução do experimento 2 - Threads variáveis
Fonte: Elaborado pela autora

A tabela 6 contém todos os tempos de execução deste experimento enquanto que a tabela 4 apresenta os *Speedups* do algoritmo paralelo em relação ao serial. Nos 3 casos de teste os *Speedups* foram maiores do que um, ou seja, não só o a implementação em serial apresentou um melhor tempo de execução, mas na pior aceleração obtida este tempo conseguiu ser pelo menos 10 vezes mais rápido que a execução em serial. Um ponto importante a ser destacado sobre esses resultados obtidos é que a medida que o número de conjunto de parâmetros aumentou o *Speedup* diminuiu.

Tipo Arquivo	**Speedup**
256	22,28
512	18,65
1024	10,45

Tabela 4 – Speedup do experimento 2

Na teoria o paralelismo deveria resultar em um aumento de velocidade de execução linearmente, o que significa dizer que ao dobrar a quantidade de elementos de processamento o tempo de execução deveria ser reduzido pela metade, no entanto, os resultados obtidos não apontam nesta direção, e isso é explicado através da lei de Amdahl que foi apresentada na seção 3.4. O aumento potencial da velocidade de um algoritmo em uma plataforma de computação paralela é limitado a porção sequencial do problema que não pode ser paralelizado. Se a porção sequencial de um programa representa X% do tempo de execução, não se pode obter mais que X vezes de aumento de velocidade, independente de quantos processadores são adicionados. Os resultados então levam a acreditar que o *speedup* tende a ser linear para poucos elementos de processamento, como no experimento 2 onde a proporção entre o aumento do *speedup* foi de $\cong$ 3,5 , mas decai a um valor quase constante para uma grande quantidade de elementos como no caso do experimento 1.

8 Conclusões

Este trabalho apresentou a proposta de otimizar um robô que opera no mercado *ForEx* através da computação paralela na plataforma CUDA. Para isso, foram criados dois ambientes de testes de robô que simulam operações do mercado tomando como base séries temporais reais e conjuntos de parâmetros que definem o comportamento da estratégia do robô. O primeiro ambiente utiliza o paradigma da computação serial e suas rotinas são executadas na CPU e o segundo utiliza o paradigma da computação paralela e suas rotinas são executadas na GPU. Para validar e avaliar o desempenho dos algoritmos eles foram submetidos a dois tipos de experimentos, no primeiro foram coletados 4 tempos de execução para cada solução, e no segundo 3 tempos para cada, totalizando assim 7 testes. Em seguida, esses valores serviram para cálculo de *speedup*, métrica utilizada para validar o modelo.

Os resultados obtidos demonstram que dentre os testes realizados apenas em apenas um deles o algoritmo serial obteve o menor tempo de execução, ou seja, dentro do escopo de validação a otimização do robô foi alcançada. No melhor desempenho do algoritmo paralelo, a implementação conseguiu ser 22 vezes mais rápida que o serial sob as mesmas condições. No experimento 1, onde o número de Threads é fixo, apesar do algoritmo serial ter obtido um tempo de execução menor para um dos testes, ainda assim o *speedup* médio para a solução demonstra um ganho de 18 vezes no tempo de processamento. Para o experimento 2 cujo de número de Threads é variável, o ganho médio foi de 17 vezes no tempo de processamento.

Diante destes números é possível afirmar que a solução proposta conseguiu otimizar o tempo de execução do robô e os resultados poderiam ter sido ainda melhores caso o *hardware* em que os testes foram realizados possuísse mais núcleos de processamento ou mesmo permitisse o uso de funções e bibliotecas que não estão disponíveis para o equipamento em questão. Um exemplo a ser citado é o uso de rotinas assíncronas para transferência de dados da memória do *host* para o *device*. Como explicado no decorrer do trabalho, este tipo de operação é a que mais eleva os tempos de processamento na plataforma CUDA. O uso de rotinas de transferências assíncronas permite retomar o controle de execução para CPU enquanto os dados ainda estão sendo transferidos e permitindo que a CPU ocupe-se com outras tarefas ao invés de estar ociosa esperando os dados serem copiados.

Um estudo da análise de complexidade dos algoritmos poderia enriquecer o trabalho e fornecer também um suporte matemático e teórico capaz de explicar o desempenho dos *speedups* encontrados face ao nível de granularidade estabelecido. Seria possível talvez

encontrar um modelo capaz de prever o comportamento das razões nos tempos de execução a fim de saber por exemplo se existe uma linearidade nos valores obtidos que depende da granularidade do problema, ou mesmo baseado no comportamento do algoritmo serial o que é possível deduzir do algoritmo paralelo. Realizar novos testes em diferentes arquiteturas CUDA poderia também ajudar a entender o quão intimamente o *software* está ligado ao *hardware* e o quanto os tempos encontrados podem ser melhores ou não a depender do ambiente em que estão sendo executados.

Em decorrência do bom desempenho apresentado a pesquisa abre caminho para um aprofundamento da solução, como tornar o algoritmo adaptável a outros tipos de robô do mercado *ForEx*, onde a estratégia pode ser paralelizada de forma automática. Outra linha a ser seguida é tornar a etapa de pós processamento também paralelizada, onde as métricas de desempenho do robô obtidas na etapa de execução possam apontar qual combinação de parâmetros apresentou os melhores resultados. Uma outra proposta seria criar o processo de geração do conjunto de parâmetros de input de forma automatizada e paralelizada sem a necessidade de leitura de um arquivo em memória visando diminuir ainda mais o tempo de execução total da solução.

Referências

ALVARES, L. O. ***Data mining em séries temporais***. Dissertação (Dissertação) — Departamento de informática e estatística da Universidade Federal de Santa Catarina, 2013. Citado na página 17.

ANDREW THOMAS TOM KEARNEY MANAGEMENT CONSULTING. ***Hype vs. Reality: The Coming Waves of "Robo" Adoption***: Insights from the a.t.kearney 2015 robo-advisory services study. Chicago, 2015. 32 p. Citado na página 13.

BANK FOR INTERNATIONAL SETTLEMENTS. ***Triennial Central Bank Survey***: Foreign exchange turnover in april 2016. [S.l.], 2016. Disponível em: <https://www.bis.org/publ/rpfx16fx.pdf>. Acesso em: 09.03.2017. Citado na página 13.

BARR, A. ***Host-Device Data Transfer***. 2015. Disponível em: <http://courses.cms-.caltech.edu/cs179/2015_lectures/cs179_2015_lec13.pdf>. Acesso em: 18.05.2017. Citado na página 33.

BRESSAN, A. A. ***Tomada de decisão em futuros agropecuários com modelos de previsão de séries temporais***. [S.l.], 2004. Disponível em: <http://www.rae.com.br/eletronica/index-.cfm?FuseAction=ArtigoID=1914Secao=FINANÇASVolume=3Numero=1Ano=2004>. Acesso em: 24.5.2017. Citado na página 17.

CHENG, G. ***How To Place Orders With A Forex Broker***. 2017. Disponível em: <http://www.investopedia.com/articles/forex/07/forexorders.asp>. Acesso em: 24.5.2017. Citado na página 16.

EHLERS, R. S. ***Análise de séries temporais***. Dissertação (Dissertação de Mestrado) — Instituto de Ciências Matemáticas e de Computação da Universidade de São Paulo, 2009. Citado na página 17.

GRAMA, A. et al. ***Introduction to Parallel Computing***. Edinburgh Gate, Harlow, England: Addison Wesley, 2003. Citado 2 vezes nas páginas 19 e 21.

GRAUER-GRAY, S. et al. ***Accelerating Financial Applications on the GPU***. Dissertação (Artigo) — University of Delaware, 2013. Citado na página 37.

GUILLeN, J. S. et al. ***O Potencial das Graphics Processing Units***. Dissertação (Artigo) — Universidade Estadual de Campinas, Campinas, SP, Brasil, 2012. Citado 3 vezes nas páginas 26, 27 e 29.

INDUWARA, S.; JAYASENA, S. **Fast Computation of Stock Market Indices Using GPUs**. *IEEE 8th International Conference on Industrial and Information Systems*, Sri Lanka, v. 8, n. 1, p. 18–20, 2013. Citado na página 38.

INVESTOPEDIA. ***Forex Walkthrough | Investopedia***. [S.l.], 2016. Disponível em: <http://www.investopedia.com/walkthrough/forex/>. Acesso em: 09.03.2017. Citado 2 vezes nas páginas 13 e 15.

INVESTOPEDIA. ***Bond***. 2017. Disponível em: <http://www.investopedia.com/terms-/b/bond.asp>. Acesso em: 25.3.2017. Citado na página 37.

INVESTOPEDIA. ***Money Market: Repos***. 2017. Disponível em: <http://www-.investopedia.com/university/moneymarket/moneymarket7.asp>. Acesso em: 25.3.2017. Citado na página 37.

INVESTOPEDIA. ***Multivariate Models: The Monte Carlo Analysis***. 2017. Disponível em: <http://www.investopedia.com/articles/financial-theory/08/monte-carlo-multivariate-model.asp>. Acesso em: 25.3.2017. Citado na página 37.

INVESTOPEDIA. ***Option***. 2017. Disponível em: <http://www.investopedia.com-/terms/o/option.asp?ad=dirNqo=investopediaSiteSearchqsrc=0o=40186>. Acesso em: 25.3.2017. Citado na página 37.

JENQ, J. ***Parallel Implementation of Moving Averages and Stock Market Prediction***. Dissertação (Mestrado) — Computer Science Department, Montclair State University, Montclair, New Jersey, USA, 2012. Citado na página 39.

KHRONOS. ***OpenCL - The open standard for parallel programming of heterogeneous systems***. 2017. Disponível em: <https://www.khronos.org/opencl/>. Acesso em: 25.3.2017. Citado na página 37.

KIRK, D. B.; HWU, W. mei W. ***Programming Massively Parallel Processors A Hands-on Approach***. Burlington, MA, USA: Elsevier, 2012. Citado 3 vezes nas páginas 13, 19 e 27.

MARINI, D. V. R. ***A importância do mercado internacional de moedas como opção de investimento, seus dogmas e mitos***. Dissertação (Dissertação de Mestrado) — Instituto Superior de Ciências do Trabalho e da Empresa, 2009. Citado 3 vezes nas páginas 13, 15 e 16.

MORIMOTO, C. E. ***Como o PCI Express funciona***. 2011. Disponível em: <http://www.hardware.com.br/guias/placas-mae-barramentos/como-pci-express-funciona.html>. Acesso em: 18.05.2017. Citado na página 34.

NVIDIA. ***Fermi***: Nvidias next generation cuda compute architecture. California, 2009. 22 p. Citado 2 vezes nas páginas 34 e 36.

NVIDIA. ***o que é CUDA***: Cuda programação paralela facilitada. [S.l.], 2017. Disponível em: <http://www.nvidia.com.br/object/cuda_home_new_br.html>. Acesso em: 09.03.2017. Citado 3 vezes nas páginas 13, 28 e 30.

OPENACC. ***About OpenACC***. 2017. Disponível em: <https://www.openacc.org-/about>. Acesso em: 19.05.2017. Citado na página 37.

PACHECO, P. S. ***An Introduction To Parallel Programming***. Burlington, MA, USA: Elsevier, 2011. Citado 5 vezes nas páginas 19, 21, 22, 23 e 25.

RAUBER, T.; RüNGER, G. ***Parallel Programming***. Dissertação (Artigo), Verlag, Heidelberg, Berlin, 2013. Citado 2 vezes nas páginas 20 e 21.

REVOLVY. ***OpenHMPP***. 2017. Disponível em: <https://www.revolvy.com-/topic/OpenHMPPitem_type=topic>. Acesso em: 19.05.2017. Citado na página 37.

ROSSI, P. ***O mercado internacional de moedas , o carry trade e as taxas de câmbio***. Dissertação (Dissertação de Mestrado) — Instituto de economia da Universidade Estadual de Campinas, 2010. Citado na página 15.

SOUZA, T. T. P. ***Simulações financeiras em GPU***. Dissertação (Dissertação de Mestrado) — Instituto de matemática e estatística da universidade de são paulo, 2013. Citado na página 38.

U, N. T. ***A Quick Guide to Support and Resistance***. 2017. Disponível em: <http://www.newtraderu.com/2017/02/02/quick-guide-support-resistance/>. Acesso em: 29.3.2017. Citado na página 41.

WIKIPEDIA. ***Taxonomia de Flynn***. 2016. Disponível em: <https://pt.wikipedia.org-/wiki/Taxonomia_de_Flynn>. Acesso em: 19.5.2017. Citado na página 19.

WIKIPEDIA. ***Black-Scholes***. 2017. Disponível em: <https://pt.wikipedia.org/wiki-/Black-Scholes>. Acesso em: 25.3.2017. Citado na página 37.

Apêndices

APÊNDICE A – Códigos auxiliares

```
int main(void) {
        cudaDeviceProp  prop;

        int count;
        cudaGetDeviceCount(&count);
        for (int i = 0; i< count; i++) {
                cudaGetDeviceProperties(&prop, i);
                printf("   --- General Information for device %d ---\n", i);
                printf("Name:  %s\n", prop.name);
                printf("Compute capability:  %d.%d\n", prop.major, prop.minor);
                printf("Clock rate:  %d\n", prop.clockRate);
                printf("   --- Memory Information for device %d ---\n", i);
                printf("Total global mem:  %ld\n", prop.totalGlobalMem);
                printf("Total constant Mem:  %ld\n", prop.totalConstMem);
                printf("Max mem pitch:  %ld\n", prop.memPitch);
                printf("   --- MP Information for device %d ---\n", i);
                printf("Multiprocessor count:  %d\n",
                        prop.multiProcessorCount);
                printf("Shared mem per mp:  %ld\n", prop.sharedMemPerBlock);
                printf("Threads in warp:  %d\n", prop.warpSize);
                printf("Max threads per block:  %d\n",
                        prop.maxThreadsPerBlock);
                printf("Max thread dimensions:  (%d, %d, %d)\n",
                        prop.maxThreadsDim[0], prop.maxThreadsDim[1],
                        prop.maxThreadsDim[2]);
        }
}
```

Listing 7 – Obtenção das propriedades da plataforma

```
1         Params *h_Params;
2         Statistic *h_statisticData;
3         Params *d_Params;
4         Statistic *d_statisticData;
5
6     //alocação de memória no host
7         h_statisticData = (Statistic *)malloc(paramCombCount * sizeof(Statistic));
8
9     // alocação de memória no device
10        checkCudaErrors(cudaMalloc((void**)&d_Params, paramCombCount
11                        * sizeof(Params)));
12        checkCudaErrors(cudaMalloc((void**)&d_statisticData, paramCombCount
13                        * sizeof(Statistic)));
14
15    //transferência de dados do host para o device
16        checkCudaErrors(cudaMemcpy(d_Params, h_Params, paramCombCount
17                        * sizeof(Params), cudaMemcpyHostToDevice));
```

Listing 8 – Alocação de memória e transferência de dados entre host e device

```
1         checkCudaErrors(cudaPeekAtLastError());
2         checkCudaErrors(cudaThreadSynchronize());
3         checkCudaErrors(cudaDeviceSynchronize());
4
5         checkCudaErrors(cudaMemcpy(h_statisticData, d_statisticData, paramCombCount
6                         * sizeof(Statistic), cudaMemcpyDeviceToHost));
```

Listing 9 – Validação de erros e transferência de dados do device para o host

APÊNDICE B – Detalhamento dos tempos dos experimentos

	Tempo de Execução (ms)	
Arquivo	**Serial**	**Paralelo**
Dia	509	700
Semana	10 321	3 577
Quinzena	103 383	10 547
Mês	354 594	19 012

Tabela 5 – Tempos de execução do experimento 1: Threads Fixas

	Tempo de Execução (ms)	
Arquivo	**Serial**	**Paralelo**
256	331 031	14 860
512	354 594	19 012
1024	377 919	36 177

Tabela 6 – Tempos de execução do experimento 2: Threads Variáveis