ALEX OSHIKA AVILLA
ÉRICO HISSASHI NIKAIDO
PEDRO VICTOR LOSADA CAVALCANTE

SISTEMA DE AUXÍLIO A ARBITRAGENS NO MERCADO FINANCEIRO

São Paulo
2010

ALEX OSHIKA AVILLA
ÉRICO HISSASHI NIKAIDO
PEDRO VICTOR LOSADA CAVALCANTE

SISTEMA DE AUXÍLIO A ARBITRAGENS NO MERCADO FINANCEIRO

Monografia apresentada à Escola Politécnica da Universidade de São Paulo para obtenção de título de Engenheiro de Computação.

Área de Concentração:

Engenharia de Computação.

Orientador:

Prof. Dr. Jorge Kinoshita.

São Paulo

2010

FICHA CATALOGRÁFICA

Avilla, Alex Oshika; Nikaido, Érico Hissashi; Cavalcante, Pedro Victor Losada.

Sistema de Auxílio a Arbitragens no Mercado Financeiro. / Alex Oshika Avilla, Érico Hissashi Nikaido e Pedro Victor Losada Cavalcante. -- São Paulo, 2010. 80p.

Trabalho de Formatura – Escola Politécnica da Universidade de São Paulo. Departamento de Engenharia de Computação e Sistemas Digitais.

1. Engenharia. 2. Engenharia de Computação 3. Sistema de Negociação Automatizada. I. Universidade de São Paulo. Escola Politécnica. Departamento de Engenharia de Computação e Sistemas Digitais.

AGRADECIMENTOS

Ao professor Jorge Kinoshita, pela orientação e pelo constante estímulo durante todo o trabalho.

A todos os professores da Escola Politécnica da Universidade de São Paulo, pelo conhecimento transmitido e pelos diversos desafios dados durante todo o curso de Engenharia que, com certeza, nos prepararam melhor para a execução desse trabalho e também para a nossa vida.

Aos familiares, por todo o suporte e compreensão durante o período de execução do trabalho.

Agradecimentos também aos amigos e a todos que colaboraram direta ou indiretamente na execução desse trabalho.

RESUMO

Esse trabalho tem como objetivo desenvolver uma alternativa viável aos sistemas de negociação automatizada de ativos financeiros através da mudança de paradigma de execução de algoritmos e da integração com ambiente mais familiar ao usuário final do sistema, o trader (profissional da mesa de operações), na criação de novos algoritmos. Para a execução dos algoritmos de arbitragem, o Sistema proposto utilizou placas de vídeo da NVIDIA habilitadas para o CUDA, tecnologia da própria NVIDIA de execução paralela extrema com alta densidade de operações aritméticas. Também foi desenvolvido uma integração com softwares de planilha eletrônica capazes de transformar as fórmulas das células em código C pronto para ser executado no CUDA. Por fim, foi desenvolvido um núcleo para a integração de ambas as tecnologias e para controlar a execução do algoritmo.

Palavras-chave: Engenharia, Engenharia de Computação, Sistema de Negociação Automatizada.

ABSTRACT

The main goal of the present work is to develop a viable alternative to current algorithmic trading systems through a paradigm shift of algorithm execution and also through the use of a familiar environment to the final user of the system, the trader from the trading desk, to create new algorithms. To execute arbitrage algorithms, the proposed System utilized NVIDIA CUDA-enable graphics cards. CUDA is a technology developed by NVIDIA for extreme parallel execution with high density of arithmetic operations. It was also developed an integration with a spreadsheet application capable of translating cell formulas to CUDA-ready C code. At last, a core program was developed to integrate both technologies and to control algorithm execution.

Keywords: Engineering, Computer Engineering, Program Trading, Algorithmic Trading, Algotrading.

LISTA DE LISTAGENS

LISTA DE FIGURAS

SUMÁRIO

INTRODUÇÃO

A Tecnologia de Informação é a responsável por uma grande revolução em negociação de ativos financeiros: a mudança de pregões tradicionais para pregões eletrônicos. Em 2005 o País assistiu a essa transformação quando a Bovespa – Bolsa de Valores de São Paulo, principal bolsa do Brasil – encerrou seu pregão viva-voz, com todos os seus característicos berros e frenéticos operadores falando em vários telefones especiais e deu exclusividade aos silenciosos e rápidos pregões eletrônicos, capazes de executar um volume muito maior de negociações a um custo muito mais baixo que pessoas negociando ao vivo. A BM&F seguiu a mesma tendência e, em 2009, terminou também seu pregão viva voz.

Agora uma segunda revolução acontece: a corrida para obter os sistemas mais rápidos para negociação automatizada pelos mais diversos *players* do mercado – bancos, corretoras e investidores institucionais – para conseguir ganhos antes inimagináveis graças aos avanços da computação e da compreensão dos mercados.

Uma definição genérica comumente utilizada de algotrading é “uso de algoritmos de computador para gerenciamento do processo de negociação” [1].

A utilização de computadores no fluxo de ordens no mercado financeiro começou no início da década de 1970 com alguns marcos significativos – como, por exemplo, a introdução na New York Stock Exchange, uma das principais bolsas de valores mundiais, do *designated order turnaround*: sistema capaz de repassar ordens eletronicamente para os operadores no *trading floor* executarem manualmente.

Também houve marcos importantes no mercado financeiro brasileiro. Fora a já mencionada exclusividade dada a negociação eletrônica em 2005 e 2009 pelas duas principais bolsas brasileiras, a possibilidade de *co-location* – arquitetura cujo objetivo é posicionar os sistemas de execução de *algotrading* próximos fisicamente dos sistemas de negociação da bolsa – em 2009 na BM&F e em 2010 na Bovespa foi também um movimento representativo da evolução do mercado financeiro brasileiro seguindo a mesma tendência de mercados mais desenvolvidos.

No entanto, foram identificados alguns aspectos no modelo atual de implementação de sistemas de *Program Trading* que podem ser melhorados.

Ao implementar um novo algoritmo, por exemplo, há a necessidade da comunicação entre quem entende dos mercados – normalmente alguém mais próximo da mesa de operações – e alguém que entenda dos sistemas de informação – esse mais próximo do departamento de Tecnologia de Informação da Instituição implementando o sistema. A experiência de diversas empresas é que a comunicação entre o profissional de negócios e o profissional da área técnica não ocorre sem problemas. Esse conflito é ainda mais evidente nas Instituições Financeiras no Brasil, onde o mercado ainda não é desenvolvido o suficiente e não existe um departamento de especialistas em ambos os pontos (mercado e sistemas), os chamados *quants*.

Outro aspecto passível de melhoras é a própria execução do algoritmo. Em sistemas tradicionais, processadores são usados em modo padrão, usando a capacidade de *multithreading* nativa. Como esses sistemas tradicionais dependem pesadamente da capacidade de processadores, equipamentos feitos especificamente para *Program Trading* têm custo elevado impedindo, dessa forma, a implementação em massa, permitindo assim uma maior eficiência do mercado em geral.

O desenvolvimento deste trabalhos será composta de duas partes principais e sua integração. A primeira parte aborda o problema de interface com o usuário. O sistema proposto faz a leitura dos dados a partir de um software de planilha eletrônica, especificamente o OpenOffice.org Calc, da suíte OpenOffice.org. Dessa forma o profissional especialista apenas em mercados – que geralmente também conhece softwares de planilha – pode entrar com seus algoritmos diretamente no ambiente que está familiarizado e o profissional de sistemas será responsável apenas pela infraestrutura do sistema – também o ambiente pelo qual está familiarizado.

1 TECNOLOGIAS

Para atingir seus objetivos, o Sistema proposto neste trabalho utilizará algumas tecnologias. Essas tecnologias compreendem os dois aspectos que o Sistema pretende abordar: integração com software de planilha e execução em GPU.

Para a integração com o OpenOffice.org Calc será utilizado Universal Network Objects, UNO. A escolha do OpenOffice.org foi feita por ter seu código aberto e possuir vasta documentação da tecnologia UNO. UNO é tecnologia criada para desenvolver funcionalidades adicionais ao OpenOffice.org, permitindo a utilização de diversas linguagens de programação. Em nossa proposta, a linguagem escolhida foi Python e o *binding* PyUNO, que permitirá a utilização do UNO.

Para a execução em GPU, a tecnologia a ser utilizada será CUDA, da NVIDIA. Ela permite que desenvolvedores utilizem a plataforma antes disponibilizada apenas para renderização gráfica, para cálculos em geral. Dessa forma, o desenvolvedor pode utilizar toda a capacidade de processamento das placas, reconhecidamente mais rápida para cálculos em paralelo que as placas comuns.

Uma descrição mais detalhada segue abaixo.

1.1 OpenOffice.org

OpenOffice.org é a suite de escritórios *de* código aberto líder de mercado para processamento de texo, planilhas, apresentações, gráficos, banco de dados dentre outros aplicativos. Ela está disponível para várias línguas e pode ser executada nas arquiteturas mais comuns.

O OpenOffice.org é um dos principais responsáveis pela criação do formato aberto de documentos, Open Document Format, mundialmente adotado como padrão a ser suportado em diversas plataformas e sistemas. A suíte também é capaz de ler e gravar em outros formatos comuns, como é o caso dos formatos do Microsoft Office.

Resultado de mais de vinte anos de pesquisa e desenvolvimento comunitário, o OpenOffice.org pode ser baixado da internet gratuitamente graças ao seu modelo

aberto de desenvolvimento, um dos mais bem sucedidos *cases* de software *opensource* no mundo hoje.

1.2 Universal Network Objects

OpenOffice.org provê uma interface de programação na forma de Universal Network Objects (UNO). Essa é uma interface orientada a objeto a qual o OpenOffice.org subdivide em vários objetos e garante acesso ao OpenOffice.org a programas externos em diversas linguagens.

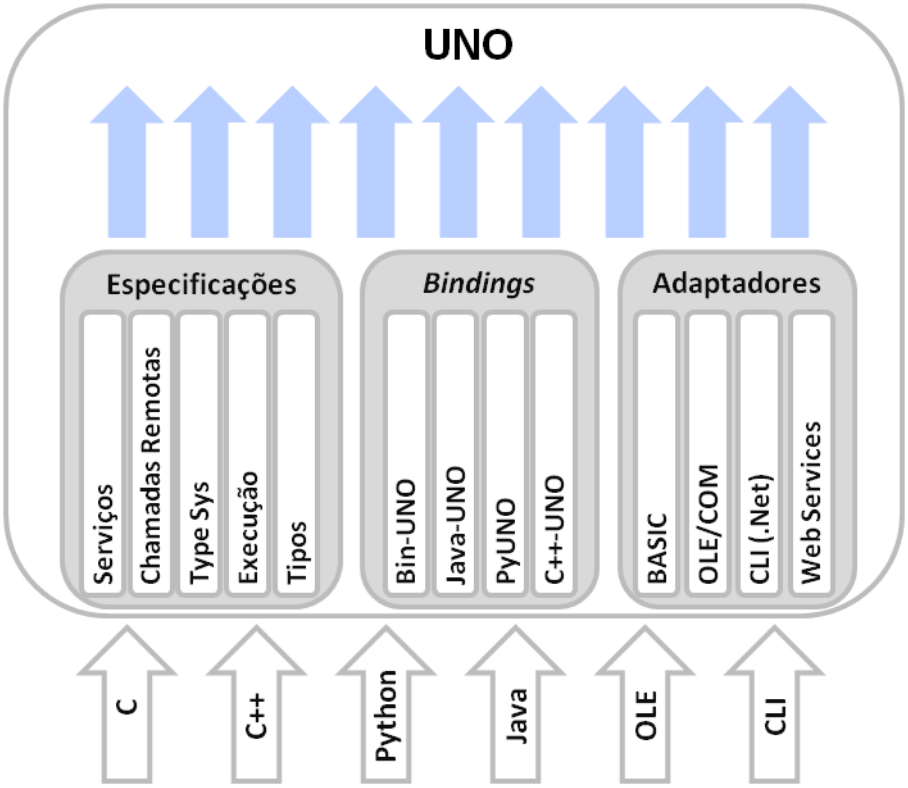


Figura 1-1. Esquematização da arquitetura do UNO.

O OpenOffice.org possui um ambiente de programação interno, o OpenOffice.org Basic que, no entanto, não atende às necessidades atuais dos programadores. Como se trata de uma linguagem de programação procedural, muitas construções linguísticas tiveram que ser adicionadas a ela só para que possa ser utilizada com UNO.

UNO é um conceito central no OpenOffice.org pois ele é o responsável por todos os objetos internos do OpenOffice.org. UNO é o responsável pela comunicação

interobjetos independentemente da linguagem na qual eles foram escritos ou em seu ambiente de execução. De fato uma chamada para um objeto do OpenOffice.org será feita corretamente a sua implementação e ambiente de execução graças ao UNO. Outra característica do UNO é a possibilidade de permitir que programas externos utilizarem objetos do OpenOffice.org remotamente através de conexões de rede e não chamadas internas do Sistema Operacional, por exemplo.

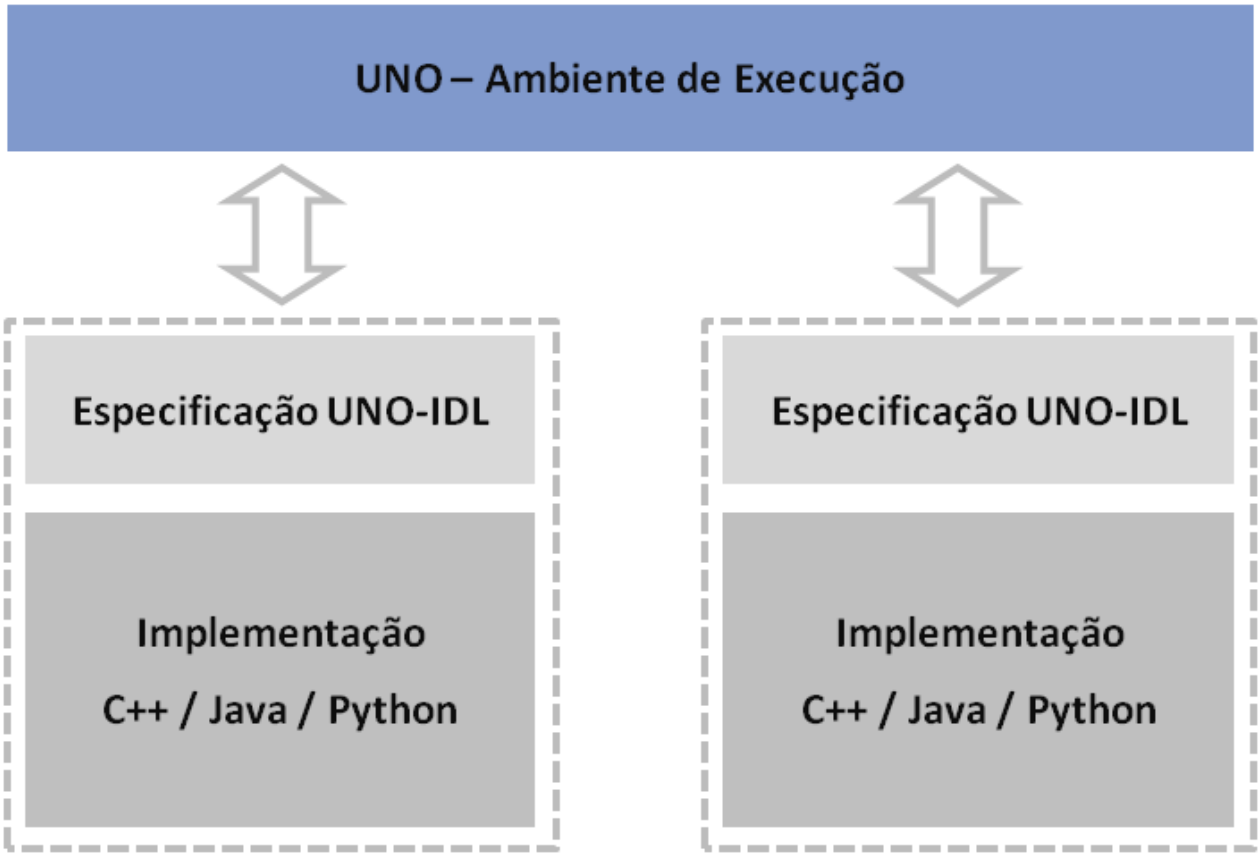


Figura 1-2. Composição de um componente UNO.

Para que um componente esteja completo, ele deve conter uma implementação de sua especificação UNO. Essa implementação pode estar escrita em qualquer uma das linguagens suportadas pelo UNO. Para ter uma linguagem disponível para UNO, uma *bridge* entre a linguagem e o UNO deve ser criada. A linguagem mais comum de

implementação é C++, Java e Python, mas outras linguagens também estão disponíveis.

Como mostrado na figura, uma chamada a um método do componente irá retornar ao UNO Runtime Environment. Ele irá traduzir a chamada para uma implementação usando registros de serviço.

Para permitir a utilização de diferentes linguagens de maneira transparente para o desenvolvedor, o UNO utiliza-se de *Proxy Objects*.

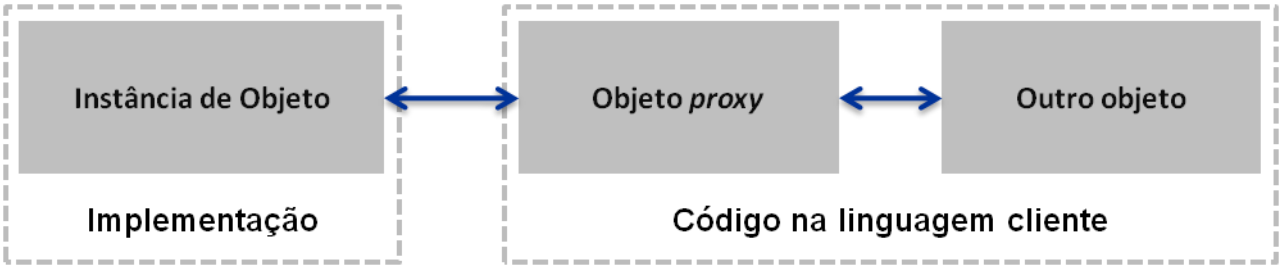


Figura 1-3. Esquematização de objetos *proxy*.

Usando-se *Proxy Objects*, o UNO permite a utilização de linguagens de programação distintas e até a utilização de outro UNO Runtime Environment que o objeto real.

O objetivo do UNO é fornecer um ambiente para objetos em rede sem limites quanto a linguagens de programação e plataformas. Objetos nessa plataforma podem se comunicar em qualquer lugar com qualquer objeto. UNO atinge esse objetivo fornecendo o seguinte *framework* fundamental:

- Objetos UNO são especificados em uma meta linguagem, chamada UNOIDL (UNO Interface Definition Language), similar ao CORBA IDL ou MIDL. De especificações em UNOIDL, cabeçalhos em uma linguagens definidas e bibliotecas podem ser geradas automaticamente para implementar objetos UNO na linguagem-alvo. Objetos UNO na forma de bibliotecas compiladas e definidas são chamados componentes. Componentes precisam suportar certas interfaces base para serem passíveis de utilização pelo ambiente UNO.
- Para instanciar componentes em um ambiente-alvo, UNO usa o conceito de fábrica. Essa fábrica é chamada de *service manager*. Ele mantém um banco de dados de componentes registrados, conhecidos pelo seu nome e que podem ser

criados por nome. Esse *service manager* podem pedir ao Linux para carregar e instanciar um objeto compartilhado em C++ ou pode pedir a uma VM Java para instanciar uma classe Java. Tudo isso é transparente ao desenvolvedor, não há necessidade de se importar com a linguagem de implementação do objeto. A comunicação é exclusiva sobre todas as chamadas de interface como especificado no UNOIDL.

- UNO fornece *bridges* para enviar chamadas de método e receber valores de retorno entre processos e entre objetos escritos em diferentes linguages de programação. Essas *bridges* remotas usam um protocolo de acesso remoto do UNO (*UNO remote protocol*, URP) para esse propósito, suportado por sockets e pipes. Ambas as pontas da *bridge* devem ser ambientes UNO, portanto um *UNO runtime environment* específico de determinada linguagem será conectado a outro processo UNO em qualquer uma das linguagens suportadas. Esses *runtime environments* são fornecidos como *language bindings*.
- A maior parte dos objetos no OpenOffice.org são capazes de se comunicar com um ambiente UNO. A especificação para as interfaces de programação do OpenOffice.org é chamada de OpenOffice.org API.

Sobre o OpenOffice.org API, trata-se de uma abordagem independente da linguagem para especificar funcionalidades do OpenOffice.org. Seus objetivos principais são dar a possibilidade a usuários para estender o OpenOffice.org com suas próprias soluções e novas funcionalidades, e fazer a implementação interna do OpenOffice.org totalmente intercambiável. Um objetivo de longo prazo do OpenOffice.org é dividir o OpenOffice.org existente em pequenos componentes que combinados fornecem todas as funcionalidades do OpenOffice.org. Tais componentes serão gerenciáveis, eles interagem entre si para prover funcionalidades de alto nível e são intercambiáveis com outras implementações equivalentes em funcionalidades, mesmo que essas outras implementações estejam em outras linguagens de programação. Quando esse objetivo de longo prazo for atingido, a API, os componentes irão fornecer um kit de construção capaz de permitir ao OpenOffice.org se adaptar em um grande espectro de soluções, não apenas como uma suíte de escritório com funcionalidades pré-definidas e estáticas.

Objetos UNO em diferentes ambientes conectam-se via uma *bridge* interprocessos. Pode-se executar chamadas às instâncias de objetos UNO, localizados em diferentes processos. Isso é feito convertendo-se o nome do método e seus argumentos em uma representação em *byte stream* e enviando-se esse pacote ao processo remoto, por exemplo, através de uma conexão por socket.

Por motivos de segurança, o OpenOffice.org não escuta por conexões remotas. Isso faz com que uma forma especial de chamada ao OpenOffice.org se faça necessário. Há duas formas de iniciar o OpenOffice.org de forma que este escute por conexões:

- Iniciar o OpenOffice.org com parâmetros especiais:
  `soffice -accept=socket,host=0,port=2002;urp;`
- Deixar os mesmos parâmetros com exceção do '`-accept=`' em um arquivo de configuração. Pode-se editar o arquivo
  `<OfficePath>/share/registry/data/org/openoffice/Setup.xcu`
  e adicionar as *tags*

```
<prop oor:name="ooSetupConnectionURL">
<value>socket,host=localhost,port=2002;urp;StarOffice.ServiceMana
ger
</value>
</prop>
```

  dentro da tag
  `<node oor:name="Office"/>`
  Essa mudança afeta toda a instalação. Ou seja: todos os programas da suíte iniciarão abertos a conexões remotas.

O modo mais comum de conexões interprocessos é a importação de uma referência a um objeto UNO através de um servidor de exportação. A forma correta de se fazer isso é usando-se o serviço `com.sun.star.bridge.UnoUrlResolver`. Sua interface principal, `com.sun.star.bridge.XunoUrlResolver` é definida da forma mostrada na Listagem 1-1. Definição de `com.sun.star.bridge.XunoUrlResolver`.Listagem 1-1.

```
interface XUnoUrlResolver: com::sun::star::uno::XInterface
{
       /** resolves an object on the UNO URL */
              com::sun::star::uno::XInterface resolve( [in] string sUnoUrl )
              raises (com::sun::star::connection::NoConnectException,
                     com::sun::star::connection::ConnectionSetupException,
                     com::sun::star::lang::IllegalArgumentException);
};
```

Listagem 1-1. Definição de `com.sun.star.bridge.XunoUrlResolver`.

A string passada para `resolve()` é chamada UNO URL. Ela deve ter o seguinte formato:

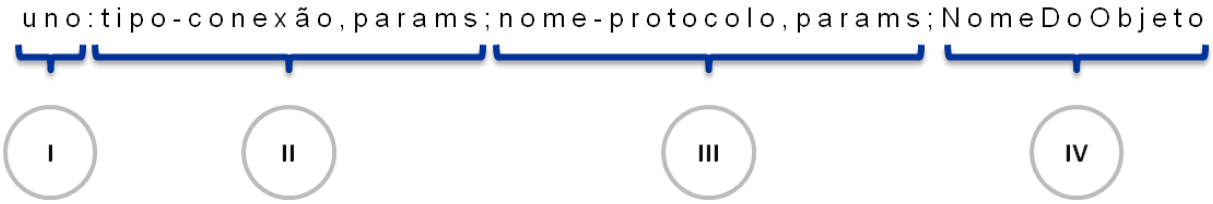


Figura 1-4. Composição de uma URL UNO.

I. *URL schema uno*: Identifica a URL como UNO URL e não a distingue de outros URLs, como `http:`, `ftp:`.

II. Uma string que identifica o tipo de conexão a ser usada no acesso a outros processos. O tipo de conexão especifica o mecanismo de transporte usado na transferência do *byte stream*, como, por exemplo, sockets TCP/IP ou pipe nomeado.

III. Uma string que identifica o tipo de protocolo usado na comunicação do *byte stream*. O protocolo sugerido é o `urp` (*UNO Remote Protocol*).

IV. O processo deve exportar um objeto por um nome distinto. Não se pode acessar um objeto UNO arbitrário (o que seria possível em CORBA, por exemplo).

Um exemplo demonstrando como se importar um objeto usando o `UnoUrlResolver` segue.

```
XComponentContext xLocalContext =
       com.sun.star.comp.helper.Bootstrap.createInitialComponentContext(null);

// initial serviceManager
XMultiComponentFactory xLocalServiceManager =  xLocalContext.getServiceManager();

// create a URL resolver
Object urlResolver = xLocalServiceManager.createInstanceWithContext(
       "com.sun.star.bridge.UnoUrlResolver", xLocalContext);

// query for the XUnoUrlResolver interface
XUnoUrlResolver xUrlResolver = (XUnoUrlResolver)
       UnoRuntime.queryInterface(XUnoUrlResolver.class, urlResolver);

// Import the object
Object rInitialObject =
       xUrlResolver.resolve("uno:socket,host=localhost,port=2002;urp;StarOffice.Serv
iceManager");

// XComponentContext
if (null != rInitialObject) {
       System.out.println("initial object successfully retrieved");
} else {
       System.out.println("given initial-object name unknown at server side");
}
```

Listagem 1-2. Demonstração de importação de objeto usando `UnoUrlResolver`.

O uso do UnoUrlResolver tem algumas desvantagens. Por exemplo, o usuário não é notificado quando a *bridge* termina por algum motivo qualquer. Também não é possível fechar a conexão interprocessos embaixo da *bridge* nem oferecer um objeto local como um objeto inicial para o processo remoto.

Toda a bridge é *threadsafe* e permite a múltiplas threads executar chamadas remotas. A thread de despacho dentro da *bridge* não pode bloquear porque ela nunca executa threads, apenas redireciona os pedidos para outras threads executarem.

- Uma chamada síncrona envia o pedido através de uma conexão e deixa a thread solicitadora esperando por resposta. Todas as chamadas que têm um valor de retorno, um parâmetro `out`, ou que lançam exceções exceto a `RuntimeException` devem ser síncronas.

- Uma chamada assíncrona (ou `oneway`) envia pedidos através de uma conexão e imediatamente retorna sem esperar por uma resposta. Atualmente especifica-se na interface IDL se um pedido é síncrono ou assíncrono através do uso do modificador `oneway`.

Para requisições síncronas, a identidade da thread é garantida. Quando process A chama o processo B e o processo B chama o processo A a mesma thread esperando em A será responsável pela nova requisição. Esse comportamento evita *deadlocks* quando o mesmo mutex é trancado novamente. Para requisições assíncronas, isso não é possível porque não tem uma thread esperando no processo A. Tais requisições são executadas em uam nova thread. A série de chamadas entre dois processos é garantida. Se duas requisições assíncronas do processo A são enviadas ao processo B a segunda requisição espera até que a primeira seja terminada.

A comunicação no UNO é baseada em chamadas para interfaces. Bridges provém a estrutura necessária para o uso das interfaces de objetos UNO entre ambientes de implementação. A chave para o *bridging* é a um ambiente intermediário chamado *binary UNO* que consiste de dados em formato binário para parâmetros e valores de retorno e um método de despacho em C usado para chamar operações arbitrárias em interfaces UNO. Um *bridge* precisa ser capaz de executar as seguintes tarefas:

- Entre a linguagem alvo e o OpenOffice.org:
    - Converter parâmetros de operações da linguagem alvo para o *binary UNO*;
    - Transformar chamadas de operações na linguagem alvo em chamadas em *binary UNO* para diferentes ambientes;
    - Transportar a operação de chamada com seus parâmetros para o OpenOffice.org e o valor de retorno de volta para a linguagem alvo.
    - Mapear os valores de retorno do *binary UNO* para a linguagem alvo.
- Entre o OpenOffice.org e a linguagem alvo; ou seja, durante *callbacks* ou quando estiver usando um componente na linguagem alvo:

- Converter parâmetros de operações do *binary UNO* para a linguagem alvo;
- Transformar chamadas de operações em *binary UNO* para chamadas na linguagem alvo;
- Tranportar chamadas de operação com seus parâmetros para a linguagem alvo e valores de retorno de volta ao OpenOffice.org;
- Converter valores de retorno da linguagem alvo para o *binary UNO*.

A API de Reflection fornece informações sobre os tipos UNO e é usada por bridges para dar apoio às converses de tipo (`com.sun.star.script.Converter`), e chamadas de método (`com.sun.star.script.Invocation` and `com.sun.star.script.XInvocation`). Fora isso, também fornece informação em tempo de execução sobre tipos e cria instâncias de certos tipos UNO, como `structs` (`com.sun.star.reflection.CoreReflection`).

Um carregador de implementações é requerido para carregar e ativar o código produzido na linguagem alvo se as implementações na linaguagem alvo precisam ser instanciadas. Isso envolve a localização de arquivos de components produzidos na linguagem alvo, e mecanismos para carregar e executar código produzido na linguagem alvo, como, por exemplo, iniciar o ambiente de execução.

O *binding* das linguagens no UNO precisa se preparar para que possa ser ligado aos componentes UNO. A forma como isso é obtido depende do ambiente alvo. Em Java, C++ e Python, um gerenciador de serviços locais no ambiente alvo é usado para instanciar a `com.sun.star.bridge.UnoUrlResolver` que conecta ao OpenOffice.org. Na *bridge Automation*, o objeto `com.sun.star.ServiceManager` é obtido do sistema COM e no OpenOffice.org Basic o gerenciador de serviços é disponibilizado através de um método especial no ambiente de execução Basic, `getProcessServiceManager()`.

1.3 PyUNO

1.3.1 Introdução

Como exposto no item anterior, para utilizar o UNO há uma gama de linguagens de programação a serem escolhidas. Para o Sistema proposto, consideramos o Python como a linguagem mais adequada por dois motivos: a) é a linguagem de mais rápido desenvolvimento dentre as opções; b) há muito mais exemplo em PyUNO que em outros *bindings*.

O *binding* para Python, PyUNO, permite aos usuários que usem a API padrão do OpenOffice.org para fazer scripts no bem conhecido ambiente Python de desenvolvimento.

Uma possibilidade fornecida pelo PyUNO também é desenvolver componentes UNO, de modo tal que dentro de processos do OpenOffice.org esses componentes possam ser chamados do Java, C++ ou mesmo OpenOffice.org Basic.

O *bridge* Python UNO permite aos usuários:

- usar a API padrão do OpenOffice.org a partir do Python.
- desenvolver componentes UNO em Python (de forma que um componente Python UNO possa ser executado dentro de processos do OpenOffice.org e possa ser chamado a partir de Java, C++ ou do StarBasic do OpenOffice.org).
- criar e invocar scripts com o framework de scripting do OpenOffice.org.

O PyUNO pode ser usado de três modos distintos:

- dentro de processos do OpenOffice.org dentro do framework de *scripting*.
- dentro de um executável python; ou seja, fora de um processo do OpenOffice.org Usa-se esse modo quando se deseja executar o script de um processo externo (como alguma requisição através de http ou cgi-script) ou quando se deseja o menor tempo de *turnaround* entre código e execução.

  Um código de exemplo encontra-se na Listagem 1-3.

```
import socket
import uno

# pegar component uno do PyUNO runtime
```

```
localContext = uno.getComponentContext()

# criar UnoUrlResolver
resolver = localContext.ServiceManager.createInstanceWithContext(
                            "com.sun.star.bridge.UnoUrlResolver", localContext )

# conectar ao office já executando
ctx = resolver.resolve(
"uno:socket,host=localhost,port=2002;urp;StarOffice.ComponentContext" )
smgr = ctx.ServiceManager

# pegar objeto central do desktop
desktop = smgr.createInstanceWithContext( "com.sun.star.frame.Desktop",ctx)

# acessar document do writer
model = desktop.getCurrentComponent()

# acessar propriedades do texto
text = model.Text

# criar um cursor
cursor = text.createTextCursor()

# inserir testo no cursor
text.insertString( cursor, "Hello World", 0 )
```

Listagem 1-3. Código de exemplo de PyUNO.

- Dentro de um processo do OpenOffice.org

  Usa-se esse modo quando se deseja utilizar esse código em diversas outras máquinas (usando pacotes UNO). Também é utilizado esse modo quando o código em Python será executado a partir de um comando vindo da GUI do OpenOffice.org.

  Um código de exemplo desse modo encontra-se na Listagem 1-4.

```
import uno
import unohelper

from com.sun.star.task import XJobExecutor

# implement a UNO component by deriving from the standard unohelper.Base class
```

```
# and from the interface(s) you want to implement.
class HelloWorldJob( unohelper.Base, XJobExecutor ):
	def __init__( self, ctx ):
		# store the component context for later use
		self.ctx = ctx

	def trigger( self, args ):
		# note: args[0] == "HelloWorld", see below config settings

		# retrieve the desktop object
		desktop = self.ctx.ServiceManager.createInstanceWithContext(
			"com.sun.star.frame.Desktop", self.ctx )

	# get current document model
	model = desktop.getCurrentComponent()

	# access the document's text property
	text = model.Text

	# create a cursor
	cursor = text.createTextCursor()

	# insert the text into the document
	text.insertString( cursor, "Hello World", 0 )

# pythonloader looks for a static g_ImplementationHelper variable
g_ImplementationHelper = unohelper.ImplementationHelper()

g_ImplementationHelper.addImplementation( \
	HelloWorldJob,                                  # UNO object class
	"org.openoffice.comp.pyuno.demo.HelloWorld",    # implementation name
		# Change this name for your own script
	 ("com.sun.star.task.Job",),)                   # list of implemented services
                                                        # (the only service)
```

Listagem 1-4. `hello_world_comp.py` – Código de exemplo no modo de execução dentro do processo do OpenOffice.org.

O código deve ser ligado a um evento de usuário. Isso pode ser através da seguinte configuração.

```
<?xml version="1.0" encoding="UTF-8"?>
<oor:node xmlns:oor="http://openoffice.org/2001/registry"
             xmlns:xs="http://www.w3.org/2001/XMLSchema"
            oor:name="Addons" oor:package="org.openoffice.Office">
<node oor:name="AddonUI">
 <node oor:name="AddonMenu">
  <node oor:name="org.openoffice.comp.pyuno.demo.HelloWorld" oor:op="replace">
   <prop oor:name="URL" oor:type="xs:string">
     <value>service:org.openoffice.comp.pyuno.demo.HelloWorld?insert</value>
   </prop>
   <prop oor:name="ImageIdentifier" oor:type="xs:string">
    <value>private:image/3216</value>
   </prop>
   <prop oor:name="Title" oor:type="xs:string">
    <value xml:lang="en-US">Insert Hello World</value>
   </prop>
  </node>
 </node>
</node>
</oor:node>
```

Listagem 1-5. `Addons.xcu` - Ligação do código a evento de usuário.

Ambos os arquivos devem ser empacotados em um único `arquivo.zip` usado um utilitário zip.

```
zip hello_world.zip Addons.xcu hello_world_comp.py
```

O arquivo resultante (`hello_world.zip`) pode ser utilizado diretamente pelo utilitário `pkgchk`, localizado no diretório de instalação do OpenOffice.org.

Diferentemente do *binding* para Java ou C++, o *binding* do Python UNO não é independente. Ele requer que o UNO binding para C++ e componentes para scripting adicional estejam presentes. Esses componentes adicionais atualmente encontram-se nas bibliotecas compartilhadas `typeconverter.uno`, `invocation.uno`, `corereflection.uno`, `introspection.uno`, `invocadapt.uno`, `proxyfac.uno`, `pythonloader.uno` (no Windows, `typeconverters.uno.dll`; no Unix: `typeconverter.uno.so`).

Também são necessários componentes para conexão interprocessos. Eles são `uuresolver.uno, connector.uno, remotebridge.uno, bridgefac.uno`

1.3.2 Implementando componentes Python UNO

Existe um carregador de components python. Ele permite a criação de instâncias de classes em python não apenas dentro do processo do python mas em qualquer processo UNO arbitrário, incluindo o próprio OpenOffice.org. Ele carrega código python *on demand* se já não estiver carregado e executa dentro da raiz do interpretador python.

Após o modulo ser importado, o carregador python procura por uma variável global com o nome `g_ImplementationHelper`, da qual é esperada uma instância de `unohelper.ImplementationHelper`. O código de exemplo da Listagem 1-6 faz um componente UNO que, embora não muito útil pois não há método UNO para obter tuples nem existe especificação do serviço com.sun.star.io.OutputStream, serve de exemplo para um componente mais genérico.

```
import unohelper
from com.sun.star.io import XOutputStream

g_ImplementationHelper = unohelper.ImplementationHelper()

class TupleOutputStream( unohelper.Base, XOutputStream ):
	# The component must have a ctor with the component context as argument.
	def __init__( self, ctx ):
		self.t = ()
		self.closed = 0

	# idl void closeOutput();
	def closeOutput(self):
		self.closed = 1

	# idl void writeBytes( [in] sequence<byte>seq );
	def writeBytes( self, seq ):
		self.t = self.t + seq	# simply add the incoming tuple to the member
```

```
        # idl void flush();
        def flush( self ):
                pass


        # convenience function to retrieve the tuple later (no UNO function, may
        # only be called from python )
        def getTuple( self ):
                return self.t

# add the TupleOutputStream class to the implementation container,
# which the loader uses to register/instantiate the component.
g_ImplementationHelper.addImplementation( \
        TupleOutputStream,"org.openoffice.pyuno.PythonOutputStream",
                ("com.sun.star.io.OutputStream",),)
```

Listagem 1-6. Exemplo de criação de componente no PyUNO.

Para carregar esse código dentro do OpenOffice.org pode-se utilizar o utilitário `pkgchk`, como anteriormente descrito.

Há a possibilidade de instanciar esse código, por exemplo, a partir do OpenOffice.org Basic.

```
tupleStrm = createUnoService( "com.sun.star.io.OutputStream" )
tupleStrm.flush()
```

Listagem 1-7. Exemplo de código em OpenOffice.org Basic para instanciar componentes do PyUNO.

Pode-se também utilizar o parâmetro `out` do UNO para retornar múltiplos valores. A Listagem 1-8 abaixo demonstra uma forma de fazê-lo e a Listagem 1-9 tem a especificação UNOIDL correspondente.

```
long foo( [in] long first, [inout] long second, [out] third )
```

Listagem 1-8. Demonstração do parâmetro `out`.

```
class Dummy( XFoo ):
        def foo( self, first,second,third):
        # Note: the value of third is always None, but it must be there
        # as a placeholder if more args would follow !
        return first,2*second,second + first
```

Listagem 1-9. Especificação em UNOIDL do código da Listagem 1-8.

Um método como esse poderia ser chamado como mostrado na Listagem 1-10.

```
ret,second,third = unoObject.foo( 2, 5 , None )
print ret,second,third    # results into 2,10,7
```

Listagem 1-10. Chamada de método com parâmetro `out`.

Para valores puramente `out`, o valor *dummy* `None` é usado, como observa-se na Listagem 1-10.

Deve-se notar, no entanto, que:

- o número correto de valores de retorno deve estar na chamada ou na implementação do código; caso contrário, um `RuntimeException` será lançado durante a chamada.
- um método `void` sempre retorna um `None` seguido de possíveis parâmetros de saída; então quando houver método `void` com apenas um parâmetro, deve-se atribuir a saída a duas variáveis (sendo que a primeira será `None`).

1.3.3 Exemplo de código (TableSample)

Para o desenvolvimento do projeto, precisou-se estudar profundamente Python UNO e o estudo julgado mais importante foi de exemplos de código, sendo que esse exemplo (retirado de [4]), talvez tenha sido o mais importante.

```
import uno

# a UNO struct later needed to create a document
from com.sun.star.text.ControlCharacter import PARAGRAPH_BREAK
from com.sun.star.text.TextContentAnchorType import AS_CHARACTER
from com.sun.star.awt import Size

from com.sun.star.lang import XMain

def insertTextIntoCell( table, cellName, text, color ):
      tableText = table.getCellByName( cellName )
```

```
        cursor = tableText.createTextCursor()
        cursor.setPropertyValue( "CharColor", color )
        tableText.setString( text )

def createTable():
        """creates a new writer document and inserts a table with some data (also known
as the SWriter sample)"""
        ctx = uno.getComponentContext()
        smgr = ctx.ServiceManager
        desktop = smgr.createInstanceWithContext( "com.sun.star.frame.Desktop",ctx)

        # open a writer document
        doc = desktop.loadComponentFromURL( "private:factory/swriter","_blank", 0, () )

        text = doc.Text
        cursor = text.createTextCursor()
        text.insertString( cursor, "The first line in the newly created text
document.\n", 0 )
        text.insertString( cursor, "Now we are in the second line\n" , 0 )

        # create a text table
        table = doc.createInstance( "com.sun.star.text.TextTable" )

        # with 4 rows and 4 columns
        table.initialize( 4,4)

        text.insertTextContent( cursor, table, 0 )
        rows = table.Rows

        table.setPropertyValue( "BackTransparent", uno.Bool(0) )
        table.setPropertyValue( "BackColor", 13421823 )
        row = rows.getByIndex(0)
        row.setPropertyValue( "BackTransparent", uno.Bool(0) )
        row.setPropertyValue( "BackColor", 6710932 )

        textColor = 16777215

        insertTextIntoCell( table, "A1", "FirstColumn", textColor )
        insertTextIntoCell( table, "B1", "SecondColumn", textColor )
        insertTextIntoCell( table, "C1", "ThirdColumn", textColor )
        insertTextIntoCell( table, "D1", "SUM", textColor )
```

```
	values = ( (22.5,21.5,121.5),
		(5615.3,615.3,-615.3),
		(-2315.7,315.7,415.7) )
	table.getCellByName("A2").setValue(22.5)
	table.getCellByName("B2").setValue(5615.3)
	table.getCellByName("C2").setValue(-2315.7)
	table.getCellByName("D2").setFormula("sum <A2:C2>")

	table.getCellByName("A3").setValue(21.5)
	table.getCellByName("B3").setValue(615.3)
	table.getCellByName("C3").setValue(-315.7)
	table.getCellByName("D3").setFormula("sum <A3:C3>")

	table.getCellByName("A4").setValue(121.5)
	table.getCellByName("B4").setValue(-615.3)
	table.getCellByName("C4").setValue(415.7)
	table.getCellByName("D4").setFormula("sum <A4:C4>")



	cursor.setPropertyValue( "CharColor", 255 )
	cursor.setPropertyValue( "CharShadowed", uno.Bool(1) )

	text.insertControlCharacter( cursor, PARAGRAPH_BREAK, 0 )
	text.insertString( cursor, " This is a colored Text - blue with shadow\n" , 0 )
	text.insertControlCharacter( cursor, PARAGRAPH_BREAK, 0 )

	textFrame = doc.createInstance( "com.sun.star.text.TextFrame" )
	textFrame.setSize( Size(15000,400))
	textFrame.setPropertyValue( "AnchorType" , AS_CHARACTER )

	text.insertTextContent( cursor, textFrame, 0 )

	textInTextFrame = textFrame.getText()
	cursorInTextFrame = textInTextFrame.createTextCursor()
	textInTextFrame.insertString( cursorInTextFrame, "The first line in the newly
created text frame.", 0 )
	textInTextFrame.insertString( cursorInTextFrame, "\nWith this second line the
height of the rame raises.",0)
	text.insertControlCharacter( cursor, PARAGRAPH_BREAK, 0 )

	cursor.setPropertyValue( "CharColor", 65536 )
	cursor.setPropertyValue( "CharShadowed", uno.Bool(0) )
```

```
      text.insertString( cursor, " That's all for now !!" , 0 )

g_exportedScripts = createTable,
```

Listagem 1-11. `TableSample` – Exemplo de código completo em PyUNO.

Um módulo UNO fornecerá um *binding* do IDL da API. Daí surge a necessidade de se importar artefatos específicos de um módulo UNO. Para isso, precisa-se especificar o caminho `com.sun.star` e importar interfaces e serviços.

```
import uno

# a UNO struct later needed to create a document
from com.sun.star.text.ControlCharacter import PARAGRAPH_BREAK
from com.sun.star.text.TextContentAnchorType import AS_CHARACTER
from com.sun.star.awt import Size

from com.sun.star.lang import XMain
```

Listagem 1-12. `TableSample` – Importação de interfaces e serviços.

Pode-se dividir a classe como um processo de:

- criar uma nova janela.
- criar um documento do OpenOffice.org Writer.
- inserir texto.
- criar uma tabela dentro do documento do Writer.
- manipular o formato e o conteúdo.
- inserir um quadro com texto dentro.
- inserir uma string.

O método `loadcomponentFromURL()` cria e carrega um documento do OpenOffice.org Writer, gerando a variável. Texto é uma variável que irá obter o objeto do documento com o módulo `Text`. A função `createTextCursor()`, de acordo com sua descrição em UNOIDL, cria uma nova instância de serviço `TextCursor` a qual pode ser usada para explorar o conteúdo do contexto do `Text`.

```
      """creates a new writer document and inserts a table with some data (also known
```

```
as the SWriter sample)"""
        ctx = uno.getComponentContext()
        smgr = ctx.ServiceManager
        desktop = smgr.createInstanceWithContext( "com.sun.star.frame.Desktop",ctx)

        # open a writer document
        doc = desktop.loadComponentFromURL( "private:factory/swriter","_blank", 0, () )

        text = doc.Text
        cursor = text.createTextCursor()
        text.insertString( cursor, "The first line in the newly created text
document.\n", 0 )
        text.insertString( cursor, "Now we are in the second line\n" , 0 )
```

Listagem 1-13. `TableSample` – Criação de novo documento do writer e de cursor.

A criação de uma instância de tabela é feita através da função `createTextCursor()`, cujo caminho é com.`sun.star.text.TextTable`.

```
        # create a text table
        table = doc.createInstance( "com.sun.star.text.TextTable" )
```

Listagem 1-14. `TableSample` – Criação de instância de tabela.

Cria-se a tabela através do método `Initialize` (`table.initialize(4,4)`). Com `table` inicializado, lida-se com as propriedades da tabela assim como métodos e objetos da tabela como linhas, colunas, cursores e valores. Usa-se então a função `insertTextContent(cursor, table, 0)` para colocar um cursor dentro da tabela.

```
        # with 4 rows and 4 columns
        table.initialize( 4,4)

        text.insertTextContent( cursor, table, 0 )
        rows = table.Rows
```

Listagem 1-15. `TableSample` – Determina-se tamanho da tabela e insere-se cursor dentro da tabela.

Em seguida, o foco é dado ao estilo da tabela e métodos como `setPropertyValue` são utilizados para dar valor às propriedades dos estilos.

```
	table.setPropertyValue( "BackTransparent", uno.Bool(0) )
	table.setPropertyValue( "BackColor", 13421823 )
	row = rows.getByIndex(0)
	row.setPropertyValue( "BackTransparent", uno.Bool(0) )
	row.setPropertyValue( "BackColor", 6710932 )
```

Listagem 1-16. `TableSample` – Alterações no estilo da tabela.

A seguir, no código, são alterados no cabeçalho onde é inserido texto na célula através do `insertTextIntoCell( table, "A1", "FirstColumn", textColor )`, assim como a cor do texto é alterada.

```
	textColor = 16777215

	insertTextIntoCell( table, "A1", "FirstColumn", textColor )
	insertTextIntoCell( table, "B1", "SecondColumn", textColor )
	insertTextIntoCell( table, "C1", "ThirdColumn", textColor )
	insertTextIntoCell( table, "D1", "SUM", textColor )
```

Listagem 1-17. `TableSample` – Altera cor do texto e insere-se conteúdo dentro da tabela.

Agora sera inserido conteúdo dentro das células da tabela. Serão utilizados os métodos `setValue()` e `setFormula()` em `getCellByName()`.

```
	values = ( (22.5,21.5,121.5),
		(5615.3,615.3,-615.3),
		(-2315.7,315.7,415.7) )
	table.getCellByName("A2").setValue(22.5)
	table.getCellByName("B2").setValue(5615.3)
	table.getCellByName("C2").setValue(-2315.7)
	table.getCellByName("D2").setFormula("sum <A2:C2>")
```

Listagem 1-18. `TableSample` – Conteúdo é inserido dentro da tabela.

1.4 CUDA

1.4.1 Introdução

CUDA é a arquitetura de computação paralela da NVIDIA que possibilita aumentos significativos na performance de computação pelo aproveitamento da potência da GPU (unidade de processamento gráfico).

Com milhões de GPUs habilitadas para CUDA vendidas até o momento, desenvolvedores de software, cientistas e pesquisadores continuam achando os mais diversos usos para a CUDA, incluindo processamento de vídeos e imagens, biologia e química computacionais, simulação de dinâmica de fluidos, reconstrução de imagens de Tomografia Computadorizada, análise sísmica, traçado de raios entre outros.

A tecnologia baseia-se na mudança do paradigma do “processamento central” na CPU para um “coprocessamento” na CPU e na GPU. Para permitir esse novo paradigma em computação, a NVIDIA inventou a arquitetura de computação paralela CUDA inclusa em diversos modelos de placas gráficas, representando uma base instalada significativa para os desenvolvedores de aplicativos.

Do lado dos desenvolvedores de software, espera-se que quase todas as principais aplicações de vídeo serão em breve aceleradas pela tecnologia CUDA,

A tecnologia CUDA tem sido recebida com entusiasmo pela área de pesquisa científica. Uma das aplicações da tecnologia no momento é na aceleração do AMBER, um programa de simulação de dinâmica de moléculas utilizado por mais de 60 mil pesquisadores no meio acadêmico e em empresas farmacêuticas no mundo inteiro para acelerar a descoberta de novos remédios.

Especificamente no mercado financeiro, a Numerix e a CompatibL anunciaram recentemente o suporte a CUDA para um novo aplicativo de risco de terceiros e reportam ter alcançado um ganho de 18x na velocidade de seus aplicativos.

A adoção do CUDA como tecnologia de execução de algoritmos está crescendo, demonstrando como a tecnologia está rendendo um bom retorno sobre investimento. Atualmente, há mais de 700 clusters de GPUs instalados no mundo inteiro em empresas que fazem parte da Fortune 500, desde a Schlumberger e a Chevron no setor energético até o BNP Paribas no setor bancário.

Quanto à compatibilidade com sistemas operacionais, o Linux, o Microsoft Windows 7 e o Snow Leopard da Apple já possuem todos suporte completo ao processamento na

GPU, sendo possível utilizá-la como um processador paralelo de finalidade geral acessível a qualquer aplicativo.

A discrepância obtida nas operações de ponto flutuante entre CPUs e GPUs é dada pela especialização que a GPU tem em computação intensiva, altamente paralela – exatamente o que se faz em uma placa gráfica – e por isso é desenhada de forma a ter mais transistors focados em processamento de dados que em caching de dados e controle de fluxo, como mostrado na Figura 1-5.

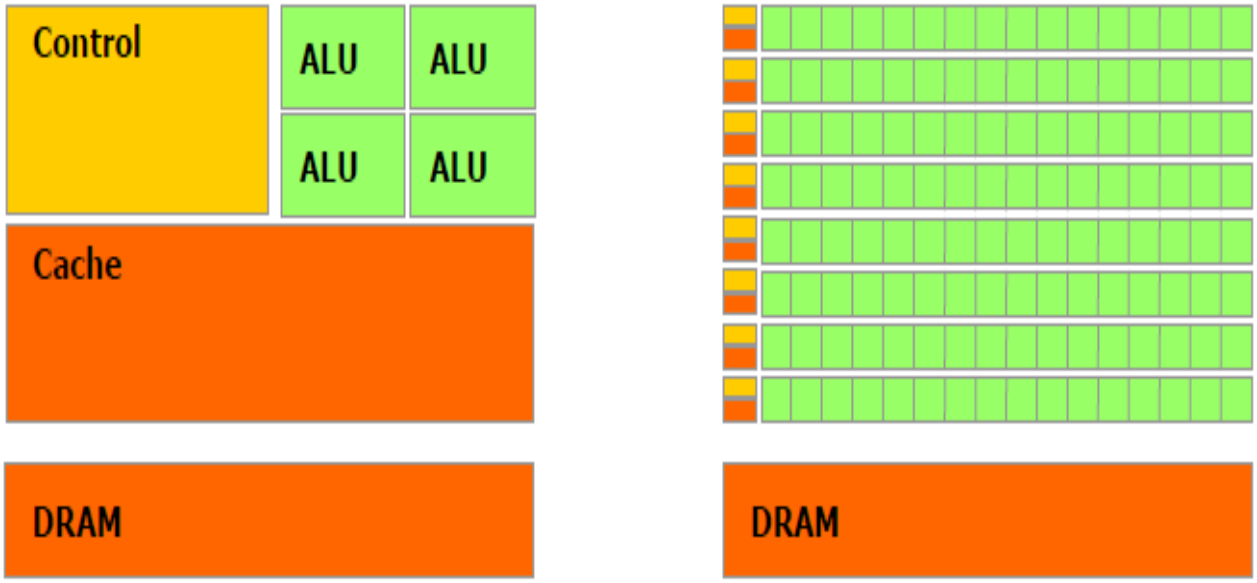


Figura 1-5. Representação esquemática demonstrando o maior número de transistors nos GPUs da NVIDIA comparados a um processador comum. Fonte: NVIDIA.

Mais especificamente, a GPU é especialmente feita para resolver problemas que podem ser descritos em computação de dados paralela – o mesmo programa é executado para vários elementos de dados em paralelo – com forte intensidade em operações aritméticas. Como o mesmo programa é executado para cada elemento de dados, há um requerimento menor por controle de fluxo sofisticado e também como é executado para diversos elementos de dados simultaneamente e tem alta intensidade em operações aritméticas, a latência de acesso a memória é escondida pelos cálculos no lugar de caches de dados enorme.

1.4.2 Arquitetura

A arquitetura do CUDA consiste de diversos componentes. Na Figura 1-6, observa-se:

1. Engines de computação paralela dentro das GPU's da NVIDIA.
2. Suporte a nível de kernel do Sistema Operacional para inicialização, configuração, etc.
3. Driver em modo usuário, provendo API em nível de dispositivo para desenvolvedores.
4. Arquitetura PTX (*Instruction Set Architecture*, ISA) para a computação paralela e funções.

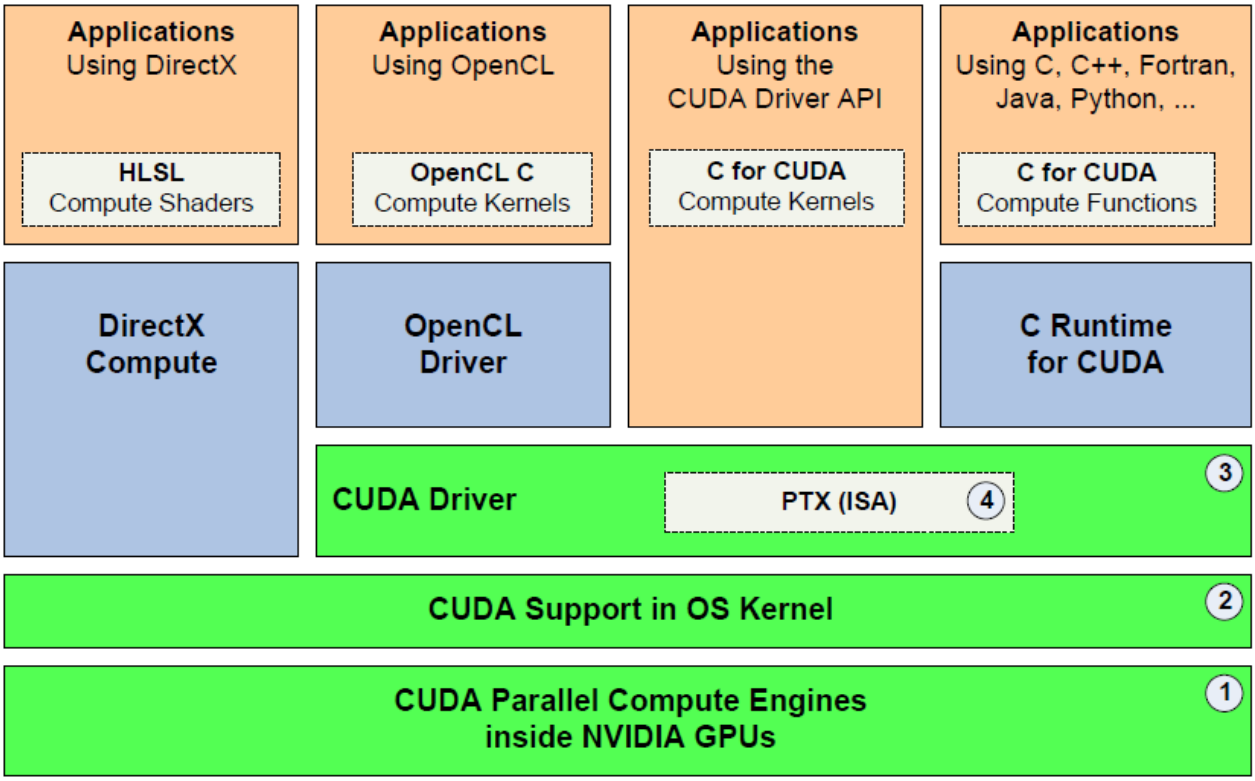


Figura 1-6. Arquitetura CUDA com seus diversos componentes. Fonte: NVIDIA.

A arquitetura do CUDA permite, de maneira análoga à do UNO, que várias linguagens de programação sejam utilizadas na tecnologia.

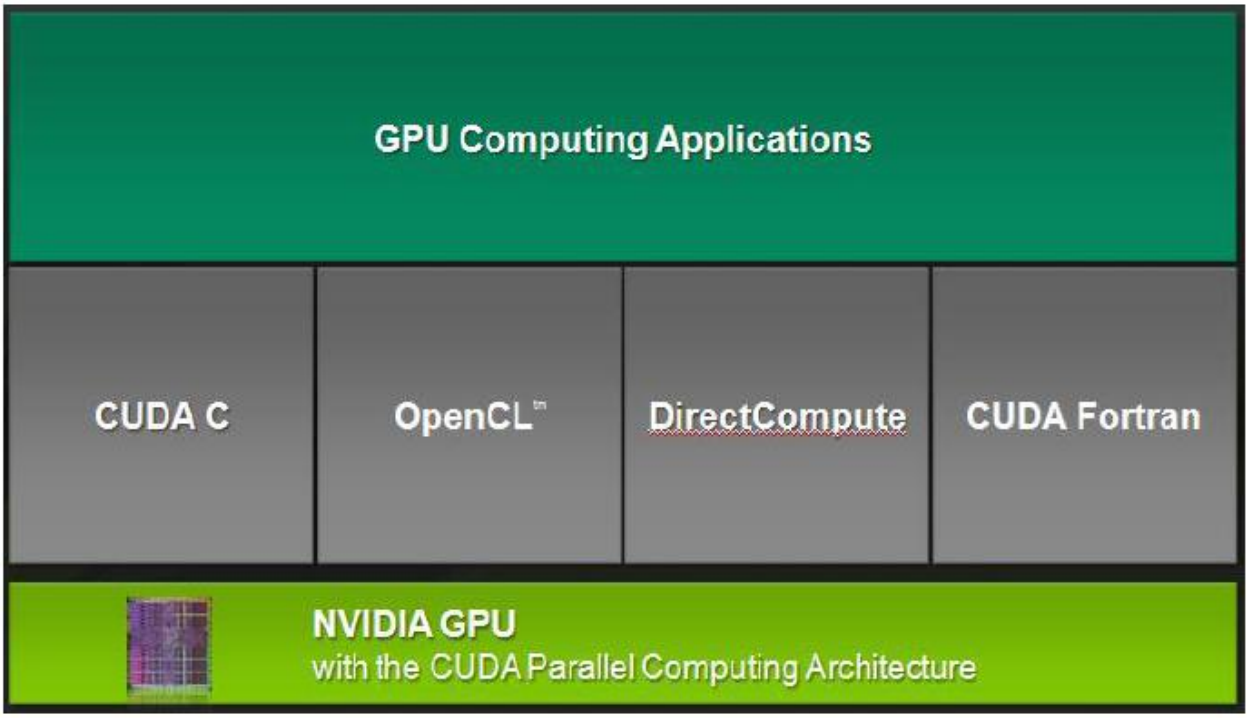


Figura 1-7. Arquitetura possibilitando a utilização de diversas linguagens de programação para o CUDA. Fonte: NVIDIA.

1.4.3 Modelo de Programação

CUDA C estende a linguagem C permitindo ao programador definir funções em C, chamados *kernels*, que, quando chamamos, são executados N vezes em paralelo por N diferentes *threads*, oposto às funções comuns em C.

Um *kernel* é definido usando-se a declaração `__global__` e o número de threads CUDA que irão executar o kernel é definido na chamada do kernel, usando uma sintaxe <<< ... >>> de execução. Cada thread que executa um kernel recebe um único *thread ID* que é acessível de dentro do kernel através da variável `threadIdx`.

A Listagem 1-19 abaixo é um código de exemplo que adiciona dois vetores A e B de tamanho N e guarda o resultado em C.

```
// Kernel definition
__global__ void VecAdd(float* A, float* B, float* C)
{
```

```
        int i = threadIdx.x;
        C[i] = A[i] + B[i];
}
int main()
{
        // Kernel invocation with N threads
        VecAdd<<<1, N>>>(A, B, C);
}
```

Listagem 1-19. Função simples para cálculo da soma de dois vetores.

No exemplo da Listagem 1-19 cada uma das N threads que executam `VecAdd()` realizam uma adição apenas.

Tem-se que `threadIdx` é um vetor de 3 componentes, de forma que cada thread pode ser identificada por um *thread index* unidimensional, bidimensional ou tridimensional, formando blocos de *thread* igualmente unidimensionais, bidimensionais e tridimensionais.

O índice de uma thread e seu thread ID relacionam-se de forma intuitiva: para um bloco unidimensional, eles são o mesmo; para um bloco bidimensional (D_x, D_y), a thread ID de uma thread de índice (x, y) é ($x + yD_y$); para um bloco tridimensional (D_x, D_y, D_z), a relação é que a thread ID de uma thread de índice (x, y, z) é ($x + yD_x + zD_xD_y$).
A Listagem 1-20 mostra um exemplo com um código que adiciona duas matrizes A e B de tamanhos NxN e guardam o resultado em C.

```
// Kernel definition
__global__ void MatAdd(float A[N][N], float B[N][N], float C[N][N])
{
        int i = threadIdx.x;
        int j = threadIdx.y;
        C[i][j] = A[i][j] + B[i][j];
}

int main()
{
        // Kernel invocation with one block of N * N * 1 threads
        int numBlocks = 1;
```

```
        dim3 threadsPerBlock(N, N);
        MatAdd<<<numBlocks, threadsPerBlock>>>(A, B, C);
}
```

Listagem 1-20. Função simples para cálculo da soma de duas matrizes em blocos bidimensionais.

Há um limite para o número de threads por bloco, dado que todas as threads de um bloco devem residir no mesmo processador *core* e devem compartilhar os recursos limitados de memória desse processador. Nos GPUs atuais, uma bloco de thread pode ter até 1024 threads.

No entanto, o kernel pode ser executado por múltiplos blocos de thread de mesmo número de threads, de forma tal que o total de threads executando o kernel é igual ao tamanho de cada bloco multiplicado pelo número de threads por bloco.

Blocos são geralmente organizados em grades de uma ou duas dimensões de blocos de threads, como ilustrado na Figura 1-8. O número de blocos de threads em uma grade é geralmente ditado pelo tamanho do dado sendo processado ou pelo número de processadores no sistema, que pode ser muito excedido.

O número de threads por bloco e o número de blocos por grade especificado na sintaxe `<< ... >>` pode ser do tipo `int` ou `dim3`.

Cada bloco dentro da grade pode ser identificado por um índice unidimensional ou bidimensional, acessível de dentro do kernel através da variável embutida `threadIdx`. A dimensão de cada bloco de thread é acessível de dentro do kernel também através da variável embutida `blockDim`.

Estendendo o exemplo anterior `MatAdd()` para manipular múltiplos blocos, o código obtido deve ser semelhante ao da Listagem 1-21.

```
// Kernel definition
__global__ void MatAdd(float A[N][N], float B[N][N], float C[N][N])
{
        int i = blockIdx.x * blockDim.x + threadIdx.x;
        int j = blockIdx.y * blockDim.y + threadIdx.y;
        if (i < N && j < N)
                C[i][j] = A[i][j] + B[i][j];
}

int main() {
```

```
        // Kernel invocation
        dim3 threadsPerBlock(16, 16);
        dim3 numBlocks(N / threadsPerBlock.x, N / threadsPerBlock.y);
        MatAdd<<<numBlocks, threadsPerBlock>>>(A, B, C);
}
```

Listagem 1-21. Exemplo da Listagem 1-20 para múltiplos blocos.

Um tamanho comumente escolhido é o 16x16 (256 threads), também usado no exemplo da Listagem 1-21. A grade é criada com blocos suficientes para se ter uma thread por elemento na matriz como antes. Por simplicidade, esse exemplo assume que o número de threads por grade em cada dimensão é divisível pelo número de threads por grade na dimensão dada, o que não precisa necessariamente ser o caso.

Blocos de threads devem ser capazes de serem executados independentemente. Ou seja, podem executar em qualquer ordem, em paralelo ou em série. Essa necessidade de independência permite a blocos de thread serem colocados em qualquer ordem em qualquer número de cores como ilustrado na Figura 1-8, permitindo a programadores escrever código que escala bem com o número de *cores*.

Threads dentro de um bloco podem cooperar através do compartilhamento de dados por *shared memory* (memória compartilhada) e por sincronização da execução para coordenar o acesso à memória. Mais precisamente, o programador pode requisitar sincronização no kernel chamando a função intrínseca `__syncthreads()`; `__syncthreads` age como uma barreira onde todas as threads do bloco devem esperar antes que seja permitida a continuação.

Para uma cooperação eficiente, espera-se que a *shared memory* seja uma memória de baixa latência localizada próxima ao processador *core* (como uma memória L1) e aque `__syncthreads()` seja leve.

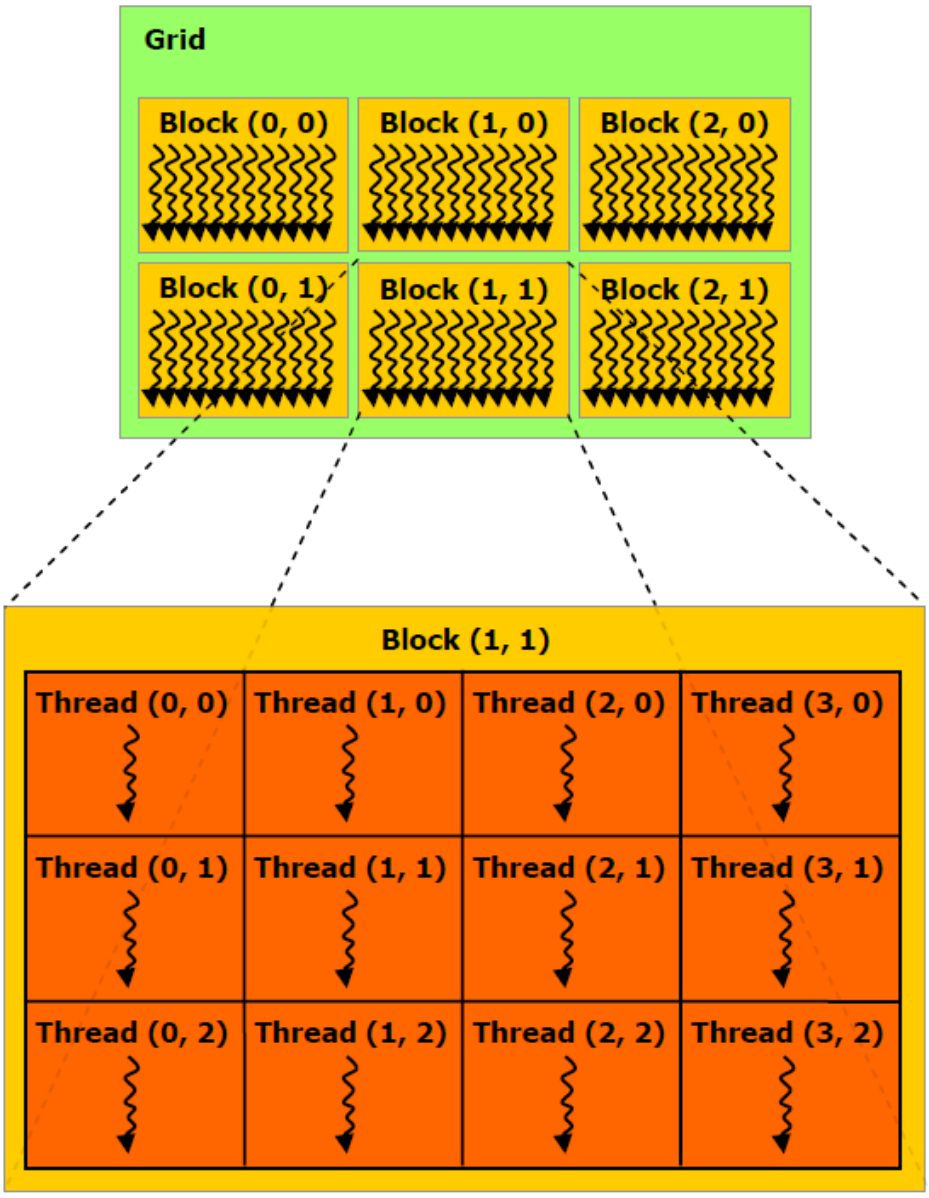


Figura 1-8. Organização de blocos e grades no *layout* de threads de um dispositivo CUDA. Fonte: NVIDIA.

Threads do CUDA podem acessar memória de diversos espaços de memória distintos durante sua execução, como ilustrado na Figura 1-9.

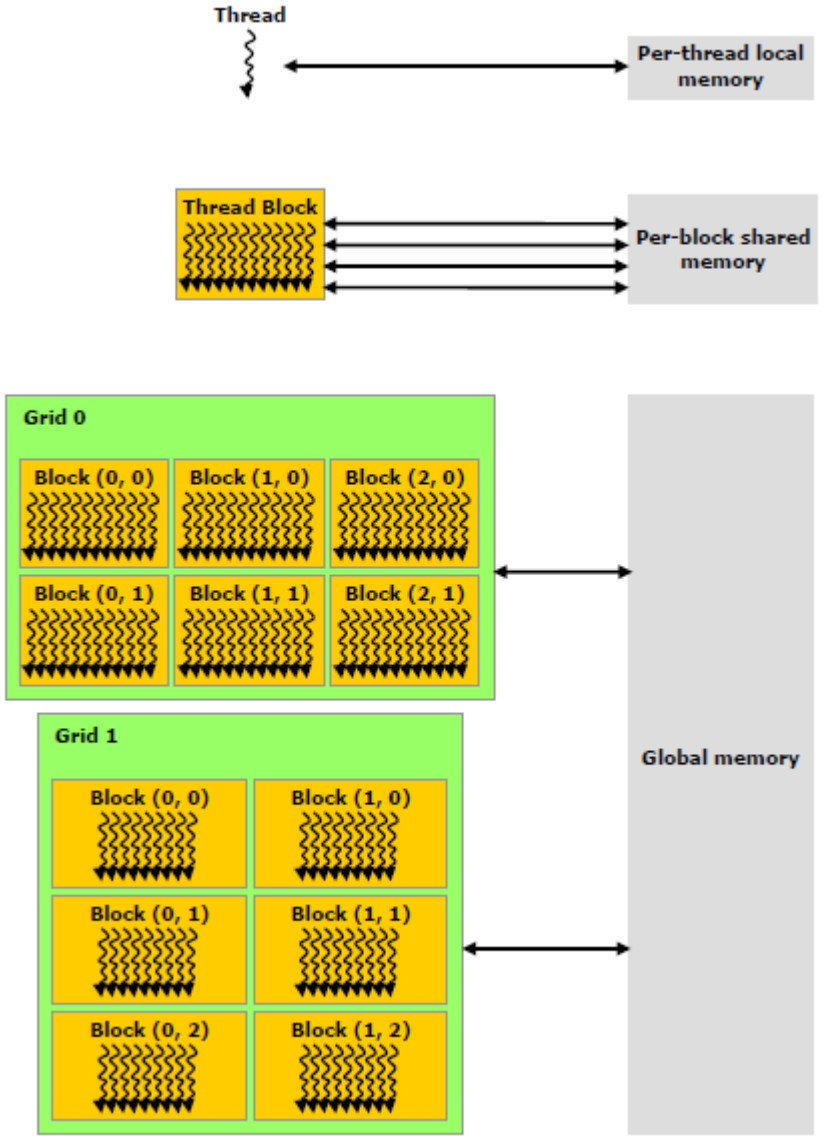


Figura 1-9. Acesso das threads do CUDA a diversos espaços de memória. Fonte: NVIDIA.

Cada thread tem sua memória local privada. Cada bloco de threads tem uma shared memory visível a todas as threads do bloco e durante a vida do bloco.Todas as threads tem acesso à mesma memória global.

Há também duas memórias somente leitura acessíveis a todas as threads: o espaço das constantes e das texturas. A global, constante, e a de textura são otimizadas para

diferentes usos de memória. Memória de texturas também oferece diferentes modos de endereçamento, assim como filtragem de dados para específicos formatos.
A memória global constante e a memória para texturas são persistentes por execuções de kernel pela mesma aplicação.

1.4.4 Interface de Programação

Duas interfaces são suportadas para escrever programas em CUDA: CUDA C e o CUDA *Driver API*. Uma aplicação tipicamente usa um ou outro, mas também pode usar ambos.
CUDA C expõe o CUDA programming model como um conjunto mínimo de extensões à linguagem C. Qualquer código fonte que contenha algumas dessas extensões devem ser compilados com o `nvcc`. Essas extensões permitem aos programadores definir um kernel como uma função C e usar uma nova sintaxe para especificar a grade e a dimensão do bloco cada hora que a função for chamada.
O CUDA *driver API* é uma API em C mais baixo nível que provê funções para carregar kernels como módulos de binário CUDA ou código assembly, para inspecionar seus parâmetros e para iniciá-los. Código binário e em assembly são obtidos compilando kernels escritos em C.
CUDA C possui uma API de tempo de execução e ambas as APIs de tempo de execução e *driver API* fornecem funções para alocar e desalocar memória do dispositivo, transferir dados entre memória do CPU e do dispositivo, gerenciar sistemas com diversos dispositivos, etc.
A API de tempo de execução é construída em cima do CUDA *driver API*. Inicialização, contexto e gerenciamento de módulo são todos implícitos e o código resultante é mais conciso.
Em contraste, o CUDA *driver API* requer mais código, é mais difícil de programar e debugar, mas oferece um nível maior de controle e é independente da linguagem utilizada já que lida com código binário e assembly.

Kernels podem ser escritos usando o CUDA ISA (Instruction Set Architecture), chamado PTX. No entanto é mais efetivo o uso de uma linguagem de alto nível como C. Em ambos os casos, kernels precisam ser compilados em código binário pelo `nvcc` para ser executado em um dispositivo.

O compilador `nvcc` é na verdade um *compiler driver* e serve para simplificar o processo de compilação de código C e PTX: ele fornece opções de linha de comando simples e familiares e as executa invocando um conjunto de ferramentas que implementam diferentes estágios de compilação.

Códigos fonte compilados pelo `nvcc` podem incluir um mix de código para o *host* (CPU) e código do dispositivo (GPU). O *workflow* básico consiste em separar código do dispositivo do código do *host* e compilar o código do dispositivo para a forma assembly (código PTX) ou sua forma binária (objeto *cubin*). O código do *host* gerado pode ser tanto código C deixado para ser compilado por outra ferramenta ou código objeto diretamente, deixando que o `nvcc` chame o compilador do `host` na última etapa de compilação.

Aplicações podem

- Carregar e executar código PTX ou objetos *cubin* no dispositivo usando o CUDA *driver API* e ignorar o código gerado para o host (se houver).
- Ligar o código gerado pelo *host*; o código gerado para o *host* inclui código PTX e objetos cubin como um vetor de dados globais inicializados e uma tradução da sintaxe `<< ... >>` no código necessário para as chamadas de função em tempo de execução do CUDA C para carregar e iniciar cada kernel compilado.

Qualquer código PTX carregado por uma aplicação em tempo de execução é compilado para código binário pelo driver do dispositivo. A isso dá-se o nome de *just-in-time compilation*. *Just-in-time compilation* aumenta o tempo de carregamento da aplicação, mas permite que aplicações se beneficiem da atualização dos compiladores. Também é a única forma das aplicações rodarem em dispositivos que não existiam a época da compilação da aplicação.

Código binário é específico para cada arquitetura. Um objeto *cubin* é gerado usando a opção de compilador `-code` que especifica uma arquitetura-alvo. Por exemplo, compilar

com `-code=sm_13` produz código binário para dispositivos com capacidade de computação 1.3. Compatibilidade binária é garantida de uma revisão *minor* para a próxima, mas não para a anterior ou entre diferenças maiores. Em outras palavras, um objeto *cubin* gerado para a capacidade de computação X.y só é garantido executar em dispositivos de capacidade X.z onde z≥y.

1.4.5 CUDA C

O CUDA C provê um caminho simples para usuários familiares com a linguagem de programação C conseguirem escrever facilmente seus programas para execução em dispositivos.

Ela consiste de um conjunto mínimo de extensões para a linguagem de programação C e uma biblioteca de tempo de execução.

A biblioteca de tempo de execução é implementada pela biblioteca dinâmica `cudart` e todos os seus *entry points* estão marcados com o prefixo `cuda`.

Não há uma função de inicialização explícita para a execução; o sistema é inicializado na primeira chamada em tempo de execução (mais especificamente a chamada de qualquer função exceto funções do dispositivo e funções de versionamento). Há a necessidade de se tomar cuidado com o momento em que tais funções são chamadas pela primeira vez para a verificação de erros.

Uma vez que o *runtime* esteja inicializado em um thread do *host*, qualquer recurso (memória, evento, etc.) alocado via alguma chamada de função de tempo de execução na thread do *host* somente será válida dentro do contexto da thread do *host*. Logo apenas chamadas de tempo de execução feitas pela thread do *host* (cópias de memória, chamadas de kernel, ...) podem operar nesses recursos. Isso acontece porque o contexto CUDA é criado em plano de fundo como parte da inicialização e configurado como padrão para a thread do *host*, e não pode ser associado a mais nenhuma outra thread do *host*.

Para mostrar a estrutura de um código em C para CUDA genérico, a Listagem 1-22 obtida do kit de desenvolvimento de CUDA disponibilizado pela NVIDIA e a explicação somente das partes que tangem ao CUDA estão abaixo reproduzidas.

```
/*
 * Copyright 1993-2010 NVIDIA Corporation.  All rights reserved.
 *
 * Please refer to the NVIDIA end user license agreement (EULA) associated
 * with this source code for terms and conditions that govern your use of
 * this software. Any use, reproduction, disclosure, or distribution of
 * this software and related documentation outside the terms of the EULA
 * is strictly prohibited.
 *
 */

/* Vector addition: C = A + B.
 *
 * This sample is a very basic sample that implements element by element
 * vector addition. It is the same as the sample illustrating Chapter 3
 * of the programming guide with some additions like error checking.
 *
 */

// Includes
#include <stdio.h>
#include <cutil_inline.h>

// Variables
float* h_A;
float* h_B;
float* h_C;
float* d_A;
float* d_B;
float* d_C;
bool noprompt = false;

// Functions
void Cleanup(void);
void RandomInit(float*, int);
void ParseArguments(int, char**);
```

```
// Device code
__global__ void VecAdd(const float* A, const float* B, float* C, int N)
{
    int i = blockDim.x * blockIdx.x + threadIdx.x;
    if (i < N)
        C[i] = A[i] + B[i];
}

// Host code
int main(int argc, char** argv)
{
    printf("Vector addition\n");
    int N = 50000;
    size_t size = N * sizeof(float);
    ParseArguments(argc, argv);

    // Allocate input vectors h_A and h_B in host memory
    h_A = (float*)malloc(size);
    if (h_A == 0) Cleanup();
    h_B = (float*)malloc(size);
    if (h_B == 0) Cleanup();
    h_C = (float*)malloc(size);
    if (h_C == 0) Cleanup();

    // Initialize input vectors
    RandomInit(h_A, N);
    RandomInit(h_B, N);

    // Allocate vectors in device memory
    cutilSafeCall( cudaMalloc((void**)&d_A, size) );
    cutilSafeCall( cudaMalloc((void**)&d_B, size) );
    cutilSafeCall( cudaMalloc((void**)&d_C, size) );

    // Copy vectors from host memory to device memory
    cutilSafeCall( cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice) );
    cutilSafeCall( cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice) );

    // Invoke kernel
    int threadsPerBlock = 256;
    int blocksPerGrid = (N + threadsPerBlock - 1) / threadsPerBlock;
    VecAdd<<<blocksPerGrid, threadsPerBlock>>>(d_A, d_B, d_C, N);
    cutilCheckMsg("kernel launch failure");
```

```
#ifdef _DEBUG
    cutilSafeCall( cudaThreadSynchronize() );
#endif

    // Copy result from device memory to host memory
    // h_C contains the result in host memory
    cutilSafeCall( cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost) );

    // Verify result
    int i;
    for (i = 0; i < N; ++i) {
        float sum = h_A[i] + h_B[i];
        if (fabs(h_C[i] - sum) > 1e-5)
            break;
    }
    printf("%s \n", (i == N) ? "PASSED" : "FAILED");

    Cleanup();
}

void Cleanup(void)
{
    // Free device memory
    if (d_A)
        cudaFree(d_A);
    if (d_B)
        cudaFree(d_B);
    if (d_C)
        cudaFree(d_C);

    // Free host memory
    if (h_A)
        free(h_A);
    if (h_B)
        free(h_B);
    if (h_C)
        free(h_C);

    cutilSafeCall( cudaThreadExit() );

    if (!noprompt) {
        printf("\nPress ENTER to exit...\n");
```

```
        fflush( stdout);
        fflush( stderr);
        getchar();
    }

    exit(0);
}

// Allocates an array with random float entries.
void RandomInit(float* data, int n)
{
    for (int i = 0; i < n; ++i)
        data[i] = rand() / (float)RAND_MAX;
}

// Parse program arguments
void ParseArguments(int argc, char** argv)
{
    for (int i = 0; i < argc; ++i)
        if (strcmp(argv[i], "--noprompt") == 0 ||
                    strcmp(argv[i], "-noprompt") == 0)
             {
            noprompt = true;
            break;
        }
}
```

Listagem 1-22. `VecAdd` – Exemplo completo de código em CUDA.

Como explicado no item 1.4.3, uma função do CUDA é definida pelo marcador `__global__` antes de sua declaração em C normal. Em outras palavras, a função VecAdd está sendo executada dentro de uma thread da GPU. O código abaixo também demonstra a obtenção do número da thread pela própria thread.

```
// Device code
__global__ void VecAdd(const float* A, const float* B, float* C, int N)
{
    int i = blockDim.x * blockIdx.x + threadIdx.x;
    if (i < N)
        C[i] = A[i] + B[i];
}
```

Listagem 1-23. `VecAdd` – Método de adição de vetores.

A função `cutilSafeCall` serve para que cada chamada ao CUDA tenha um retorno não abrupto em caso de erro e mostre uma mensagem de erro ao invés de falhar silenciosamente.

A função `cudaMalloc` é responsável por alocar espaço na memória do GPU (*device*), diferentemente da função `malloc` do C que aloca espaço na memória principal (*host*). Para copiar memória entre o host e o device também se utiliza uma função especial do CUDA, `cudaMemcpy`, responsável por realizar a cópia eficientemente em baixo nível. O parâmetro cudaMemcpyHostToDevice especifica que a memória deve ser copiada do host para o device, enquanto a utilização do parâmetro cudaMemcpyDeviceToHost especifica o inverso.

```
    // Allocate vectors in device memory
    cutilSafeCall( cudaMalloc((void**)&d_A, size) );
    cutilSafeCall( cudaMalloc((void**)&d_B, size) );
    cutilSafeCall( cudaMalloc((void**)&d_C, size) );

    // Copy vectors from host memory to device memory
    cutilSafeCall( cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice) );
    cutilSafeCall( cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice) );
```

Listagem 1-24. `VecAdd` – Alocação e cópia de memória do *host* para o *device*.

```
    // Copy result from device memory to host memory
    // h_C contains the result in host memory
    cutilSafeCall( cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost) );
```

Listagem 1-25. VecAdd - Cópia de memória do *device* para o *host*.

E o ultimo ponto a ser analisado no código fornecido é a chamada da própria função `VecAdd`. Observa-se que na chamada deve ser especificado o número de blocos e o número de threads por bloco que deve ser utilizado, como especificado e exemplificado no item 1.4.3.

1.4.6 PTX: Parallel Thread Execution

Assim como CPUs, as GPUs também possuem seus códigos de máquina. No caso das GPUs da NVIDIA que suportam CUDA, esse código é o PTX. É para ele que o nvcc compila o código dado.

A explicação do conjunto de instruções PTX foge ao escopo desse trabalho.

1.5 Negociação de ativos financeiros

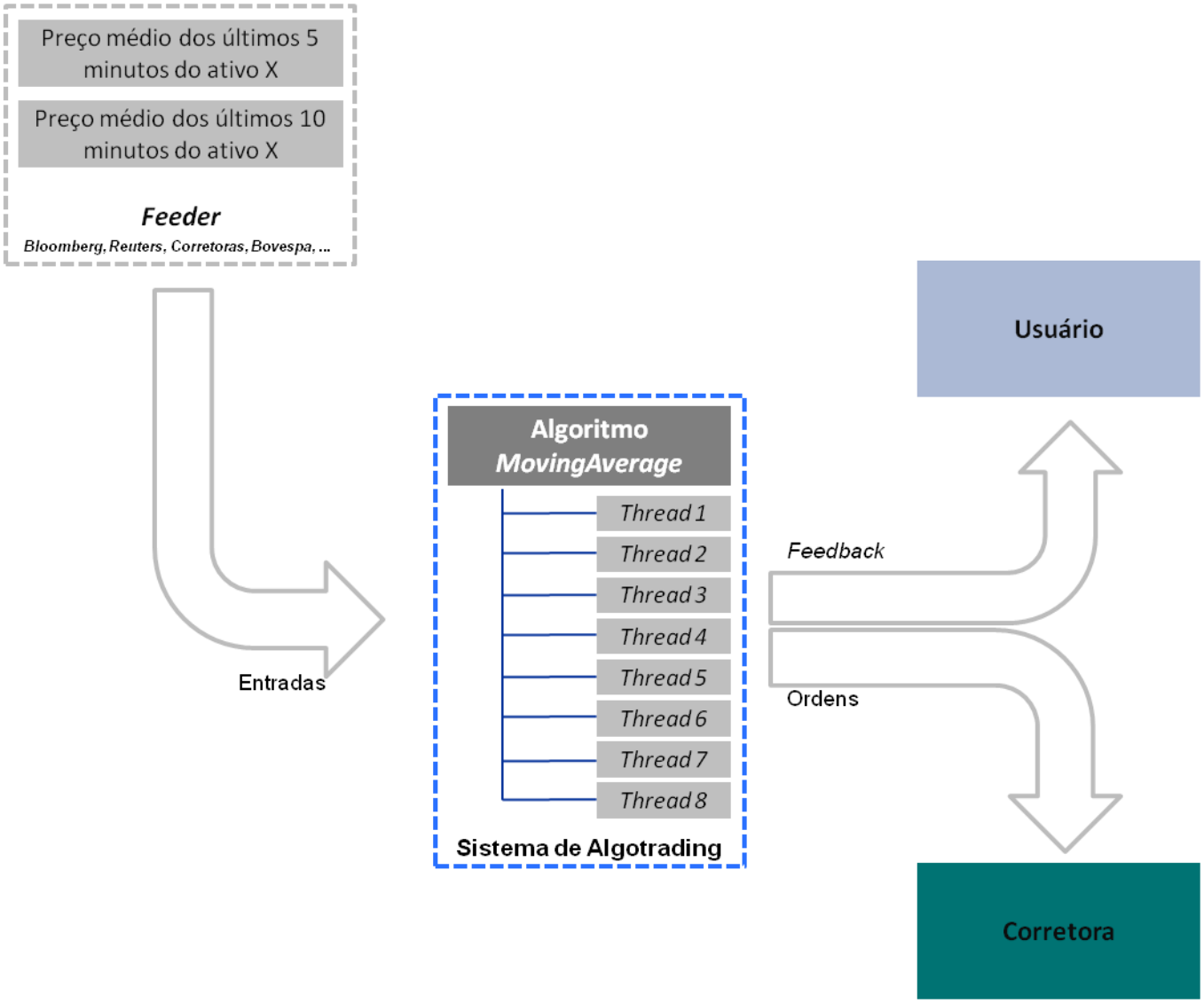


Figura 1-10. Processo de execução de algoritmos em sistema *8-core* comum.

A imagem que existe sobre Bolsas de Valores é a de *traders* fazendo grandes apostas em dinheiro em cima de sua intuição e ganhando sempre muito dinheiro.
Embora durante muitos anos as bolsas brasileiras tenham tido no trader o grande personagem de seus pregões, a tendência observada nos últimos tempos é a de que o volume de negociações observadas por eles está perdendo importância para o volume gerado pelas negociações automatizadas.
Primeiro serão abordados algoritmos de negociação automatizada e, em seguida, será abordado o protocolo FIX, de comunicação com bolsas de valores.

1.5.1 VWAP

Um dos algoritmos mais conhecidos nos mercados financeiros modernos é a negociação por *Volume Weighted Average Price* (VWAP). Uma definição mais informal do algoritmo é a de que VWAP é simplesmente a média ponderada pelo volume do preço do título e que o algoritmo VWAP é a tentativa de se negociar próximo ao VWAP.
Uma das grandes motivações no VWAP é a saída de grandes posições em certos títulos, normalmente feitos por grandes investidores institucionais, que não querem mexer no preço da ação enquanto se desfazem dessas posições.
Um serviço comum oferecido por corretoras é a compra direta dessas grandes posições detidas por esses grandes fundos para vendê-las depois a mercado. Para isso, as corretoras cobram uma taxa por esse serviço e uma das perguntas que ficam é quanto se deve cobrar por um serviço desses. O algoritmo VWAP de negociação está muito relacionado à resposta dessa pergunta.
VWAP também é utilizado como *benchmark* para investidores que desejam ser passivos em suas negociações, como os grandes fundos institucionais citados anteriormente.
Como o algoritmo VWAP tem com objetivo estar em linha com o mercado, argumenta-se que tais execuções diminuem os custos de transação dado que as ordens são executadas ao mínimo de *spreads*.

1.5.2 Moving Average

O *Moving Average* é um algoritmo muito simples, no entanto bastante genérico, utilizado para provar o conceito do Sistema. O algoritmo baseia-se na seguinte regra:
"Se a média móvel dos últimos T_1 minutos for menor que X% da média móvel dos últimos T_2 minutos do preço de determinado ativo, enviar uma ordem de compra".
Há também a regra oposta para venda de ativos mas dadas as restrições de venda a descoberto em alguns mercados e também o fato do Sistema ser apenas uma prova de conceito, apenas a versão de compra foi utilizada.
O intuito do algoritmo é observar tendências de alta rápidas do preço do ativo, tentando entrar em meio a uma tendência de alta.

1.6 Protocolo FIX

A principal bolsa de valores operando no Brasil, a BM&F Bovespa, utiliza o protocolo FIX para a comunicação de ordens e obtenção de dados.
O projeto do Financial Interface eXchange iniciou-se em 1992 por um grupo de instituições e corretoras interessadas em alinhar seus processos de negociação. Essas firmas perceberam que a indústria como um todo seria beneficiada das eficiências derivadas da comunicação eletrônica de indicações, ordens e execuções. O resultado foi o FIX, um padrão aberto de mensagens controlado por nenhuma entidade, que pode ser estruturado para se moldar aos requerimentos de negócios de cada empresa. Os benefícios são:

- de uma perspective do fluxo de negócios, o FIX provê à instituições e corretoras um modo de reduzir ligações telefônicas e papeladas desnecessárias;
- de uma perspectiva tecnológica, um padrão aberto alavanca o desenvolvimento da iniciativa de forma a criar ligações eficientes com um grande número de contra-partes;
- para fornecedores de tecnologia, o FIX provê a indústria com um padrão já pronto e aceito pelo mercado, de forma a facilitar o reconhecimento desses produtos pelos clientes.

O protocolo FIX não foi implementado no Sistema pelo fato do grupo não ter contrato com nenhuma corretora ou com nenhuma bolsa diretamente que possua a alternativa de utilização direta do FIX. Para obter informações do mercado e envio de ordens, o Sistema utilizou simulações do mercado.

2 DETALHAMENTO DA SOLUÇÃO

2.1 Arquitetura da Solução

Para atingir os objetivos do Sistema, foi desenvolvida uma arquitetura focada na interação com o usuário mais simples possível e com o maior aproveitamento possível da capacidade de processamento do CUDA. A arquitetura do Sistema está explicada a seguir.

2.2 Componentes

2.2.1 Tradutor

O Tradutor é o componente que irá transformar as planilhas do OpenOffice.org Calc em código C.
Atualmente sua implementação é feita em PyUNO, de forma que o mesmo código que integra com o OpenOffice.org Calc também faz a transformação da planilha em código C.
As planilhas usadas na implementação do Sistema tem um formato específico, explicado na Figura 2-1.

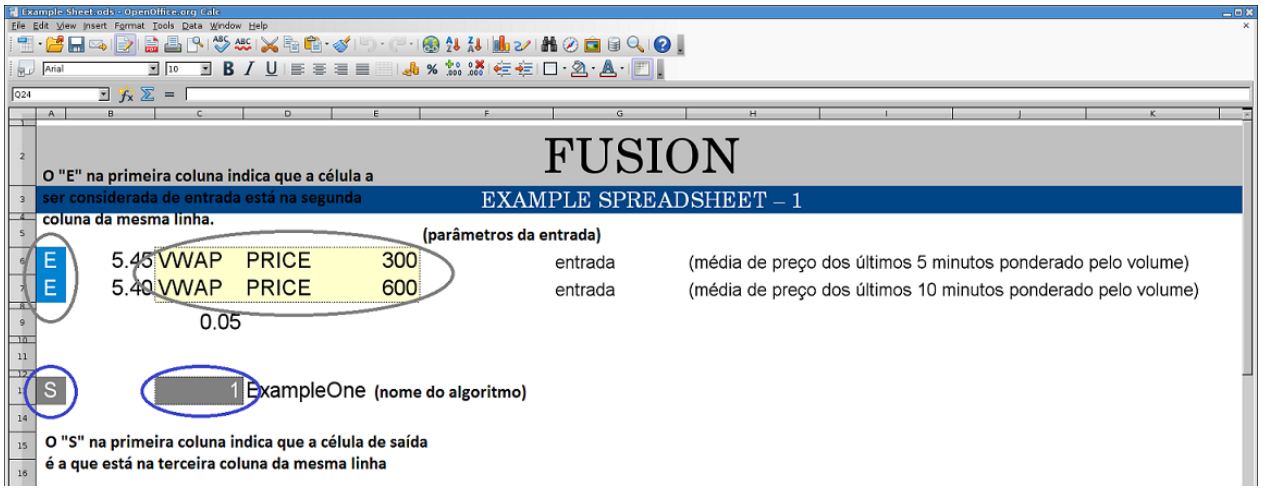
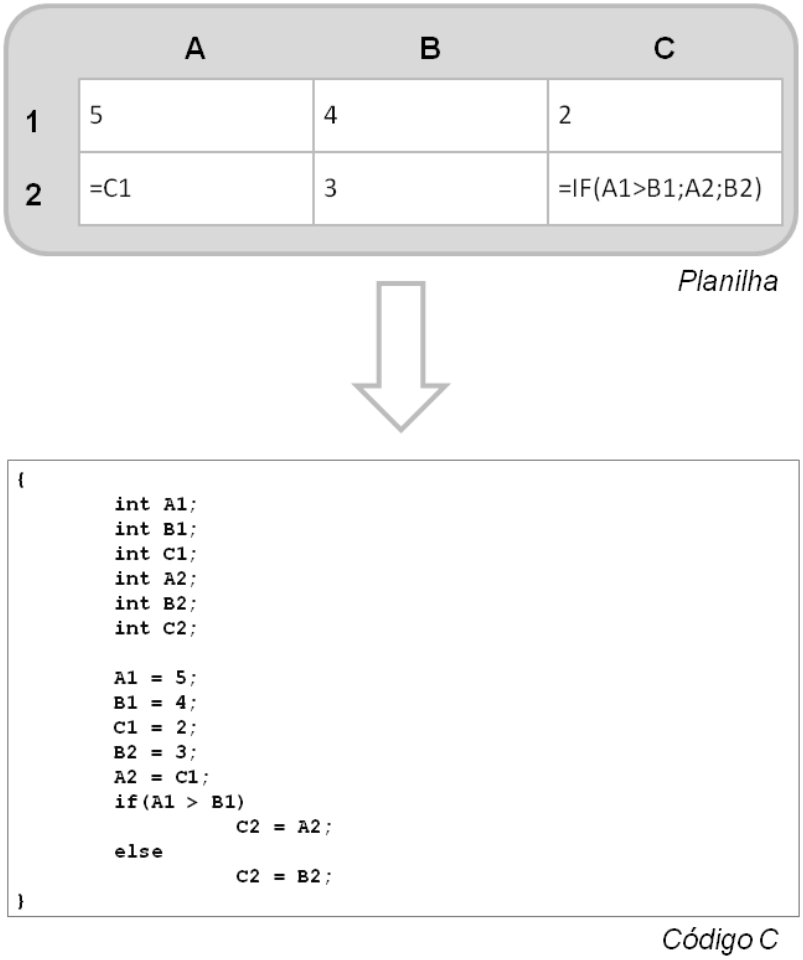


Figura 2-1. Formato esperado das planilhas eletrônicas.

Para a transformação do conteúdo das células em código C, foi usada uma técnica de programação muito parecida com a de compiladores: identificação de tokens e posterior tradução (“de para”) para código C, como exemplificado na Figura 2-2.



	A	B	C
1	5	4	2
2	=C1	3	=IF(A1>B1;A2;B2)

Planilha

```
{
        int A1;
        int B1;
        int C1;
        int A2;
        int B2;
        int C2;

        A1 = 5;
        B1 = 4;
        C1 = 2;
        B2 = 3;
        A2 = C1;
        if(A1 > B1)
                C2 = A2;
        else
                C2 = B2;
}
```

Código C

Figura 2-2. Exemplo de tradução de planilha eletrônica para código C.

Essa tradução é facilitada pelo fato de planilhas terem uma estrutura muito parecida com o código C, no sentido de que, resolvidos os problemas de dependência, a tradução é quase imediata.

Para cada célula da planilha a ser traduzida, são criadas variáveis para guardar o valor. Com relação a dependência entre células, ela é resolvida através da averiguação de quais células estão sendo utilizadas pela célula corrente que ainda não foram calculadas. Essa estratégia tem uma desvantagem: planilhas com referência circular não podem ser utilizadas. No entanto, isso não é visto como uma desvantagem verdadeira dado que tem-se como convenção que não é boa prática a produção de planilhas com referência circular, dada a dificuldade de ser fazer o debug das mesmas por outras pessoas, de modo similar a várias técnicas em programação que também não são consideradas boas práticas.

A Figura 2-3 demonstra como é o processo de traduçao de planilhas em código C.

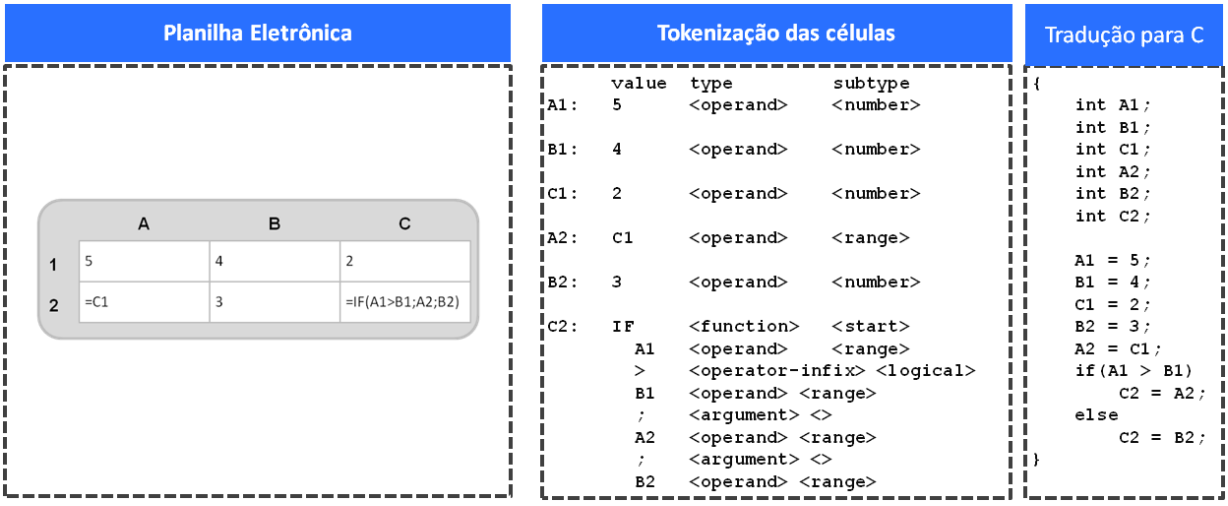


Figura 2-3. Processo de tradução de planilhas em código C para CUDA.

2.2.2 Núcleo – Executor

O Executor é o componente responsável pela execução do código gerado pelo Tradutor no CUDA.

A função de chaveamento de processos do kernel de um Sistema Operacional é muito semelhante ao que é feito pelo Executor.

No caso do SO, após a execução de um processo, o kernel é responsável por:

1. guardar os registradores usados no processo;
2. executar a preparação do próximo processo (como recuperar o valor dos registradores);
3. chamar a próxima instrução do processo.

No caso do Sistema, após a execução de um algoritmo, o Executor é responsável por:

1. guardar as saídas geradas;
2. preparar os dados para execução do algoritmo;
3. chamar a função do algoritmo;

Da mesma maneira, o módulo responsável pelo chaveamento de processos do kernel de um Sistema Operacional de propósito geral deve poder receber as seguintes ordens:

1. Adicionar processo à lista de execução;
2. Remover processo da lista de execução.

No Sistema, o Executor pode receber as seguintes ordens:

1. Adicionar algoritmo à lista de execução;
2. Remover algoritmo da lista de execução.

O processo de funcionamento do Executor está demonstrado na Figura 2-4, como já explicado nessa seção.

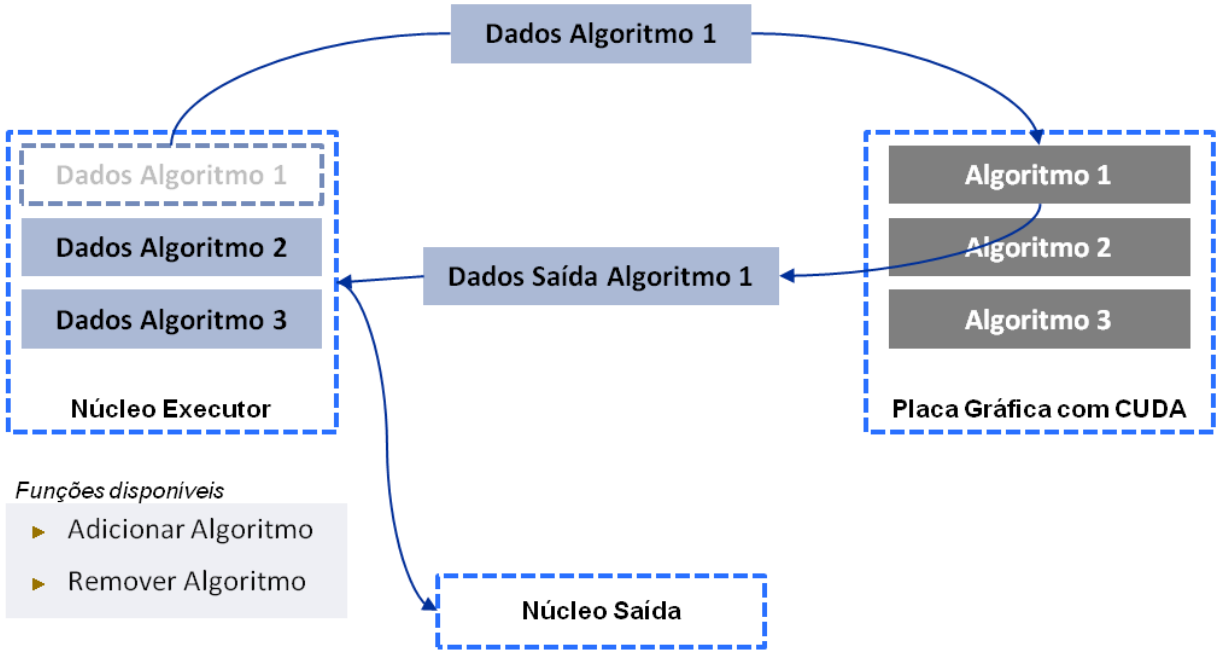


Figura 2-4. Arquitetura do Núcleo Executor.

2.2.3 Núcleo – Linha de Comando

A interface de Linha de Comando foi criada para a interação entre o usuário e o Núcleo (basicamente o Executor). Ela irá transmitir as ordens dada ao Núcleo pelo Usuário.

Os comandos criados são simples pois é uma interface temporária, que não está de acordo com a facilidade de uso sugerida na proposta.

A ideia é que ela se transforme em uma interface de configuração do Sistema, e não seja utilizada para adição e remoção de algoritmos do Executor.

A Figura 2-5 mostra como é feito o envio de ordens do Usuário ao Executor.

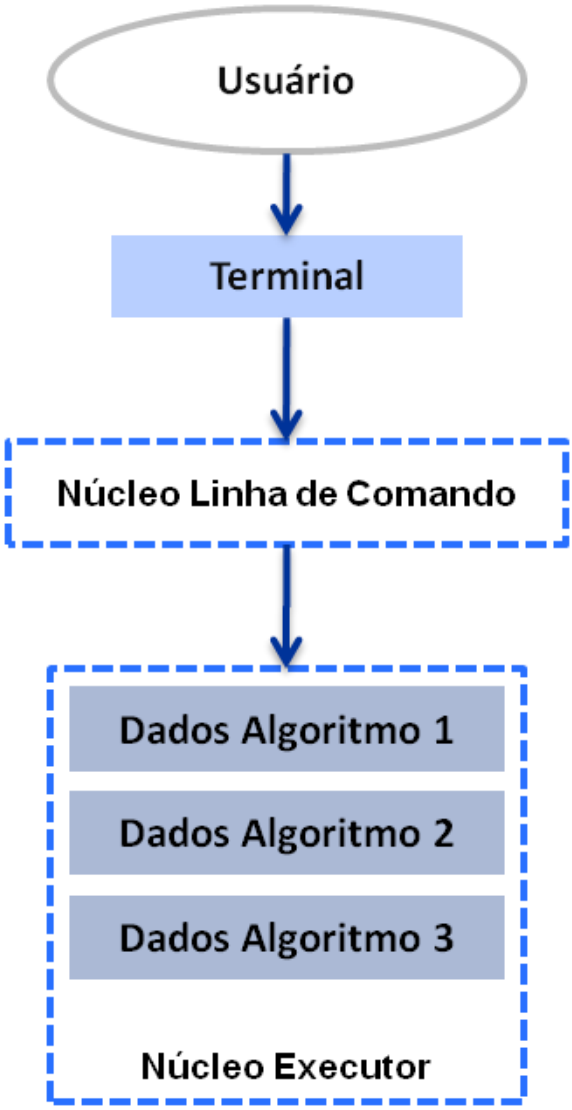


Figura 2-5. Esquema simples de utilização do Núcleo Linha de Comando.

2.2.4 Núcleo – Saída

Para o armazenamento das operações encontradas pelos algoritmos executados no Sistema, este conta com o móduo Saída.

O módulo Saída é basicamente a integração do Sistema com algum banco de dados. Para evitar que o Sistema fique dependente de um banco de dados específico, foi criado o módulo Saída para que a interface do Executor com o armazenamento da saída dos algoritmos seja padronizado e feito por outro módulo.

A Figura 2-6 demonstra como é a integração com banco de dados pelo módulo Saída.

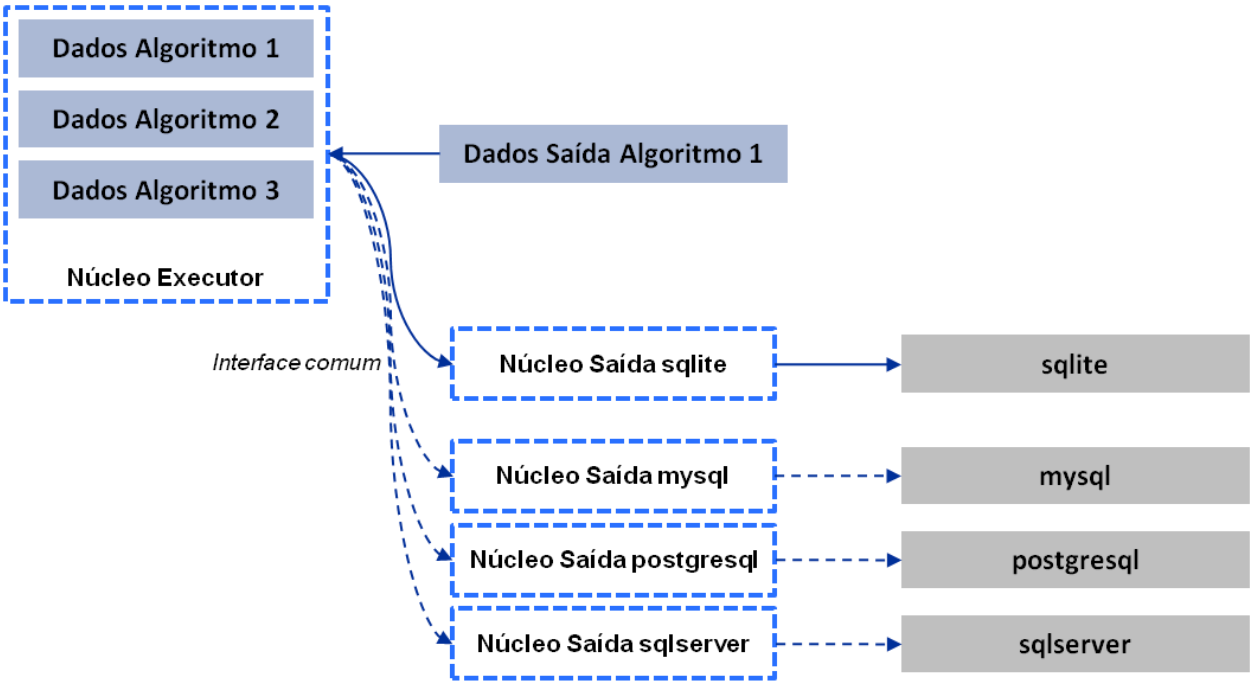


Figura 2-6. Arquitetura do Núcleo Saída.

Uma opção feita no Sistema é de que a ligação entre o Executor e o módulo Saída é feita em tempo de compilação, e não execução. Essa opção tem a desvantagem de impossibilitar a mudança de banco de dados sem a recompilação do Sistema, mas foi considerada a melhor pois é a forma de ligação que menos atrasos dá ao Executor. Fora isso, também foi considerado que um Sistema de alta performance como esse não deve ter seu banco de dados alterado com uma frequência que justifique a alteração dessa estratégia.

2.3 Processos

Além da solução tecnológica, o Sistema também conta com soluções nos processos do Cliente.

De maneira geral, o Sistema sugere que haja maior eficiencia entre o desejo do trader e a operação em si do Sistema.

2.3.1 Criação de algoritmo

Como mencionado anteriormente, a criação de algoritmos nas empresas que utilizam algotrading é ineficiente pelo fato de que o profissional que entende os mercados – mais próximo da mesa de operações – e o profissional que entende de Sistemas – da equipe de Tecnologia de Informação terem que se comunicar (e, daí, perder eficiência) na criação de algoritmos. O ideal seria o profissional da mesa ser o responsável pelo desenvolvimento dos algoritmos, evitando que falhas de comunicação entre o *trader* e o pessoal de TI sejam responsáveis por demoras no *deployment* de novos algoritmos.

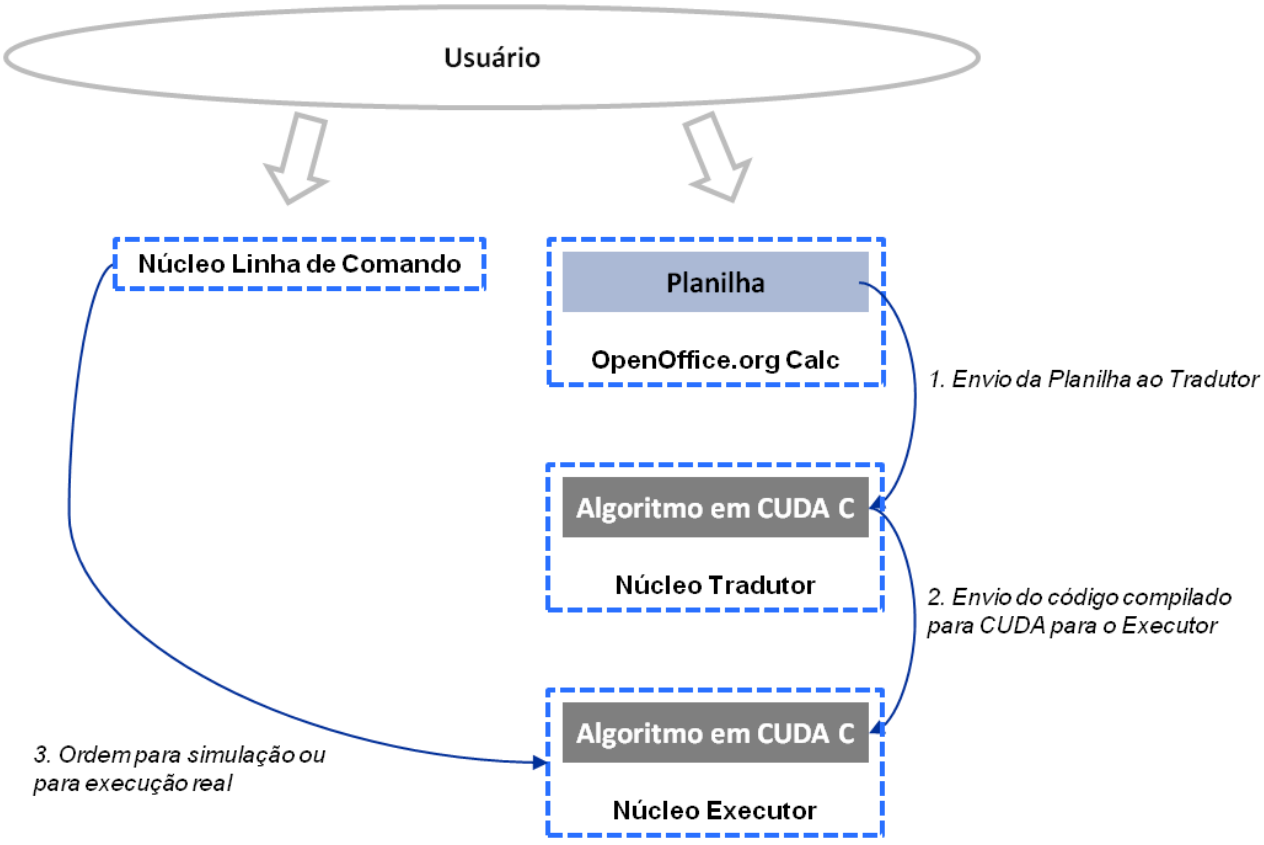


Figura 2-7. Fluxo de criação de algoritmos.

O trader, profissional que entende dos mercados, em geral, não tem uma formação de Tecnologia de Informação e, por isso, não sabe programar. Fora que a programação de

sistemas de algotrading é, em si, complicada, pelos requerimentos de alta disponibilidade e baixíssima taxa de falhas que devem ter.
No entanto, traders entendem muito bem de softwares de planilha eletrônica. Embora o software utilizado pelo mercado seja, na verdade, o Microsft Office Excel, os conceitos são os mesmos do OpenOffice.org Calc, inclusive as funções utilizadas.

Uma possibilidade aberta também pelo Sistema é a de deixar o algoritmo apenas para simulação, sem executar as ordens por ele calculadas. A simulação não tem o objetivo de medir o nível de sucesso nas negociações que o algoritmo tem, e sim se o algoritmo possui alguma falha básica em si e se o Sistema consegue executá-lo bem, dado que o Sistema é relativamente novo e será responsável pela gestão de grandes somas de dinheiro. Para validar a eficiencia do algoritmo deve-se utilizar uma ou ambas das seguintes opções:

a. *Backtesting* com dados de mercado históricos para comprovar eficiência em dados passados;
b. Usar matemática formal com teorias de finanças para comprovar eficiência;

O motivo pelo qual simulação não pode ser utilizada para tal fim é que, enquanto as duas opções dadas fornecem uma base muito mais sólida de testes, seja por quantidade de observações (*backtesting*) ou por demonstrações matemáticas, a simulação é capaz de fornecer apenas algumas observações durante um curto espaço de tempo.

2.3.2 Adição e Remoção de Algoritmo do pool *de Execução*

No Sistema proposto acima, temos a interface de linha de comando para que o usuário possa inserir e remover algoritmos *on-the-fly*, sem a necessidade de reiniciar o sistema.

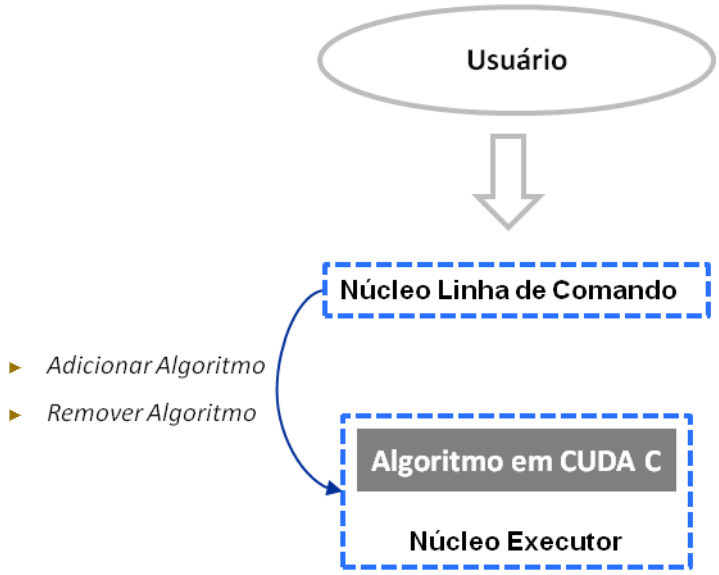


Figura 2-8. Fluxo de controle de algoritmos pelo Usuário.

Alguns sistemas de algotrading necessitam dessa reinicialização do sistema, o que pode acarretar em perdas – caso o sistema seja reinicializado durante o período de funcionamento do mercado – ou em atrasos no *deployment* de novos algoritmos – caso tenha que se esperar fechar o mercado para a reinicialização do sistema.

2.3.3 Execução de Algoritmo

A Figura 2-9 demonstra o fluxo completo de execução de um algoritmo. Observa-se a entrada de dados de um feeder (software que possui dados do mercado), a execução do algoritmo, o envio da ordem para a corretora e o log das informações.

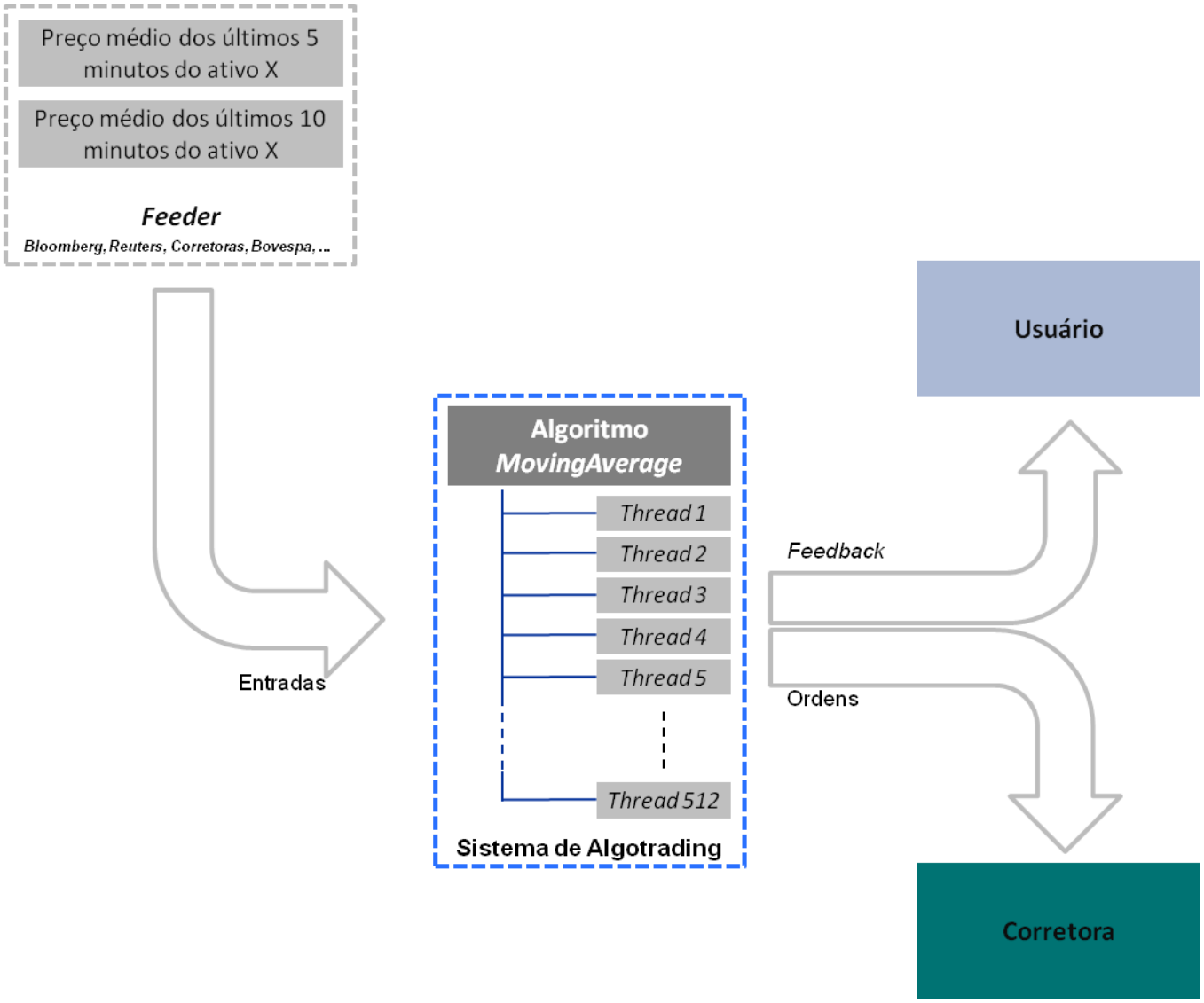


Figura 2-9. Processo de execução de algoritmos utilizando CUDA.

3 RESULTADOS

No mundo acadêmico, implementações de sistemas para negociação de ativos financeiros em tempo real são muito pouco exploradas. Mais comumente são encontradas referências a algoritmos, mas sem propostas sobre como executá-los.

No mercado são encontrados diversos produtos que se propõe a executar esses algoritmos encontrados pelo usuário. E há produtos voltados ao mercado financeiro que utilizam CUDA para execução. No entanto não há nada sendo oferecido no mercado que fornece ao usuário a possibilidade de usar um software de planilha eletrônica como entrada.

A implementação do Sistema teve o seguinte status em seus módulos:

a. Tradutor: traduz para código C células com aritmética (exemplo: ”`=C9`”) e apenas função IF (exemplo: ”`=IF(C9=C8;1;0)`“ dentre todas as disponíveis no Excel;
b. Executor: consegue adicionar e remover algoritmos de maneira simples, usando simulação de dados de mercado e comunicação com a bolsa (ou seja: sem integração real);
c. Linha de Comando: consegue enviar comandos simples ao Executor para adição e remoção de algoritmos em tempo real;
d. Saída: consegue logar em um banco de dados (sqlite) as saídas ativas (de compra ou venda) dos algoritmos.

A implementação mais incompleta foi a do Tradutor, pois para uma prova de conceito não há necessidade de uma implementação extensiva das funções do Excel.

Sobre o quesito abrangência, o Sistema será confrontado com os softwares disponíveis no mercado nos quesitos de número de ativos financeiros que podem ser utilizados, facilidade de uso e performance de execução.

Sobre custo, o custo do hardware do Sistema proposto será comparado ao custo de hardware de um sistema utilizando CPU de alta performance.

E, finalmente, sobre desempenho, a performance do Sistema em um algoritmo, o *Moving Average*, será confrontada com a performance do mesmo algoritmo implementado em CPU.

3.1 Abrangência

3.1.1 Progress Apama

O Progress Apama é atualmente o sistema de *algotrading* mais aceito pelo mercado. Poucas informações técnicas estão disponíveis sobre o software Apama da empresa Progress. No entanto, sabe-se que o software não utiliza plataforma especial de execução para seus algoritmos, deixando tudo a cargo de um hardware genérico de alto custo.

Com relação à interface com o usuário, o Apama utiliza uma linguagem própria não intuitiva para o profissional da mesa de operações, como demonstra a Figura 3-1

.

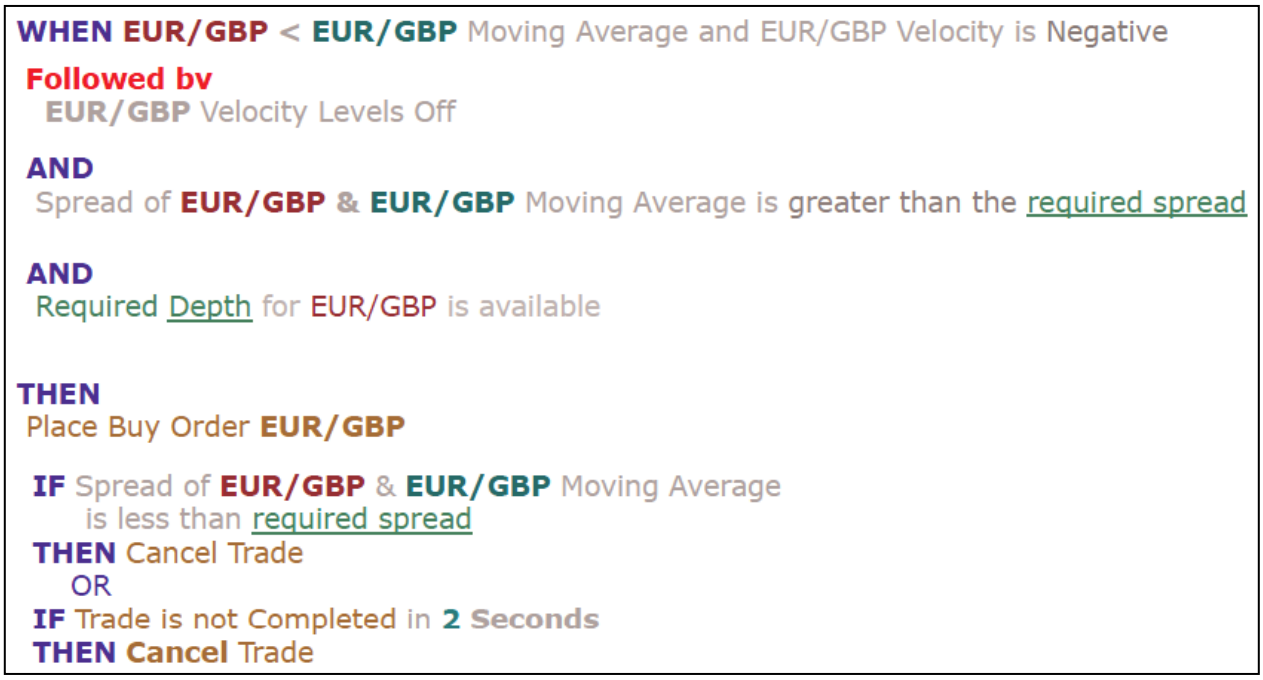


Figura 3-1. Exemplo de código na linguagem própria do Progress Apama. Fonte: Apresentação do Progress Apama a evento para clientes e usuários no Brasil. Outubro, 2009.

3.1.2 Hanweck Associates

Como exemplo de empresa que utiliza CUDA em softwares para o mercado financeiro, temos a Hanweck Associates.
A Hanweck Associates é uma firma especializada em investimentos e gestão de riscos que utiliza CUDA para análise do mercado americano.
A empresa fornece o seguinte dado sobre o CUDA: o que antes precisava de 60 servidores para analisar todo o mercado americano de ações, hoje ela utiliza 12 placas de vídeo com CUDA.

3.1.3 SciComp Inc.

A empresa SciComp fornece produtos para o mercado financeiro utilizando CUDA.
Dos seus produtos, o que mais se assemelha ao Sistema proposto é o SciFinance. Esse produto afirma transformar código em uma linguagem própria da SciComp não intuitiva ao profissional da mesa de operações em código C para execução no CUDA. Para indicar quais partes do código deve ser executado no CUDA, o usuário deve marcar o bloco com uma marca especial da linguagem própria.
Maiores informações técnicas não estão disponíveis publicamente.

3.2 Custo

Com relação ao custo, o Sistema proposto possui um relação custo-benefício muito superior ao de sistemas tradicionais.
Enquanto sistemas de alto desempenho atual, contando com dois processadores de última geração, 16GB ou mais de memória e um Sistema Operacional proprietário pode custar mais de R$ 20.000,00 dependendo do tamanho do pedido, o Sistema proposto, contando com um modelo um pouco mais defasado de processador, 6GB ou mais de memória e Linux como Sistema Operacional e com uma placa de vídeo de última geração não custa mais de R$ 6.000,00.
Isso acontece pelo fato de placas de vídeo com CUDA, para o propósito de algotrading, serem muito mais eficientes e econômicas que processadores de propósito geral.

3.3 Desempenho

Para execução de um algoritmo simples de média móvel, o CUDA permitiu uma redução média de tempo de aproximadamente 90,1% no cálculo para 5.000 ativos financeiros e, em um caso hipotético com 100.000 ativos, a redução alcançou 99,4%, como mostra a Figura 3-2.

A Listagem 3-1 possui o código utilizado para fazer as medições de performance do software. O código da função `genExampleOne` foi produzido diretamente pelo Núcleo - Tradutor e todo o resto (inclusive o `genExampleOneCPU`) foi desenvolvido pelo grupo.

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#define THREADS 5000
#define BLOCK_SIZE 512
#define PRICE_MAX 50.0
#define DIFF 0.005

__global__ void initializeCuda()
{
}

/* código criado pelo Núcleo Tradutor */
__global__ void genExampleOne(float *C13, float *B6, float *B7)

{
  float C9;
  int i = blockIdx.x * blockDim.x + threadIdx.x;

  C9 = B6[i]-B7[i];
  if(C9>B7[i]*0.005)
    C13[i] = 1;
  else
    C13[i] = 0;

}
```

```
void genExampleOneCPU(float *r, float *p5, float *p10)
{
  int i;

  for(i = 0; i < THREADS; i++)
  if(p5[i]-p10[i]>p10[i]*DIFF)
    r[i] = 1;
  else
    r[i] = 0;
}

float *prices5min, *prices10min, *cudaAnswers;
float *prices5min_d, *prices10min_d, *cudaAnswers_d;
float *correct, *cpuAnswers;

float normalized_rand()
{
  return (float)rand()/RAND_MAX;
}

void print_answers()
{
  int i;

  for(i = 0; i < THREADS; i++)
  {
    printf("%d:\t", i);
    printf("%.4g\t\t%.4g\t\t", prices5min[i], prices10min[i]);
    printf("%2.1g\t\t%2.1g%%\t\t", prices5min[i]-prices10min[i],
prices5min[i]/prices10min[i]-1);
    printf("%.0g\t\t", correct[i]);
    printf("%.0g%s\t\t", cudaAnswers[i], cudaAnswers[i] == correct[i]?" (OK)":"
(NOK)");
    printf("%.0g%s\n", cpuAnswers[i], cpuAnswers[i] == correct[i]?" (OK)":" (NOK)");
  }
}

int main()
{
  int up;
  int i;
```

```
struct timespec start_cpu, end_cpu, start_gpu, end_gpu;

/* inicializa random */
srand(time(0));

int nBytes = THREADS*sizeof(float);
int nBlocks = THREADS/BLOCK_SIZE + 1;

/* aloca memoria para CPU */
prices5min = (float *)malloc(nBytes);
prices10min = (float *)malloc(nBytes);
correct = (float *)malloc(nBytes);
cpuAnswers = (float *)malloc(nBytes);
cudaAnswers = (float *)malloc(nBytes);

/* aloca memoria para GPU */
cudaMalloc((void **)&prices5min_d, nBytes);
cudaMalloc((void **)&prices10min_d, nBytes);
cudaMalloc((void **)&cudaAnswers_d, nBytes);

/* inicializa valores */
for(i = 0; i < THREADS; i++)
{
  prices10min[i] = normalized_rand()*PRICE_MAX; // preco de 10min
  up = normalized_rand() > 0.5?1:-1;       // define movimento (up ou down)

  if(normalized_rand() > 0.5)                     // dentro do range
    prices5min[i] = prices10min[i] * (1 + 0.5 * DIFF * up);
  else                                           // fora do range
    prices5min[i] = prices10min[i] * (1 + 1.5 * DIFF * up);

  if((prices5min[i] - prices10min[i]) > prices10min[i]*(DIFF))
    correct[i] = 1;
  else
    correct[i] = 0;
}

/* copia memoria para GPU */
cudaMemcpy(prices5min_d, prices5min, nBytes, cudaMemcpyHostToDevice);
cudaMemcpy(prices10min_d, prices10min, nBytes, cudaMemcpyHostToDevice);

/* execucao na CPU */
```

```
  clock_gettime(CLOCK_REALTIME, &start_cpu);
  genExampleOneCPU(cpuAnswers, prices5min, prices10min);
  clock_gettime(CLOCK_REALTIME, &end_cpu);

  /* execucao na GPU */
  initializeCuda<<<nBlocks, BLOCK_SIZE>>>();

  clock_gettime(CLOCK_REALTIME, &start_gpu);
  genExampleOne<<<nBlocks, BLOCK_SIZE>>>(cudaAnswers_d, prices5min_d, prices10min_d);
  clock_gettime(CLOCK_REALTIME, &end_gpu);


  /* copia memoria para CPU */
  cudaMemcpy(cudaAnswers, cudaAnswers_d, nBytes, cudaMemcpyDeviceToHost);

  /* calculo do tempo */
  double time_cpu = (double)( end_cpu.tv_nsec - start_cpu.tv_nsec ) / (double)1000000;
  double time_gpu = (double)( end_gpu.tv_nsec - start_gpu.tv_nsec ) / (double)1000000;

  /* exibe resultados */
  print_answers();

  /* exibe tempos */
  printf("threads: %d\n", THREADS);
  printf("Tempo CPU (em ms): %f\n", time_cpu);
  printf("Tempo GPU (em ms): %f\n", time_gpu);

  return 0;
}
```

Listagem 3-1. Código usado para benchmarking de performance entre o algoritmo rodando em uma placa NVIDIA com CUDA habilitado e o mesmo algoritmo sendo executado em um processador Intel i7.

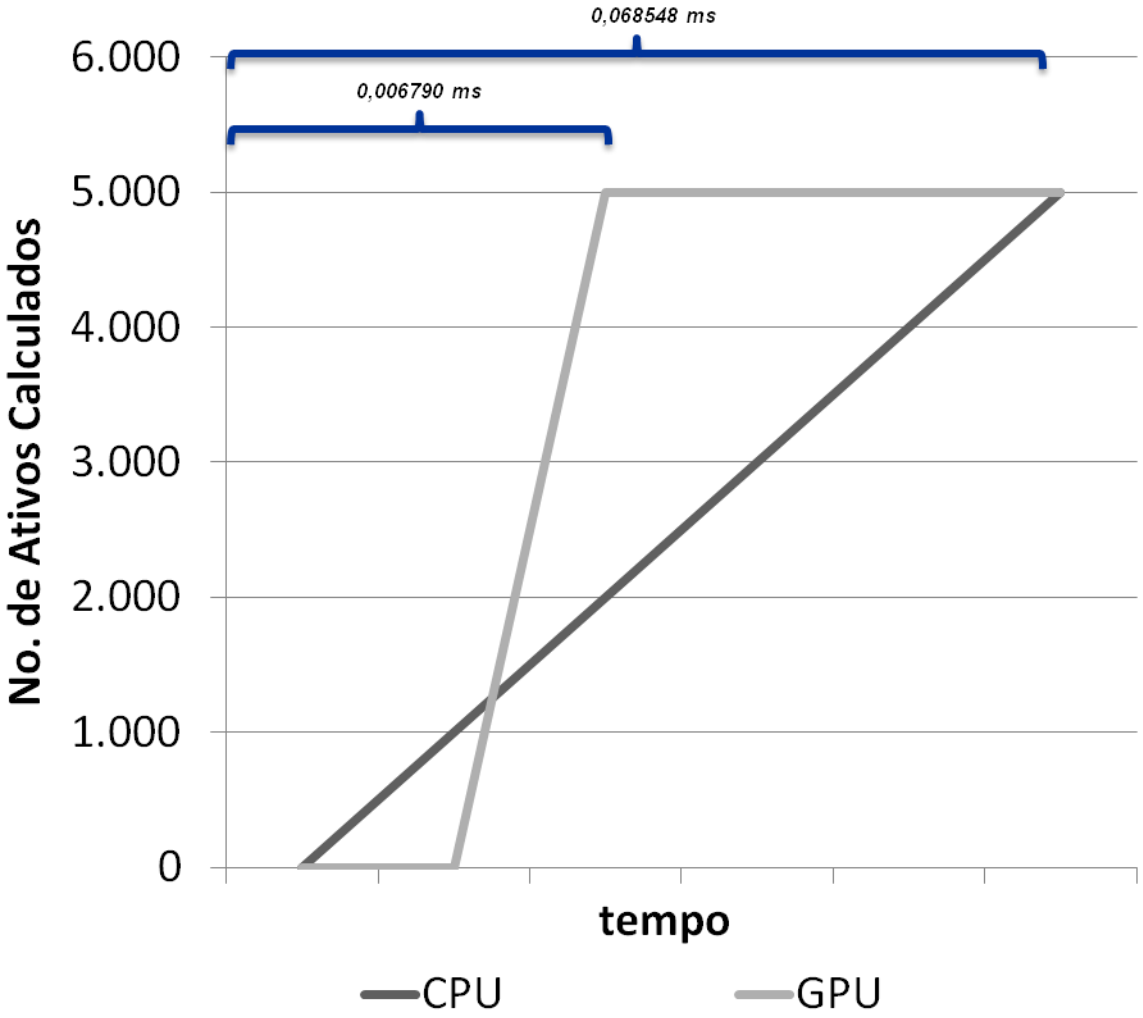


Figura 3-2. Gráfico ilustrativo do modo de funcionamento e performance qualitativa de execuções na GPU e no CPU.

# ativos	Tempo de Execução		
	CPU (ms)	GPU (ms)	Redução
100	0,001521	0,006931	-355,7%
200	0,002696	0,006962	-158,2%
500	0,006644	0,006822	-2,7%
1.000	0,013124	0,007470	43,0%
2.000	0,026078	0,006772	74,0%
5.000	0.068548	0.006790	90,1%
10.000	0,136326	0,007177	94,7%
20.000	0,275279	0,007369	97,3%
50.000	0,689085	0,007795	98,8%
100.000	1.612720	0.009667	99,4%

Figura 3-3. Tempo de execução no CPU e na GPU para diversos número de ativos.

Item	Descrição
CPU	Intel i7 860 @ 2.80Ghz
Memória Principal	6 Gb de RAM
Placa de Vídeo	NVIDIA GeForce GTS 250
Memória da Placa de Vídeo	1 Gb de RAM

Figura 3-4. Descrição do equipamento utilizado no benchmarking.

4 CONCLUSÕES

O Sistema de algotrading proposto nesse trabalho foi sabatinado com diversos profissionais do mercado financeiro e professores e doutorandos do PCS da Poli-USP. A ideia do Sistema teve críticas muito boas e ajudou o grupo na melhor concepção do Sistema.

O grupo considera que a implementação pode ser considerada de sucesso. Nela, os dois principais objetivos foram atingidos: desempenho superior de algoritmos e facilidade de uso.

Com relação ao desempenho, como demonstrado no item 3.3 anterior, o grupo conseguiu alcançar uma redução de 90% com relação à execução tradicional no CPU para um algoritmo relativamente genérico, reforçando a excelente tática de utilizar GPUs na execução de algoritmos de negociação automática de ativos. Esses algoritmos se encaixam perfeitamente na descrição dos melhores tipos de algoritmos que GPUs conseguem executar: são compostos basicamente de operações aritméticas e não necessitam de controles de fluxo muito sofisticados. O maior exemplo do fato de não necessitarem de controles de fluxo sofisticados é o fato de poucos utilizarem o laço `for` em sua implementação.

Mostrou-se que a tradução de planilhas do Excel para código C é possível mesmo em uma implementação relativamente limitada. Embora somente a função IF() do Excel esteja completamente implementada em C – além das operações aritméticas das células – o conceito foi provado através de exemplos de algoritmos reais transformados em C, como o MovingAverage do item 1.5.2.

A motivação maior por trás da tradução das planilhas em Excel para código C diretamente era a necessidade de melhorias no processo de criação de algoritmos, pois esses são atualmente feitos por profissionais que entendem dos mercados, os *traders*, e profissionais que entendem de sistemas, do departamento de TI, e esse grupo de trabalho normalmente não consegue uma comunicação plena, acarretando em atrasos e implementações não ótimas dos algoritmos inicialmente propostos. Com a tradução de planilhas de Excel para código C diretamente sendo possível, as perspectivas para otimização do processo de criação de algoritmos aumentaram pois agora é possível

que *traders* façam seus próprios algoritmos diretamente, sem precisar da intermediação do pessoal do departamento de TI. De maneira análoga, os profissionais do departamento de TI podem se concentrar na sua especialidade – sistemas de computação – e tratar da infraestrutura necessária para que o Sistema proposto nesse trabalho possa rodar.

Em resumo, o grupo considerou o trabalho bem-sucedido em seus objetivos e considera que ele pode dar origem a mais trabalhos acadêmicos e a produtos para se disponibilizar no mercado.

5 REFERÊNCIAS

[1] HENDERSHOTT, T.; JONES, C.; MENKVELD, A. **Does Algorithmic Trading Improve Liquidity?** 2008. Disponível em:
http://papers.ssrn.com/sol3/papers.cfm?abstract_id=1100635

[2] **Understading UNO**. Disponível em:
http://wiki.services.openoffice.org/wiki/Uno/Article/Understanding_Uno

[3] **Professional UNO**. Disponível em:
http://api.openoffice.org/docs/DevelopersGuide/ProfUNO/ProfUNO.xhtml

[4] **PyUNO Samples**. Disponível em:
http://wiki.services.openoffice.org/wiki/PyUNO_samples

[5] KAKADE, S. M.; KEARNS, M.; MANSOUR, Y.; ORTIZ, L., E. **Competitive Algorithms for VWAP and Limit Order Trading**. 2004. Disponível em:
http://www.cis.upenn.edu/~mkearns/papers/vwap.pdf