

UNIVERSIDADE FEDERAL FLUMINENSE
POLO UNIVERSITÁRIO DE RIO DAS OSTRAS
FACULDADE FEDERAL DE RIO DAS OSTRAS
BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

Luiz Guilherme Oliveira dos Santos

Um Estudo de Mapeamento de Sistemas Multiagentes baseados
em FIPA para Arquiteturas GPGPU

Rio das Ostras-RJ

2010

LUIZ GUILHERME OLIVEIRA DOS SANTOS

UM ESTUDO DE MAPEAMENTO DE SISTEMAS MULTIAGENTES BASEADOS EM FIPA PARA
ARQUITETURAS GPGPU

Trabalho de Conclusão de Curso
apresentado ao Curso de Graduação
em Bacharelado em Ciência da Computação da
Universidade Federal Fluminense, como
requisito parcial para obtenção do Grau de
Bacharel.

Orientador: Prof^a. Dr^a. FLÁVIA CRISTINA BERNARDINI

Rio das Ostras-RJ

2010

LUIZ GUILHERME OLIVEIRA DOS SANTOS

UM ESTUDO DE MAPEAMENTO DE SISTEMAS MULTIAGENTES BASEADOS EM FIPA PARA
ARQUITETURAS GPGPU

Trabalho de Conclusão de Curso
apresentado ao Curso de Graduação
em Bacharelado em Ciência da Computação da
Universidade Federal Fluminense, como
requisito parcial para obtenção do Grau de
Bacharel.

Aprovada em Julho de 2010.

BANCA EXAMINADORA

Prof^a. Dr^a. FLÁVIA CRISTINA BERNARDINI - Orientadora
UFF

Prof. Dr. CARLOS BAZÍLIO MARTINS
UFF

Prof. Dr. ESTEBAN WALTER GONZALEZ CLUA
UFF

Rio das Ostras-RJ
2010

“O estado de espírito de uma pessoa se revela em seus sonhos. Devemos nos esforçar para fazer dos sonhos nossos aliados.”

(Yamamoto Tsunetomo - Hagakure, O Código Samurai)

“Os homens devem moldar seu caminho, a partir do momento em que você vir o caminho em tudo que fizer, você se tornará o caminho.”

(Miyamoto Musashi - O Livro dos Cinco Anéis)

Este trabalho é dedicado aos meus pais, Miguel Borges dos Santos e Lanúzia Oliveira da Câmara Santos, que mesmo nos momentos de dificuldade sempre acreditaram que a educação era a prioridade na vida dos seus filhos.

Agradecimentos

Gostaria de agradecer a todos os professores lotados e bolsistas que lecionaram no Pólo Universitário de Rio das Ostras, por acreditarem no projeto do pólo e emprestar (sem direito a devolução) seu conhecimento em prol da formação dos estudantes aqui matriculados.

À minha orientadora Flávia Cristina Bernardini, por me ter aceito como aluno orientando e me ajudar na execução desse trabalho.

Ao pessoal do MediaLab de Niterói, em especial o Professor Esteban Clua, e aos alunos Erick Passos, Marcelo Zamith e João Gazola por criarem mecanismos de comunicação e troca de informações que ajudaram na execução desse trabalho.

Listas de Figuras

1.1	À Direita em (a), um modelo do leão mecânico de Da Vinci. À Esquerda em (b) Exibição de uma réplica do Leão no Château du Clos Lucé.[Buchanan, 2002]	1
1.2	À Direita (a) O carro autônomo construído por alunos da Universidade de Standford cruza o deserto de Las Vegas em 7 horas. À Esquerda (b) Robô Mars Rover da NASA. Ambos são exemplos de autonomia em agentes [Buchanan, 2005].	4
1.3	À Direita (a) uma imagem do jogo International Soccer(1982). À Esquerda (b) uma imagem do jogo Pro Evolution Soccer(2009). Cada vez mais os personagens gráficos se tornam mais realistas, contudo com o realismo é necessário aprimorações em sua inteligência.	5
2.1	Estrutura Básica de um Agente	8
2.2	Exemplo de uma Máquina de Estados para um Agente Inteligente.	8
2.3	Taxonomia dos Agentes	10
2.4	Taxonomia do Comportamento dos Agentes em um Sistema Multiagente.	11
2.5	Taxonomia dos Sistemas Multiagentes	13
2.6	Estrutura de uma Mensagem em FIPA-ACL	15
2.7	Exemplo de Comunicação utilizando FIPA-ACL	15
2.8	Ontologia de Controle dos Agentes	16
2.9	Arquitetura do JADE	18
2.10	Relacionamento entre os elementos da Arquitetura do JADE	18
3.1	Arquitetura de GPU Utilizando Vertex Shaders e Pixel Shaders	22
3.2	Diferença de desempenho entre CPU e GPU em GFlops	22
3.3	Diferença de Arquitetura entre CPU e GPU	23
3.4	Exemplo de Arquitetura capaz de suportar CUDA	23
3.5	Um programa multithread é partitionado em blocos de threads que executam independentemente. Assim quanto maior o número de núcleos da GPU, menor será o tempo de execução.	24
3.6	Sintaxe para definição e execução do Kernel	25
3.7	Exemplo de uso do <code>threadIdx</code>	25
3.8	Grid de Threads	26
3.9	Exemplo de uso do <code>dimBlock</code> e <code>blockIdx</code>	27

3.10 Exemplo de uso do <code>__syncthreads()</code>	27
3.11 Hierarquia de Memória das Threads em CUDA	28
3.12 Exemplo da Programação Heterogênea existente em CUDA	29
4.1 Problema dos Caixotes	31
4.2 Speed-Up em relação aos cenários.	34
4.3 Speed-Up em relação ao número de agentes.	34
4.4 Speed-Up em relação ao número de caixotes.	34
4.5 Representação do Sistema Bidimensional com o Algoritmo A*	36
4.6 Speed-Up em relação aos cenários.	39

Lista de Tabelas

2.1	Descrição de Relações Sociais entre os Agentes	11
4.1	Especificação dos Casos de Teste	33
4.2	Tabela comparativa entre o Método de Dijkstra e o Método A*	37
4.3	Especificação dos Casos de Teste	39

Lista de Algoritmos

1	Algoritmo Para Mover os Caixotes.	32
2	Algoritmo A*.	37
3	Movimentação dos Agentes no Mapa.	38

Glossário

IA: Inteligência Artificial

MAS: *Multiagent System*

AOP: *Agent-Oriented Programming*

FIPA: *Foundation for Intelligent Physical Agents*

JADE: *Java Agent DEvelopment Framework*

RAM: *Random Access Memory*

CPU: *Central Processing Unit*

FLOP: *FLoating point Operations Per Second*

GPU: *Graphics Processing Unit*

GPGPU: *General Purpose Programming Using a Graphics Processing Unit*

CUDA: *Compute Unified Device Architecture*

Speed-up: Métrica de desempenho entre diferentes implementações.

Sumário

Agradecimentos	vi
Lista de Figuras	vii
Lista de Tabelas	ix
Lista de Algoritmos	x
Glossário	xi
Resumo	xiv
Abstract	xv
1 Introdução	1
2 Agentes Inteligentes	7
2.1 Sistemas Multiagentes	7
2.2 Programação Orientada a Agentes (AOP)	12
2.2.1 Definição	12
2.2.2 Especificações FIPA	14
2.2.3 Framework JADE	17
3 Arquitetura de GPU's	20
3.1 GPU Computing	21
3.2 CUDA (<i>Computer Unified Device Architecture</i>)	23
3.2.1 Hierarquia de Threads	25
3.2.2 Sincronização	26
3.2.3 Hierarquia de Memória	27
4 Estudo de Casos	30
4.1 Motivação e Trabalhos Relacionados	30
4.2 Cenários Unidimensionais: O Problema dos Caixotes	30
4.2.1 Mapeando do JADE para o CUDA	31

4.2.2	Análise de Desempenho	33
4.3	Cenários Bidimensionais: Pathfinding A*	35
4.3.1	O Algoritmo A*	35
4.3.2	Mapeando de JADE para CUDA	38
4.3.3	Análise de Desempenho	39
5	Conclusões e Trabalhos Futuros	40
	Referências Bibliográficas	41
A	Especificações do FIPA	48
B	Diagrama de Classes do Caso Unidimensional	50
C	Assinatura das funções do Caso Unidimensional	52
D	Tempos do Caso Unidimensional	53
E	Diagrama de Classes do Caso Bidimensional	64
F	Assinatura das funções do Caso Bidimensional	66
G	Tempos do Caso Bidimensional	69

Resumo

Atualmente muitos problemas podem ser resolvidos utilizando agentes inteligentes. Com um cenário específico, esses agentes interagem com o cenário e com os outros agentes simulando comportamentos previamente programados. Quando há vários agentes em um cenário interagindo entre si, temos um Sistema Multiagente. Um dos modelos mais usados para a implementação de Sistemas Multiagentes hoje em dia é o FIPA, e um dos *frameworks* mais utilizados é o JADE. Porém há limitações ao simular multidões com um grande número de agentes devido ao atual poder de processamento das CPU's. Com a popularização das chamadas GPGPU's é possível ter um novo nível de escalabilidade e desempenho de algoritmos quando os mesmos são executados direto na GPU. Uma tecnologia que está difundida hoje em dia é o CUDA. Este trabalho visa investigar ambas implementações, mostrar semelhanças, diferenças, dificuldades, vantagens e desvantagens de mapear JADE para CUDA.

Palavras-chave: Inteligência Artificial, Sistemas Multiagentes, GPGPU, CUDA

Abstract

Recently, many problems can be solved using intelligent agents. With an especific scenario, these agents interacts among each other and with the scenario, simulating predefined roles. When we have many agents interecting like that we call it a Multiagent system. One of the most commonly pattern used by developers is FIPA, and JADE framework. It is hard to simulate large crowds with many agents because of the limited processing power we have though. In the recent year, we notice a popularization of what we call GPGPU, and now it is possivel to have a new level of escalability and performance when algorithms run directly on GPU. The most widespread tecnology is CUDA. These works wants to investigate both implementations, showing similarities, differences, difficulties, advantages and disadvantages of mapping JADE to CUDA.

Keywords: Artificial Intelligence, Multiagent Systems, GPGPU, CUDA

Capítulo 1

Introdução

O conceito de “criar inteligência” é algo bem mais antigo do que a era dos computadores. Na mitologia grega há um mito em que Hefesto, o ferreiro divino, filho de Zeus com Hera, criador de vários serventes mecânicos [McCorduck, 2004]. Em especial Talos, um “homem” de bronze, cujo dever é patrulhar as praias de Crete. A partir desse e de outros mitos, e da criatividade humana, muitos aparelhos mecânicos foram criados a partir do estudo das características humanas e de animais. Um desses grandes exemplos é o Leão Mecânico criado por Leonardo Da Vinci em 1515 D.C. [Buchanan, 2002] - Figura 1.1.

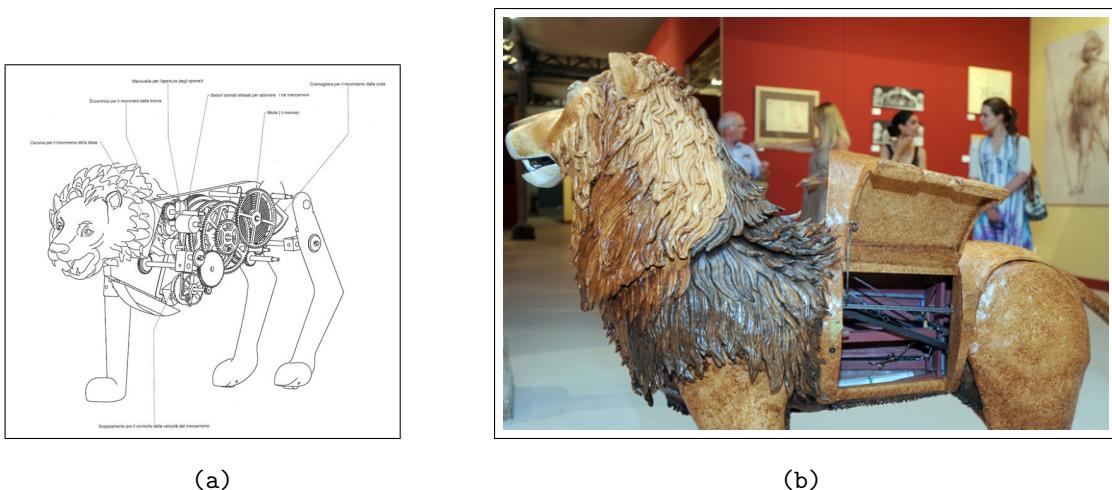


Figura 1.1: À Direita em (a), um modelo do leão mecânico de Da Vinci. À Esquerda em (b) Exibição de uma réplica do Leão no Château du Clos Lucé. [Buchanan, 2002]

Mesmo com todo o desenvolvimento de maquinário nos séculos subsequentes devido à revolução industrial, ainda faltava propor como criar o intelecto, como entender a forma humana de pensar. René Descartes, no século XVII apresentou uma primeira discussão sobre mente e matéria, e dos problemas que podem ser visualizados com essa distinção. Ele propôs que os corpos de animais não eram nada mais que máquinas complexas. Ao longo desse século muitos filósofos trabalharam em variações desse modelo.

No meio do século XIX, matemáticos como Augustus DeMorgan e George Boole trabalharam na criação de uma lógica simbólica, para a substituição da lógica de síslogística de Aristóteles - 5 A.C., criando

as primeiras notações de atomicidade, classes e operadores binários, que depois ficou conhecido como álgebra booleana. Em 1879, Gottlob Frege apresenta a lógica como um ramo de investigação sistemática mais fundamental do que a matemática ou a álgebra [Frege, 1979]. Mais tarde, esse trabalho teve seu conceito refinado e expandido por Bertrand Russell, Alfred Tarski, Kurt Gödel, Alonzo Church entre outros. No início do século XX, [Russell, 1906] e [Whitehead and Russell, 1910] revolucionaram a lógica formal, propondo uma completa mudança na axiomatização da lógica proposicional, a partir de disjunções negação de primitivas. Com isso, foi possível unir filosofia e análise de conhecimento.

Com a iminência da Segunda Guerra Mundial, muitos esforços em prol do crescimento tecnológico foram feitos. Alan Turing, em 1936-1937 desenvolveu o modelo da Máquina Universal, que muitos consideram até hoje como a origem do computador programável [Davis, 2000]. Com sua supervisão foram construídos dez dos primeiros computadores, os Colossus I, com o objetivo de decodificar códigos nazistas. Baseado nisso John Von Neumann criou o primeiro modelo que usava uma CPU. Esse modelo, além de implementar a máquina de Turing, continha uma arquitetura sequencial. O primeiro computador criado a partir desse modelo foi o ENIAC, que pesava quase 30 toneladas, ocupava um espaço de 200 metros quadrados, tinha 10 mil capacitores e conseguia fazer 5 mil adições por segundo. Não se via tal evolução desde o ábaco e as réguas de cálculo.

Começam então os primeiros trabalhos da recente área que veio a ser conhecida como Inteligência Artificial. Warren McCulloch e Walter Pitts em 1943 propuseram um modelo de neurônios artificiais baseados em conhecimentos de fisiologia básica, da função dos neurônios, da análise da lógica proposicional criada por Russell e Whitehead, e da teoria da computação criada por Turing [McCulloch and Pitts, 1943]. Em 1949, Donald Hebb demonstra uma regra de atualização simples para atualizar neurônios que é utilizada até os dias de hoje, e é chamada de **aprendizagem de Hebb** [Hebb, 1949].

Alan Turing [Turing, 1950] propõe uma simples questão sobre o desenvolvimento da época - “As máquinas podem pensar?”. Nesse trabalho ele propôs um jogo em que temos 3 pessoas, o Jogador A (Homem), o jogador B (Mulher) e o jogador C (Interrogador, de qualquer sexo). O objetivo do jogador A é enganar o interrogador, nesse caso, convencer o jogador C de que ele é uma mulher. O objetivo do jogador B é ajudar o interrogador, convencendo-o de que ele é uma mulher. Por fim o objetivo de C é descobrir quem é homem ou mulher. Para que tons de voz ou aparência não interfiram no jogo, todos ficam em uma sala separada e se comunicam através de mensagens escritas por uma máquina. Turing propôs que se fosse possível substituir A por uma máquina e assim convencer C, então A seria uma máquina inteligente. Esse jogo ficou conhecido como “Teste de Turing”.

O interesse sobre esse novo campo de conhecimento começou a crescer, apesar dele ainda não estar bem fundamentado. Tanto a comunidade científica quanto escritores como Isaac Asimov começaram a tecer teorias sobre como seria a vida do ser humano assistida por um robô pensante, e suas implicações para a sociedade em que vivemos [Asimov, 1950].

No verão de 1956, o recém graduado John McCarthy organizou o primeiro seminário de Inteligência Artificial, dando nome ao termo, no Dartmouth College. O objetivo do seminário era reunir pessoas interessadas em estudar teorias de autômatos, redes neurais e estudo da inteligência. Nele foi demonstrado o primeiro programa de IA, o *Logic Theorist* ou LT criado por Allen Newell e Heber Si-

mon [McCorduck, 2004]. Tal programa demonstrava boa parte dos teoremas do segundo capítulo do livro [Whitehead and Russell, 1910], de forma mais simples. Apesar de não trazer significativamente nada de novo para a comunidade, esse seminário reuniu os principais nomes dessa área nos anos subsequentes.

Nos primórdios a IA era repleta de atenção e sucesso, porém tais conquistas eram erroneamente exaltadas e eram alcançadas de forma limitada e incompleta. Nessa época o ferramental era precário. Linguagens ainda estavam em desenvolvimento e os computadores eram vistos como objetos de luxo que efetuavam operações aritméticas de forma automatizada. Alguns preferiam simplesmente dizer que “a máquina nunca será capaz de realizar X ”¹, e os cientistas tinha como meta responder um X de cada vez.

Nesse entusiasmo inicial foram criados programas inovadores como o *General Problem Solver* (Solucionador geral de problemas), ou GPS [Newell et al., 1959], que imitava protocolos humanos para resolução de problemas, o Geometry Theorem Prover [Gelertner, 1959], que podia demonstrar teoremas que eram bastante complicados, programas para jogos de damas [Samuel, 1959] entre outros. É necessário citar também que nessa mesma época foi criado o LISP, uma linguagem de alto nível que se tornou dominante na IA por diversos anos. Com ela foram criados programas como o SAINT [Slagle, 1963], que resolvia problemas de cálculo integral, e o ANALOGY [Evans, 1968], que resolvia problemas clássicos de analogia geométrica em testes de QI.

As primeiras dificuldades surgiram ao perceber que a maioria dos programas não continha um conhecimento adquirido sobre o assunto. Eram baseados simplesmente em manipulações e táticas simples. Imagine por exemplo um programa de tradução de ideogramas em japonês para português. Cada símbolo é interpretado de uma maneira diferente a cada contexto. A máquina não entendia essa interpretação de contexto e gerava traduções equivocadas. Até hoje a tradução é uma ferramenta em desenvolvimento, mas já teve um grande progresso ao longo dos anos.

Outro grande problema era que a maior parte dos programas de IA na época resolviam problemas experimentando diferentes combinações de passos até encontrar a solução. Essa estratégia funciona bem para problemas pequenos, mas era relativamente difícil de se alcançar quando o escopo era aumentado. Alguns pesquisadores culpavam o hardware que ainda não estava totalmente “desenvolvido” para a realização de tais experimentos. A visão desses pesquisadores começou a ficar negativa e problemas e teoremas maiores não puderam ser provados.

Os investimentos na área da IA foram sendo gradualmente reduzidos e a área passava por uma crise intelectual que só terminaria nos anos 80. Novas áreas foram criadas para suprir tal fracasso como o estudo de Sistemas Especialistas. Esses sistemas tinham por objetivo primário ajudar o profissional na realização de sua tarefa, e não realizá-la por si só. Um exemplo disso foi o MYCIN [Shortliffe and Buchanan, 1984], que ajudava médicos na identificação de infecções sanguíneas.

O crescimento da demanda por Sistemas Especialistas, levou a uma demanda por melhores ferramentas de representação de conhecimento. Foram criadas ferramentas baseadas em lógica como o PROLOG e o PLANNER, e outras soluções mais estruturadas como a idéia de frames de Minsky [Minsky, 1975].

¹Turing citou muitos X para essa frase, dentre eles: Ser amável, diligente, bonito, amigável, ter iniciativa, senso de humor, distinguir o certo do errado, cometer enganos, apaixonar-se, gostar de morangos e creme, fazer alguém se apaixonar por ela, aprender a partir da experiência, usar palavras corretamente, ser o sujeito de seu próprio pensamento, ter diversidade de comportamento quanto ao homem, fazer algo realmente novo.

Nos anos 80, a IA volta a ganhar força a partir de sistemas bem sucedidos comercialmente como o R1 da empresa DEC (*Digital Equipment Corporation*) [Polit, 1984], programa que ajudava a configurar pedidos de novos sistemas de computadores. O montante economizado por ano com o uso desse software chegava aos 40 milhões de dólares. Nessa mesma época, em Standford foi realizada a primeira conferência nacional da AAAI (*American Association of Artificial Intelligence*, Associação americana de Inteligência Artificial em inglês). A IA ressurge como ciência e leva à evolução de técnicas desprezadas como as Redes Neurais, a partir de novos métodos como à Retropropagação [Werbos, 1994], e cria novos campos de trabalho como Redes Bayesianas, Mineração de Dados, e o surgimento da Teoria de Agentes.

Em 1987, foi ratificada uma arquitetura completa para o agente [Laird et al., 1987]. Assim foi possível aprofundar os estudos no funcionamento interno dos agentes, incorporando ambientes reais e entradas sensoriais contínuas. Com a popularização da internet, os agentes passaram a ser base para muitas ferramentas existentes, como sistemas de auxílio à construção de websites, mecanismos de pesquisa, filtros de spam, tradutores online, entre outros.

Até a década de 90, a IA focou em problemas voltado para a inteligência individual. Porém, algo que faz os seres humanos diferente de máquinas, além de sermos capazes de pensar, é nosso relacionamento em sociedade. O fato de criarmos linguagens simbólicas e formas de comunicação que vão além das palavras, possibilitam um poder de cooperação, coordenação, negociação e vivência uns com os outros. Cada vez o uso de agentes inteligentes nos ajudam a ter processos autônomos, como o uso de carros inteligentes, exploração marinha em grandes profundidades, exploração do espaço, entre outros - Figura 1.2.



(a)



(b)

Figura 1.2: À Direita (a) O carro autônomo construído por alunos da Universidade de Standford cruza o deserto de Las Vegas em 7 horas. À Esquerda (b) Robô Mars Rover da NASA. Ambos são exemplos de autonomia em agentes [Buchanan, 2005].

Um outro grande uso dos agentes é relacionado ao entretenimento digital. Com o uso de personagens cada vez mais realistas, é possível ter outro nível de imersão em jogos, e aplicativos de realidade virtual e aumentada. Outro uso importante é em robótica, sendo possível dar assistência a médicos durante cirurgias. Mesmo com os *frameworks* existentes, alguns problemas não podem ser implementados em tempo real devido ao atual poder de processamento existente.

Atualmente o poder computacional proporcionado pelas placas gráficas chama tamanha atenção



Figura 1.3: À Direita (a) uma imagem do jogo International Soccer(1982). À Esquerda (b) uma imagem do jogo Pro Evolution Soccer(2009). Cada vez mais os personagens gráficos se tornam mais realistas, contudo com o realismo é necessário aprimorações em sua inteligência.

que um dos desafios da IA é fazer com que os agentes possam ser executados diretamente em GPU. Com essa possibilidade, é possível criar uma quantidade maior de agentes e em alguns casos estudar o desempenho de diversos algoritmos distribuídos diretamente em GPU. A grande vantagem da simulações de muitos agentes (*Crowd Simulations*) seriam simulações de emergência, multidões em tempo real, maior escalabilidade em algoritmos distribuídos entre agentes, entre outros. Por exemplo, uma empresa de aviação pode fazer simulações reais de como funciona saídas de emergência de aviões em diversas situações, e não precisaria fazer isso fisicamente, o que gera custos muito altos.

Um grande problema é a não existência de um *framework* que possibilite tal criação. Um programador deveria abstrair toda teoria de agentes e criar estruturas para criação de sistemas multiagentes em GPU. Um dos objetivos desse trabalho é explorar a escalabilidade de sistemas multiagentes, principalmente quando esses possuem grandes populações para mapear problemas reais como trâfego de trânsito, simulações de multidões entre outros. A partir de tal mapeamento, podemos determinar quais algoritmos podem ser feitos em placas gráficas, como isso deve ser feito, o como contornar as restrições impostas pela arquitetura da GPU.

Outro objetivo é analisar a diferença entre um *framework* já existente e o paradigma GPU Computing para um programador de agentes. Para tal foi utilizado uma padronização existente, o FIPA, e um *framework* existente que segue tal padronização, o JADE. Serão mostradas similaridades e diferenças entre ambas a implementações com o intuito de mostrar a forma como é aplicada a teoria de agentes em ambas a implementações.

Por fim, esse trabalho propõe dois estudo de casos para ilustrar esse mapeamento de uma implementação em paradigma orientado a agentes para paradigma GPU Computing de Sistemas Multiagentes. Em cada um desses estudos é falado sobre os objetivos dos estudos, o mapeamento propriamente dito, as vantagens e desvantagens de cada implementação, e ao final é feito uma análise de desempenho.

Esse documento é organizado da seguinte maneira:

No Capítulo 2 são descritos os sistemas multiagentes e suas principais características.

No Capítulo 3 são introduzidos conceitos relativos às arquiteturas GPGPU, e à linguagem CUDA.

No Capítulo 4 são mostrados estudos de caso realizados, e como esses estudos de caso nos levaram a analisar o de sistemas multiagentes para o paradigma GPGPU.

No Capítulo 5 são feitas as conclusões desse trabalho, bem como futuros trabalhos que podem ser desenvolvidos à partir desse estudo.

Capítulo 2

Agentes Inteligentes

2.1 Sistemas Multiagentes

Uma linha de estudo da IA é o estudo do ser humano como indivíduo, e não apenas como massa céfálica. Esse indivíduo pensa, raciocina, interage com o meio e toma decisões que afetam a si próprio, ao meio onde vive, e a outras pessoas nesse meio.

Dessa forma, inteligência é não somente o ato de raciocinar, decidir, aprender, planejar. Um ser dotado de inteligência é aquele que consegue integrar esses 4 aspectos básicos e adaptá-los para a situação em que se encontra. Essa união geral é o que chamamos de comportamento individual do ser humano.

Em Ciência da Computação, chamamos essa forma de criar e executar um comportamento individual de Agente Inteligente¹. Para [Wolldridge, 2002], “um agente é um sistema computacional situado em algum ambiente, sendo capaz de realizar, de forma independente (autonomante), ações nesse ambiente, descobrindo o que necessita para satisfazer seus objetivos ao invés de ter que receber essa informação”. Para [Franklin and Graesser, 1996], “um agente autônomo é um sistema situado dentro e de parte de um ambiente, ambiente este que o agente percebe e nele age ao longo do tempo, perseguindo um objetivo dentro de sua própria agenda.”. Mas em geral a mais usada e mais completa é a de [Russell and Norvig, 1995]:

“Um agente é uma entidade real ou virtual, capaz de agir num ambiente, de se comunicar com outros agentes, que é movida por um conjunto de inclinações (sejam objetivos individuais a atingir ou uma função de satisfação a otimizar); que possui recursos próprios; que é capaz de perceber seu ambiente (de modo limitado); que dispõe (eventualmente) de uma representação parcial deste ambiente; que possui competência e oferece serviços; que pode eventualmente se reproduzir e cujo comportamento tende a atingir seus objetivos utilizando as competências e os recursos que dispõe e levando em conta os resultados de suas funções de percepção e comunicação, bem como suas representações internas.”

Um exemplo simplificado de um modelo de agente descrito por [Russell and Norvig, 1995] pode ser visto na Figura 2.1. Analogamente ao ser humano o agente pode ser munido de câmeras, detectores

¹Alguns livros e fontes se referem ao Agente Inteligente como Artefato Inteligente

infra-vermelhos e outros instrumentos de entrada de dados (direta ou indireta) agindo como sentidos (olfato, paladar, visão, audição, tato).

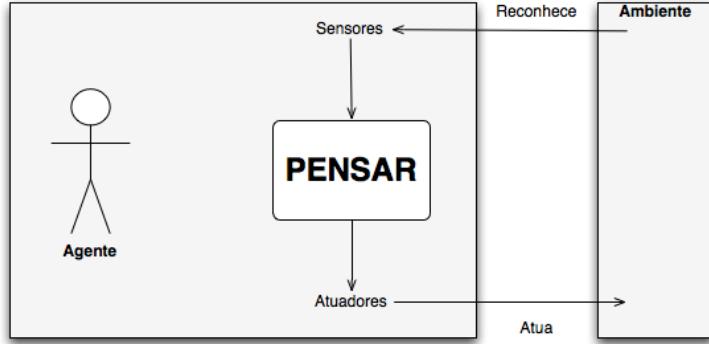


Figura 2.1: Estrutura Básica de um Agente

Com essas informações, o agente deve ser capaz de **perceber** o ambiente ao qual está presente. Em geral, a escolha da decisão tem influência na ordem dessas percepções. Com isso, é possível mapear o agente como uma função matemática $f(x_1, x_2, \dots, x_n)$, onde x_1, x_2, \dots, x_n é a **sequência de percepção** do agente.

O agente pode conter diversos estados. Por exemplo, um mundo onde o agente corre se o ambiente sentir perigo, anda se sentir fora de perigo e para caso esteja no lugar desejado, temos a seguinte máquina de estados na Figura 2.2. Ela demonstra claramente um aspecto importante do conceito de agentes inteligentes, a **autonomia de decisão**. Ou seja, o próprio agente é capaz de escolher em qual estado estará apenas com as informações obtidas pelos seus sensores, e possui a **competência para decidir** a mudança de estados. Todas essas mudanças e informações devem satisfazer às metas do agente, que deverá estar licitado em sua **agenda própria**.

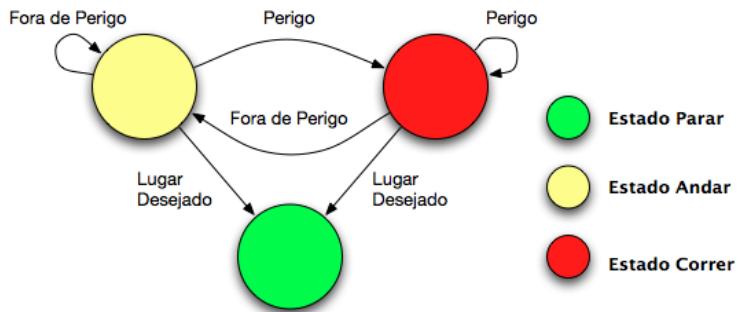


Figura 2.2: Exemplo de uma Máquina de Estados para um Agente Inteligente.

Agentes podem ter uma arquitetura reativa ou cognitiva (algumas vezes chamada de deliberativa). O agente reativo é baseado no modelo estímulo-resposta, e em arquiteturas de subsunção (*Subsumption Architecture*) conforme proposto por [Brooks, 1986], para resolver problemas na área de robótica. Esse tipo de agente é usado especialmente quando há um grande número de agentes e a eficiência do sistema é algo crucial.

Em contrapartida, os agentes cognitivos são baseados em planejamento e tomada de decisão. Em IA, o planejamento é um programa que recebe como entrada uma descrição de conjuntos possíveis de decisão e suas respectivas implicações. Cabe ao algoritmo do programa definir, de acordo com as metas estabelecidas, um plano detalhando quais decisões tomar para chegar ao estado-objetivo. Esse processo envolve os seguintes passos:

1. Representar o estado atual do ambiente
2. Representar o estado desejado do ambiente
3. Determinar um conjunto de ações atômicas possíveis de serem executadas
4. Com essas ações, determinar a melhor forma de sair do estado atual, chegar ao estado desejado (ou num caso de uma aproximação, algo que satisfaça as condições), utilizando o conjunto de ações proposto e um resolvidor de problemas (algoritmo)

Existem muitos modelos de planejamentos criados: Lineares como o STRIPS [Fikes and Nilsson, 1971], não lineares como o NONLIN [Tate, 1977], abstratos como o ABSTRIPS [Sacerdoti, 1974] e hierárquicos como o NOAH [Sacerdoti, 1977].

No planejamento clássico temos que o ambiente é estático, ou seja, será o mesmo até o final da execução, determinístico e facilmente observável. Porém, esse tipo de problema representa poucos casos encontrados. O planejamento não-clássico admite que o ambiente é estocástico, ou seja, ele é dinâmico e parcialmente observável.

Por fim, o agente pode ser um agente estrutural, ou seja, ter características humanas, por exemplo as NPCs² em um jogo, ou agentes comportamentais, como por exemplo agentes responsáveis pelo filtro de spam de um email. O ambiente do agente também pode ser restrito ao computador utilizado, ou pode agir em uma rede interna (Intranet) ou externa (Internet). A Figura 2.3 representa toda a taxonomia aqui falada, e as divide em eixos, segundo a classificação de [Garcia and Sichman, 2003].

Sistemas computacionais que implementam agentes de software podem ser monoagentes ou multiagentes. Um sistema monoagente descreve apenas como um agente resolve um problema de modo **isolado**. Nesse contexto, não há nenhum tipo de **cooperação, competição e negociação** entre agentes. Ao final da década de 70, muitos cientistas tentaram usar a cooperação entre agentes para resolver problemas [Erman et al., 1980], [Lenat, 1975]. Em 1980 foi criado um protocolo de negociação entre agentes [Smith, 1980] baseado na noção do mercado econômico. Alguns ambientes foram desenvolvidos anos mais tarde, por exemplo [Cardozo, 1987], [Corkill and Lesser, 1993] e [Gasser et al., 1987]. Apenas em 95, foram criados abordagens para resolver problemas de modo cooperativo [Boissier, 1993], [Sichman, 1995], [Durfee and Rosenschein, 1994]. Apesar de existir problemas em que um agente autônomo consegue resolver problemas sozinho, o ser humano tem poucas dessas situações normalmente. A internet, redes sociais, dentre outros recursos para conectar pessoas mostrou-se algo poderoso em situações mais complexas [Malone and Crowston, 1994]. Em uma sociedade moderna, cada pessoa tem um conhecimento e perícia limitada.

²Non Player Character, são personagens em um jogo que não possuem entrada de dados pelo jogador.

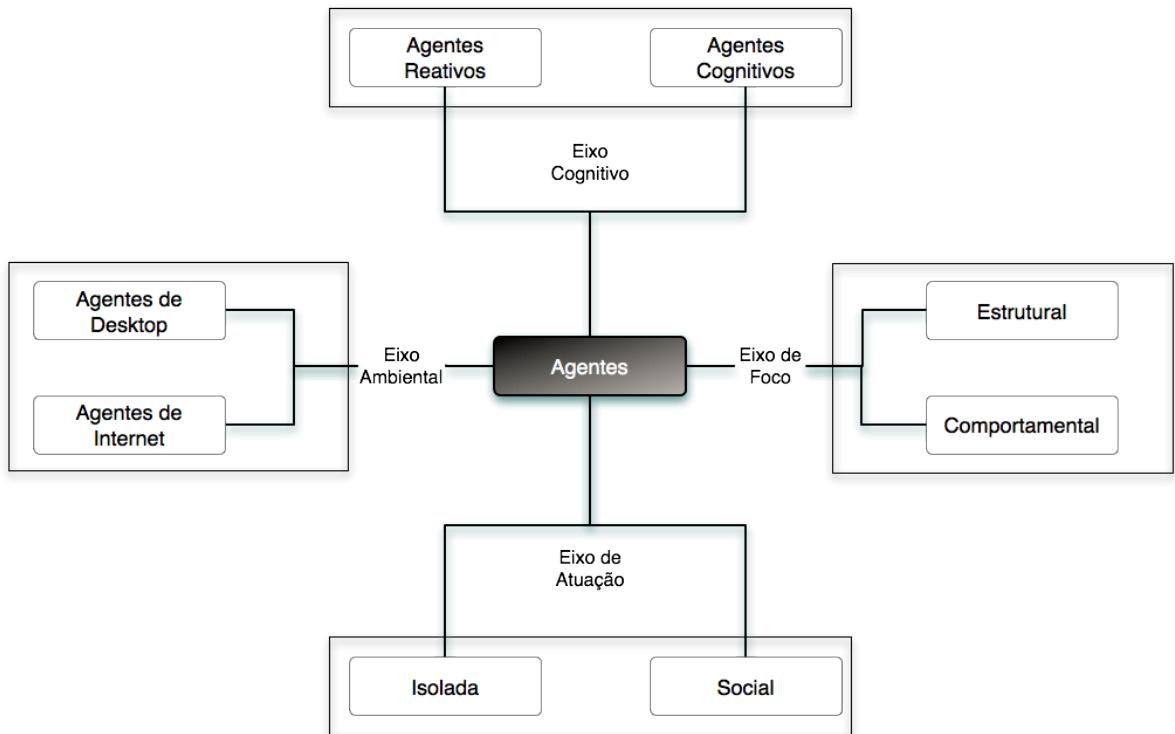


Figura 2.3: Taxonomia dos Agentes

Com a descentralização do problema, várias questões foram levantadas para o desenvolvimento de uma inteligência distribuída entre os agentes, tais como: infraestrutura, interações entre os agentes, planejamento distribuído, protocolos de comunicação e protocolos de negociação.

Da mesma forma que não há um consenso sobre o que é um agente, há muitas definições para um sistema multiagente. Nesse trabalho iremos usar a definição de [Wolldridge, 2002]:

“Um Sistema Multiagente é um sistema que consiste de um número de agentes que interagem uns com os outros. Além disto, para interagir de forma eficaz, os agentes deste sistema devem ser capazes de cooperar, se coordenar e negociar entre si.”

Uma grande questão é o porquê de usar soluções descentralizadas ou distribuídas. Soluções centralizadas em geral mais confiáveis e em alguns casos até mais eficiente. Porém, a computação distribuída pode levar a possibilidade de resolver problemas antes impossíveis por limitação de recursos ou desempenho. Ainda, alguns problemas ficam fáceis de modelar de forma distribuída [Passos et al., 2008]. Por outro lado, não há como prever o resultado da simulação, e conflitos precisam ser tratados durante sua execução.

Suponha que dentro de um determinado ambiente existam dois agentes, *a* e *b*. Como vimos anteriormente, cada agente possui uma agenda própria e objetivos dentro dessa agenda a serem alcançados. Em um sistema distribuído, esses agentes dependem de relações sociais para realizar esses objetivos. A Tabela 2.1 lista algumas dessas relações.

Interferência Social	É o efeito da ação individual do agente <i>a</i> no agente <i>b</i> . Essa ação pode ser direta ou indireta, e a interferência pode ser positiva (visando ajudar ambos a cumprir seu objetivo) ou negativa (depreciando um dos agentes ou ambas as partes).
Interação Social	É a combinação da interferência do agente <i>a</i> no agente <i>b</i> com a interferência do agente <i>b</i> no agente <i>a</i> .
Dependência	Para que o agente <i>a</i> atinja determinado objetivo, inatingível se estivesse sozinho, pois o agente <i>b</i> pode interferir positivamente , por meio de uma determinada ação, dizemos que o agente <i>a</i> não é autônomo para atingir um determinado objetivo, sendo assim dependente do agente <i>b</i> .
Delegação	Acontece quando o agente <i>a</i> determina que o agente <i>b</i> deva realizar uma dada tarefa. Para tal, o agente <i>a</i> deverá utilizar um mecanismo de planejamento social e exercer uma influência sobre o agente <i>b</i> e convencê-lo a realizar a ação. Essa influência pode ser dada também através de um Escambo Social entre os agentes.
Compromisso	Acontece quando um agente <i>a</i> firma um compromisso com o agente <i>b</i> de realizar uma determinada tarefa.

Tabela 2.1: Descrição de Relações Sociais entre os Agentes.

Um dos desafios do sistema multiagente é sua coordenação. O nível de coordenação usado no sistema determina sua complexidade, tipos de comunicação entre os agentes e forma de implementação do sistema. Na Figura 2.4 é mostrado uma taxonomia de coordenação segundo [Wolldridge, 2002]. Nesse modelo os agentes podem ser cooperativos ou competitivos.

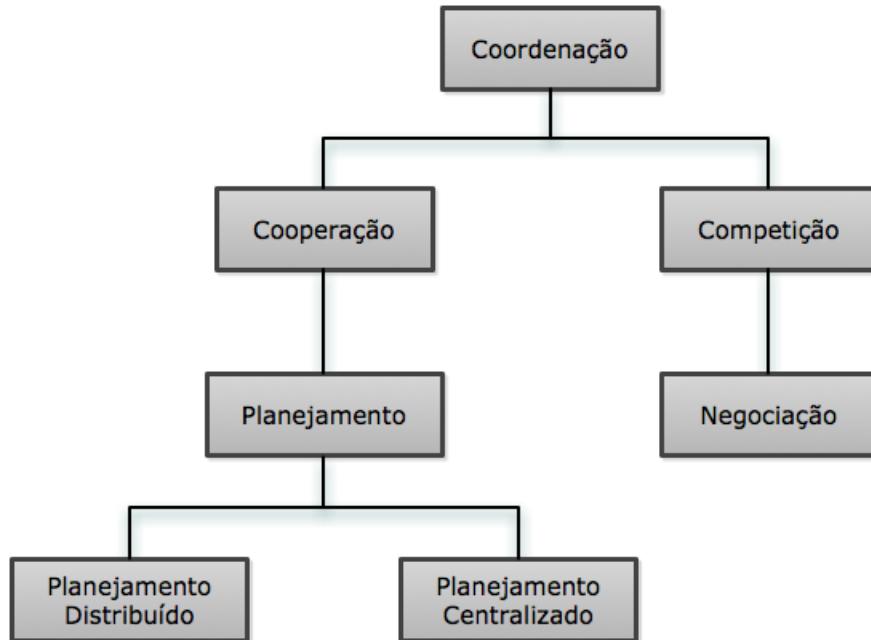


Figura 2.4: Taxonomia do Comportamento dos Agentes em um Sistema Multiagente.

A cooperação entre os agentes pode ser obrigatória (Symbiose), ou não vital (Proto-cooperação). Por exemplo, um filtro de spam que tem uma quantidade limitada de usuários para atuar, se dois filtros trocarem informações entre si ambos terão uma melhor base de dados e filtrarão os dados eficientemente.

Mas essa troca não é algo obrigatório, então eles cooperam entre si. Se esses filtros fossem usados para o mesmo usuário em horários diferentes, a troca de informações seria algo necessário para manter a integridade do sistema, tornando isso obrigatório. Se um desses filtros apenas recebesse a informação, ele não prejudicaria o outro, mas também não o ajudaria (Comensalismo).

A competição, por sua vez, visa determinar o agente que chegue primeiro ao estado-objetivo. Para tal, a ação direta ou indireta (Amensalismo) de um agente no sistema acaba prejudicando 1 ou mais agentes (incluindo a si mesmo em certos casos) nesse mesmo sistema. Se tomamos o caso de um leilão, a cada lance feito por qualquer agente, todos os agentes ficam mais distante do objetivo (competição que prejudica ambas as partes), mas se um agente resolve bloquear a comunicação dos outros agentes, ele acaba eliminando sua concorrência (Predação). Um outro caso curioso é quando temos duas inteligências em um jogo de xadrez jogando contra si e uma delas passa a considerar movimentos possíveis do outro com base nas jogadas anteriores (Parasitismo).

Os sistemas podem ser abertos ou fechados, ou seja, sua composição pode (Sistemas Abertos) ou não (Sistemas Fechados) ser alterada dinamicamente. Os componentes desse sistema (Agentes) podem ser idênticos uns aos outros (Sistema Homogêneo) ou podem ter papéis e algoritmos diferentes (Sistema Heterogêneo).

Na Taxonomia de [Garcia and Sichman, 2003], também classificamos os sistemas por granularidade: caso hajam muitos agentes no sistema (ordem de milhares) dizemos que esse é um sistema com alta granularidade, caso contrário o sistema tem uma baixa granularidade. Além disso o sistema pode ser uma simulação social ou uma resolução social. A simulação estuda porquê os agentes interagem para resolver determinada tarefa, a resolução foca no objetivo de desenvolver novas técnicas de resolução a partir das interações. Essa taxonomia pode ser vista na Figura 2.5.

2.2 Programação Orientada a Agentes (AOP)

2.2.1 Definição

A programação orientada a agentes, em inglês *Agent-Oriented Programming* ou AOP, é um paradigma de computação centrado na teoria de agentes desenvolvido para facilitar o desenvolvimento de sistemas multiagentes. Seu modelo de arquitetura é intrinsecamente *peer-to-peer*³, já que cada agente é autônomo para executar suas funções e comunicar-se com outros agentes dentro de um mesmo ambiente.

Quando adotamos uma iniciativa orientada a agentes para resolver um dado problema, há várias questões a serem resolvidas, como a comunicação entre os agentes. Ao invés de pedir que os desenvolvedores criem padrões e a infraestrutura necessária para a aplicação, é mais conveniente (e barato) criar uma aplicação multiagente em cima de uma camada intermediária AOP que irá prover uma estrutura geral, permitindo que os desenvolvedores foquem na produção do sistema.

Os sistemas multiagentes podem ser facilmente modelados para qualquer linguagem. Em particular, as linguagens orientadas a objeto são consideradas ideais, pois o conceito de agente não é tão distante

³Em português conhecida como ponto-a-ponto. Baseada no desmembramento da arquitetura cliente-servidor conhecida na web, que descentralizou as funções na rede. Usado popularmente em aplicativos de DNS e compartilhamento de arquivos.

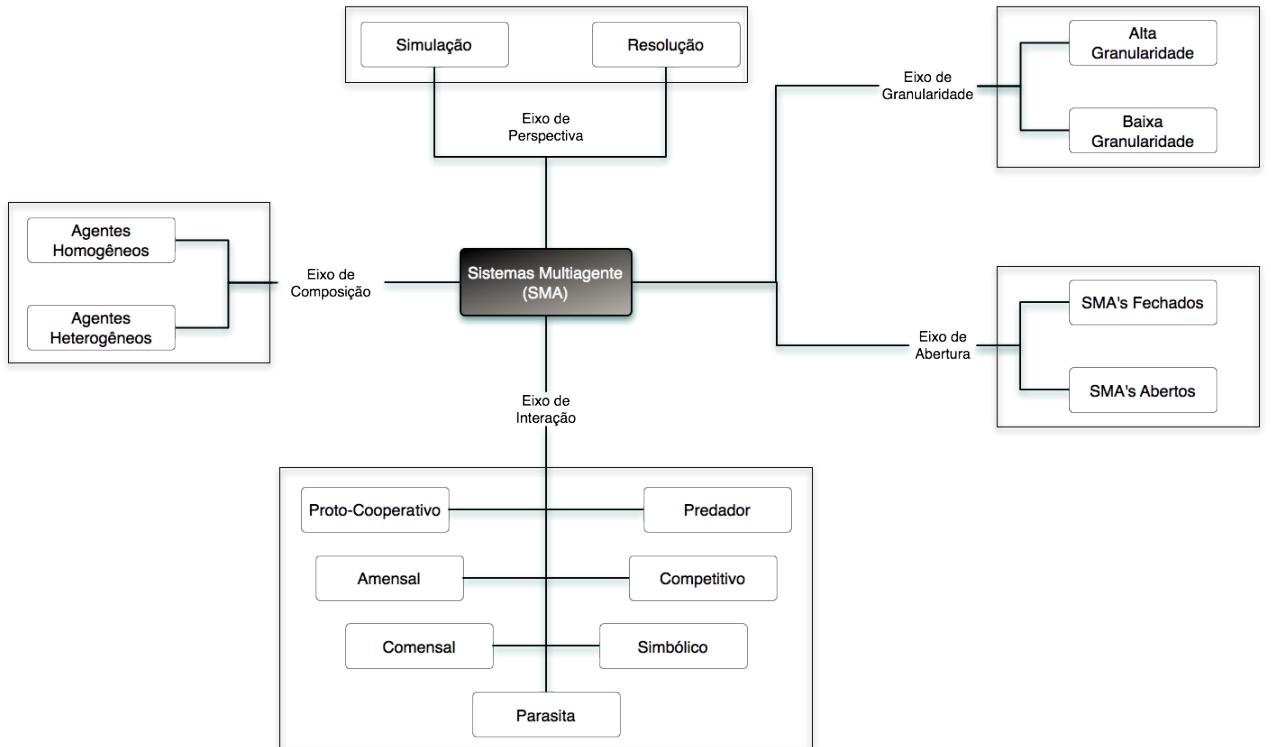


Figura 2.5: Taxonomia dos Sistemas Multiagentes

do conceito de objeto. Os objetos contém muitas características comuns como encapsulamento, e frequentemente, herança e comunicação. Porém, existem diferenças: agentes são autônomos; tem a capacidade de ter um comportamento dinâmico; e cada agente do sistema tem sua própria linha de execução.

Uma linguagem AOP deve prover um ferramental focado nas principais características de um sistema multiagente. Ele deve prover basicamente a estrutura mínima necessária do agente. Mas adicionalmente deve ser capaz de modelar planos, desejos, metas, papéis e normas de um agente. Atualmente há várias linguagens disponíveis como [Bordini et al., 2006]. Algumas foram feitas apenas incorporando alguma teoria de agentes, enquanto outras usam extensivamente as peculiaridades do paradigma. Temos algumas linguagens que são puramente imperativas como FLUX [Thielscher, 2005] and JACK Agent Language [Winikoff, 2005]. Em contrapartida, temos linguagens híbridas, que combinam funções declarativas e imperativas como 3APL [Dastani et al., 2005] e Jason [Bordini et al., 2005].

Outro fator é saber em quais plataformas e *frameworks*⁴ essa aplicação irá rodar. A maioria prove formas de executar a aplicação em diferentes tipos de hardware e sistemas operacionais, em geral com uma camada intermediária para abstrair as operações essenciais como comunicação e coordenação. Muitas dessas plataformas provêm um suporte aos padrões e funcionalidades descritas no FIPA (detalhado na Seção 2.2.2), e alguns suportam tipos especiais de agentes como os agentes móveis[Lange and Oshima, 1998]

Nesse trabalho iremos focar no JADE (descrito na Seção 2.2.3) [Bellifemine et al., 2007] que é

⁴Em português arcabouço. Em geral um *framework* é um conjunto de ferramentas desenvolvidas em uma ou mais linguagens que encapsulam funções e acessos a hardware ajudando o desenvolvedor a criar sua aplicação rapidamente. Exemplo: OpenGL, .NET.

multiplataforma, utiliza extensivamente o padrão FIPA e suporta vários tipos diferentes de arquitetura.

2.2.2 Especificações FIPA

FIPA, em inglês *Foundation For Intelligent Physical Agents*, foi criada em 1996 e é uma organização não lucrativa com o intuito de desenvolver padrões para o desenvolvimento de aplicações relacionadas a teoria de agentes. Inicialmente ela foi formada por membros acadêmicos e organizações industriais, esboçando um conjunto de estatutos para guiar a produção de especificações respeitando leis e abrangendo interesses privados e acadêmicos. Nessa época, os softwares baseados em agentes já eram bastante difundidos no meio acadêmico, mas eram limitados em aplicações comerciais.

Os principais princípios seguidos pela organização são os seguintes:

1. A teoria de agentes provê um novo paradigma para solucionar problemas antigos e atuais;
2. Algumas das tecnologias baseadas em agentes já atingiram um nível de maturidade aceitável;
3. É necessário especificar e padronizar tecnologias baseadas em agentes;
4. A padronização de tecnologias genéricas se mostrou possível, e levou a bons resultados por outras instituições de padronização;
5. A padronização da mecânica interna dos agentes não é o objetivo primário, porém é algo necessário posteriormente para que haja uma interoperabilidade aberta para a linguagem e sua infraestrutura.

Comunicação Entre os Agentes

O FIPA define como padrão de comunicação o FIPA-ACL. Ele foi baseado em diversas tecnologias e padrões de internet feitos durante os anos 90 como OMG, DCE, W3C e GGF. O modelo orientado a serviço é essencialmente uma pilha de comunicação com várias subcamadas dentro da aplicação, ao invés de uma camada única.

FIPA-ACL é baseado na teoria que as mensagens representam ações. Esse protocolo contém um conjunto de 22 ações comunicativas que foram baseados no ARCOL, proposto pela France Telécon. Neste, toda ação é descrita tanto de forma narrativa, quanto de forma semântica baseada em lógica modal [Garson, 1984], que especifica o efeito de enviar uma mensagem nas atitudes mentais do receptor e do emissor. Essa forma de lógica é similar ao modelo BDI (*Beliefs, Desire, Intention*) [Rao and Georgeff, 1995]. A maioria das mensagens eram comuns tais como: “Informar”, “Requisitar”, “Concordo”, “Rejeito” e “Não entendi”⁵. Com isso, FIPA define que um agente deve ser capaz de receber uma mensagem FIPA-ACL e responder ao menos “Não entendi” se a mensagem não pode ser processada.

Baseado nessas ações, FIPA definiu uma série de protocolos de interação, cada um consistindo em uma sequência de ações comunicativas com o intuito de coordenar ações de multiplas mensagens, como rede de comunicação para estabelecer acordos entre agentes para vários tipos de definição de tarefas.

⁵Todas essas comunicações e palavras chave são feitas em inglês no FIPA, então no código, essas mensagens são “inform”, “request”, “agree”, “reject” e “not understood”

Como mencionado, a comunicação em FIPA é dividida em várias sub-camadas baseada nas camadas OSI ou TCP/IP. Nesse contexto temos:

Sub-camada 1 - Transporte: Em FIPA-ACL, a camada mais baixa é a camada de transporte. Foram definido protocolos de transporte para IIOP [IIOP, 1999], WAP e HTTP.

Sub-camada 2 - Encoding: Nessa camada, FIPA define diversas representações de mensagem utilizando uma estrutura de dados como XML, Strings e Bit-Efficient. A última tem como intuito, a comunicação em conexões com banda de rede reduzida.

Sub-camada 3 - Estrutura da Mensagem: Em FIPA a Estrutura da Mensagem é especificada diferentemente do seu *encoding* para melhorar a flexibilidade. Um aspecto importante dessa camada é que é necessário colocar informações extras, além do emissor e do receptor, como por exemplo tempos limites para respostas. A estrutura de uma mensagem em FIPA-ACL pode ser vista na Figura 2.6 e um exemplo de linguagem pode ser visto na Figura 2.7.

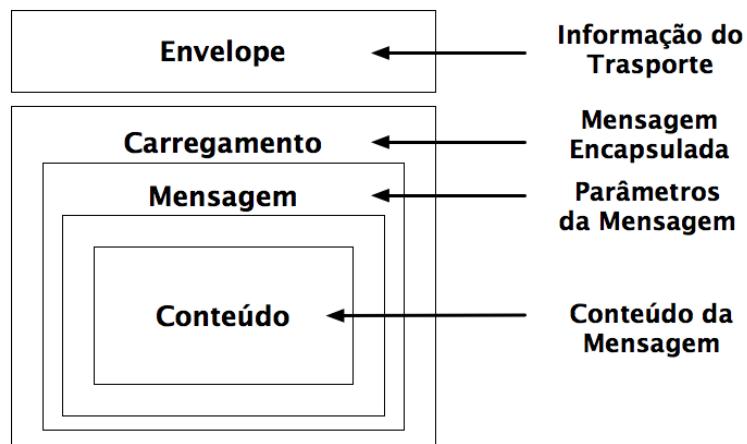


Figura 2.6: Estrutura de uma Mensagem em FIPA-ACL

```

(request :sender (agent-identifier :name alice@mydomain.com)
:receiver (agent-identifier :name bob@yourdomain.com)
:ontology travel-assistant :language FIPA-SL
:protocol fipa-request
:content
  """(action
    (agent-identifier :name bob@yourdomain.com)
    (book-hotel :arrival 15/10/2006 :departure 05/07/2002 ... )
  )"""
)
  
```

Figura 2.7: Exemplo de Comunicação utilizando FIPA-ACL

Sub-camada 4 - Ontologia: As informações passadas por uma mensagem podem estar contidas ou declaradas através de uma ontologia ou modelo conceitual. Apesar de FIPA permitir o uso de ontologias para representar o conhecimento, esse tipo de especificação não é propriamente obrigatória. Mas, é possível referenciar Ontologias Web se necessário.

Sub-camada 5 - Representação do Conteúdo: A atual forma das mensagens em FIPA determina que ela pode ser de qualquer forma, mas há padrões determinados para o uso de fórmulas lógicas, predicados lógicos, expressões algébricas e a combinação desses itens. A linguagem mais usada nesse contexto é o FIPA-SL, utilizando exemplos de fórmulas lógicas como: *not*, *or*, *implies*, *equiv*, etc., e exemplos de operações algébricas como *any* e *all*.

Sub-camada 6 - Ações Comunicativas: Classificação da mensagem em termos de ação, e o que isso implica. Exemplo: “Concordo”, “Rejeito”...

Sub-camada 7 - Protocolo de Interação: FIPA define muitos protocolos de interação como o “*request*”, que é um protocolo de negociação entre agentes que podem concordar ou rejeitar a proposta.

Controle dos Agentes

Além da comunicação, FIPA estabelece como ponto chave o controle dos agentes. Foi criado um modelo de referência para a criação, registro, localização, comunicação, migração e operação dos agentes. Esse modelo pode ser visualizado na Figura 2.8.

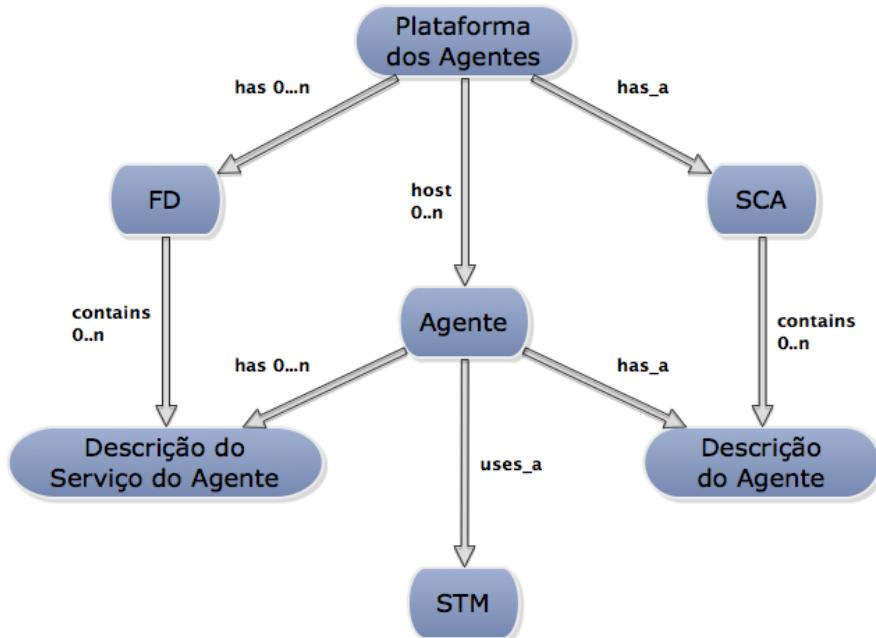


Figura 2.8: Ontologia de Controle dos Agentes

A seguir, são descritos os itens da Figura 2.8:

Plataforma dos Agentes: Provê a estrutura física necessária onde os agentes serão executados. A Plataforma dos Agentes consiste em componentes de controle (descritos a seguir), sistema operacional, ferramental básico, e dos próprios agentes, e de qualquer outro software de suporte à execução. Uma plataforma pode se espalhar por diversos computadores, e os agentes não precisam estar todos alocados no mesmo host.

Agente: É um agente computacional, conforme descrito na seção 2.1. A implementação desse agente não é cabível ao FIPA, que fica responsável apenas pela estrutura e codificação de mensagens usadas para troca de informações entre os agentes. Cada agente deve ter pelo menos 1 dono, e ao menos uma noção de identidade, que é descrito no FIPA como AID (*Agent Identifier*), que rotula um agente e o torna unicamente identificável. Um agente também deve ter um número de registro no protocolo de transporte para que possa enviar/receber mensagens.

Facilitador de Diretórios (FD): O FD é um componente opcional que prove um catálogo de agentes. Ele mantém uma lista precisa e completa de todos os agentes e deve ser capaz de ter um controle de acessos, inibindo um acesso não autorizado de um agente a tal informação. O agente deve requerer ao FD uma modificação de sua própria estrutura, para manter a consistência nessa lista. Porém o FD não garante a autenticidade da lista.

Sistema de Controle de Agentes (SCA): Responsável por administrar o controle da Plataforma dos Agentes, como a criação, exclusão e migração de agentes. Cada agente deve se registrar no SCA e receber sua AID, que é registrada e mantém a informação sobre cada agente, como seu estado atual. Mudanças de estrutura devem ser autorizadas pelo SCA. Quando a vida de um agente se extingue, o mesmo é des registrado do SCA.

Serviço de Transporte de Mensagens (STM): O STM é provido pela FIPA-ACL.

No Apêndice A. são listadas as principais normas FIPA, e o site da organização para acesso da documentação completa.

2.2.3 Framework JADE

JADE é um software desenvolvido pela Telecom Itália em 1998, antigamente com o codenome CSELT. Com o financiamento da *European Commission* (Projeto FACTS, ACTS AC317), o projeto tomou rumos maiores e deixou de ser apenas um software validado pelas especificações FIPA, e se tornou uma camada intermediária poderosa para o desenvolvimento de sistemas multiagentes. Toda API foi focada em usabilidade e simplicidade, visando tornar a curva de aprendizado menor.

Em 2000, sua licença foi alterada para LGPL⁶ e continuou a ser distribuída pela Telecom Itália. Como a LGPL não impõe nenhuma restrição sobre o software, é permitido ao desenvolvedor criar uma aplicação utilizando partes do código, assim como usar outras bibliotecas na criação da aplicação.

Ele foi todo desenvolvido em JAVA. Em seu site, <http://jade.tilab.com> é possível baixar o código, documentação, exemplos de uso e outras coisas acerca do JADE. Nessa seção iremos nos restringir a mostrar como o JADE implementa o FIPA.

Na Figura 2.9 é mostrado um diagrama contendo os principais elementos da arquitetura do JADE. Uma plataforma JADE é composta por n containers de agentes distribuídos pela rede. Os agentes atuam em containers que contenham o processo *JADE runtime* e todos os serviços necessários para sua execução. Existe um container especial chamado de *Main Container* ou Container Principal, que representa o início

⁶Em inglês *Library Gnu Public License*, ela permite ao desenvolvedor usar, distribuir e modificar o software desde que o mesmo refencie os autores do projeto.

da aplicação. Ele deve ser o primeiro a ser chamado, e todos os outros containers devem se registrar nele. Na Figura 2.10 são representadas as relações existentes entre os elementos da arquitetura do JADE.

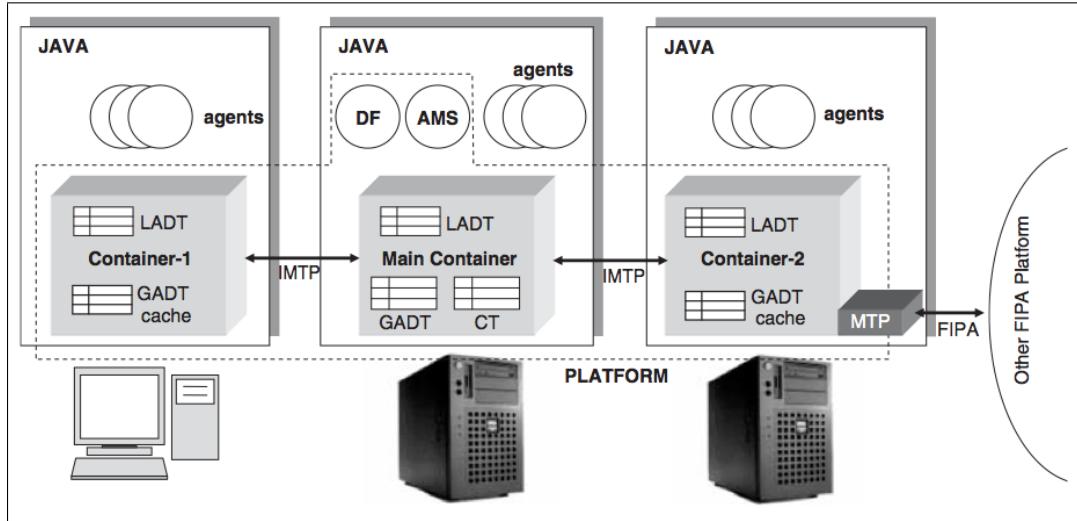


Figura 2.9: Arquitetura do JADE

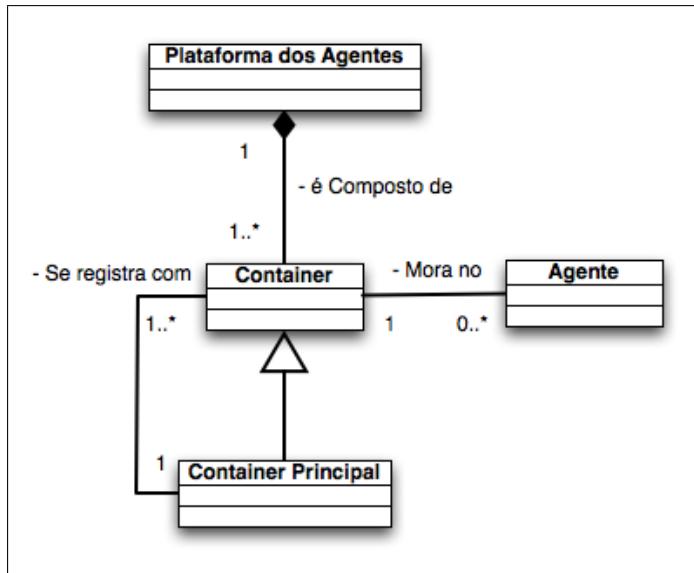


Figura 2.10: Relacionamento entre os elementos da Arquitetura do JADE

Inicialmente, o container principal é responsável por:

- Administrar a Tabela de Containers (CT), registrando todas as referências de objetos e endereços de transporte de todos os containers.
- Administrar um descriptor global de agentes (GATD), que registra todos os agentes presentes na plataforma e suas respectivas características e status.
- Hospedar o Sistema de Controle dos Agentes (AMS) e o Facilitador de Diretórios (DF) como dois agentes especiais que provêm os serviços necessários.

Todos os agentes implementados em JADE são classes que herdam atributos da classe `Agent`. Essa classe prove ao desenvolvedor métodos importantes como o `getLocalName()` que diz o nome único (UID) do agente, e o método `setup()` que é o método principal do agente. Nesse método são configuradas como e quais comportamentos serão usados pelo agente. Os comportamentos são implementados a partir de uma ou mais classes privadas que herdam atributos da classe `Behaviour`. Nessa classe existem vários métodos especiais como:

`action()`: Método principal da classe. Nesse método são implementadas as ações do agente. Ele abstrai o loop de execução das ações do agente. Para não entrar em um loop infinito o método `done()` deve ser implementado.

`done()`: Checa se o *behaviour* terminou, retornando verdadeiro caso verdadeiro.

`block()`: Bloqueia a ação do agente por um determinado tempo

`onStart()`: Método invocado antes do início da execução de um *behaviour* de um agente. Usado para ajustar suas configurações iniciais.

`onEnd()`: Método invocado na remoção do *behaviour* de um agente. Usado para ajustar as configurações finais de um agente.

`reset()`: Restaura o *behaviour* para seu estado inicial.

`myAgent()`: Retorna o agente para qual aquele *behaviour* pertence.

Existem duas formas de iniciar os agentes. A primeira é iniciar por linha de comando. Essa forma é considerada difícil para inicializar muitos agentes, pois todos devem ser especificados no momento da execução do JADE. Um outra forma é utilizando a classe `Runtime`. Ela cria uma abstração da instância de inicialização do JADE, e o programador pode associar vários *containers* e agentes à essa execução. Maiores informações sobre o JADE podem ser encontradas em [Bellifemine et al., 2007], [Bellifemine et al., 2008].

Capítulo 3

Arquitetura de GPU's

Em menos de duas décadas, houve um crescimento espantoso de microprocessadores baseados numa central única de processamento (*CPU* em inglês), como processadores da família Intel[®] e AMD[®]. Com esse crescimento de demanda, as pesquisas levaram esses processadores a um rápido incremento de performance e redução de custos melhorando assim o mercado de computadores pessoais. Esses microprocessadores trouxeram Giga FLOPS para CPU's e centenas de Giga FLOPS para clusters e servidores. Com isso as aplicações começaram a ficar maiores, com melhores interfaces gráficas e resolvendo mais problemas do dia-a-dia, criando assim um mercado promissor.

Durante esse crescimento, os desenvolvedores sempre confiavam em novos avanços de hardware para melhorar a velocidade de suas aplicações, ou seja, o software iria executar “mais rápido” a cada nova geração de processador, pois este viria com um *clock*¹ maior. Porém, a partir de 2003 a indústria começou a ter problemas. À medida que os processadores ficavam mais rápidos, eles também esquentavam mais e consumiam mais energia. Com isso as fabricantes, optaram por usar modelos *Multi-Core* e *Many-Core*, onde existiam várias unidades de processamento, ou núcleos, dentro de um mesmo chip a fim de melhorar o poder de processamento do microprocessador.

Essa mudança provocou uma mudança gigantesca na indústria. Tradicionalmente a maioria das aplicações eram escritas baseadas na arquitetura de Von Neumann [Neumann, 1945] eram seqüenciais. A expectativa de que a cada nova geração de processadores o programa iria executar de forma mais rápida e eficiente não era mais válida. Em um processador com n cores, um programa seqüencial iria executar apenas em um desses n núcleos, tornando-o assim ineficiente. Com o desaparecimento da melhora de desempenho em novos microprocessadores, os desenvolvedores veem reduzidas as chances de crescimento de toda a indústria.

Por outro lado, abre-se a oportunidade para programas com mais de uma linha de execução, ou paralelos. Nesse tipo de programa, múltiplas linhas de execução, chamadas de *threads*, cooperam para que um o programa em si execute de forma mais rápida e eficiente. Essa prática é deveras nova para a indústria. Porém, comunidades de pesquisadores em computação de alto desempenho já fazem uso dessa solução a décadas. Há tempos atrás apenas aplicações em larga escala justificavam tal investimento.

¹Frequência de processamento de instruções de um processador

Porém, com o intuito de tirar o máximo proveito dos microprocessadores *Multi-Core* e *Many-Core*, o desenvolvimento de programas paralelos cresceu vertiginosamente.

3.1 GPU Computing

Uma GPU (Graphics Processing Unit) é um microprocessador em geral acoplado às placas gráficas, que é dedicado a calcular operações de ponto flutuante. Dentro de uma GPU há um *acelerador gráfico* que contém um conjunto de operações matemáticas implementadas em hardware comumente usadas para o processamento gráfico. Os primeiros controladores gráficos apareceram nos anos 70's e eram usados pelos Atari 8 bits como interface de comunicação com as televisões. Somente no início dos anos 90 surgem comercialmente os primeiros chips aceleradores 2D do mercado. Com isso, em 95 a maioria dos fabricantes de placas gráficas já suportavam esse padrão, suprimindo assim as soluções integradas de vídeo. Nos anos subsequentes a tecnologia de placas gráficas continuou evoluindo com a criação de diversas APIs (*application programming interfaces*) como o DirectDraw para windows 95.

Em 1997, surgem soluções 3D integrando CPU e GPU em video-games como Playstation da Sony e o Nintendo 64 da Nintendo, aumentando a demanda por hardwares específicos 3D. No mundo dos PC's houveram alguns fracassos comerciais como a S3 ViRGE, ATI Rage, e a Matrox Mystique que eram essencialmente placas com chips 2D, mas com um suporte à operações em 3D. Mesmo com soluções mais viáveis, a performance 3D só era possível com placas de vídeo que tivessem aceleração dedicada a isso (limitando a renderização 2D totalmente) como a 3dfx Voodoo.

Com o advento do interesse comercial proporcionado pelas placas gráficas, boas soluções em API começaram a surgir, como o OpenGL e o DirectX. Ambas implementações e o desenvolvimento conjunto com os fabricantes de GPU's transformaram radicalmente o pipeline de execução das placas gráficas de um simples rasterizador para algo bem mais complexo.

Em 1999 a NVidia lança a Geforce 256 (ou NV10), trazendo pela primeira vez a tona os Pixel Shaders e Vertex Shaders, mostrados na Figura 3.1. Toda parte de transformações e iluminação já existentes no OpenGL, foram implementadas em Hardware, dando início a uma nova era de processadores gráficos. Os programadores poderiam usar Shaders, programas que rodavam nos Pixel Shaders e Vertex Shaders, para alterar o modo com que a placa gráfica processava vértices e pixels.

Em 2003 foi lançada a Geforce 5 (ou NV30), que pela primeira vez se equipara a uma CPU em Processamento de Ponto Flutuante, ou FLOPS. A partir daí temos o fenômeno ilustrado na Figura 3.2, enquanto o processamento GFLOPS² aumenta a cada ano nas GPU's de forma acentuada, nas CPU's esse aumento é bem menor, gerando uma profunda diferença. Atualmente chegamos a um ponto em que essa diferença de desempenho motiva muitos desenvolvedores a usar a GPU em parte de suas aplicações não-gráficas. Quanto mais escaláveis essas aplicações são e quanto maior carga de trabalho puder ser dividida, melhor para que ela seja executada.

A grande diferença entre as CPU's e GPU's está na forma em que ambas são construídas. O design fundamental de seus processadores é ilustrado na Figura 3.3. O design da CPU é feito de forma a

²Giga FLOPS

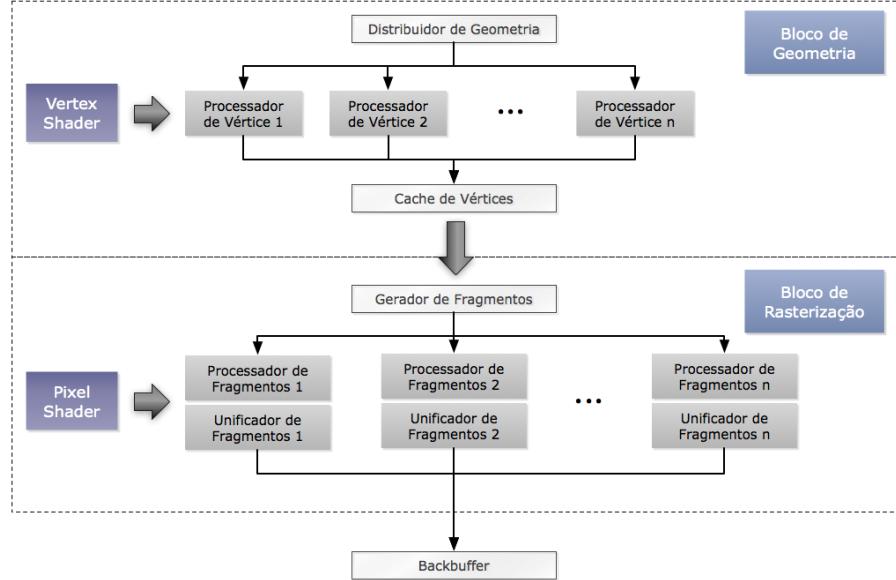


Figura 3.1: Arquitetura de GPU Utilizando Vertex Shaders e Pixel Shaders

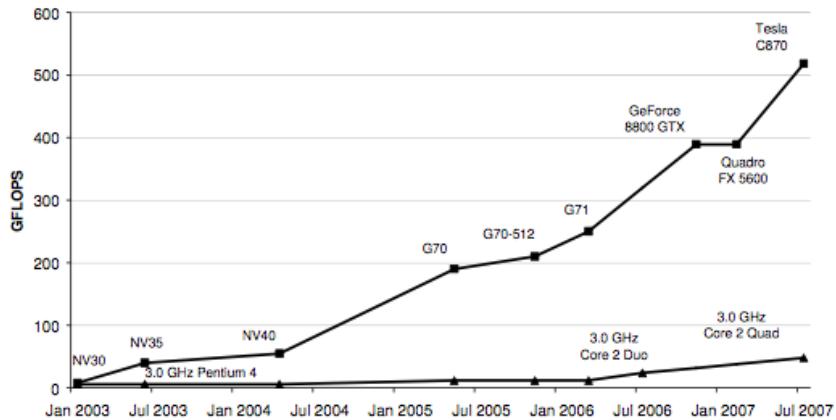


Figura 3.2: Diferença de desempenho entre CPU e GPU em GFlops

tirar proveito de um código seqüencial. Ele contem uma unidade lógica de controle mais sofisticada que permite instruções desde uma única thread de execução até execuções em paralelo, porém criando uma abstração para parecer uma execução seqüencial. As GPU's por sua vez, com a pressão das indústrias de entretenimento, maximizaram a área utilizada para o cálculo de ponto flutuantes. Com isso, foi otimizado o uso massivo das threads em detrimento à execuções seqüenciais, minimizando o uso da unidade de controle, caches, e forçando os fabricantes a desenvolverem barramentos de baixa latência. Como resultado, a maior parte da área dos chips são reservados a unidades de cálculo de ponto flutuante. Uma grande desvantagem desse modelo são suas restrições impostas, tais como recursão e alocação dinâmica. Apenas as placas gráficas atuais baseadas na arquitetura *Fermi* [Fermi Architecture, 2010] tem suporte a essas função, mas de forma limitada.

Um dos fatores importantes para a adoção no mercado além de desempenho, visibilidade e presença em mercado, é a facilidade com que é possível se programar. Até 2006, era tortuoso programar



Figura 3.3: Diferença de Arquitetura entre CPU e GPU

aplicações não-gráficas em chips gráficos, pois além de ter que usar uma API como OpenGL ou DirectX, era necessário converter todo o código para um paradigma diferente. Esse Paradigma é chamado de GPGPU (*General Purpose Programming Using a Graphics Processing Unit*). Mesmo um desenvolvedor com alto conhecimento nessas APIs tinha dificuldades e grandes limitações na tradução do algoritmo, de forma a tirar proveito total da performance da GPU. Em 2007, a NVidia mudou este cenário com o lançamento do CUDA (*Computer Unified Device Architecture*), que será explicado com mais detalhes na Seção 3.2. O programador passou a ter uma maior facilidade ao portar seus programas para ser executado em placas gráficas. Esse novo paradigma passou a se chamar *GPU Computing*.

3.2 CUDA (*Computer Unified Device Architecture*)

O CUDA, Computer Unified Device Architecture, trouxe uma inovação no modo de programar GPUs e em sua arquitetura interna. Na Figura 3.4 é mostrada uma arquitetura formada por 16 *streaming multiprocessors* (SMs), organizados em 8 blocos de *streaming processors* (SPs), num total de 128 (16×8). Os processadores com suporte a Vertex e Pixel Shaders foram substituídos por processadores que podem executar ambas as tarefas, quanto suportar aplicações de uso geral.

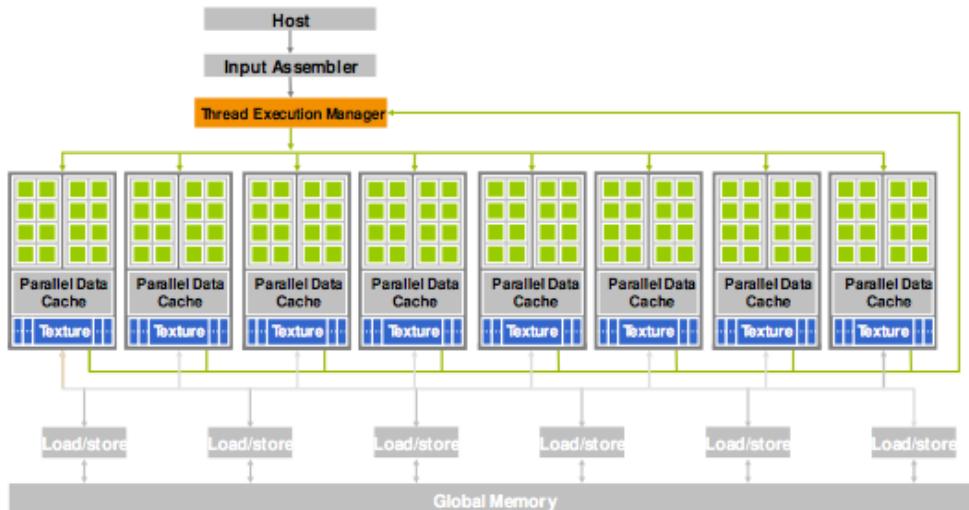


Figura 3.4: Exemplo de Arquitetura capaz de suportar CUDA

Para um desenvolvedor, a arquitetura do CUDA consiste em um *host* que é a CPU tradicional que ele está habituado e um ou mais *devices*, que são as GPUs equipadas com microprocessadores com um amplo número de operações aritméticas e um grande número de processadores. Para tal, CUDA usa uma extensão de C como programação de alto nível, além de prover outras linguagens e APIs como FORTRAN, OpenCL e DirectCompute.

Apesar do paralelismo surgir como uma solução para o problema da evolução dos processadores, descrito na lei de Moore, temos o problema da escalabilidade, ou seja, a medida que aumentamos o número de threads por núcleo temos uma perda de desempenho. Para suprir isto, temos 3 abstrações em principais: hierarquia de threads, hierarquia de memória, e barreiras de sincronização. Elas são apresentadas ao desenvolvedor como algo intrínseco à linguagem. Essas abstrações provem uma fina granularidade de concorrência de dados por threads, combinado com uma alta granularidade de concorrência de dados por tarefas.

Isso leva o desenvolvedor a dividir seu programa em um conjunto de sub problemas que podem ser solucionados independentemente em paralelo por *blocos de threads*, e cada bloco será particionado em pequenas partes que serão resolvidas de forma cooperativa entre cada thread. Na Figura 3.5 é ilustrado como o controlador das threads escalona os blocos de threads em GPUs com quantidade diferente de núcleos.

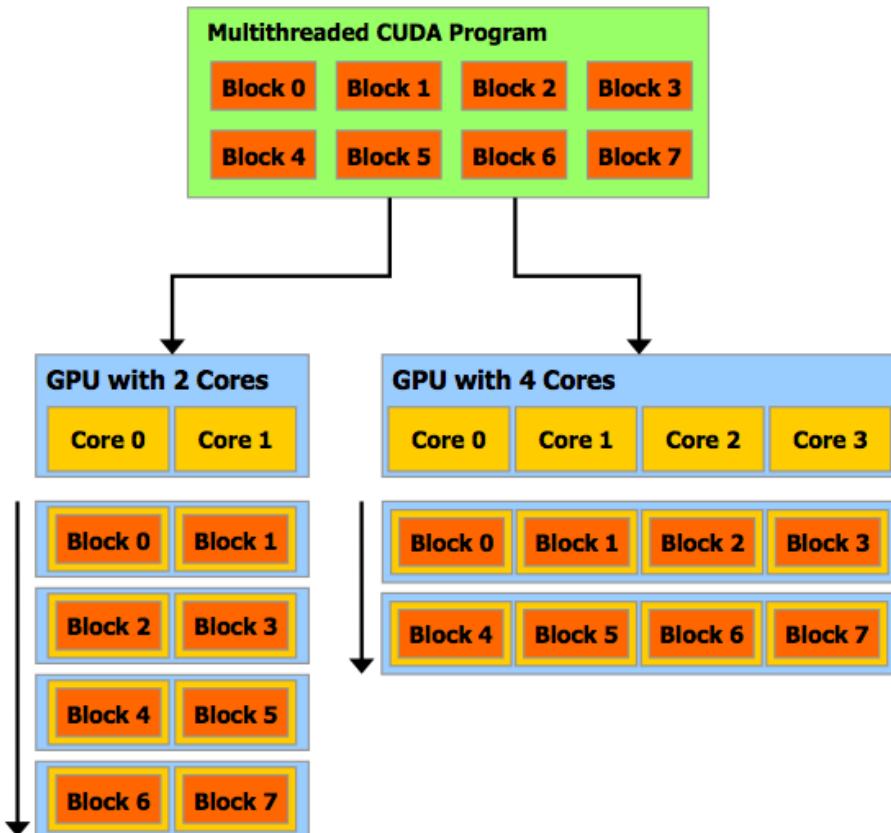


Figura 3.5: Um programa multithread é partitionado em blocos de threads que executam independentemente. Assim quanto maior o número de núcleos da GPU, menor será o tempo de execução.

Kernel

Para o acesso no *device*, CUDA define funções especiais chamadas de *Kernels*. Um Kernel é definido utilizando a macro `__global__` antes da função. Esse tipo de função é acionada apenas pelo *host* e executa diretamente no *device*. Ao executar, o desenvolvedor deve dizer o número de threads e blocos utilizando a sintaxe `<<<...>>>`. Na Figura 3.6 é ilustrado como se dá a criação e a chamada de um kernel a partir do host.

```

1 // Definicao do Kernel
2 __global__ void foo(float* A, float* B, float* C)
3 {
4 }
5
6 int main(){
7     //Chamada do Kernel
8     foo<<<1, N>>>(A,B,C);
9 }
```

Figura 3.6: Sintaxe para definição e execução do Kernel

3.2.1 Hierarquia de Threads

Quando executado, cada thread do kernel recebe uma *ID* única, que pode ser acessada por todas as threads do bloco. Essa identificação é tri-dimensional, estando assim esta thread em um block unidimensional, bi-dimensional ou tri-dimensional. Por conveniência, existe uma variável tri-dimensional chamada `threadIdx` que retorna as 3 componentes de cada thread. Na Figura 3.7 é exemplificado o seu uso. Nesse caso é criado um bloco bidimensional de threads e cada thread fica responsável por fazer a adição em um determinado ponto da matriz.

```

1 // Definicao do Kernel
2 __global__ void MatAdd(float A[N][N], float B[N][N], float C[N][N])
3 {
4     int i = threadIdx.x;
5     int j = threadIdx.y;
6     C[i][j] = A[i][j] + B[i][j];
7 }
8 int main()
9 {
10    ...
11    // Chamada de Kernel com um bloco de N * N * 1 threads
12    int numBlocks = 1;
13    dim3 threadsPerBlock(N, N);
14    MatAdd<<<numBlocks, threadsPerBlock>>>(A, B, C);
15 }
```

Figura 3.7: Exemplo de uso do `threadIdx`

Há um limite no número de threads por bloco, já que todas as threads de um bloco irão residir num determinado núcleo e devem partilhar da memória deste núcleo. Atualmente esse número é 512

threads. Lembrado sempre que o kernel é executado por múltiplos blocos de mesmo tamanho, sendo o total de threads = Número de blocos × Número de threads por bloco.

Assim como as threads podem ser organizadas em blocos, os blocos podem ser organizados em *grids* unidimensionais e bidimensionais, como mostrado na Figura 3.8. Usando a sintaxe `<<<...>>>` podemos especificar o número de blocos por grid. Para isso, é necessário o uso da variável do tipo `int` ou `dim3`. Cada bloco pertencente a um grid terá um identificador unidimensional ou bidimensional que pode ser obtido através da variável `blockIdx`, assim como sua dimensão pode ser obtida através da variável `blockDim`.

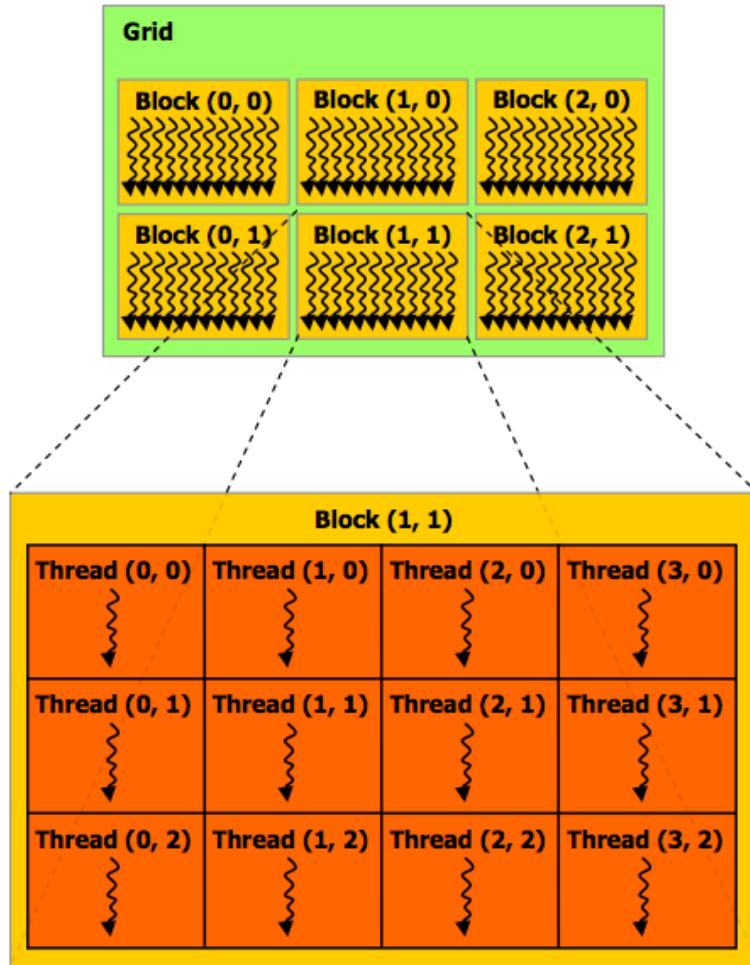


Figura 3.8: Grid de Threads

Na Figura 3.9 temos um exemplo mais prático. Na linha 12 declaramos um grid bidimensional 16×16 , e na linha 13 declaramos que o grid terá tamanho $(\frac{N+\text{dimBlock.x}-1}{\text{dimBlock.x}}, \frac{N+\text{dimBlock.y}-1}{\text{dimBlock.y}})$. Ou seja, se $N = 20$, teremos matrizes 20×20 , blocos 16×16 , e um grid 2×2 .

3.2.2 Sincronização

A cooperação entre threads de um determinado bloco é realizada a partir da *memória compartilhada*. CUDA provê a função intrínseca `__syncthreads()`, que atua como uma barreira, fazendo com que as

```

1 // Definicao do Kernel
2 __global__ void MatAdd(float A[N][N], float B[N][N], float C[N][N]){
3     int i = blockIdx.x * blockDim.x + threadIdx.x;
4     int j = blockIdx.y * blockDim.y + threadIdx.y;
5     if(i < N && j < N)
6         c[i][j] = A[i][j] + B[i][j];
7 }
8
9 int main(){
10    ...
11    //Chamada do Kernel
12    dim3 dimBlock(16, 16);
13    dim3 dimGrid((N + dimBlock.x - 1) / dimBlock.x,
14                  (N + dimBlock.y - 1) / dimBlock.y);
15    MatAdd<<<dimGrid, dimBlock>>>(A,B,C);
16 }
```

Figura 3.9: Exemplo de uso do `dimBlock` e `blockIdx`

threads de um determinado bloco fiquem bloqueadas até que todas as threads do bloco alcancem o mesmo ponto, evitando uma condição de corrida. Ou seja, prevenir uma inconsistência nos dados compartilhados devido ao acesso não atômico das threads.

Na Figura 3.10 temos um exemplo do uso desta função. Na linha 8, ela é usada para sincronizar todas as threads antes de modificar o vetor compartilhado `vet`.

```

1 // Definicao da variavel compartilhada entre as threads
2 __shared__ float vet[N];
3
4 // Definicao do Kernel
5 __global__ void corners(float vet[N]){
6     int index = threadIdx.x;
7
8     __syncthreads()
9     if((index + 1) < N && (index - 1) > 0){
10         vet[index] = vet[index]+ vet[index+1] + vet[index-1];
11     }
12 }
13
14 int main(){
15    ...
16    // Chamada do Kernel
17    corners<<<1,N>>>(vet [N]);
18 }
```

Figura 3.10: Exemplo de uso do `__syncthreads()`

3.2.3 Hierarquia de Memória

Uma thread em CUDA pode acessar dados de diversos espaços de memória durante sua execução. Como podemos ver na Figura 3.11, cada thread tem um espaço reservado exclusivamente para ela, cada bloco tem um espaço de memória compartilhado entre as threads inerentes ao bloco e existe um espaço global acessado por todas as threads da execução.

Para complementar, há dois espaços de memória somente de leitura acessível por todas as threads: a memória constante e memória de texturas. Elas persistem através das execuções do kernel dentro de uma mesma aplicação.

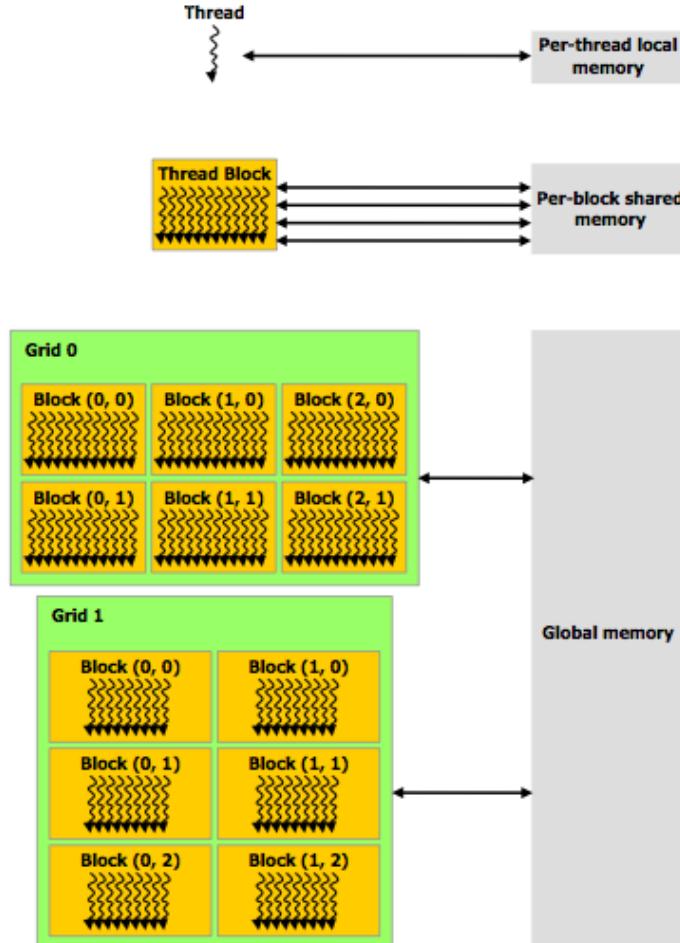


Figura 3.11: Hierarquia de Memória das Threads em CUDA

Programação Heterogênea (*Host* e *Device*)

O modelo de programação do CUDA assume que a GPU (*Device*) atua como um coprocessador da CPU (*Host*). Assim sendo, o código é iniciado no *host* e, em um dado momento da execução, o *device* é acionado, bloqueando a ação do *host* até que a execução no *device* seja finalizada. Isso pode se repetir diversas vezes durante a aplicação e a configuração de blocos, grids e número de threads pode variar a cada chamada do *device*. Isso é ilustrado na Figura 3.12.

Tanto *host* quanto o *device* mantém sua própria memória RAM, denominada *memória do host* e *memória do device* respectivamente. Todo gerenciamento de *memória do device*, como alocação desalocação, é feito pela API do CUDA através da CUDA Runtime, assim como a transferência de dados entre o *host* e o *device*.

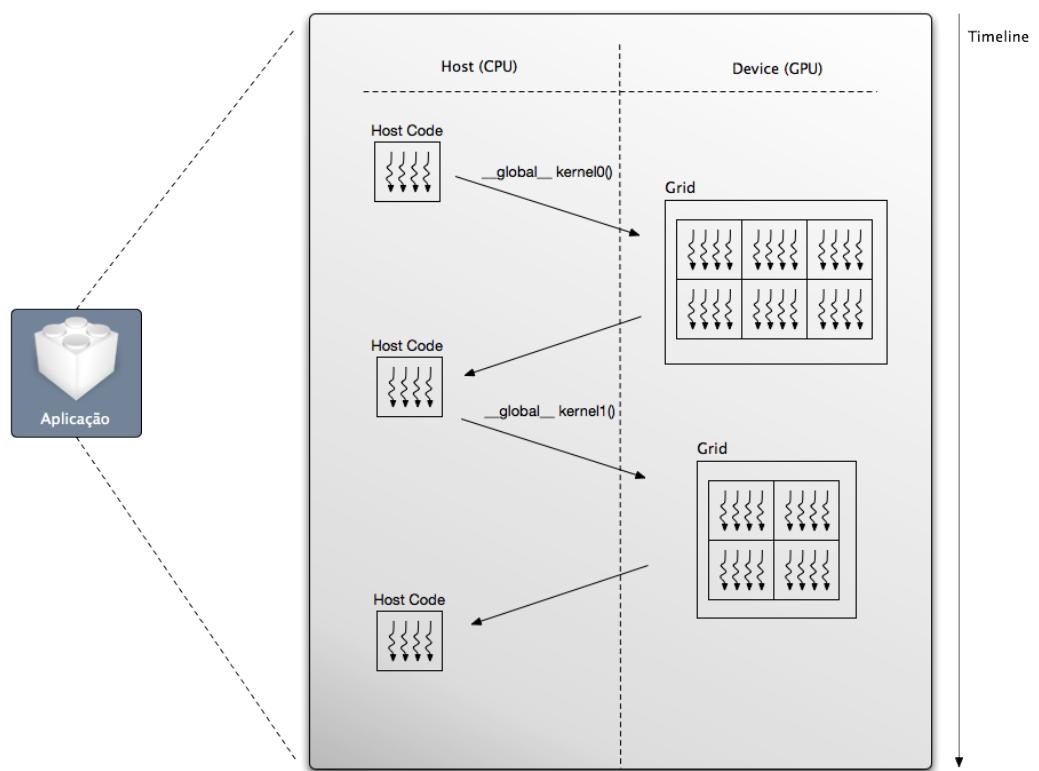


Figura 3.12: Exemplo da Programação Heterogênea existente em CUDA

Capítulo 4

Estudo de Casos

4.1 Motivação e Trabalhos Relacionados

Recentemente, presenciamos um grande uso de tecnologias orientadas a agente, sendo tais soluções consideradas inovadoras. Um sistema multiagente tem aspectos interessantes, pois permite que criemos um subproblema baseado em satisfações e objetivos de cada agente, de tal forma que o problema como um todo é adaptado para um paradigma distribuído. Como exemplos de trabalhos temos criação de veículos autônomos [Stone and Veloso, 1997], Coordenação de Sinais de Trânsito [Bazzan, 2005], Sistemas de evacuação [Dimakis et al., 2009], entre outros.

Paralelamente a esse contexto, vimos um grande aumento do uso de paralelismo em placas gráficas e paradigmas GPGPU como solução de muitos problemas propostos. Em alguns desses trabalhos [Passos et al., 2008], [Bleiweiss, 2008], começa a aparecer uma convergência entre o uso de alternativas GPGPU e a teoria de agentes. Porém, não temos ferramentas para o desenvolvimento direto de sistemas multiagentes em GPGPU como temos para outros sistemas, como por exemplo os *frameworks* JADE e JADEX [Jad,].

O propósito deste trabalho é explorar como mapear um sistema multiagente criado a partir de uma arquitetura distribuída, para arquiteturas GPGPU. Para tal utilizamos o *Framework* JADE e os Padrões FIPA nas implementações, e portamos as aplicações para CUDA. Desenvolvemos dois casos de testes que serão detalhados a seguir e mostram vantagens e desvantagens de tais mapeamentos.

4.2 Cenários Unidimensionais: O Problema dos Caixotes

Esse problema tem como objetivo transportar os caixotes de um lado do mapa ao outro. Apesar de ser considerado simples, se o número de agentes for muito alto o tempo de execução da simulação pode aumentar muito, mesmo em uma arquitetura distribuída. O objetivo principal desse estudo de caso é ver como funcionam as funções básicas (a criação do ambiente, dos agentes e a interação entre os mesmos) e como ambos os sistemas se comportam com um número elevado de agentes.

Na Figura 4.1 é mostrado como o problema é contextualizado. Em (a) inicializamos o **ambiente**

sem nenhum caixote. Logo após, em (b) inicializamos um número n de pacotes em posições aleatórias em um lado do mapa. Em (c), m agentes, $m \leq n$ são inicializados e posicionados aleatoriamente ao longo do cenário. Note que, como estamos lidando com um cenário unidimensional os agentes só tem 3 opções: (1) carregar o caixote, (2) mover para a direita, (3) mover para a esquerda. Em (d) vemos uma representação do sistema em execução. Os agentes são autônomos pois cada um decide se anda para algum lado ou se move o caixote independentemente da decisão de outro agente. O sistema termina quando todos os caixotes são movidos.

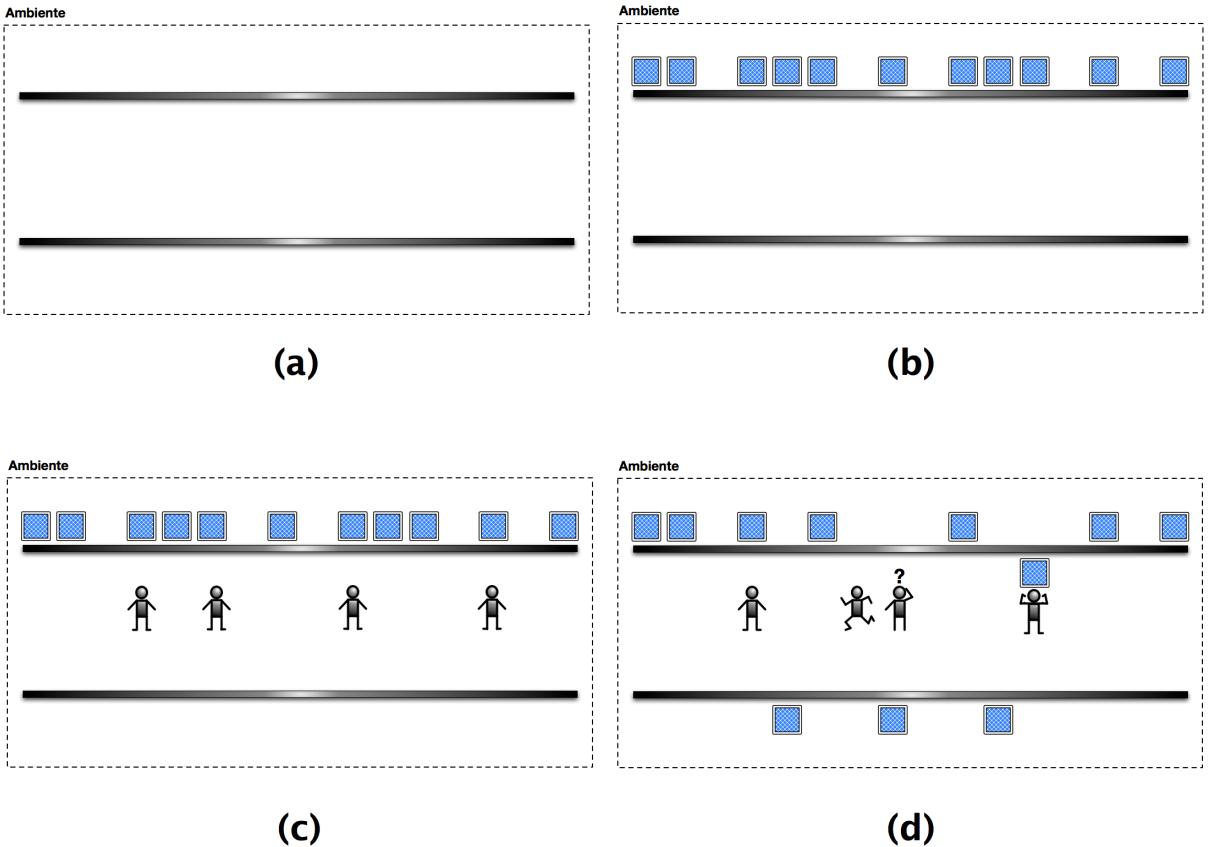


Figura 4.1: Problema dos Caixotes

O sistema é classificado como um sistema multiagente cooperativo de planejamento independente. Além disso ele é fechado e possui uma alta granularidade no casos de teste. Todos os agentes são homogêneos e seguem o Algoritmo 1 descrito a seguir. Nesse algoritmo temos que a única informação dos agentes é sua posição e, a partir dela, o agente deve verificar se há um caixote nessa posição (linha 1) e movê-lo (linha 2). Caso contrário, ele decide se anda para a direita ou para a esquerda, ou se faz um movimento aleatório (linhas 5 a 10).

4.2.1 Mapeando do JADE para o CUDA

JADE segue estritamente os padrões FIPA e mapeia cada agente criado como uma linha de execução única em um **container** de agentes. Um dos grandes problemas encontrado na passagem de uma implementação para outra é o paradigma adotado por ambas. Em JADE, todos os agentes são objetos cuja inteligência

Algoritmo 1 Algoritmo Para Mover os Caixotes.

Requer: *Pos*: Posição do Agente.

```

1: se Há um caixote na posição do agente então
2:   MoverCaixote(pos);
3: fim se
4: enquanto Todas os caixotes não foram movidos faça
5:   se Há um caixote do lado direito então
6:     MoverAgente(direita);
7:   senão se Há um caixote do lado esquerdo então
8:     MoverAgente(esquerdo);
9:   senão
10:    MoverAgente(Aleatório);
11:  fim se
12: fim enquanto

```

é implementada através de classes especiais chamadas de **behaviours**, e é programada como um método especial inerente ao objeto. CUDA por sua vez, só permite a chamada de uma função direta em GPU, o *kernel*. Todos os agentes foram mapeados como um vetor e, quando o *kernel* é executado, cada agente é representado pela sua respectiva linha de execução criada.

A camada de abstração no JADE ajuda a informar o desenvolvedor em que parte do código mexer para que o agente mude sua forma de pensar e subentende que há um loop de execução do sistema, criando métodos especiais como a **start()** e **done()**. O desenvolvedor em CUDA ainda deve ligar com algumas restrições de linguagem como a não possibilidade de alocação dinâmica e recursão.

Em JADE, utilizando o paradigma AOP, todos os agentes foram mapeados como objetos descendentes da classe **Agent**. Cada um deles tem uma classe privada descendente da classe **Behaviour** chamada de **PersonBehaviour**. Essa classe é responsável por determinar as ações do agente e verificar quando é necessário terminar a execução. Para tal, são usados dois métodos especiais da classe, os métodos **action()** e **done()** respectivamente.

Para a criação do ambiente e inicialização dos caixotes, há um outra classe chamada de **Scenario**, que também é responsável por criar e colocar os agentes nas suas posições. Todos os agentes são colocados no mesmo container chamado de **AgentContainer**. Ao final da execução os tempos são calculados e os agentes são excluídos.

Apesar do CUDA ser considerado uma linguagem com uma pequena curva de aprendizagem e melhorar significativamente o desempenho e escalabilidade das aplicações, seu mapeamento a partir do JADE não foi tão simples. Uma das maiores dificuldades foi sair do paradigma AOP para o GPGPU, e perder todas as facilidades da camada intermediária que eram oferecidas no JADE.

Todos os agentes foram mapeados em um único vetor, onde o índice do mesmo era sua identificação e o valor desse índice continha a posição atual do agente. A função global *kernel* criou um número *m* de *threads*, correspondentes ao número máximo de agentes no sistema, e cada *thread* indexando uma posição do vetor. Antes da execução é necessário alocar todos os recursos necessários na GPU através da função **cudaMalloc()** e copiá-los através da função **cudaMemcpy()**, como o ambiente com os caixotes e a posição inicial de cada agente.

Note que, o *kernel* é nossa função **action()** no JADE. Caso existissem mais de um tipo de agente

ou um agente que trocasse seu *behaviour* dinamicamente durante a execução, o problema seria mapeado de uma forma totalmente diferente, e todos esses *behaviours* deveriam estar estaticamente implementados no *kernel*.

Analogamente, o método `done()` do JADE, que analisava se todos os caixotes foram movidos, é implementado no CUDA. Nesse caso usamos uma função do tipo *device*, visto que ela não seria lançada pelo usuário, e todas as linhas de execução teriam acesso a mesma. Quando todos os caixotes são movidos, a execução é finalizada, utilizamos novamente a função `cudaMemcpy()` para copiar os novos dados da GPU para a CPU, e liberamos a memória utilizando a função `cudaFree()`. Nos apêndices B e C é possível ver o diagrama de classes utilizadas em JADE e as assinatura das funções utilizadas em CUDA respectivamente.

4.2.2 Análise de Desempenho

Neste trabalho, utilizamos uma máquina com CPU Intel® Core 2 Duo¹ 2.26GHz², 4GB³ de RAM DDR3⁴ e uma GPU NVidia Geforce 9400M⁵ com 256MB de memória DDR3.

Foram testadas 22 instâncias, que podem ser vistas na Tabela 4.1, e cada experimento foi executado 10 vezes. **Scn** significa a identificação do cenário, **# Agentes** representa o número de agentes usados no experimento, e **# Caixotes** representa o número de caixotes a sempre movidos pelos agentes em cada execução. No apêndice D estão as tabelas de tempo das 10 execuções de cada instância em milisegundos.

Scn	# Agentes	# Caixotes
1	250	500
2	500	500
3	250	1,000
4	500	1,000
5	1,000	1,000
6	250	2,000
7	500	2,000
8	1,000	2,000
9	1,500	2,000
10	2,000	2,000
11	250	5,000
12	500	5,000
13	1,000	5,000
14	1,500	5,000
15	2,000	5,000
16	5,000	5,000
17	250	10,000
18	500	10,000
19	1,000	10,000
20	1,500	10,000
21	2,000	10,000
22	5,000	10,000

Tabela 4.1: Especificação dos Casos de Teste

Em cada execução, foi calculado o *speed-up* entre as implementações em JADE e em CUDA. Chamamos de *speed-up* a diferença entre o tempo calculado utilizando JADE para executar o Algoritmo 1 dividido pelo tempo calculado em CUDA para a mesma tarefa. Calculamos o desvio padrão para as 10 execuções do sistema. Nas Figuras 4.2, 4.3 e 4.4 é mostrado o speed-up do sistema em relação aos cenários, agentes e caixotes respectivamente. As barras verticais representam o desvio padrão calculado. Note que na Figura 4.2 os casos de teste não puderam ser calculados, pois o JADE não conseguiu criar mais de 5000 agentes⁶.

¹Linha da intel que possui dois núcleos e através da tecnologia Hyperthreading emula mais dois núcleos. Mais informações em <http://www.intel.com>

² 10^9 Hertz. Representa o número de instruções processadas por ciclo do processador

³ $2^{30} = 1073741824$ Bytes

⁴DDR3 SDRAM ou Taxa Dupla de Transferência Nível Três de Memória Síncrona Dinâmica de Acesso Aleatório

⁵GPU com 16 CUDA Cores com Frequência de 51 GFlops, Interface de Memória 128 bits, e velocidade de transferência de 21GB/s. Mais informações em http://www.nvidia.com/object/product_geforce_9400m_g_us.html

⁶O erro gerado foi “Unable to create new native thread.”

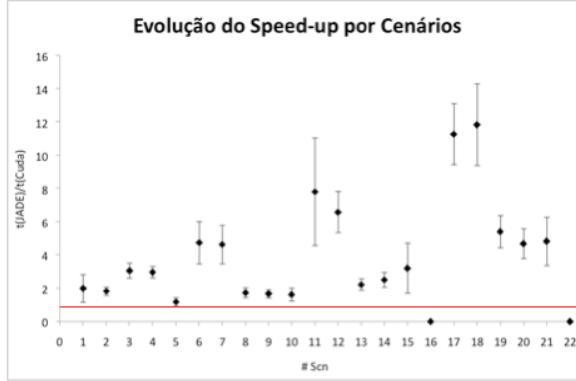


Figura 4.2: Speed-Up em relação aos cenários.

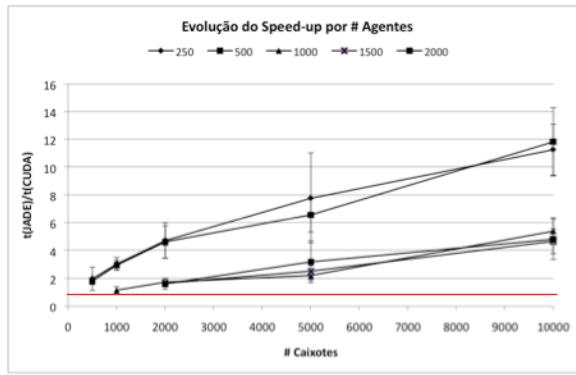


Figura 4.3: Speed-Up em relação ao número de agentes.

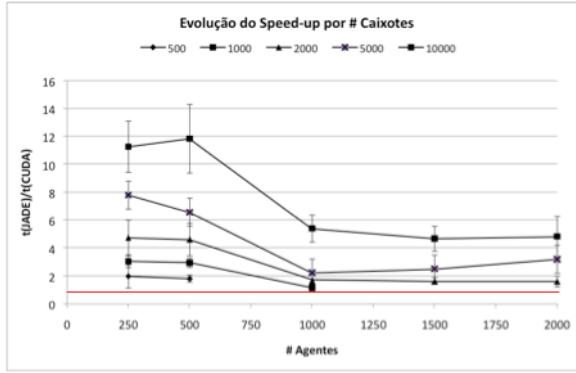


Figura 4.4: Speed-Up em relação ao número de caixotes.

Notamos um decaimento no speed-up a medida que adicionamos mais caixotes no sistema até um ponto onde eles se mantêm praticamente constantes (1000 caixotes). Esperamos que com o uso de GPU's mais poderosas esse "ponto de equilíbrio" se mostre a frente do que nesses testes. A cada crescimento de agentes o speed-up aumenta, devido ao fato de que no JADE é necessário criar manualmente cada thread, enquanto que no CUDA ao chamar a função *kernel*, todas as threads do sistemas estarão criadas.

4.3 Cenários Bidimensionais: Pathfinding A*

Esse é um problema mais complexo que o anterior, tanto em relação à “inteligência” dos agentes quanto ao uso de estruturas e algoritmos. Neste problema, o agente deverá ser capaz de encontrar o caminho de sua origem até o ponto final que denominaremos de *checkpoint*. O objetivo é avaliar funções mais complexas como busca e sincronização entre os agentes.

Na Figura 4.5 é contextualizado o problema proposto. Em (a) iniciamos o cenário bidimensional de tamanho $m \times n$. Em (b) definimos o *checkpoint* representado pelo espaço em verde, e as posições iniciais dos agentes representado pelos espaços em vermelho. Em (c) vemos a inicialização dos agentes. Com as informações limitadas (sua posição, posição final, mapa), o agente em (d) deve calcular sua rota. Note que cada agente calcula essa rota em separado, tendo espaços sendo mapeados na rota de mais de um agente. Em (e) vemos o programa em execução, e um dos problemas existentes acontece quando dois agentes pretendem ir para a mesma posição, esse problema deverá ser tratado durante a execução. O estado final do sistema é visto em (f), onde todos os agentes alcançaram o *checkpoint*.

Diferentemente do primeiro sistema, temos agora um sistema competitivo com planejamento individual, porém os agentes continuam sendo homogêneos. Escolhemos como algoritmo de busca de caminho o algoritmo A*. Tal escolha deu-se devido ao seu baixo consumo computacional e sua heurística se aproxima do caminho ideal, encontrado em alguns algoritmos determinísticos como o Dijkstra.

4.3.1 O Algoritmo A*

O algoritmo A*⁷ [Nilson, 1971] é um algoritmo de busca que usa uma heurística de um custo mínimo ou um caminho com distância mínima e utiliza o princípio da programação dinâmica. O conhecimento heurístico usado pelo método é uma estimativa da distância que falta até o nó objetivo. O A* avalia o nó candidato, n , para um caminho estendido a partir da soma de custo ou distância ao longo do caminho até agora percorrido, chamado de $g(n)$, mais a estimativa do custo ou distância até o nó objetivo, chamado de $h(n)$. A função de avaliação é dada pela Equação 4.1.

$$f(n) = g(n) + h(n) \quad (4.1)$$

No Algoritmo 2 são encontrados os passos do algoritmo A* no problema proposto. Ele usa todas as informações disponíveis para o agente, o mapa $m \times n$, a posição inicial do agente, chamada de *pos*, e seu objetivo final, chamado de *checkpoint*. Inicialmente, o algoritmo cria duas listas, chamadas de *ListaAberta* e de *ListaFechada*, ambas devem iniciar o algoritmo vazias. A execução inicia colocando a posição inicial do agente na lista aberta. Como a *ListaAberta* está com 1 item dentro dela, e ele ainda não é o *checkpoint*, a variável *A* recebe o primeiro item da *ListaAberta*, o coloca na *ListaFechada* e para todos os adjacentes ela avalia primeiramente se o ponto é válido, ou seja, se o ponto não é um obstáculo ou não está fora dos limites da matriz. Se o ponto satisfaz essas condições, a função *vê* se algum adjacente já está presente na *ListaAberta*. Caso positivo, os valores daquele nó são atualizados caso o caminho

⁷Pronuncia-se A Estrela

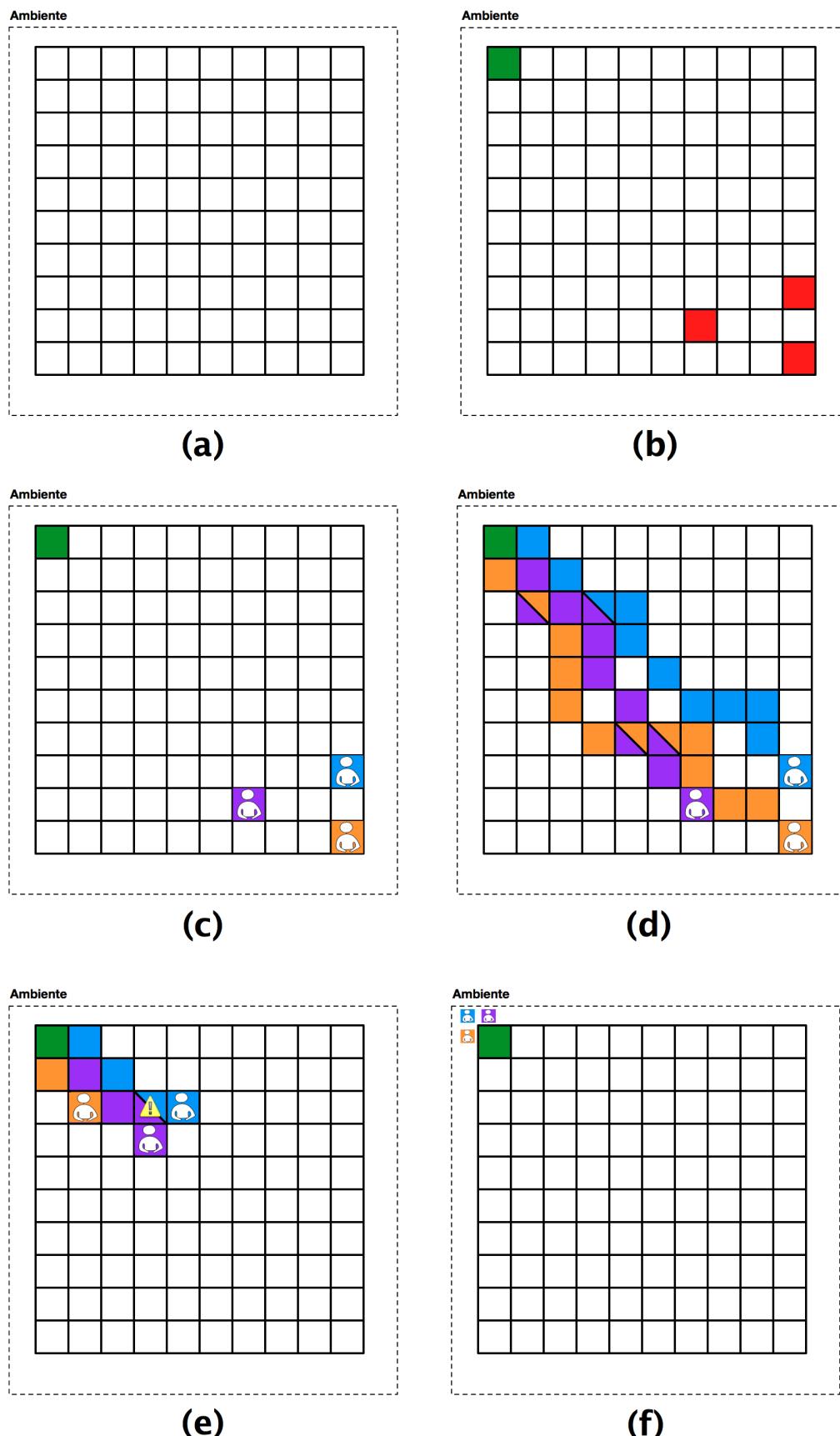


Figura 4.5: Representação do Sistema Bidimensional com o Algoritmo A*

encontrado seja menor. Caso contrário, nada é feito. Se o ponto não estiver na lista ele é adicionado. Ao adicionar todos os adjacentes válidos, a lista é ordenada. Essa repetição é feita até que o ponto desejado seja encontrado.

Note que, quando o *checkpoint* é adicionado na *ListaFechada*, não significa que essa lista é o caminho. Deve-se organizá-la, de forma a ver quem é o pai de cada nó adicionado e então encontrar o caminho do *checkpoint* até o nó de início. Isso é feito na linha 19.

Algoritmo 2 Algoritmo A*.

Requer: *Map*: $m \times n$ Mapa Bidimensional.

Requer: *Pos*: Posição do Agente.

Requer: *Checkpoint*: Posição desejada do Agente.

Condição: *ListaAberta* = *ListaFechada* = $\{\phi\}$.

```

1: ListaAberta  $\leftarrow$  Pos;
2: enquanto Checkpoint  $\notin$  ListaFechada and ListaAberta  $\neq \{\phi\}$  faça
3:   A  $\leftarrow$  ListaAberta[0];
4:   ListaFechada  $\leftarrow$  A;
5:   Remover(A,ListaAberta);
6:   para todo Adjacentes Ai de A faça
7:     se Aix, y não é um obstáculo e  $0 \leq x \leq m$ ,  $0 \leq y \leq n$  então
8:       se  $\neg(\exists P, P\{x, y\} = A_i\{x, y\}$  e  $P \in ListaAberta)$  então
9:         ListaAberta  $\leftarrow A_i;
10:        Ordenar(OpenList);
11:        senão se Ai{G} < P{G} então
12:          Remover(P,OpenList);
13:          OpenList  $\leftarrow A_i;
14:          Ordenar(OpenList);
15:        fim se
16:      fim se
17:    fim para
18:  fim enquanto
19:  Path  $\leftarrow$ Organizar(ClosedList); // Função que irá pegar todos os pontos necessários do caminho.
20: Retorne Path;$$ 
```

	Método de Dijkstra	Método A*
Estrutura de Dados	Conjunto de Marcações Permanentes em cada nó	Lista Ordenada
Interfaces de Conhecimento	Vetor a_{ij} para distância do caminho	Medida para distância do caminho, gerador incremental de passos, estimador de distância até o objetivo
Direcionamento	Não há	Através de função de avaliação
Métodos Intrínsecos	Busca <i>best-first</i> para o nó com caminho mais curto até o nó base	SP-2
Complexidade no Pior Caso	$O(b^d) = O(n^2) = O(N)$	$O(b^k)$, onde b é o fator de partição e k é o número de níveis.
Complexidade Média	$O(n^2)$	$O(N \exp(C\phi(N)))$, onde $\phi(N) = \log(N)^k$
Critério de Solução	Otimização	Otimização
Característica Principal	Busca Completa	Busca completa caso a função heurística admita tal fato.

Tabela 4.2: Tabela comparativa entre o Método de Dijkstra e o Método A*

A complexidade e desempenho do A* vem sendo estudada de ambas as formas, empírica e teórica.

No melhor caso, o algoritmo A* é $O(N)$, ou seja, ele encontra o caminho diretamente ao nó objetivo, onde N é o número de nós do nó base ao nó objetivo. No pior caso, o algoritmo é $O(b^N)$, ou seja, uma complexidade exponencial para o tamanho do caminho, onde b é o fator de partição. Estudos mostram que a complexidade do algoritmo A* depende diretamente da função heurística usada na avaliação da função, tendendo a complexidade exponencial caso tal função seja muito precisa [Stefik, 1995]. O algoritmo segue a complexidade média $O(N \exp(C\phi(N)))$, onde $\phi(N) = \log(N)^k$ [Pearl, 1984]. A Tabela 4.2 mostra um quadro comparativo entre os algoritmos A* e o algoritmo de Dijkstra.

4.3.2 Mapeando de JADE para CUDA

Todos os agentes do sistema seguem os passos do Algoritmo 3. Nele, cada agente primeiramente calcula seu caminho e espera por todos os agentes do sistema terem calculado o mesmo. Após, a execução é iniciada. A cada execução na repetição, o agente deve verificar se há algum outro agente na sua posição desejada. Caso sim, ele esperará até que a posição esteja livre. Com isso é possível tratar o problema das colisões anteriormente comentado.

Algoritmo 3 Movimentação dos Agentes no Mapa.

Requer: *Map*: $m \times n$ Mapa Bidimensional.

Requer: *Pos*: Posição do Agente.

Requer: *Checkpoint*: Posição desejada do Agente.

```

1: Path  $\leftarrow$  A*(Map, Pos, Checkpoint);
2: Esperar();
3: enquanto Path  $\neq \{\phi\} faça
4:   NewPos  $\leftarrow$  PróximaPosição(Path);
5:   se A Posição NewPos Está Vazia então
6:     MoverAgente(NewPos);
7:   senão
8:     Esperar();
9:   fim se
10: fim enquanto$ 
```

Novamente foi usado um **behaviour** em JADE para implementar esse algoritmo nos agentes. Esses últimos implementados como uma classe que herda os atributos da classe **Agent**. Em CUDA, o *kernel* novamente foi usado para controlar a execução. Cada agente foi indexado por uma thread, e foram usados dois vetores para identificar a posição inicial de cada agente.

Os problemas começam a surgir quando o algoritmo A* é implementado. Como CUDA não admite estruturas dinâmicas, foi necessário prever o número máximo de nós no pior caso da execução do algoritmo e alocar na GPU essa memória caso fosse necessário. Para execuções que demandem uma grande alocação de memória, isso pode torná-las ineficiente. Outro problema foi a função de ordenação. Em JADE foi usada o algoritmo quicksort. Porém, por sua natureza recursiva, ele não pode ser implementado em CUDA. Foi usado então ao algoritmo Bolha, bem mais ineficiente.

A sincronização no JADE é feita a partir da diretiva **synchronized**, usada nos métodos para criar uma espécie de monitor. Quando o agente quiser trocar de posição, ele o fará a partir do método **setNewPos**, e nela, só entrará um agente de cada vez. Em CUDA há algo semelhante a isso, a cada tentativa de mudança é usado a função intrínseca **_sync()**, que sincroniza as threads na execução. Em

CUDA, esse é o único modo de sincronização existente. Nos apêndices E e F é possível ver o diagrama de classes utilizadas em JADE e as assinatura das funções utilizadas em CUDA respectivamente.

4.3.3 Análise de Desempenho

Como nos casos unidimensionais, utilizamos uma máquina com CPU Intel® Core 2 Duo 2.26GHz, 4GB de RAM DDR3 e uma GPU NVidia Geforce 9400M com 256MB de memória DDR3.

Foram testadas 4 instâncias que podem ser vistas na Tabela 4.3, e cada experimento foi executado 10 vezes. **Scn** significa a identificação do cenário, **# Agentes** representa o número de agentes usados no experimento, e **# Mapa** representa a dimensão do mapa utilizado em cada execução. No apêndice G estão as tabelas de tempo das 10 execuções de cada instância em milisegundos.

Scn	# Agentes	# Mapa
1	3	6x6
2	5	10x10
3	10	20x20
4	43	40x40

Tabela 4.3: Especificação dos Casos de Teste

Em cada execução foi calculado o *speed-up* entre as implementações em JADE e CUDA, assim como seus respectivos desvios padrões. Na Figura 4.6 é mostrado a evolução da execução pelos cenários, as barras verticais no gráfico mostram o desvio padrão em cada caso de teste.

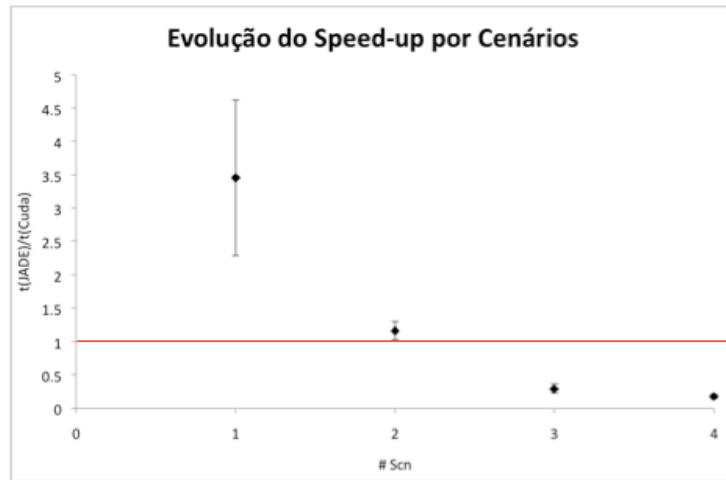


Figura 4.6: Speed-Up em relação aos cenários.

É possível notar que a implementação em JADE passa a ganhar a implementação em CUDA a partir do terceiro cenário. Tal fenômeno pode ser explicado devido à complexidade do algoritmo de *Pathfinding A** ser $O(N \exp(C\phi(N)))$ no caso médio, o que tornaria a implementação em GPU ineficiente. Outro fator relevante é a impossibilidade de criação de estruturas dinâmicas em GPU, o que levou a prever o número de estruturas máximas utilizadas, criando um uso excessivo de memória em GPU. Tais circunstâncias poderão ser melhor verificadas utilizando uma GPU mais poderosa.

Capítulo 5

Conclusões e Trabalhos Futuros

Nesse trabalho foram inicialmente descritos conceitos de Sistemas Multiagentes, e sua evolução ao longo do tempo. Essa evolução nos trouxe a um patamar de padronização do uso da tecnologia, acompanhado de um novo paradigma, o paradigma orientado a agentes (AOP), e de um padrão FIPA para esse paradigma. Paralelamente a isso, vimos que atualmente temos a possibilidade de usar arquiteturas gráficas para a programação geral, as chamadas GPGPU. Esse é um campo de pesquisa novo e ainda em exploração.

Mostramos como mapear dois estudos de caso entre os paradigmas AOP e o GPGPU, utilizando o *framework* JADE para AOP e a linguagem CUDA para GPGPU. Foram comparadas suas diferenças, similaridades, vantagens e desvantagens de usar uma ou outra. Pudemos observar uma dificuldade no JADE em tratar muitos agentes. Em CUDA, no entanto, a escalabilidade se manteve dentro do aceitável. No entanto, caso um programador de um Sistema Multiagente pretenda utilizar CUDA e FIPA, não há recursos pré-disponíveis como um *framework* para auxiliá-lo durante o desenvolvimento. Com isso, é necessário um alto conhecimento em Arquiteturas GPGPU e em estrutura de dados para abstrair os conceitos do AOP. Outro grande problema ainda existente nesse mapeamento é a comunicação entre os agentes. O FIPA-ACL impõe padrões de comunicação e negociação entre agentes que não foi testado em GPGPU. Tais comunicações podem vir a se tornar um gargalo durante as execuções na GPU devido ao número de sincronizações que devem ser feitas. É necessário ver como portar certos algoritmos, assim como feito com o A*, para GPU. Boa parte desses algoritmos precisam ser avaliados cuidadosamente antes de serem rodados em GPU devido às suas restrições.

Futuramente esse trabalho visa expor mais casos de teste com o intuito de abranger minúcias do FIPA e determinar em quais casos é melhor usar AOP, e em quais casos utilizar GPGPU. Contudo, é possível, com essas informações, criar uma camada de abstração que pode se integrar a *frameworks* já existentes, ou mesmo criar um *framework* próprio para criação de agentes em GPU's. A criação desse *framework* possibilitaria implementações mais robustas com uma menor curva de aprendizagem para o desenvolvedor. Como exemplo de aplicações podemos citar sistemas de trâfego de trânsito, simulação de multidões com uma maior inteligência, sistemas de evacuações de emergência, entre outros.

Outro trabalho interessante seria utilizar o BDI em GPU's. Como falado na Seção 2.2, esse modelo permite definir os desejos, intenções e pensamento do agente que interfere em suas ações. Com

isso teremos uma alternativa ao modelo FIPA previamente apresentado. Com ele seriam possíveis outros tipos de implementações usadas na teoria de agentes, podendo-se até criar uma comparação entre ambos os modelos, FIPA e BDI.

Por fim, acreditamos que CUDA pode ter um uso muito importante para algoritmos em um sistema de agentes. Tal estudo nos mostrou que devemos investigar e mostrar esses casos com cautela e, à partir disso, criar soluções que facilitem e abstraiam o trabalho do desenvolvedor.

Referências Bibliográficas

- [Jad,] <http://jadex.informatik.uni-hamburg.de/xwiki/>.
- [IIO, 1999] (1999). Internet inter-orb protocol specification, common object request broker architecture.
- 2.2. Object Management Group.
- [Asimov, 1950] Asimov, I. (1950). I, robot. Série dividida em 9 contos, publicados separadamente.
- [Bazzan, 2005] Bazzan, A. L. C. (2005). A distributed approach for coordination of traffic signal agents. In *Autonomous Agents and Multiagent Systems*, volume 10, pages 131–164. Springer Link.
- [Bazzan, 2010] Bazzan, A. L. C. (2010). *Atualizações em Informática 2010*, chapter 3 - IA Multiagente: Mais Inteligência, Mais Desafios. Editora Puc-Rio. Esse livro faz parte das JAI's, Jornada de Atualização em Informática oferecida pela Sociedade Brasileira de Computação.
- [Bellifemine et al., 2007] Bellifemine, F., Caire, G., and Greenwood, D. (2007). *Developing multi-agent systems with JADE*. Wiley Series in Agent Technology.
- [Bellifemine et al., 2008] Bellifemine, F., Caire, G., Poggi, A., and Rimassa, G. (2008). Jade: A software framework for developing multi-agent applications. lessons learned. *Information and Software Technology*, 50(10–21).
- [Bleiweiss, 2008] Bleiweiss, A. (2008). Gpu accelerated pathfinding. In *Proceedings of the 23rd ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware*, pages 65–74. NVidia.
- [Boissier, 1993] Boissier, O. (1993). *Problème du Contrôle dans un Système Intégré de Vision. Utilisation d'un Système Multi-Agent*. PhD thesis, Institut National Polytechnique de Grenoble, Grenoble, France.
- [Bordini et al., 2006] Bordini, R., Braubach, L., Dastani, M., Seghrouchni, A. E. F., Gomez-Sanz, J., Leite, J., OHare, G., Pokahr, A., and Ricci, A. (2006). A survey of programming languages and platforms for multi-agent systems. *Informatica*, 30(33–44).
- [Bordini et al., 2005] Bordini, R., Hübner, J., and Vieira, R. (2005). Jason and the golden fleece of agent-oriented programming. In *Multi-Agent Programming*, volume 15, pages 3–37.
- [Boulic and Renault, 1991] Boulic, R. and Renault, O. (1991). 3d hierarchies for animation. In Magnenat-Thalmann, N. and Thalmann, D., editors, *New Trends in Animation and Visualization*. John Wiley & Sons ltd.

- [Brooks, 1986] Brooks, R. A. (1986). A robust layered control system for a mobile robot. *IEEE Jornal*, 2(1):14–23.
- [Buchanan, 2002] Buchanan, B. G. (2002). Timeline of ai: A brief history of artificial intelligence. <http://www.aaai.org/AITopics/pmwiki/pmwiki.php/AITopics/BriefHistory>. A chronological list of significant events in the history of AI, prepared for the Introduction to AI class at the University of Pittsburgh and updated with links to relevant online articles.
- [Buchanan, 2005] Buchanan, B. G. (2005). A (very) brief history of artificial intelligence. In *American Association for Artificial Intelligence 25th Anniversary Issue*, pages 53–60.
- [Buck et al., 2004] Buck, I., Foley, T., Horn, D., Sugerman, J., Fatahalian, K., Houston, M., and Hanrahan, P. (2004). Brook for gpus: Stream computing on graphics hardware. In *31st International Conference on Computer Graphics and Interative Techniques — SIGGRAPH'2004*.
- [Cardozo, 1987] Cardozo, E. (1987). *DPSK: a kernel for distributed problem solving*. PhD thesis, Carnegie Mellon University, Pittsburg, PE.
- [Corkill and Lesser, 1993] Corkill, D. G. and Lesser, V. R. (1993). The distributed vehicle monitoring testbed: a tool for investigating distributed problem solving networks. In *AI Magazine*, volume 4, pages 15–33.
- [Dastani et al., 2005] Dastani, M., van Birna Riemsdijk, M., and Meyer, J.-J. (2005). Programming multi-agent systems in 3apl. In *Multi-Agent Programming*, volume 15, pages 39–67.
- [Davis, 2000] Davis, M. (2000). *Engines of Logic: Mathematicians and the origin of the Computer*. Norton & Company.
- [Dimakis et al., 2009] Dimakis, N., Filippoupolitis, A., and Gelenbe, E. (2009). Distributed building evaluation simulator for smart emergency management. *The Computer Journal. Accepted to be published*. Available at http://sa.ee.ic.ac.uk/publications/DBES_CJ.pdf.
- [Durfee and Rosenschein, 1994] Durfee, E. H. and Rosenschein, J. S. (1994). Distributed problem solving and multi-agent systems: comparisons and examples. In *13th International Workshop on Distributed Artificial Intelligence, Seattle, WA*.
- [Erman et al., 1980] Erman, L. D., Hayes-Roth, F., Lesser, V. R., and Reddy, D. R. (1980). The hearsay-ii speech-understanding system: integrating knowledge to resolve uncertainty. In *Computing Surveys*, volume 12, pages 213–253.
- [Evans, 1968] Evans, T. (1968). A program for the solution of a class of geometric-analogy intelligence-test questions. In *Semantic Information about knowledge*, pages 271–353. MIT Press, Cambridge, Massachusetts.
- [Fermi Architecture, 2010] Fermi Architecture (2010). http://www.nvidia.com/object/GTX_400_architecture.html.

- [Fikes and Nilsson, 1971] Fikes, R. E. and Nilsson, N. J. (1971). Strips: a new approach to the application of theorem proving to the problem solving. In *Artificial Intelligence*, volume 2, pages 189–208.
- [Flores-Menderz, 1999] Flores-Menderz, R. (1999). Towards a standardization of multi-agent system frameworks. *ACM Crossroads Magazine*. <http://www.acm.org/crossroads/xrds5-4/multiagent.html>.
- [Franklin and Graesser, 1996] Franklin, S. and Graesser, A. (1996). Is it an agent or just a program? a taxonomy for autonomous agents. In Muller, J., Wooldridge, M., and Jennings, N., editors, *Intelligent Agents III. Agent Theories, Architectures, and Languages. LNAI*, volume 1193, pages 21–35.
- [Frege, 1979] Frege, G. (1979). *Begriffsschrift*. Halle.
- [Garcia and Sichman, 2003] Garcia, A. C. B. and Sichman, J. S. (2003). *Sistemas Inteligentes, Fundamentos e Aplicações*, chapter Agentes e Sistemas Multiagentes. Manole.
- [Garson, 1984] Garson, J. W. (1984). Quantification in modal logic. In *Handbook of Philosophical Logic*, volume II: Extensions of Classical Logic, page 249–307. D. Reidel Publishing Company.
- [Gasser et al., 1987] Gasser, L., Braganza, C., and Herman, N. (1987). Mace: a flexible testbed for distributed ai research. In *Distributed Artificial Intelligence*, pages 119–152. London, UK, Pitman Publishing Co.
- [Gelertner, 1959] Gelertner, H. (1959). Realization of a geometry-theorem proving machine. In *Proceedings of an International Conference on Information Processing*, pages 273–282.
- [Ghouloum et al., 2007] Ghouloum, A., Smith, T., Wu, G., Zhou, X., Fang, J., Guo, P., So, B., Rajagopalan, M., Chen, Y., and Chen, B. (2007). Future-proof data parallel algorithms and software on intel multicore architecture. *Intel Technology Journal*, 11(4).
- [Group, 2009] Group, K. (2009). OpenCL overview. http://www.khronos.org/developers/library/overview/opencl_overview.pdf.
- [Hebb, 1949] Hebb, D. O. (1949). *The Organization of Behavior*. Psychology Press; New edition edition (June 15, 2002).
- [Knuth, 1984] Knuth, D. E. (1984). *The TeX Book*. Addison-Wesley, 15th edition.
- [Laird et al., 1987] Laird, J., Newell, A., and Rosenbloom, P. (1987). Soar: An architecture for general intelligence. *Artificial Intelligence*, 33(1):1–64.
- [Lange and Oshima, 1998] Lange, D. and Oshima, M. (1998). *Programming and Deploying Java Mobile Agents with Aglets*. Addison-Wesley.
- [Lenat, 1975] Lenat, D. B. (1975). Beings: knowledge as interacting experts. In *Readings in Distributed Artificial Intelligence, San Mateo, CA*, pages 161–168. Morgan Kaufmann Publishers Inc.

- [Lester, 2004] Lester, P. (2004). A* para iniciantes. http://www.policyalmanac.org/games/aStarTutorial_port.htm. Traduzido por Reynaldo N. Gayoso.
- [Malone and Crowston, 1994] Malone, T. W. and Crowston, K. (1994). The interdisciplinary study of coordination. *ACM Computing Surveys*.
- [McCorduck, 2004] McCorduck, P. (2004). *Machines Who Think: A Personal Inquiry into the History and Prospects of Artificial Intelligence*. Carnegie Mellon.
- [McCulloch and Pitts, 1943] McCulloch, W. and Pitts, W. (1943). A logical calculus of ideas immanent in nervous activity. In *Bulletin of Mathematical Biophysics*.
- [Minsky, 1975] Minsky, M. L. (1975). A framework for representing knowledge. In *The Psychology of Computer Vision*, pages 211–277.
- [Neumann, 1945] Neumann, J. V. (1945). First draft of a report on the edvac. In *Origins of Digital Computers: Selected Papers*, pages 383–392. Moore School of Electrical Engineering, Univ. of Penn., Philadelphia, Springer-Verlag, Berlin Heidelberg.
- [Newell et al., 1959] Newell, A., Simon, H., and Simon, J. C. (1959). Report on a general problem-solving program. In *Proceedings of the International Conference on Information Processing*, pages 256–264.
- [Nilson, 1971] Nilson, N. J. (1971). *Problem-solving methods in Artificial Intelligence*. McGraw-Hill.
- [NVIDIA, 2007] NVIDIA (2007). CUDA description. <http://www.nvidia.com/cuda>.
- [Passos et al., 2008] Passos, E. B., Joselli, M., Zamith, M., Rocha, J., Clua, E. W. G., Montenegro, A., Conci, A., and Feijo, B. (2008). Supermassive crowd simulation on gpu based on emergent behavior. In *SBGames 2008*.
- [Pearl, 1984] Pearl, J. (1984). *Heuristics: Intelligent Search Strategies for Computer Problem Solving*. Addison-Wesley.
- [Polit, 1984] Polit, S. (1984). R1 and beyond: Ai technology transfer at dec. *AI Magazine*, 5(4).
- [Rao and Georgeff, 1995] Rao, A. S. and Georgeff, M. P. (1995). Bdi agents: from theory to practice. In *Proceedings of the 1st International Conference on Multi-Agent Systems*, page 312–319.
- [Russel, 1906] Russel, B. (1906). The theory of implication. *American Journal of Mathematics*.
- [Russell and Norvig, 1995] Russell, S. and Norvig, P. (1995). *Artificial Intelligence, A Modern Approach*. Prentice Hall, second edition edition.
- [Sacerdoti, 1974] Sacerdoti, E. D. (1974). Planning in a hierarchy of abstraction spaces. In *Artificial Intelligente*, volume 5, pages 115–135.
- [Sacerdoti, 1977] Sacerdoti, E. D. (1977). A structure for plans and behaviours. Technical report, North Holland.

- [Samuel, 1959] Samuel, A. (1959). Some studies in machine learning using the game of checkers. *IBM Journal of Research and Development*, 3(3):210–229.
- [Shortliffe and Buchanan, 1984] Shortliffe, E. H. and Buchanan, B. G. (1984). *Rule-Based Expert Systems: The MYCIN Experiments of the Stanford Heuristic Programming Project*. Addison-Wesley.
- [Sichman, 1995] Sichman, J. S. (1995). *Du raisonnement social chez les agents: une approche fondée sur la théorie de la dépendance*. PhD thesis, Institut National Polytechnique de Grenoble, Grenoble, France.
- [Slagle, 1963] Slagle, J. (1963). A heuristic program that solves symbolic integration problems in freshman calculus. *Jornal of the Association for Computing Machinery*, 10(4).
- [Smith and Jones, 1999] Smith, A. and Jones, B. (1999). On the complexity of computing. In Smith-Jones, A. B., editor, *Advances in Computer Science*, pages 555–566. Publishing Press.
- [Smith, 1980] Smith, R. G. (1980). The contract net protocol: High level communication and control in a distributed problem solver. In *IEEE Transactions on Computers*, volume 29, pages 1104–1113.
- [Stefik, 1995] Stefik, M. (1995). *Introducing to Knowledge Systems*. Morgan Kaufmann.
- [Stone and Veloso, 1997] Stone, P. and Veloso, M. (1997). Multiagent systems: A survey from a machine learning perspective. *Autonomous Robotics*, 8:345–383.
- [Tate, 1977] Tate, A. (1977). Generating project networks. In *Proceedings of the 5th international joint conference on Artificial intelligence*, volume 2, pages 888–893.
- [Thielscher, 2005] Thielscher, M. (2005). Flux: A logic programming method for reasoning agents. *Theory and Practice of Logic Programming*, 5:533–565.
- [Turing, 1950] Turing, A. (1950). Computing machinery and intelligence. *Journal of the Mind Association*, LIX:433–460.
- [Valckenaers et al., 2006] Valckenaers, P., Sauter, J., Sierra, C., and Rodriguez-Aguilar, J. (2006). Applications and environments for Multi-Agent Systems. *Autonomous Agents and Multi-Agent Systems*, 14(1):61–85.
- [Vega and Cantú-Paz, 2010] Vega, F. F. and Cantú-Paz, E. (2010). *Parallel and Distributed Computational Intelligence*, volume 269 of *Studies in Computational Intelligence*. Springer.
- [Werbos, 1994] Werbos, P. J. (1994). *The Roots of Backpropagation: From Ordered Derivatives to Neural Networks and Political Forecasting (Adaptive and Learning Systems for Signal Processing, Communications and Control Series)*. Wiley-Interscience.
- [Weyns et al., 2005] Weyns, D., Schumacher, M., Ricci, A., Viroli, M., and Holvoet, T. (2005). Environments in multiagent systems. *The Knowledge Engineering Review*, 2(20):127–141.
- [Whitehead and Russell, 1910] Whitehead, A. N. and Russell, B. (1910). *Principia Mathematica*. Cambridge University.

[Winikoff, 2005] Winikoff, M. (2005). Jack intelligent agents: An industrial strength platform. In *Multi-Agent Programming*, page 175–193. Kluwer.

[Wolldridge, 2002] Wolldridge, M. (2002). *An Introduction to Multiagent Systems*. John Wiley and Sons.

[Wolldridge, 2009] Wolldridge, M. (2009). *An Introduction to Multiagent Systems*. John Wiley and Sons, second edition edition.

Apêndice A

Especificações do FIPA

A seguir há um conjunto de especificações de padrões FIPA. Essas, e outras não referenciadas estão disponíveis para download em <http://www.fipa.org>.

- FIPA1. Specification SC00001, FIPA Abstract Architecture Specification.
- FIPA8. Specification SC00008, FIPA SL Content Language Specification.
- FIPA14. Specification SI00014, FIPA Nomadic Application Support Specification.
- FIPA23. Specification SC00023, FIPA Agent Management Specification.
- FIPA26. Specification SC00026, FIPA Request Interaction Protocol Specification.
- FIPA27. Specification SC00027, FIPA Query Interaction Protocol Specification.
- FIPA28. Specification SC00028, FIPA Request When Interaction Protocol Specification.
- FIPA29. Specification SC00029, FIPA Contract Net Interaction Protocol Specification.
- FIPA30. Specification SC00030, FIPA Iterated Contract Net Interaction Protocol Specification.
- FIPA33. Specification SC00033, FIPA Brokering Interaction Protocol Specification.
- FIPA34. Specification SC00034, FIPA Recruiting Interaction Protocol Specification.
- FIPA35. Specification SC00035, FIPA Subscribe Interaction Protocol Specification.
- FIPA36. Specification SC00036, FIPA Propose Interaction Protocol Specification.
- FIPA37. Specification SC00037, FIPA Communicative Act Library Specification.
- FIPA61. Specification SC00061, FIPA ACL Message Structure Specification.
- FIPA67. Specification SC00067, FIPA Agent Message Transport Service Specification.
- FIPA69. Specification SC00069, FIPA ACL Message Representation in Bit-Efficient Specification.
- FIPA70. Specification SC00070, FIPA ACL Message Representation in String Specification.
- FIPA71. Specification SC00071, FIPA ACL Message Representation in XML Specification.
- FIPA75. Specification SC00075, FIPA Agent Message Transport Protocol for IIOP Specification.
- FIPA84. Specification SC00084, FIPA Agent Message Transport Protocol for HTTP Specification.
- FIPA85. Specification SC00085, FIPA Agent Message Transport Envelope Representation in XML Specification.
- FIPA88. Specification SC00088, FIPA Agent Message Transport Envelope Representation in Bit Efficient Specification.

FIPA91. Specification SI00091, FIPA Device Ontology Specification.

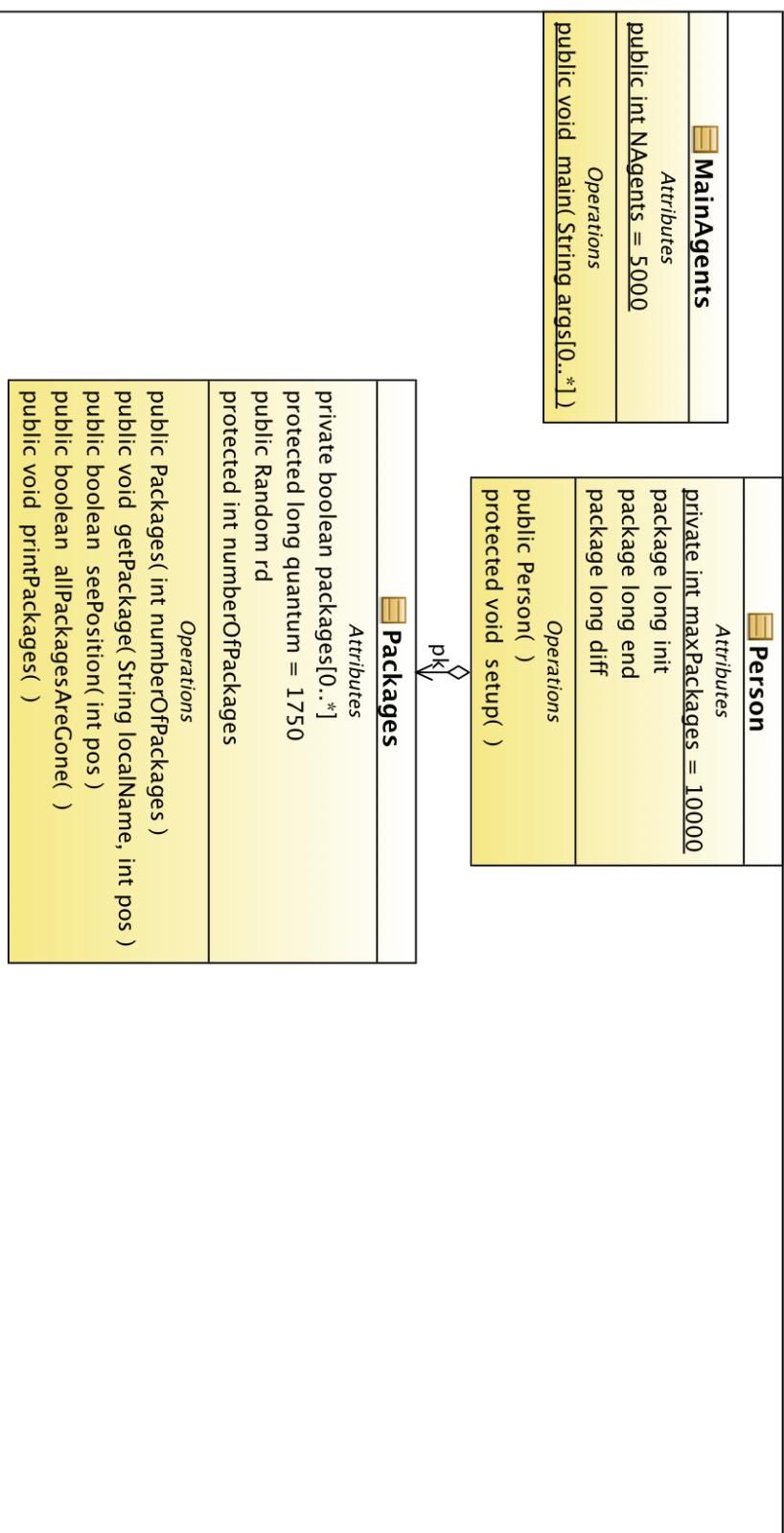
FIPA94. Specification SC00094, FIPA Quality of Service Specification.

FIPAHST. História do FIPA. Disponível online via <http://www.fipa.org/subgroups/ROFS-SG.html>.

Apêndice B

Diagrama de Classes do Caso Unidimensional

Class Diagram 2



Apêndice C

Assinatura das funções do Caso Unidimensional

```
1 //Cuda Variables
2 void** d_agents;
3 void** d_packages;
4
5 //Cuda Functions
6 int InitCUDA();
7 __global__ void kernel(void** agents, int num_agents, void** packages,
8                      int num_packs);
```

```
1 //Host Variables
2 int* agents;
3 int* packages;
4
5 //Host Functions
6 void printVet(int* vet, int tam);
```

Apêndice D

Tempos do Caso Unidimensional

1 - Número de agentes: 250, Número de Caixotes: 500

Execução	Tempo JADE	Tempo CUDA
1	171	68
2	42	66
3	205	66
4	139	64
5	201	63
6	123	62
7	74	64
8	116	63
9	73	64
10	137	63
Média	128,1	64,3
Desvio Padrão	54,58	1,83

2 - Número de agentes: 500, Número de Caixotes: 500

Execução	Tempo JADE	Tempo CUDA
1	146	71
2	121	66
3	105	68
4	98	65
5	124	61
6	90	64
7	133	65
8	124	66
9	124	64
10	128	64
Média	119,3	65,4
Desvio Padrão	16,87	2,67

3 - Número de agentes: 250, Número de Caixotes: 1000

Execução	Tempo JADE	Tempo CUDA
1	208	72
2	159	66
3	256	68
4	235	65
5	250	69
6	194	67
7	175	66
8	200	67
9	197	65
10	194	69
Média	206,8	67,4
Desvio Padrão	31,36	2,17

4 - Número de agentes: 500, Número de Caixotes: 1000

Execução	Tempo JADE	Tempo CUDA
1	198	72
2	204	66
3	159	67
4	227	66
5	189	66
6	215	65
7	194	64
8	209	65
9	165	66
10	199	64
Média	195,9	66,1
Desvio Padrão	20,95	2,28

5 - Número de agentes: 1000, Número de Caixotes: 1000

Execução	Tempo JADE	Tempo CUDA
1	166	162
2	216	167
3	149	158
4	279	161
5	189	161
6	136	160
7	168	162
8	211	159
9	201	161
10	179	162
Média	189,4	161,3
Desvio Padrão	40,74	2,41

6 - Número de agentes: 250, Número de Caixotes: 2000

Execução	Tempo JADE	Tempo CUDA
1	279	71
2	325	69
3	220	68
4	476	65
5	245	63
6	245	64
7	295	65
8	337	66
9	293	67
10	430	66
Média	314,5	66,4
Desvio Padrão	82,15	2,41

7 - Número de agentes: 500, Número de Caixotes: 2000

Execução	Tempo JADE	Tempo CUDA
1	538	70
2	343	69
3	286	75
4	253	65
5	244	66
6	276	65
7	303	66
8	299	64
9	292	61
10	263	66
Média	309,7	66,7
Desvio Padrão	85,06	3,83

8 - Número de agentes: 1000, Número de Caixotes: 2000

Execução	Tempo JADE	Tempo CUDA
1	254	161
2	279	164
3	347	160
4	310	159
5	267	160
6	352	161
7	217	159
8	288	162
9	206	160
10	261	160
Média	278,1	160,6
Desvio Padrão	48,59	1,51

9 - Número de agentes: 1500, Número de Caixotes: 2000

Execução	Tempo JADE	Tempo CUDA
1	282	164
2	376	160
3	243	161
4	261	163
5	245	162
6	280	162
7	251	160
8	249	159
9	253	161
10	247	159
Média	268,7	161,1
Desvio Padrão	40,16	1,66

10 - Número de agentes: 2000, Número de Caixotes: 2000

Execução	Tempo JADE	Tempo CUDA
1	365	158
2	216	162
3	311	159
4	181	162
5	231	163
6	298	160
7	296	161
8	181	162
9	297	162
10	236	161
Média	261,2	161
Desvio Padrão	61,06	1,56

11 - Número de agentes: 250, Número de Caixotes: 5000

Execução	Tempo JADE	Tempo CUDA
1	457	78
2	666	70
3	647	77
4	377	65
5	352	66
6	441	66
7	430	65
8	1047	66
9	588	65
10	329	66
Média	533,4	68,4
Desvio Padrão	216,4	5,02

12 - Número de agentes: 500, Número de Caixotes: 5000

Execução	Tempo JADE	Tempo CUDA
1	522	72
2	456	75
3	433	69
4	489	66
5	487	68
6	421	77
7	562	65
8	504	66
9	337	64
10	309	66
Média	452	68,8
Desvio Padrão	79,8	4,44

13 - Número de agentes: 1000, Número de Caixotes: 5000

Execução	Tempo JADE	Tempo CUDA
1	422	161
2	392	169
3	307	163
4	400	159
5	343	161
6	413	168
7	267	160
8	294	162
9	356	161
10	424	159
Média	361,8	162,3
Desvio Padrão	57,21	3,5

14 - Número de agentes: 1500, Número de Caixotes: 5000

Execução	Tempo JADE	Tempo CUDA
1	429	160
2	465	165
3	365	177
4	424	167
5	342	165
6	440	166
7	414	169
8	284	161
9	510	159
10	492	170
Média	416,5	165,9
Desvio Padrão	69,38	5,36

15 - Número de agentes: 2000, Número de Caixotes: 5000

Execução	Tempo JADE	Tempo CUDA
1	392	160
2	453	160
3	430	158
4	385	161
5	351	160
6	971	162
7	238	161
8	684	159
9	873	160
10	373	161
Média	515	160,2
Desvio Padrão	243,02	1,14

16 - Número de agentes: 5000, Número de Caixotes: 5000

Execução	Tempo JADE	Tempo CUDA
1	0	159
2	0	160
3	0	158
4	0	163
5	0	161
6	0	158
7	0	169
8	0	160
9	0	159
10	0	158
Média	0	160,5
Desvio Padrão	0	3,37

17 - Número de agentes: 250, Número de Caixotes: 10000

Execução	Tempo JADE	Tempo CUDA
1	802	76
2	1026	75
3	613	74
4	718	70
5	770	71
6	872	77
7	927	70
8	1069	76
9	728	71
10	769	75
Média	829,4	73,5
Desvio Padrão	143,31	2,72

18 - Número de agentes: 500, Número de Caixotes: 10000

Execução	Tempo JADE	Tempo CUDA
1	557	73
2	866	72
3	677	78
4	757	65
5	770	66
6	921	64
7	817	64
8	911	65
9	679	66
10	994	65
Média	794,9	67,8
Desvio Padrão	133,56	4,8

19 - Número de agentes: 1000, Número de Caixotes: 10000

Execução	Tempo JADE	Tempo CUDA
1	1072	173
2	736	161
3	1170	163
4	1056	167
5	660	161
6	904	161
7	760	165
8	801	160
9	758	161
10	924	159
Média	884,1	163,1
Desvio Padrão	169,62	4,23

20 - Número de agentes: 1500, Número de Caixotes: 10000

Execução	Tempo JADE	Tempo CUDA
1	937	161
2	908	160
3	815	161
4	606	159
5	763	161
6	660	161
7	949	159
8	617	163
9	656	160
10	608	161
Média	751,9	160,6
Desvio Padrão	141,19	1,17

21 - Número de agentes: 2000, Número de Caixotes: 10000

Execução	Tempo JADE	Tempo CUDA
1	969	159
2	734	185
3	857	159
4	1283	160
5	815	160
6	797	162
7	727	159
8	505	160
9	551	162
10	606	164
Média	784,4	163
Desvio Padrão	225,89	7,9

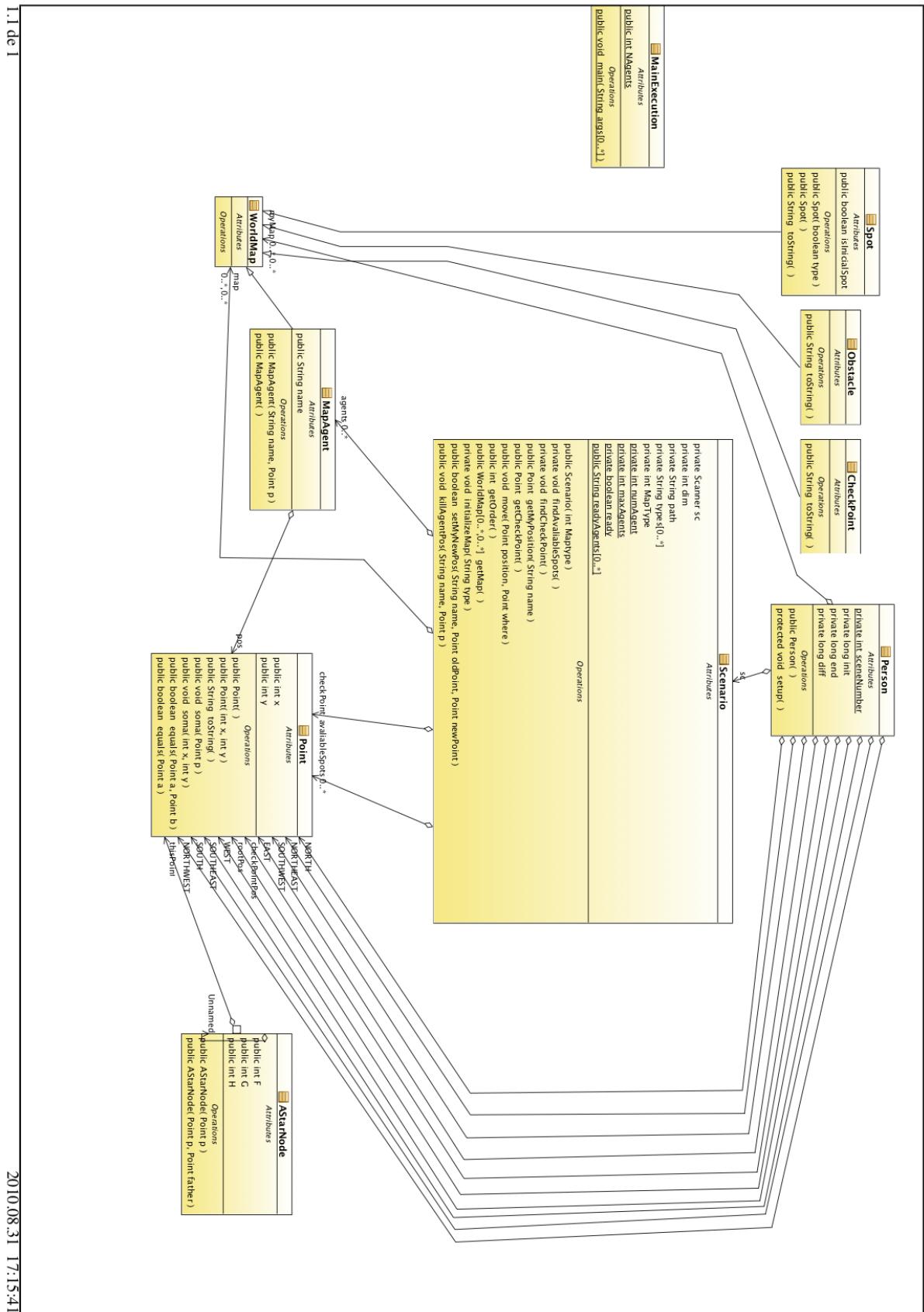
22 - Número de agentes: 5000, Número de Caixotes: 10000

Execução	Tempo JADE	Tempo CUDA
1	0	161
2	0	162
3	0	160
4	0	162
5	0	163
6	0	159
7	0	166
8	0	162
9	0	163
10	0	166
Média	0	162,4
Desvio Padrão	0	2,27

Apêndice E

Diagrama de Classes do Caso Bidimensional

Caso Bidimensional



1.ide1

2010.08.31 17:15:41

Apêndice F

Assinatura das funções do Caso Bidimensional

```
1 //Device Structs
2
3
4
5 __device__ struct AStarNode{
6     int2 p;
7     int2 father;
8     int F;
9     int G;
10    int H;
11 };
12
13 __device__ struct passByReference{
14     struct AStarNode* list;
15     int begin;
16     int end;
17 };
```

```
1 //Device Varaibles
2
3
4 __device__ int* d_agentsX;
5 __device__ int* d_agentsY;
6 __device__ int* d_map;
7 __device__ int* d_checkPoint;
8 __device__ int* d_dim;
9 __device__ int* d_numAgents;
10 __device__ int2* d_walkThru;
11 __device__ int* d_tamWalk;
12 __device__ AStarNode* d_closedList;
13 __device__ AStarNode* d_openList;
14 __device__ int* d_closedListIndex;
15 __device__ int* d_openListBegin;
16 __device__ int* d_openListEnd;
```

```

1 //Host Structs
2
3
4 struct point{
5     int x;
6     int y;
7 }typedef Point;
8
9
10 struct list{
11     Point *p;
12     struct list *prox;
13 }typedef List;

```

```

1 //Cuda Functions
2
3 int InitCUDA();
4
5 --global-- void kernel(int exec, int* agentsX, int* agentsY, int* nAgents, int* map,
6     int* dim, int* checkPoint, int2* walkThru, int* tamWalk,
7     struct AStarNode* closedList, struct AStarNode* openList,
8     int* closedListIndex, int* openListBegin, int* openListEnd);
9
10 --device-- struct passByReference addAdjacents(struct AStarNode root, int index,
11     int* map, int dim, int jump, struct AStarNode* openList, int begin, int end,
12     int2 endPoint);
13
14 --device-- void initConstants();
15
16 --device-- struct passByReference inTheList(struct AStarNode a, int index, int jump,
17     struct AStarNode* openList, int begin, int end);
18
19 --device-- int calculateH(int2 P, int2 To);
20
21 --device-- struct AStarNode* sort(struct AStarNode* list, int index, int jump,
22     int begin, int end);
23
24 --device-- int pointInside(struct AStarNode* list, int tam, int2 p, int index,
25     int jump);

```

```

1 //Host Functions
2
3 //Copy Functions
4 void copyAgentVectors(Point **agents);
5 void copyMapMatrix(int **map, int dim);
6 void copycheckPoint(Point* checkPoint);
7 void copyVariables(int dim, int numAgents);
8 void copyWalkThru(int dim, int numAgents);
9
10 //Get Funcions
11 Point **getAgentVectors(Point **agents);
12 int **getMapMatrix(int** sceneMap, int dim);
13 void getWalkThru(int dim, int numAgents);
14
15 //Free functions
16 void freeAgentVectors(Point **agents);
17 void freeMapMatrix(int **map, int dim);
18 void freecheckPoint(Point* checkPoint);
19 void freeVariables();
20 void freeWalkThru();
21
22 //Matrix Stuff
23 int **getMap(char fileName[], int *dim, Point** checkPoint, List** aS);
24
25 //List Stuff
26 List* initList();
27 List* insertList(List *l, Point *p);
28 List* removeList(List *l, Point *p);
29 int lengthList(List *l);
30 void freeList(List *l);
31
32 //Agent Stuff
33 Point** initAgents(int maxAgents, int numberOfAgents ,List* aS);
34
35 //Auxiliar Functions
36 void printMat(int **mat, int dim);
37 void printAgents(Point** agents);

```

Apêndice G

Tempos do Caso Bidimensional

1 - Mapa: 6x6, Número de Agentes: 3

Execução	Tempo JADE	Tempo CUDA
1	23	6,42
2	9	6,53
3	30	6,55
4	15	6,45
5	28	6,59
6	11	6,57
7	32	6,51
8	26	6,44
9	25	6,56
10	26	6,43
Média	22,5	6,51
Desvio Padrão	8,02	0,06

2 - Mapa: 10x10, Número de Agentes: 5

Execução	Tempo JADE	Tempo CUDA
1	36	32,5
2	40	32,34
3	40	32,47
4	44	32,52
5	32	32,46
6	39	32,51
7	45	32,48
8	32	32,51
9	33	32,46
10	38	32,45
Média	37,9	32,47
Desvio Padrão	4,65	0,05

3 - Mapa: 20x20, Número de Agentes: 10

Execução	Tempo JADE	Tempo CUDA
1	121	303,16
2	61	303,12
3	61	300,81
4	105	302,64
5	62	302,24
6	91	301,29
7	97	303,11
8	72	301,74
9	90	300,86
10	109	302,82
Média	86,9	302,18
Desvio Padrão	21,83	0,94

4 - Mapa: 40x40, Número de Agentes: 43

Execução	Tempo JADE	Tempo CUDA
1	209	997,69
2	164	993,36
3	152	997,74
4	171	997,42
5	185	996,07
6	155	976,3
7	236	996,12
8	154	994,68
9	148	997,83
10	221	998,93
Média	179,5	994,61
Desvio Padrão	31,84	6,65