UNIVERSIDADE FEDERAL FLUMINENSE

LUIZ GUILHERME OLIVEIRA DOS SANTOS

UMA ARQUITETURA PARALELA BASEADA EM GPUS PARA JOGOS E SIMULAÇOES EM TEMPO REAL PARA SISTEMAS MULTIAGENTES

NITERÓI

2013

UNIVERSIDADE FEDERAL FLUMINENSE

LUIZ GUILHERME OLIVEIRA DOS SANTOS

UMA ARQUITETURA PARALELA BASEADA EM GPUS PARA JOGOS E SIMULAÇOES EM TEMPO REAL PARA SISTEMAS MULTIAGENTES

Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Computação da Universidade Federal Fluminense como requisito parcial para a obtenção do Grau de Mestre em Computação. Área de concentração: Computação Visual

Orientador:
ESTEBAN WALTER GONZALES CLUA

Co-orientador:
DANIELA GORSKI TREVISAN

NITERÓI

2013

LUIZ GUILHERME OLIVEIRA DOS SANTOS

UMA ARQUITETURA PARALELA PARA SISTEMAS MULTIAGENTES BASEADA EM GPUS PARA JOGOS E SIMULAÇOES EM TEMPO REAL

Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Computação da Universidade Federal Fluminense como requisito parcial para a obtenção do Grau de Mestre em Computação. Área de concentração: Computação Visual

Aprovada em Junho de 2013.

BANCA EXAMINADORA

Prof. Esteban Walter Gonzales Clua - Orientador, UFF

Prof. Daniela Gorski Trevisan - Co-Orientador, UFF

Karin Breitman, EMC

Prof. José Viterbo Filho, UFF

Prof. Flávia Cristina Bernardini, UFF

Niterói

2013

Para minhas avós, Iridete Maria de Oliveira e Maria da Visitação Borges dos Santos, que faleceram recentemente. Que ambas descansem em paz.

"In life, unlike chess, the game continues after checkmate"

Isaac Asimov

Agradecimentos

Aos meus pais, Miguel Borges e Lanúzia Oliveira, e ao meu irmão Marcos Vinicius pelo apoio incondicional durante a realização do meu mestrado, sempre apoiando financeiramente e moralmente, tornando-se indispensáveis em mais essa etapa da minha vida.

À o meu orientador Esteban Clua, que acreditou no meu potencial e me deu a oportunidade para realizar este trabalho.

À Flávia Bernardini pelas incontáveis ajudas, discussões, revisões neste trabalho e por estar sempre me ajudando desde meu tempo na graduação.

À Ubiratam de Paula, Rafaelli Coutinho, Carlos Heitor, Magno Mathias, Leandro Oliveira, Flávio Vinicius, Welton Barbosa, João Bentes e Jean Zahn, que foram meus companheiros de moradia durante esses últimos 2 anos, tornando-a muito mais agradável.

Aos amigos do projeto da marinha do brasil: Giancarlo Taveira, Christian Ruff, Diego Barboza, William Jefferson e Pitágoras Alcântara pelos bons momentos durante e após minha estadia neste projeto.

Aos amigos do MediaLab , por fazer com que o ambiente possa sempre ficar mais agradável e divertido.

Aos colegas de trabalho da Nano Games, Schlumberger e EMC que me acompanharam profissionalmente durante o período do Mestrado.

À Daniele Crespo, por me apoiar e me instigar durante este período.

À CAPES pela ajuda financeira, sem o qual esse trabalho não poderia ser realizado.

A todos os demais que, de alguma forma, contribuíram para a realização desse trabalho.

Resumo

Muitos jogos e simulações fazem uso de agentes inteligentes para melhorar suas possibilidades de design e criar situações diferentes e não usuais para jogadores. Essa natureza dinâmica e o uso comum dos agentes e do paradigma orientado a agentes hoje em dia motiva a investigação de Sistemas Multi Agentes (MAS) e sua padronização. O conceito principal de um MAS é quebrar um problema em pequenos subproblemas, cada um com seus interesses e objetivos, sendo FIPA uma das coleções de padrões mais utilizados hoje em dia. Porém, quando lidamos com um número grande de agentes para aplicações de tempo real, como multidões em jogos e soluções de realidade virtual, é difícil atingir um número satisfatório de agentes devido as restrições computacionais impostas pela CPU. Com o advento das arquiteturas paralelas de GPU, e a possibilidade de criar programas genéricos que possam ser executados nelas, tornou-se possível modelar essas aplicações massivas. Neste trabalho é proposto um novo modelo de padronização de aplicações baseados em agentes baseado no FIPA, utilizando arquiteturas de GPU, possibilitando a modelagem de simulações com grandes quantidades de entidades envolvidas. É apresentado também diferentes abordagens entre CPU, GPU e GPU com kerneis concorrentes para agentes heterogêneos, revelando suas vantagens, desvantagens e restrições de uso. Os resultados obtidos nessas simulações são bastante promissores e mostram que a GPU pode ser uma escolha viável para aplicações baseadas em agentes com natureza massiva.

Palavras-chave: GPU Computing, FIPA, Sistemas Multi Agente, Programação Paralela

Abstract

Many video games and simulations make use of agents to improve their design possibilities and create different and unusual situations for players. This dynamic nature and common use of agents and agent paradigm motivates the investigation of Multi-Agent Systems (MAS) and its standardization. The main concept of a MAS is to break a problem into sub problems related to constraint satisfaction issues. Each subproblem is subcontracted to different problem solving agents with their own interests and goals, being FIPA one of the most commonly collection of standards used nowadays. Although, when dealing with a huge set of agents for real time applications, such as swarms in games and virtual reality solutions, it is hard to compute a massive crowd of agents due to the computational restrictions in CPU. With the advent of parallel GPU architectures and the possibility to run general algorithms inside it, it became possible to model such massive applications. In this work a novel standardization of agent applications based on FIPA using GPU architectures is proposed, making possible the modelling of higher crowd behaviours. There is also a comparison in different approaches between CPU, GPU and Concurrent Kernel GPU for heterogeneous agents, shows its restrictions, advantages and disadvantages. The obtained results in these simulations were very promising and show that GPUs may be a choice for massively agent-based applications.

Keywords: GPU Computing, FIPA, Multi Agent Systems, Parallel Programming

Lista de Figuras

Lista de Tabelas

Lista de Abreviaturas e Siglas

IA	:	Inteligência Artificial;
AOP	:	*Agent-Oriented Programming*;
CUDA	:	*Compute Unified Device Architecture*;
GPGPU	:	*General Purpose Programming Using a Graphics Processing Unit*;
CPU	:	*Central Processing Unit*;
GPU	:	*Graphics Processing Unit*;
FIPA	:	*Foundation For Intelligent Physical Agents*;
JADE	:	*Java Agent DEvelopment Framework*;
MAS	:	*Multi Agent System*;
SIMD	:	*Single Instruction Multiple Data*;
SM	:	*Stream Multiprocessor*;
GB	:	*Giga Byte*;
DDR3	:	*Double Data Rate Type 3*;
GDDR	:	*Graphics Double Data Rate*;

Sumário

Capítulo 1

Introdução

1.1 Definição do Tema

Até a década de 90, o campo da Inteligência Artificial focou-se em problemas voltados para a inteligência individual, como Redes Neurais e Sistemas Especialistas [21]. 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.

No Campo da Computação, chamamos essa forma de modelar e executar um comportamento individual de Agente Inteligente[1]. Para [46], "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 [10], "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.". Outra definição geral, considerada uma das mais comuns e completa é a de [33]:

> *"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*

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

> *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."*

Em 1987, foi ratificada uma arquitetura completa para o agente [16]. 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.

Um sistema multi agente (em inglês *multi agent system* ou MAS) possui uma grande flexibilidade na modelagem de problemas reais, permitindo que sejam modeladas como subdivisões de um problema de satisfação de uma ou mais restrições em diferentes, e individuais, especificações de agentes, a partir de seus interesses e objetivos. Com isso, múltiplos agentes se tornaram fundamentais para o desenvolvimento de ótimos comportamentos de inteligência artificial em games e simuladores, como o de multidões.

O objetivo principal desse trabalho é criar uma arquitetura de software que possa prover ao programador um arcabouço simples para a criação massiva de agentes dentro de um sistema multi agente. Pensando em termos de escalablidade, e por ser uma tecnologia relativamente nova e promissora, utilizamos como hardware as GPUs. Outra grande preocupação foi a de criar uma arquitetura que fosse parecida com as utilizadas atualmente, para que essa transição seja facilitada. Foi escolhido o FIPA como ponto de apoio desse nosso arcabouço devido a sua difusão não somente por membros acadêmicos, mas também por grandes empresas.

1.2 Motivação

A natureza dinâmica e distribuída de um agente motiva a criação de padrões para melhorar a colaboração entre grupos de pesquisa e membros da indústria que trabalham com MAS. Mendez [9] descreve cada modelo de um MAS proposto por esses grupos, e conclui que "Os modelos da arquitetura de ambientes abertos é composto por áreas logicamente distribuídas onde o agente existe. Os agentes básicos nesta arquitetura são mínimos, sendo alguns exemplos os coordenadores locais de área, servidores de endereço, e servidores de

domínios compartilhados"[2]. Um deses modelos, utilizado nesse trabalho, é a *Foundation for Intelligent Physical Agents*, conhecido como FIPA.

O principal conceito de um MAS é simular o mundo real e suas interações, especialmente quando composto por muitas entidades, *e.g.* um prédio com muitas pessoas durante uma evacuação de emergência, uma comunidade de animais como abelhas, iterações biológicas entre células e enzimas, e assim por diante. Em uma aplicação como jogos ou simulações, a criação de muitos indivíduos com diferentes comportamentos e/ou objetivos se difundiu bastante. Existem vários trabalhos na literatura que exploram essa dinamicidade [31, 22, 34, 28], mas quando se trata de aplicações interativas como jogos, existem muitas restrições computacionais a serem cuidadosamente analisadas, já que esse tipo de processamento pode ser dispendioso. Alguns trabalhos na literatura exploram as limitações de hardware para criar ambientes com grande concentração de agentes [30, 42]. Outros exploram problemas relacionados a simulação, como colisões [11], *Path-Planning* [47], alta diversidade de comportamentos [37], e assim por diante. [17] explora o uso de diferentes modelos de *hardware* para melhorar seus resultados e criar o maior número de agentes possíveis.

1.3 Contribuições Alcançadas

O objetivo inicial desse trabalho era criar a possibilidade de um *framework* para a criação de agentes em GPU. O trabalho teve início com uma ideia base já existente na indústria e o desenvolveu para suportar a criação de agentes dentro da GPU. Com isso um paradigma já existente e bem difundido foi estendido possibilitando uma gama de novas possibilidades de aplicações que utilizem agentes massivos.

Outra grande contribuição é o mapeamento de restrições na criação dos agentes em GPU que será discutido ao longo de toda essa dissertação, mostrando casos em que o seu uso traz benefícios significativos, e casos a serem evitados.

Este trabalho já obteve um retorno bastante positivo da comunidade científica por ser considerado um novo campo de pesquisa, e com isso obtivemos alguns trabalhos já publicados[5, 6, 7] em diversas conferências na área de Entretenimento Digital.

[2]Tradução livre do trecho: "The architecture models open environments composed of logically distributed areas where agents exist. The basic agents in this architecture are minimal agents, such as local area coordinators, yellow page servers, and cooperation domain servers"

1.4 Organização da Dissertação

Este trabalho é organizado da seguinte maneira:

- O capítulo 2 descreve o FIPA, um dos padrões mais utilizados para a programação de sistemas multi agente, e dá detalhes sobre o seu funcionamento e uso nos dias de hoje.
- O capítulo 3 fala sobre *GPU Computing*, falando de forma geral como ela funciona e seu uso no campo da Inteligência Artificial.
- O capítulo 4 fornece detalhes de como funciona a arquitetura proposta por este trabalho, como que é a comunicação entre CPU e GPU, e suas vantagens e desvantagens.
- No capítulo 5 é mostrado dois casos de teste para validação dessa arquitetura. Em ambos os casos é mostrado como foi implementado, qual foi a metodologia de análise e os resultados do mesmo.
- O capítulo 6 contem a conclusão desse trabalho, assim como sugestões para trabalhos futuros.

Capítulo 2

Agentes Inteligentes

2.1 Definição

A teoria de agentes é considerada uma das tecnologias mais emergentes e inovadora para o desenvolvimento de sistemas distribuídos. Apesar de ainda não ser usada em larga escala, muitos trabalhos estão desenvolvidos com base nesta teoria, e produtos estão migrando da academia para a indústria, como o FIPA [8].

Agentes podem se basear em duas diferentes arquiteturas: lógica e reativa [9]. A primeira é baseada em sistemas de conhecimento, onde o programador deve representar um ambiente complexo e criar regras para manipular o agente de acordo com mecanismos de interpretação lógica. O segundo geralmente é baseado em comportamentos de escolha de decisão. Diferentemente do método baseado em lógica, o reativo não necessita de um sistema de interpretação lógica, mas apenas de um modelo de comunicação com os dados do ambiente, com o objetivo de receber informações relevantes a sua execução, e atuar neste ambiente de acordo com o processamento desses dados.

A definição formal de uma arquitetura bem estruturada para o desenvolvimento é requisito fundamental para a criação de agentes e sistemas multi agentes, facilitando o trabalho de um programador e provendo uma série de serviços que podem ser usados durante uma execução. Esse esforço se limita não somente a parte de implementação, mas a estruturação de como o agente funciona. De fato, os esforços iniciais no campo da Teoria dos Agentes focou basicamente na criação de arquiteturas de execução dos agentes[3, 29]. Atualmente existem basicamente dois tipos de agentes:

Agentes Cognitivos : Toda sua arquitetura é baseada no tradicional sistema de representação e interpretação de conhecimento lógico, que é simbolicamente representado

e com sua manipulação ele pode atingir seus "desejos" e "objetivos". As vantagens desse modelo é que ele pode ser facilmente modelado em fórmulas lógicas que se assemelham mais ao pensamento humano. Suas desvantagens são que esse modelo não é sempre preciso, e sua computação pode requerer bastante tempo computacional dependendo do número de regras criadas. Um representação desse modelo é ilustrada na figura 2.1.

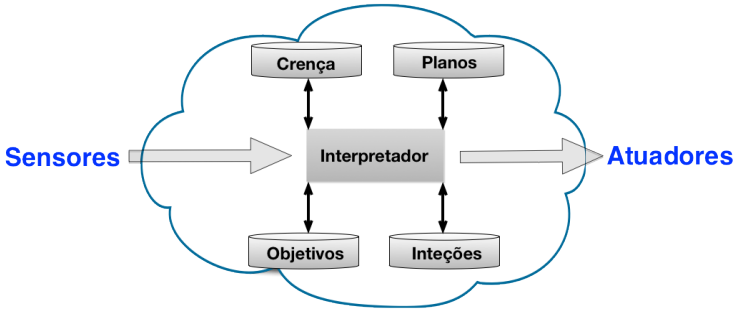

Figura 2.1: Representação de um Agente Cognitivo.

Agentes Reativos : São baseadas nos agentes que fazem tomadas de decisão baseados em dados a partir de um *input* sensorial. Diferentemente dos Agentes Cognitivos, não há necessidade de um de um modelo simbólico de entrada, e assim, não é necessário utilizar nenhum sistema de interpretação lógica. Brooks [3] descreve que um "comportamento inteligente" pode ser criado sem a necessidade de uma representação explícita ou pensamento abstrado provido pela simbologia que geralmente é utilizada em IA. A arquitetura supracitada define camadas de máquinas de estados que estão diretamente conectadas ao "sensores" do agente. Assim o sistema de tomada de decisão é feito através dos objetivos que são diretamente mapeados no agente. A decisão pode ou não alterar o estado atual do agente, levando ele em outras iterações a atuar no ambiente de forma diferente. A figura 2.1 ilustra esse modelo, mostrando os estados e suas decisões.

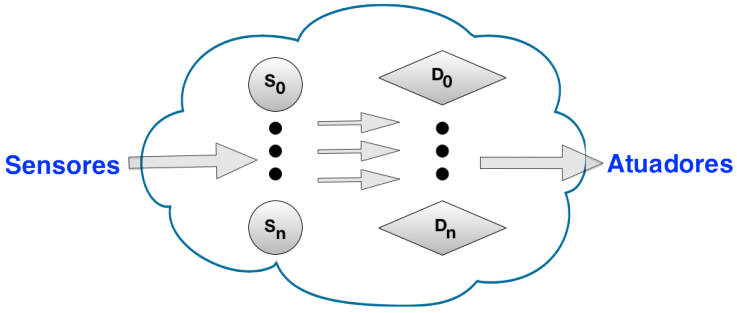

Figura 2.2: Representação de um Agente Reativo. Nele podemos ver vários estados S_0 a S_n e várias decisões D_0 a D_n.

Neste trabalho, o agente utilizado é o agente reativo. Dentro de qualquer arquitetura de agente, o mesmo é considerado totalmente autônomo, pois ele deve agir sem nenhuma intervenção sobre seus atos, e estados internos. Ele pode ou não interagir com outros agentes do meio, essa cooperação pode levar a soluções fiéis de problemas complexos. Sua reatividade deve responder diretamente a eventos ocorridos no seu meio, e não deve ser influenciada por outros agentes, ao menos que esses também influenciem o meio. Por fim ele é considerado pró-ativo, pois tem a habilidade de exibir seus objetivos e fazer com que eles sejam alcançados.

Um dos requisitos essenciais para a criação de agentes são as ferramentas e tecnologias utilizadas. Os agentes podem ser criados em qualquer tipo de linguagem, arquitetura de hardware e dependem bastante da disponibilidade de poder computacional atual, bibliotecas de software entre outros. Em sua maioria, são utilizadas linguagens orientadas a objeto para a programação de agentes devido à sua semelhança com um agente, suas características, encapsulamento e herança.

2.2 FIPA

2.2.1 A Organização

FIPA é um acrônimo para *Foundation For Intelligent Physical Agents* [8] e é uma organização não lucrativa, que desenvolve padrões para a criação de aplicações baseadas em agentes. Essa organização, fundada em 1996, é composta por membros tanto acadêmicos quanto da indústria desde sua criação. Essas soluções são amplamente utilizadas tanto academicamente quanto em soluções comerciais. Outros padrões como o MASIF(*Mobile Agent System Interoperability Facility*)[13], são usados com fins mais específicos, e não são tão genéricos quanto o FIPA. A organização se baseia nos seguintes princípios:

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 para que posteriormente haja uma interoperabilidade aberta para a linguagem e sua infra estrutura.

Durante a evolução do FIPA, dois conceitos principais foram desenvolvidos: o FIPA-ACL (para propósitos de comunicação) e o *Agent Management Framework*. FIPA-ACL é o padrão de comunicação entre agentes baseado nos padrões da internet dos anos 90, tai como OMG, DCE, W3C e GGF. O *Agent Management Framework* foca em como criar, operar e manipular os agentes. Ele define a criação, registro, localização, comunicação e processo operacional dos agentes. Neste trabalho, usaremos agentes como entidades que podem ser facilmente paralelizáveis e utilizados em problemas comuns de IA aplicado a jogos e simulação.

O FIPA determina que deve haver uma camada abstrata onde todos os serviços dos agentes devem ser providos, assim, o programador pode desenvolver sua aplicação no topo dessa camada. Por outro lado, todos os agentes devem ser autônomos, agindo como uma aplicação *peer-to-peer* e deve haver um controlador, chamado de Container. O Container possui um Agent Description dos agentes e tem a permissão de iniciar os agentes e controlar o ambiente do qual o agente se situa. Esse paradigma é chamado de *Agent Oriented Programming* — AOP. Existem vários arcabouços, plataformas e aplicações baseadas no paradigma AOP, como o JADE [1], FLUX [38] e o JACK [45]. Seu uso hoje em dia são os mais variados possível, como "robôs" que coletam informações na web, serviços orientado a mídias[12] , sistemas de evacuação com um número massivo de pessoas [41], e assim por diante. As principais normas do FIPA estão listadas no apêndice A.

2.2.2 Controle dos Agentes

Cada agente recebe um número de identificação único (AID ou *Agent Identifier*), que o rotula e é usado em todos os serviços providos. Além disso cada agente possui pelo menos um controlador, que não o controla diretamente, apenas tem a função de iniciar e interromper sua execução. Outro protocolos podem requerer um registro a mais além do AID, como por exemplo o protocolo de comunicação que é feito pelo FIPA-ACL.

Juntamente com o as definições de comunicação, o controle de agentes é peça fundamental no padrão FIPA. Este especifica um modelo de referência para a criação, registro, localização, comunicação, migração e operação dos agentes. Isso é feito através de vários serviços que podem ser utilizados por todos os agentes durante sua execução. Dos serviços

providos podemos citar a *Plataforma de Agentes*, o *Facilitador de Diretórios* e o *Sistema de Controle de Agentes*.

A Plataforma de Agentes consiste em componentes de controle, ferramental básico, e dos próprios agentes, e de qualquer outro software de suporte a execução. Uma plataforma pode se espalhar por diversos computadores, e os agentes não precisam estar todos alocados no mesmo *host*.

O Facilitador de Diretórios é um componente opcional que provê um catálogo de agentes. Ele mantém uma lista completa de todos os agentes, além do controle do acesso a informação. Para alterar sua própria estrutura, o agente deve requisitar isso ao facilitador, porém, essa lista nem sempre tem a obrigatoriedade de manter sua consistência.

Por fim, o Sistema de Controle de Agentes(SCA) é responsável por administrar a Plataforma dos Agentes, sendo responsável pela criação, exclusão, migração, entre outros. Quando uma vida de um agente se extingue, o mesmo é eliminado do SCA.

Capítulo 3

GPU Computing

3.1 Visão Geral

Uma GPU, *Graphics Processing Unit*, é uma arquitetura de *hardware* originalmente desenvolvida para computação gráfica, mas com uma grande capacidade de cálculos de ponto flutuante em paralelo. Por muitos anos essa arquitetura estava restrita apenas para renderização, e até 2006, a única forma de programar aplicações genéricas na GPU era utilizando alguma API, como OpenGL e DirectX, criando diversas estruturas complexas, e ao final mapeando-as para *Pixels* e *Vertex Shaders*, gerando um número grande de soluções impraticáveis ou pouco otimizáveis. A figura **??** ilustra como funcionava a as arquiteturas baseadas em *Pixel* e *Vertex Shaders* e seus diferentes estágios.

Nesse paradigma, o dado de entrada é processado como um *stream*, i.e, toda a informação é processada em blocos paralelos e esse *stream* é alimentado dentro das unidades de processamento da GPU, cada uma executando uma cópia do mesmo código, seguindo o modelo SIMD(*Single Instruction Multiple Data*). Em 2006, a NVidia propôs o primeiro modelo de arquitetura unificada de cores de GPU, não mais necessitando de diferentes processadores para diferentes tipos de *shaders* como era feito anteriormente. No mesmo ano, os programadores tiveram acesso ao uma série de ferramentas em diferentes linguagens como C/C++, Fortran, OpenCL, entre outras para criar programas que não mais necessitavam de uma API específica. Essa arquitetura é conhecida como *Computer Unified Device Architecture*, também conhecida como *CUDA*[24]. Esse novo paradigma é intitulado *GPU Computing*. A figura 3.1 ilustra essa arquitetura.

É importante frisar que a *GPU Computing* ainda está sendo desenvolvida, e existe muitos desafios e restrições que dificultam o mapeamento direto, modelagem de um algo-

Figura 3.1: Arquitetura das placas baseadas na arquitetura Kepler da NVidia

ritmo sequencial e até mesmo algoritmos paralelos já existentes. O modelo de programação se organiza de uma forma diferente das CPUs. Um conjunto de *threads* é organizado em blocos que podem cooperar entre si, que estão dentro de um *grid*. O número de blocos e *threads* geralmente são definidas de forma empírica, dependendo das definições do hardware utilizado e das otimizações do algoritmo. Devido ao pequeno controle de execução que, em geral, as arquiteturas SIMD oferecem, é importante evitar repetições aninhadas e divergências de código no algoritmo. Em geral, é uma boa estratégia utilizados os índices dos blocos e *threads* para acessar localmente o dados e distribuir o processamento dos dados entre as *threads*, sendo assim, quando um *Cuda kernel* executa, apesar de todas as *threads* executarem o mesmo código, cada uma possui um certo grau de independência.

A arquitetura Fermi [25] incluiu uma nova habilidade, chamada de execução de *kernel* concorrente. Essa habilidade permite o programador executar vários pequenos e diferentes *kernels* em paralelo, possibilitando o uso máximo de todos os recursos e poder computacional da GPU. A figura **??** ilustra essa habilidade. Os blocos *kernel* preenchem ao máximo os *stream multiprocessors* de uma GPU, em paralelo, economizando tempo e permitindo uma melhor ocupação da GPU.

3.1.1 Warp

Cada *thread* é executado por um único core, cada bloco de *threads* é executado em um *Strem Multiprocessor*(SM), que consiste em uma sequência de cores. Um conjunto de

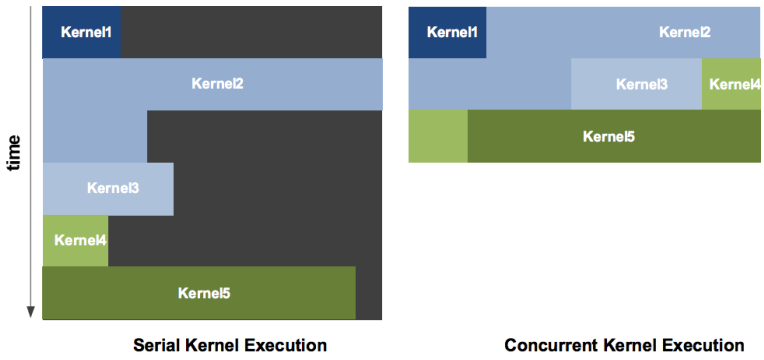

Figura 3.2: Funcionamento de *Kernels* concorrentes dentro de uma única GPU

threads executando em paralelo em um bloco é chamado de *Warp*. O programador não possui controle sobre como esses *warps* serão organizados dentro da GPU. *Threads* possuem acesso a uma memória privada, utilizada para registradores e chamadas de funções. Elas possuem também acesso a uma memória global, para sincronizações e dados compartilhados para outras threads, mas é recomendável que se evite a utilização dessa memória, pois além de ser uma memória de acesso lento, pode-se gerar conflitos de dados.

A GPU executa uma instrução para todas as *threads* de um mesmo *warp*, antes de ir para a próxima instrução. Esse tipo de comportamento deve-se à forma que a GPU é construída, visando minimizar o processamento de uma instrução, fazendo com que a mesma seja feita por diversas *threads* em paralelo. Ela funciona muito bem em funções aritméticas, onde o processamento é simples. Esse modo de executar as instruções é chamado de *SIMT (Single Instruction Multiple Thread)*.

Se uma *thread* do *warp* executar uma instrução que requer sua interrupção, i.e acesso a memória, o *warp* é retirado da execução, e outro *warp* é colocado para execução no *Stream multiprocessor*. Esse escalonamento de *warps* pode ser usado para otimizar outras tarefas, como pipeline de operações de ponto flutuante, e divergências de código resultantes de instruções *if-then-else*. Essa não tolerância a latências na execução é consequência do pouco espaço reservado pela GPU ao controle das *threads*, *cache* e mecanismos como antecipação de *pipeline* de execução comuns em CPUs.

3.1.2 Restrições da GPU

Apesar de bastante versátil, e permitir que o programador crie qualquer tipo de programa geral dentro da GPU, algumas diretivas devem ser obedecidas para que evite um mal uso dos recursos dentro da GPU, comprometendo assim a performance do algoritmo. Em diferentes aplicações as restrições impostas pela GPU pode criar um fator limitante ao

programador, exigindo que o mesmo utilize de artifícios como estruturas de dados mais complexas ou mesmo modificação extrema do algoritmo para melhorar sua execução nessa arquitetura.

Por exemplo, uma instrução do tipo *if-then-else* é considerada ineficiente dentro da GPU. Devido ao SIMT explicado anteriormente, quando uma *warp* entra em uma divergência de código, ou seja, duas ou mais threads dentro de um mesmo *warp* executam instruções distintas, a GPU irá executar a porção de código do *if* para todas as *threads* que estiverem dentro do *if*, e depois necessitará executar as instruções do *else* para todas as *threads* remanescentes. Essas duas passadas são sequenciais, aumentando assim o tempo de execução de um *warp*. Uma forma de evitar essa segunda passada desnecessária é trocar instruções do tipo *if-then-else* por expressões aritméticas, como demonstrado a seguir:

```
if(a < 0){
  b = x;
}
else{
  b = y;
}
```

Substituir por:

```
b = ( a < 0 ) * x + ( a >= 0 ) * y;
```

Outra grande restrição da GPU se dá no acesso a sua memória, em especial a memória global. A natureza massivamente paralela da GPU proporciona o processamento de diversos dados por suas *threads*, porém, os dados iniciais na maioria das vezes estão guardados na memória global, e eles devem ser acessados de forma a otimizar o uso da cache da GPU e evitar faltas de páginas.

Ao requisitar um dado em um endereço de memória da GPU, devido ao tamanho da palavra, os endereços vizinhos também são requisitados. Caso *threads* de um mesmo *warp* requisitarem endereços contíguos de memória, isso levará a menos acessos a global, e a menos interrupções de I/O, diferentemente de um acesso for menos linear. A figura 3.1.2 mostra uma alocação na memória de uma matriz 2D, e duas formas diferentes de acessá-la, uma forma linear e não linear. A forma não linear precisa de diversas chamadas para acessar os dados, enquanto na forma linear, um acesso traz consigo vários dados.

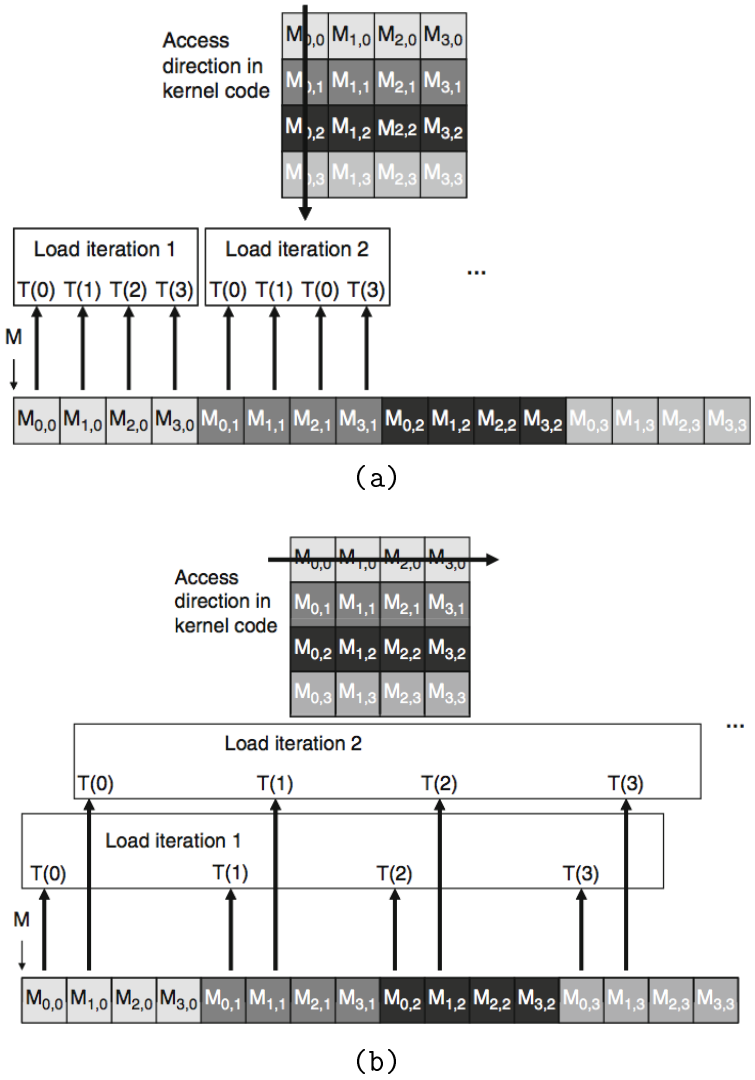

Figura 3.3: (a) Acesso linear da memória, (b) Acesso não linear da memória[14]

3.2 Uso de agentes em GPU

Desde 2006, o uso de unidades de processamento gráficos (GPUs) se tornou não somente uma nova área de pesquisa mas é utilizado por muitas aplicações e sistemas operacionais para evitar gargalos de processamento em seus algoritmos. Quando as GPUs se tornaram baratas e totalmente programáveis, muitos pesquisadores começaram a utilizar esse poder computacional com o intuito de criar um número grande de agentes com comportamentos melhorados, de acordo com as limitações impostas [2, 39]. Entretando, o mapeamento de comportamentos de agentes para uma GPU não é algo trivial, dado as restrições impostas por esta arquitetura e a complexidade dos algoritmos utilizados em inteligência artificial aplicados para agentes e sistemas multi agentes. Muitas heurísticas tentam evitar a complexidade $O(n^2)$ utilizando estruturas mais complexas e árvores de decisão. Apesar

desses algoritmos terem bons resultados para um número pequeno de agentes, dificilmente eles atingem uma boa escalabilidade [40].

Diferentemente de outras pesquisas, a linha de pensamento estabelecida nesse trabalho é que a padronização do processo de criação de agentes em arquiteturas GPU é necessária não apenas para melhorar as implementações atuais em GPU, mas para facilitar a programação dos agentes. Essa linha foi proposta anteriormente em trabalhos passados [5, 6], onde nós exploramos o processo de mapeamento de agentes feitos em outro arcabouço que já utilizava o padrão FIPA, intitulado JADE [1], para *GPU Computing*.

Existem trabalhos na literatura que criam [32] ou modificam [15] arquiteturas já existentes de sistemas multi agentes com o objetivo de melhorar ou resolver um problema em específico. Essas aplicações, assim como esta, vislumbram alcançar não somente uma melhor performance, mas também um modo mais fácil de se programar agentes inteligentes, explorando todos os recursos possíveis.

Capítulo 4

Arquitetura Proposta para Criação de Agentes em GPU

A proposta dessa arquitetura é um pouco diferente da usual. Definimos ela para problemas em que haja uma grande quantidade de agentes interagindo em um meio. Para isso necessitamos de um grande poder computacional e paralelo. A natureza SIMD da GPU e sua escalabilidade foi escolhida pois é capaz de suportar uma ordem de grandeza de mais de 10^6 agentes sem perder sua escalabilidade. FIPA foi utilizada como base, mas , por questões de restrições da GPU, não é possível implementar todos os padrões dispostos no FIPA, Mas a cada nova versão dessa arquitetura essas restrições diminuem.

Como essa arquitetura foi bastante baseada na forma em como o FIPA atua, e em algumas implementações já existentes, podem haver semelhanças entre outras soluções já existentes na indústria. Esse um ponto positivo, pois diminui a curva de aprendizado de um programador já acostumado com esse paradigma e foca no principal destaque dessa arquitetura, que é o uso versátil da GPU para a criação de agentes.

4.1 Estrutura Básica

O ator principal desta arquitetura é o agente inteligente. Um ciclo básico deste agente é ilustrado no Algoritmo 1. Como podemos ver ela possui dois métodos básicos, o que cria todos os descritores de agentes (`CreateAgentDescriptor`) e o que inicia sua execução (`Start`). Outros métodos podem ser utilizados para inicializar o cenário ou mesmo os agentes, mas não são obrigatórios. Por fim a figura?? ilustra como essa arquitetura se conecta e quais as cardinalidades de cada conexão.

A figura 4.1 mostra uma visão geral dessa arquitetura contendo este agente genérico.

Algoritm 1 Ciclo de vida de um agente genérico

Requer: *Environment*: Mundo em que o Agente esta contido.
Requer: *State*: Estado inicial do Agente.
1: Carregar todas as variaveis;
2: **enquanto** (As condicoes de parada nao estao satisfeitas **faça**
3: Execute o comportamento do agente;
4: Interaja com o *Environment*;
5: Troque o *State* do agente;
6: **fim enquanto**

Os principais módulos são o `Container` e o `Agent Descriptor`. O `Container` é responsável por preparar e inicializar a execução do algoritmo, enquanto o `Agent Descriptor` fica a cargo da execução em si do agente.

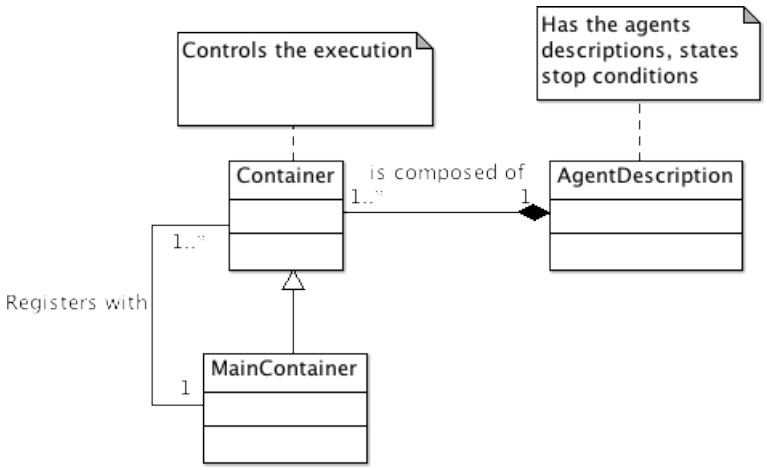

Figura 4.1: Visão geral do Modelo de Programação Orientado a Agentes (AOP) da Arquitetura Proposta

Neste modelo o agente não é programado diretamente, mas existem diversas descrições desse agente que ajudam a entender todo o fluxo de funcionamento de agente, como em qual estado que ele está, qual sua condição de parada e assim por diante. A Figura 4.1 mostra o mapeamento das funções básicas de um agente para um classe. Cada agente deve ter no mínimo um identificador único, uma função que será chamada enquanto duas condições de parada não forem aceitas, e deverá ficar responsável pelo comportamento do mesmo perante o mundo (`action`) e uma função para avaliar se o agente teve sua condição de parada satisfeita (`done`). A função `setup` fica responsável por preparar a execução desses agentes, iniciando as variáveis necessárias. Essa interface pode possuir outras funções adicionais, não limitadas apenas a essas duas. Outras variáveis podem e deverão ser necessárias para que o agente guarde o seu estado atual e sua percepção do mundo.

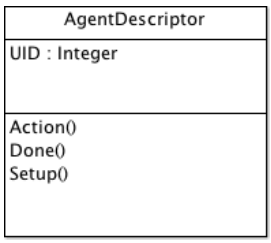

Figura 4.2: Classe básica de um Descritor de Agente

Para permitir a manutenção da estrutura orientada a agentes da aplicação, todo o agente deve estar alocado a uma classe responsável com iniciar sua execução, inicializar o cenário, finalizar a aplicação caso o objetivo global tenha sido atingido entre outras atribuições, neste caso é a classe `Container`. Dentro de uma aplicação podem existir um ou mais *containers* com diferentes agentes em diferentes cenários. Essa estrutura pode ser vista na figura 4.1.

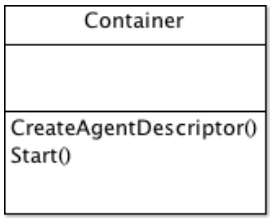

Figura 4.3: Classe básica de um Container

4.2 Conexão com a GPU

Na GPU, devido as restrições impostas pelo compilador, é necessário que o descritor de agentes seja dividido em dois arquivos, um `AgentDescriptor` e um *kernel*. O papel do `AgentDescriptor` é preparar todas as variáveis necessárias para a execução do agente e eventualmente processar parte da informação dos agentes. No kernel ficará toda a execução principal dos agentes, assim como as funções `action` e `done`. Esse mapeamento pode ser visto na figura 4.2.

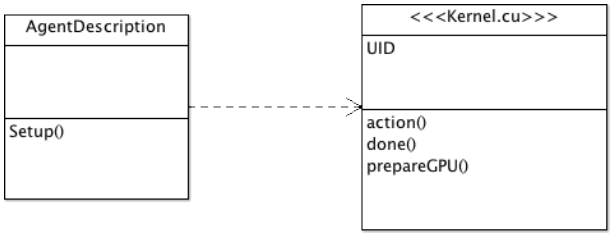

Figura 4.4: Classe básica de um Descritor de Agente utilizando a GPU

Diferentemente da CPU, a função `setup` fica responsável por instanciar o *kernel*, passar as variáveis necessárias para a execução, chamar função de alocação de memória em GPU e invocar o *kernel*. É importante notar que cada agente é instanciado como uma *thread* de um kernel em GPU. Assim, propriedades como a UID não precisam ser guardadas. Isso será explicado em mais detalhes na subseção 4.3. Toda a conexão dos agentes com os `Container` se mantém da mesma forma.

Esse tipo de modelo mostrou-se bastante versátil pois permite ao programador ou arquiteto de sistemas trocar facilmente o agente de uma dada aplicação apenas substituindo o `kernel` e o `AgentDescriptor` dentro de suas classes.

4.3 Estrutura do Kernel

Em uma abordagem típica de CPU, o agente persiste em todo o ciclo de vida da aplicação com sua descrição. Partindo do fato que um agente tem uma execução totalmente autônoma e a GPU segue o modelo SIMD, é necessário que todos os tipos de agentes contidos numa aplicação saibam seu próprio código e seus dados. Para isso, cada agente dentro da GPU é mapeado em uma *thread*, mas não necessariamente todos os dados dos agentes são processados dentro do *kernel*.

Além disso, um *kernel* pode ser utilizado para diferente configurações de agentes dependendo da aplicação. A GPU restringe o número de *threads* e blocos para números com base 2. Por exemplo, se em uma execução há 1000 agentes, devem ser criadas 1024 *threads* divididas em blocos. As *threads* excedentes são descartadas durante a execução. No algoritmo 2 pode-se visualizar como isso é feito. Se um UID de um agente é maior do quê o número máximo de agentes, essa *thread* ficará inativa durante a execução. Como cada agente é uma única thread, seu UID é uma simples conta aritmética entre a dimensão do bloco, o ID do bloco onde ele está contido e o número da *thread* deste bloco.

Algoritm 2 Kernel Concorrente

Requer: $World$: Cenario do Agente.
Requer: $State$: Posicao inicial do agente.
Requer: $NumberofAgents$: Numero maximo de agentes utilizados.
1: $UID = (BlockIndex \times BlockID) + ThreadID$;
2: **se** ($UID <$ Numero de Agentes) **então**
3: **enquanto** Condicoes de parada nao satisfeitas **faça**
4: $State =$ NextState($State$, $World$);
5: **fim enquanto**
6: **fim se**

Para testar a se capacidade de execução de *kernels* concorrentes é melhor que a de *kernels* singulares, o algoritmo 2 foi modificado para ser capaz de suportar diferentes tipos de agentes. Ele possui as mesmas restrições mostradas anteriormente, e cada agente também possui uma UID. A grande diferença é que a partir da UID é possível determinar o tipo do agente e saber qual é seu fluxo de código. O algoritmo 3 mostra como essa adaptação é feita.

Algoritm 3 Kernel Nao Concorrente

Requer: $World$: Mundo do agente.
Requer: $State$: Posicao Inicial do Agente.
Requer: $NumberofAgents$: Numero maximo de agentes utilizados.

```
1:  UID = (BlockIndex × BlockID) + ThreadID;
2:  se (UID < Numero de Agentes) então
3:     enquanto Condicoes de parada nao satisfeitas faça
4:        se UID = TypeA então
5:           State = NextStateTypeA(State, World);
6:        fim se
7:        se UID = TypeB então
8:           State = NextStateTypeB(State, World);
9:        fim se
10:    fim enquanto
11: fim se
```

Outro ponto importante é como calcular um dado número de *threads* e blocos a partir de um número de agentes na execução. Para tal, calculamos da seguinte maneira:

$$N_T = 2^{\left\lceil \frac{\log(N_A)}{\log 2} \right\rceil} \tag{4.1}$$

$$N_B = \frac{N_T}{N_W} \tag{4.2}$$

$$N_{TpB} = \frac{N_T}{N_B} \tag{4.3}$$

onde:

$\boldsymbol{N_A}$ é o número de agentes na simulação;

$\boldsymbol{N_T}$ é o número real de *threads* que serão criadas;

$\boldsymbol{N_B}$ é o número de blocos criadas pra essa execução;

$\boldsymbol{N_W}$ é o número de *threads* em um *warp*

$\boldsymbol{N_{TpB}}$ é o número de *threads* por bloco. O número total de *threads* é dado por $N_B \times N_T pB$.

Essas equações em geral minimizam o número de *warps* na execução, aumentam sua escalabilidade e calculam os blocos e *threads* em base 2. Se o número de blocos ou o número de *threads* por blocos é maior que o máximo de uma GPU, a segunda equação é relaxada, permitindo mais *warps* por blocos.

Utilizando agentes heterogêneos, o Agent Descriptor deve configurar e chamar o *kernel* mais de uma vez, dependendo da solução que está sendo implementada. Quando a GPU é utilizada, o descritor faz cópias da CPU para a GPU e vice versa. Ele também é responsável pela configuração do *kernel* e alocação e desalocação de memória da GPU. O comportamento do agente será incorporado pelo *kernel* e deverá ser específico para cada tipo de ação.

Capítulo 5

Validação da Arquitetura

Para validar nossa proposta escolhemos dois problemas distintos, o A* e o problema Seguir/Repelir. Ambos são problemas facilmente encontrados na literatura, que são utilizados em larga escala pela indústria, tem comportamentos facilmente modeláveis, e que tem dificuldades de processamento quando há um número grande de agentes.

No primeiro problema exploramos o funcionamento de um sistema homogêneo, com pouca comunicação entre os agentes, e um processamento mais complexo. O segundo por sua vez possui um processamento bem mais simples, mas tem mais acessos a memória e possui agentes heterogêneos visando explorar os *kernels* concorrentes da GPU.

Em ambos os casos utilizamos a mesma estrutura proposta, trocando basicamente o descritor de agentes para cada problema. A arquitetura se mostrou robusta para ambos os casos, e o detalhamento de cada teste assim como seus resultados podem ser vistos a seguir.

5.1 Agentes de Pathfiding

Como primeiro caso de teste, usaremos um problema de *pathfiding* para um número grande de agentes com o objetivo de testar toda a escalabilidade do sistema, funções mais avançadas como sincronização e busca heurística. Neste problema, cada agente deverá ser capaz de encontrar um caminho da sua origem até um ponto final que denominaremos de *checkpoint*.

5.1.1 O Algoritmo A*

O algoritmo A*[1] [23] 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 5.1.

$$f(n) = g(n) + h(n) \tag{5.1}$$

No Algoritmo 4 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 ser iniciadas como 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*, coloca-o 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 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.

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

[1] Pronuncia-se A Estrela

Algoritmo 4 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** Adjecentes A_i de A **faça**
7: **se** $A_i x, y$ 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** $A_i\{G\} < P\{G\}$ **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 *best-first* 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 5.1: Tabela comparativa entre o Método de Dijkstra e o Método A*

A* depende diretamente da função heurística usada na avaliação da função, tendendo a complexidade exponencial caso tal função seja muito precisa [36]. O algoritmo segue a complexidade média $O(N \exp (C\phi(N))$, onde $\phi(N) = log(N)^k$ [27]. A Tabela 4.2 mostra um quadro comparativo entre os algoritmos A* e o algoritmo de Dijkstra.

5.1.2 A Arquitetura

Para este problema, seguindo a arquitetura proposta na seção 4, o nosso problema foi modelado como mostrado na figura 5.1.2. O agente é descrito diretamente na classe `Agent Descriptor` de maneira totalmente independente do resto da aplicação, como pede o FIPA. Neste caso utilizamos uma estrutura auxiliar que contém os dados da GPU em questão para facilitar a implementação.

No caso da GPU, a única coisa que foi necessário modificar foi a classe `Agent Descriptor`. Ela agora tem como objetivo fazer o elo entre a CPU e a GPU, preparando-a, fazendo as chamadas necessárias e recebendo os dados ao final da execução. Isso é ilustrado na figura **??**.

Isso valida um dos pontos citados na seção 4, a versatilidade da arquitetura. O problema para CPU e GPU são modelados de forma bastante similar, e como veremos no

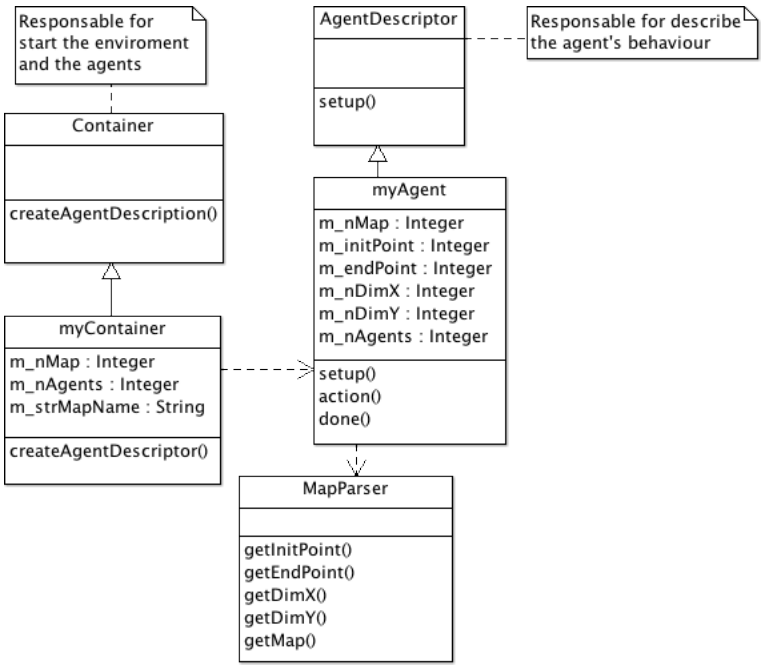

Figura 5.1: Diagrama de classes para o problema do A* em CPU

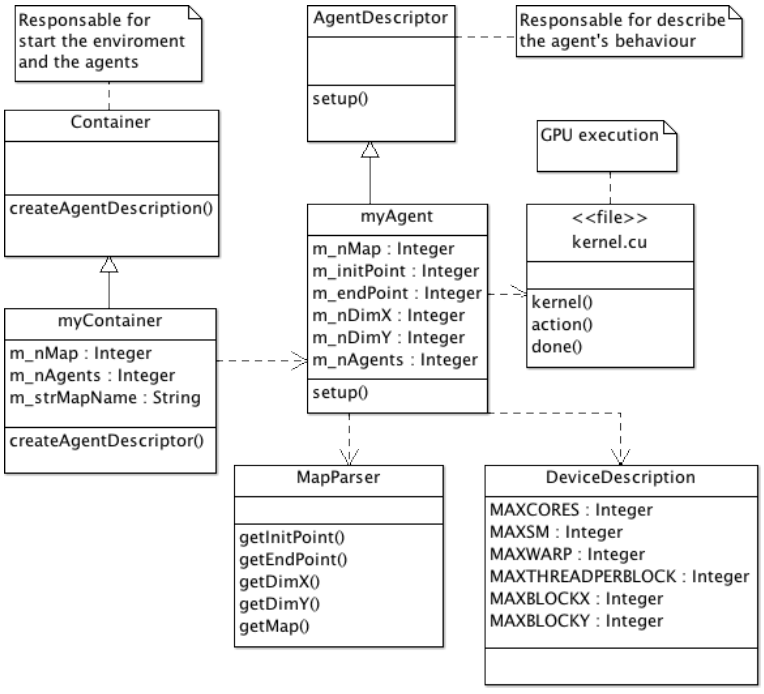

Figura 5.2: Diagrama de classes para o problema do A* em GPU

segundo casos de teste, a substituição desse agente também é relativamente simples.

5.1.3 Análise de desempenho

O principal objetivo desse caso de teste é verificar a escalabilidade da aplicação, ou seja, se a execução é possível e se mantém em um tempo razoável a medida que aumentamos

o número de agentes. Utilizamos um processador Intel Core i7 3.07GHz com 8GB de Memória DDR3 como CPU, e uma nVidia GeForce GTX 580 com 512 *Cuda Cores*, cada core com 1544MHz e com memória 1536MB GDDR5 para a GPU. Todos os cenários de teste na tabela 5.1.3 foram executados 10 vezes em um sistema operacional CentOS 6. O mapa foi fixado em uma dimensão 1000×1000. Note que as colunas N_A (número de agentes na simulação), N_B (número de blocos criados) e N_{TpB} (número de *threads* por bloco) são usadas as configurações de GPU calculadas nas Equações ???.

Número do Teste	N_A	N_B	N_{TpB}
T_0	10^0	1	1
T_1	10^1	1	16
T_2	10^2	4	32
T_3	10^3	32	32
T_4	10^4	512	32
T_5	10^5	1024	128
T_6	10^6	1024	1024

Tabela 5.2: Cenários de Testes executados

Tabela 5.1.3 mostra a média e o desvio padrão das 10 execuções de tempo. Podemos observar que, a medida que o número de agentes cresce (T_1 à T_6), a GPU mantém sua escalabilidade, perdendo tempo apenas quando o número de *warps* cresce. Entretanto, esses tempos são considerado bons para uma simulação em tempo real. Por outro lado, os tempos da CPU cresce de forma linear a medida que o número de agentes cresce. Para uma melhor percepção da evolução do tempo na implementação da GPU, a figura **??** mostra como essa evolução se dá a medida que o número de agentes cresce. É possível ver que há mudanças perceptuais a medida que é necessário mais *warps* por blocos, especialmente nos casos onde há 10^5 e 10^6 agentes.

Número	CPU		GPU	
do Teste	Tempo Médio(s)	Desvio Padrão(s)	Tempo Médio(s)	Desvio Padrão(s)
T_0	0,000027	0,000004	0,102781	0,001090
T_1	0,000240	0,000005	0,103334	0,000914
T_2	0,002377	0,000006	0,104765	0,003214
T_3	0,024976	0,000911	0,105510	0,004830
T_4	0,235692	0,000077	0,107930	0,008282
T_5	2,360808	0,003434	0,108138	0,005976
T_6	23,562007	0,000502	0,119523	0,001940

Tabela 5.3: Performance dos Algoritmos em CPU e GPU

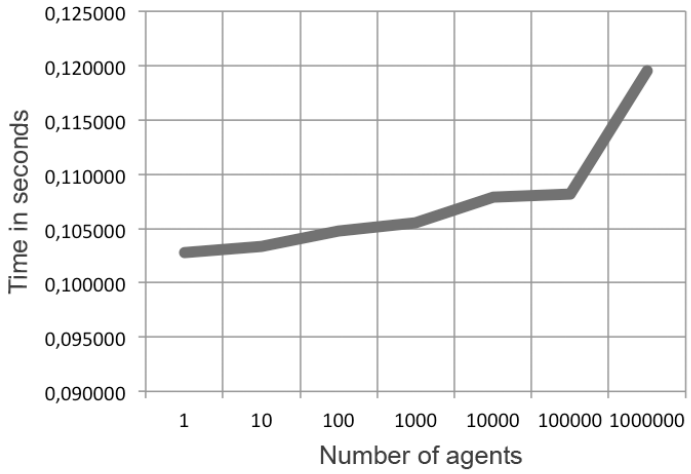

Figura 5.3: Gráfico que representa os resultados dos cenários de teste

5.2 Agentes Seguir/Repelir

5.2.1 O Problema

O problema dos Agentes Seguir/Repelir é um problema puramente paralelo que consiste, em um sistema multi agente, iterações entre agentes, tendo alguns a tendência de se aproximar de outros e alguns a tendência de se repelir em relação aos outros. Esse problema é uma simplificação do *Flocking Behaviour Problem* [31] de Reynolds. Apesar da complexidade deste problema ser considerada $O(1)$, este é um problema bem difícil de ser processado computacionalmente quando lidamos com um número grande de agentes. O objetivo desde teste é checar o comportamento dos *kernels* quando lidamos com um grande número de agentes heterogêneos, utilizando *kernels* concorrentes. Também queremos verificar como a divergência de código afeta *kernels* não concorrentes.

O algoritmo 5 consiste em procurar a melhor posição para o agente se mover. A cada iteração o agente pode mover um passo para todas as direções (norte, sul, leste, oeste, nordeste, sudeste, noroeste, sudoeste) respeitando as bordas do mapa. Por simplicidade não estamos calculando colisões entre os agentes.

O comportamento do agente que irá repelir é análogo ao agente que irá seguir. O agente checa sua posição com o agente alvo, mas ao invés de se mover para perto dele, ele se mova na direção contrária, aumentando a distância entre eles.

Algoritm 5 Comportamento de Seguir

Requer: $FollowPosition$: Posicao do agente que sera seguido.
Requer: $AgentPosition$: Posicao inicial do Agente.
Requer: $MapDim$: Dimensões do mapa utilizado.

1: **se** $FollowPosition.x > AgentPosition.x$ e $(AgentPosition.x + 1) < MapDim.x$ **então**
2: $AgentPosition.x = AgentPosition.x + 1$;
3: **senão se** $FollowPosition.x < AgentPosition.x$ e $(AgentPosition.x - 1) > 0$ **então**
4: $AgentPosition.x = AgentPosition.x - 1$;
5: **fim se**
6: **se** $FollowPosition.y > AgentPosition.y$ e $(AgentPosition.y + 1) < MapDim.y$ **então**
7: $AgentPosition.y = AgentPosition.y + 1$;
8: **senão se** $FollowPosition.y < AgentPosition.y$ e $(AgentPosition.y - 1) > 0$ **então**
9: $AgentPosition.y = AgentPosition.y - 1$;
10: **fim se**

5.2.2 A Arquitetura

Seguindo a estrutura proposta na seção 4, implementados o problema em CPU como ilustrado da figura 5.4. Existe um `container` que possui todos os agentes, e o descritor de agentes implementa todos os métodos que garantem a independência dos agentes.

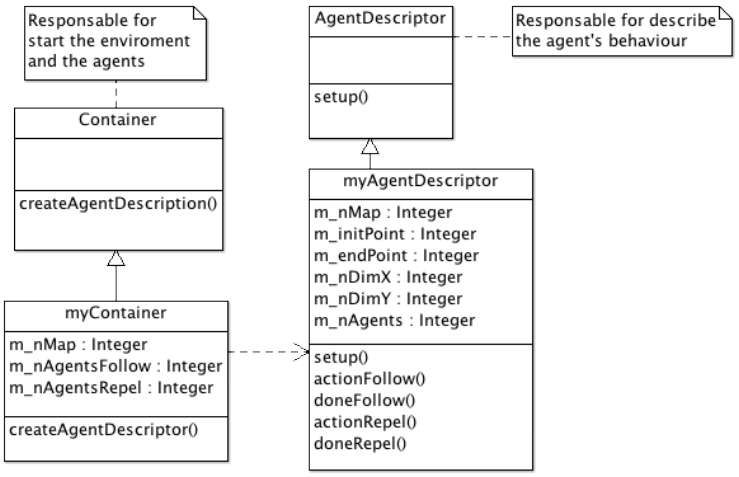

Figura 5.4: Diagrama de Classes para o Problema Seguir/Repelir em CPU

A abordagem de GPU não é muito diferente. A figura 5.5 mostra que o *kernel* é chamado externamente da classe que herda o descritor de agentes e o processamento principal desse agente foi colocado no *kernel*. Note que essa figura mostra que o mesmo descritor de agentes pode ser utilizado para agentes heterogêneos quando usamos *kernels* concorrentes.

A arquitetura proposta segue alguns dos padrões de gerenciamento de agentes defini-

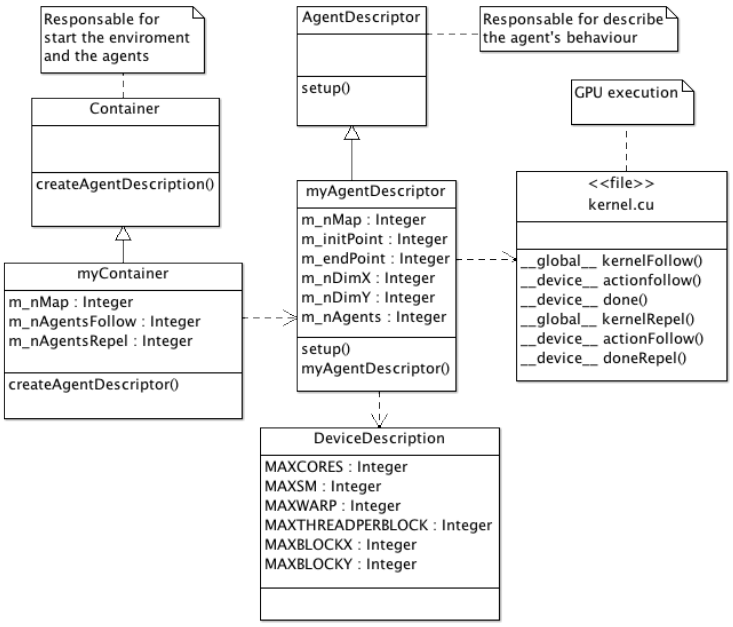

Figura 5.5: Diagrama de Classes para o Problema Seguir/Repelir em GPU

dos pelo FIPA e pode ser facilmente reproduzida e adaptada com poucas modificações. Apesar disso ainda há algumas restrições em relação às comunicações dos agentes. Essas soluções novamente refletem as restrições do paradigma SIMD que a GPU utiliza, onde não é recomendável a troca de informações entre as *threads*.

5.2.3 Análise de Performance

Para esse teste executamos 3 tipos de implementações, CPU, *kernel* concorrente e *kernel* não concorrente 10 vezes. A condição de parada foi fixada em 10000 iterações para cada agente e o mapa foi fixado em uma dimensão 1000×1000. Utilizados a mesma máquina que fora usada nos testes anteriores, uma Intel Core i7 3.07GHz com 8GB de memória DDR3 para a CPU e uma Geforce 580 com 512 *CUDA* cores, com 1544GHz para cada core e uma memória de 1536MB GDDR5 para a GPU. Todos os cenários de teste na tabela 5.2.3 foram executados no sistema operacional CentOS6.

A tabela 5.2.3 mostra o tempo médio ($M_t(s)$), e o desvio padrão ($Std(s)$)dos 10 tempos executados em segundos. Podemos observar que a medida que o número de agentes cresce (T_1 à T_{14}), os tempos de CPU crescem exponencialmente. A abordagem de *kernel* não concorrente tem um grande crescimento nos últimos casos de teste, devido à divergência de código. A abordagem com *kernel* concorrente se mantém em um estado linear, com melhores tempos comparado com a CPU em casos com um grande número de

Caso de Teste	Configuração	
	Agentes Seguir	Agentes Repelir
T_1	10	10
T_2	20	20
T_3	30	30
T_4	40	40
T_5	50	50
T_6	100	100
T_7	250	250
T_8	500	500
T_9	700	700
T_{10}	1000	1000
T_{11}	10000	10000
T_{12}	100000	100000
T_{13}	200000	200000
T_{14}	500000	500000

Tabela 5.4: Cenários de teste

agentes.

Casos de Teste	CPU		GPU	Concorrente	GPU Não Concorrente	
	M_t(s)	Std(s)	M_t(s)	Std(s)	M_t(s)	Std(s)
T_1	0,0001336336	0,0000021940	0,0056146365	0,0000315273	0,0012846592	0,0001017912
T_2	0,0002680778	0,0000039294	0,0056873984	0,0000117149	0,0014392669	0,0000706423
T_3	0,0004018784	0,0000063850	0,0058388659	0,0000252241	0,0014152941	0,0000706406
T_4	0,0005391955	0,0000223477	0,0059318189	0,0000499846	0,0014621808	0,0000785823
T_5	0,0006731868	0,0000226515	0,0059026317	0,0000322182	0,0014667651	0,0000472758
T_6	0,0013363957	0,0000075873	0,0059864922	0,0000249610	0,0014638013	0,0000273684
T_7	0,0033433557	0,0000365957	0,0064051283	0,0000339204	0,0015940154	0,0000306022
T_8	0,0067131043	0,0001431485	0,0066082928	0,0000185607	0,0016446221	0,0000390102
T_9	0,0093456626	0,0000874901	0,0067496013	0,0001361668	0,0016765914	0,0000210507
T_{10}	0,0133333445	0,0001236207	0,0068234310	0,0000492380	0,0017389414	0,0000299041
T_{11}	0,1343695045	0,0023372785	0,0388589641	0,0000393143	0,0094199421	0,0000362678
T_{12}	1,3332240820	0,0036477387	0,2631043637	0,0000900465	0,2184030116	0,0001129489
T_{13}	2,6821980476	0,0254864376	0,6884414947	0,0001957378	2,6295261574	0,0008705217
T_{14}	6,6677182913	0,0044754592	5,8546949482	0,0012406186	7,5032364716	0,0015366176

Tabela 5.5: Performance do Algoritmo em CPU e em GPU

As figuras 5.6 e 5.7 ilustram a evolução do tempo ao longo dos casos de teste. Nessas figuras, podemos observas que a abordagem de *kernel* não concorrente é melhor que a abordagem de *kernel* concorrente nos casos de teste T_{10}, T_{11} e T_{12}, porém começa a ficar pior nos casos T_{13} and T_{14}. Isso é causado pelo *overhead* da troca de contexto quando lidamos com *kernels* concorrentes. Para checar por uma diferença estatística significante entre os resultados obtidos nas execuções dos cenários T_{13} e T_{14}, consideramos duas hipóteses nulas: (**H1**) Os resultados obtidos em um *kernel* concorrente são comparados com a GPU e (**H2**) Os resultados obtidos em um *kernel* concorrente de GPU é comparado com

a CPU. Para ambas as hipóteses nulas, executamos t testes com o nível de certeza de 99% em cada cenário de teste(T_{13} e T_{14}). A hipótese nula foi rejeitada nos quatro testes.

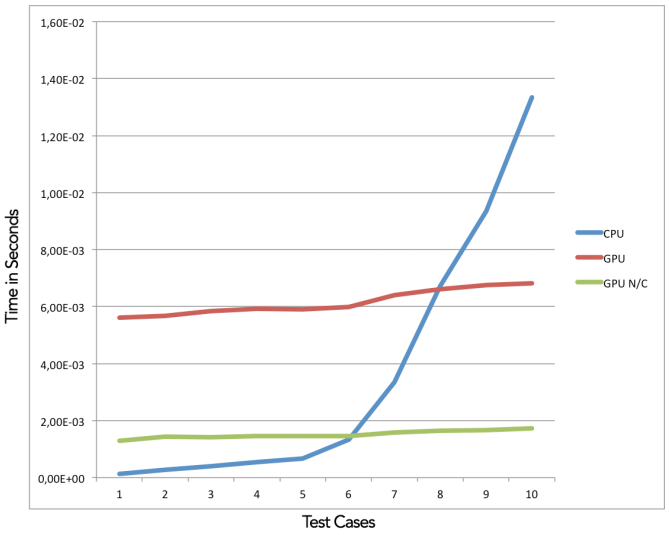

Figura 5.6: Cenários de teste 1 à 10

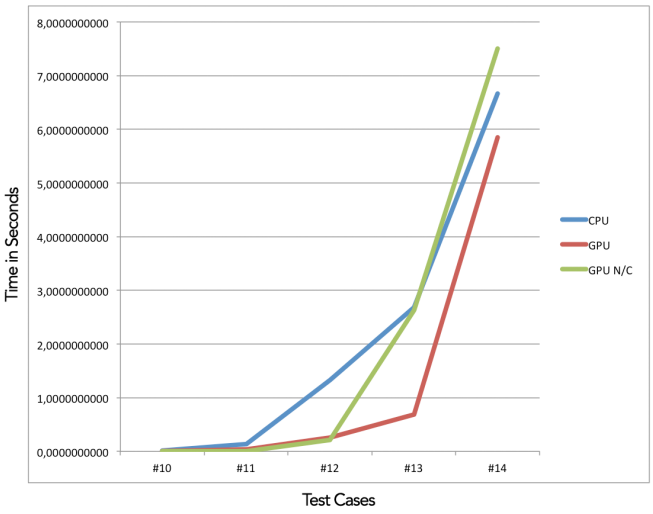

Figura 5.7: Cenários de teste 10 à 14

A arquitetura proposta segue os mesmos padrões de manutenção de agentes definidos pelo FIPA e pode ser facilmente reproduzidos e adaptados com poucas modificações. Apesar disso, ainda há muitas restrições em nossa implementação, principalmente na implementação de comunicação entre agentes. Essas restrições existem devido ao paradigma SIMD da GPU, onde não é recomendável criar divergências de código dentro de um *kernel* somente para a solucionar esses problemas. Por essas razões, essa solução é focada para programadores que querem criar agentes utilizando FIPA com poucas comunicação e e pouca heterogeneidade entre elas. Outra característica é que é possível reproduzir

inúmeros problemas com o mesmo padrão, apenas trocando algumas linhas de código, aumentando assim a produtividade e a confiabilidade do código, que é um dos grandes desafios quando desenvolvemos jogos e simuladores.

Capítulo 6

Conclusão e Trabalhos Futuros

Este trabalho apresentou uma proposta de arquitetura para sistemas multi agentes baseada nos padrões FIPA utilizando uma arquitetura massivamente paralela, a GPU. Foram expostas vantagens e desvantagens da mesma, além de dois casos de teste para sua validação. Esses casos de teste mostraram que a partir de seu uso, pode-se utilizar um número grande de agentes em uma aplicação, com taxas de quadros aceitável e criando um ambiente facilitado para sua programação.

O uso de agentes é bem difundido dentro de jogos e simulações, sendo considerada uma parte vital para a experiência do jogador. Existem diferentes abordagens que lidam com esse problema, mas, em sua maioria, a CPU é utilizada para processar essa informação. Apesar da CPU ser computacionalmente aceitável para soluções com agentes de comportamentos complexos e suportar comunicações entre eles, se um dado problema necessitar de um número grande de agentes, essa solução é impraticável para aplicações interativas.

A motivação principal deste trabalho foi facilitar e difundir o uso de GPUs para sistemas multi agentes em aplicações com grande número de entidades envolvidas. Muitos dos trabalhos nessa área focam em resolver determinado problema, não se preocupando se sua solução pode ser reaproveitada para outros problemas da mesma categoria. A arquitetura baseada em FIPA foi desenvolvida focando-se na simplicidade do seu uso, no tempo de aprendizado de um programador já acostumado a esse tipo de aplicação, e nas restrições impostas pela GPU. Basear-se em algo já bastante difundido pela academia e pela indústria para esta criação facilitou o entendimento dos problemas relacionados à programação de agentes em GPU, e possibilitou uma forma de minimizá-los.

Toda a proposta apresentada visa ser um embrião do que possa a ser futuramente

um arcabouço robusto para a criação de grandes números de agentes, acompanhando a evolução das GPUs. Os casos de teste serviram para corroborar o conhecimento das restrições da GPU no contexto dos agentes, e determinar como podemos utilizar o poder computacional dessa arquitetura dentro de um paradigma orientado a agentes. Ao longo do desenvolvimento dessa tecnologia, as restrições impostas por essa arquitetura irão diminuir, entretanto, sua natureza massiva e altamente escalável deverá se manter.

Essa dissertação se focou em prover um arquitetura inicial simples para a criação que abrangesse o maior número de tipos de agentes possível, respeitando as restrições de hardware. Espera-se que este trabalho gere diretivas para a criação de uma camada abstrata robusta para a criação de agentes em GPU, aumentando a possibilidade de modelagem de agentes, o processo produtivo de criação de agentes, e é claro, sua performance.

Todos os casos de teste de validação dessa proposta foram baseados em problemas do cotidiano dessa área, que apesar de não serem tão complexos, mostram-se fundamentais para a criação de uma IA básica em jogos e simulações, e tem dificuldades de processamento em ambientes massivos. Seus resultados se mostraram promissores e dentro da estimativa de criação de uma inteligência massiva em tempo real para esse sistemas. A escalabilidade dos agentes se manteve estável a medida que o número de agentes dentro da simulação aumentava na GPU, diferentemente da CPU, que mostrava um padrão exponencial na medida de tempo de sua execução.

As aplicações desse trabalho são diversas, desde a criação de sistemas mais robustos para IA aplicado a jogos e simulação, até a melhora de sistemas já existentes, permitindo novas possibilidades de iteração, aumentando assim a imersão do jogador e a sensação de realidade da simulação. Ele também pode ser usado para resolução de problemas de natureza distribuída embasados no conceito de sistema multi agente.

Uma crítica a este trabalho é que como nos baseamos em uma arquitetura já existente e foram feitas modificações, nem todas as características da GPU podem ser eficientemente utilizadas. Uma outra linha de pesquisa seria criar uma arquitetura do zero, mais influenciada pela arquitetura SIMD da GPU, tentando explorar ao máximo suas capacidades e evitando ao máximo suas restrições, mesmo que esta arquitetura seja diferente da utilizada atualmente. Outra possível linha de pesquisa a esse trabalho é aplicar outros tipos de agentes em GPU, não somente agentes reativos, como BDI e outras linguagens já existentes.

Este trabalho vem sendo desenvolvido a cerca de 3 anos, desde minha graduação, e foi pensado inicialmente apenas para facilitar o desenvolvimento de um sistema para

planejamento de evacuações de emergência. Ao longo desse período, vimos que explorar as limitações impostas nos traria um valor agregado tão grande quanto o sistema em si, e ajudaria outras pessoas que estivessem com a mesma dificuldade para criação de IA em GPUs. Por fim, esperamos que este trabalho agregue valor a esta nova linha de pesquisa e adicione mais possibilidades para o processamento de IA, em especial, agentes em placas gráficas, melhorando seu desempenho e raciocínio lógico.

Referências

[1] Bellifemine, F.; Caire, G.; Greenwood, D. *Developing multi-agent systems with JADE*. Wiley Series in Agent Technology, 2007.

[2] Bleiweiss, A. Gpu accelerated pathfinding. In *Proceedings of the 23rd ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware* (2008), NVidia, pp. 65–74.

[3] Brooks, R. A. Intelligence without reason. In *Proceedings of the 12th international joint conference on Artificial intelligence - Volume 1* (San Francisco, CA, USA, 1991), IJCAI'91, Morgan Kaufmann Publishers Inc., pp. 569–595.

[4] Chen, D.; Wang, L.; Tian, M.; Tian, J.; Wang, S.; Bian, C.; Li, X. Massively parallel modelling and simulation of large crowd with gpgpu. *The Journal Of Supercomputing* (2011).

[5] dos Santos, L. G. O.; Bernardini, F. C.; Clua, E. G.; da Costa, L. C.; Passos, E. Mapping multi-agent systems based on fipa specification to gpu architectures. In 3^a *Conferencia Anual em Ciencia e Arte dos Videojogos* (Instituto Superior Tecnico - Taguspark, Lisboa, Portugal, 2010), pp. 109–118.

[6] dos Santos, L. G. O.; Bernardini, F. C.; Clua, E. G.; da Costa, L. C.; Passos, E. Mapping a path-fiding multiagent system based on fipa specification to gpu architectures. In *X Simposio Brasileiro de Games e Entretenimento Digital* (2011).

[7] dos Santos, L. G. O.; Gonzales Clua, E. W.; Bernardini, F. C. A parallel fipa architecture based on gpu for games and real time simulations. In *Entertainment Computing - ICEC 2012*, M. Herrlich, R. Malaka, and M. Masuch, Eds., vol. 7522 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2012, pp. 306–317.

[8] FIPA. Foundation for intelligent physical agents. `http://fipa.org/`, 2012.

[9] Flores-Mendez, R. Towards a standardization of multi-agent system frameworks. *ACM Crossroads Magazine* (1999). `http://www.acm.org/crossroads/xrds5-4/multiagent.html`.

[10] Franklin, S.; Graesser, A. Is it an agent or just a program? a taxonomy for autonomous agents. In *Intelligent Agents III. Agent Theories, Architectures, and Languages. LNAI* (1996), J. Muller, M. Wooldridge, and N. Jennings, Eds., vol. 1193, pp. 21–35.

[11] Guy, S. J.; Chhugani, J.; Kim, C.; Satish, N.; Lin, M.; Manocha, D.; Dubey, P. Clearpath: highly parallel collision avoidance for multi-agent simulation.

In *Proceedings of the 2009 ACM SIGGRAPH/Eurographics Symposium on Computer Animation* (2009), SCA '09, ACM, pp. 177–187.

[12] Han, S. W.; Kim, J. Preparing experiments with media-oriented service composition for future internet. In *Proceedings of the 5th International Conference on Future Internet Technologies* (New York, NY, USA, 2010), CFI '10, ACM, pp. 73–78.

[13] Islam, N.; Mallah, G. A.; Shaikh, Z. A. Fipa and masif standards: a comparative study and strategies for integration. In *Proceedings of the 2010 National Software Engineering Conference* (New York, NY, USA, 2010), NSEC '10, ACM, pp. 7:1–7:6.

[14] Kirk, D. B.; mei W. Hwu, W. *Programming Massively Parallel Processors: A Hands On Approach*. Elsevier, 2010.

[15] Komma, V.; Jain, P.; Mehta, N. An approach for agent modeling in manufacturing on jade reactive architecture. *Int. Journal of Advanced Manufacturing Technology 52* (2011), 1079–1090.

[16] Laird, J.; Newell, A.; Rosenbloom, P. Soar: An architecture for general intelligence. *Artificial Intelligence 33*, 1 (1987), 1–64.

[17] Lamarche, F.; Donikian, S. Crowd of virtual humans: a new approach for real time navigation in complex and structured environments. *Computer Graphics Forum 23* (2004), 509–518.

[18] Lester, P. A* for beginners. `http://www.policyalmanac.org/games/aStarTutorial.htm`, 2004.

[19] Loscos, C.; Marchal, D.; Meyer, A. Intuitive crowd behavior in dense urban environments using local laws. In *Theory and Practice of Computer Graphics, 2003. Proceedings* (june 2003), pp. 122 – 129.

[20] Maillot, P. Using quaternions for coding 3d transformations. In *Graphic Gems*, A. S. Glassner, Ed. Academic Press, Boston, MA, 1990, pp. 498–515.

[21] Minsky, M. L. A framework for representing knowledge. In *The Psychology of Computer Vision* (1975), pp. 211–277.

[22] Musse, S. R.; Thalmann, D. Hierarchical model for real time simulation of virtual human crowds. *IEEE Transactions on Visualization and Computer Graphics 7* (2001), 152–164.

[23] Nilson, N. J. *Problem-solving methods in Artificial Intelligence*. McGraw-Hill, 1971.

[24] NVIDIA. Cuda description. `http://www.nvidia.com/cuda`, 2007.

[25] NVidia. Fermi compute whitepaper. `http://www.nvidia.com/object/IO_86776.html`, 2011.

[26] Passos, E. B.; Joselli, M.; Zamith, M.; Rocha, J.; Clua, E. W. G.; Montenegro, A.; Conci, A.; Feijo, B. Supermassive crowd simulation on gpu based on emergent behavior. In *SBGames 2008* (2008).

[27] Pearl, J. *Heuristics: Intelligent Search Strategies for Computer Problem Solving.* Addison-Wesley, 1984.

[28] Pelechano, N.; Allbeck, J. M.; Badler, N. I. Controlling individual agents in high-density crowd simulation. In *Proceedings of the 2007 ACM SIGGRAPH/Eurographics symposium on Computer animation* (Aire-la-Ville, Switzerland, Switzerland, 2007), SCA '07, Eurographics Association, pp. 99–108.

[29] Rao, A. S.; Georgeff, M. P. Bdi agents: From theory to practice. In *IN PROCEEDINGS OF THE FIRST INTERNATIONAL CONFERENCE ON MULTI-AGENT SYSTEMS (ICMAS-95* (1995), pp. 312–319.

[30] Reynolds, C. Big fast crowds on ps3. In *Proceedings of the 2006 ACM SIGGRAPH symposium on Videogames* (2006), Sandbox '06, ACM, pp. 113–121.

[31] Reynolds, C. W. Flocks, herds, and schools: A distributed behavioral model. In *ACM SIGGRAPH '87 Conference Proceedings* (1987), vol. 21, pp. 25–34.

[32] Rodriguez, S.; Julián, V.; Bajo, J.; Carrascosa, C.; Botti, V.; Corchado, J. M. Agent-based virtual organization architecture. *Eng. Appl. Artif. Intell. 24*, 5 (Aug. 2011), 895–910.

[33] Russell, S.; Norvig, P. *Artificial Intelligence, A Modern Approach*, second edition ed. Prentice Hall, 1995.

[34] S. R. Musse, D. T. A model of human crowd behavior: Group inter-relationship and collision detection analysis. In *Workshop Computer Animation and Simulation of Eurographics*, pp. 39–52.

[35] Sanna, A.; Montuschi, P. A new algorithm for the rendering of csg scenes. *The Computer Journal 9* (1997), 555–564.

[36] Stefik, M. *Introducing to Knowledge Systems.* Morgan Kaufmann, 1995.

[37] Sung, M.; Gleicher, M.; Chenney, S. Scalable behaviors for crowd simulation. Eurographics '04.

[38] Thielscher, M. Flux: A logic programming method for reasoning agents. *Theory and Practice of Logic Programming 5* (2005), 533–565.

[39] Torchelsen, R. P.; Scheidegger, L. F.; Oliveira, G. N.; Bastos, R.; Comba, J. L. D. Real-time multi-agent path planning on arbitrary surfaces. In *Proceedings of the 2010 ACM SIGGRAPH symposium on Interactive 3D Graphics and Games* (2010), I3D '10, ACM, pp. 47–54.

[40] Turner, P.; Jennings, N. Improving the scalability of multi-agent systems. In *Infrastructure for Agents, Multi-Agent Systems, and Scalable Multi-Agent Systems*, vol. 1887 of *Lecture Notes in Computer Science*. Springer Berlin, 2001, pp. 246–262.

[41] Valckenaers, P.; Sauter, J.; Sierra, C.; Rodriguez-Aguilar, J. Applications and environments for Multi-Agent Systems. *Autonomous Agents and Multi-Agent Systems 14*, 1 (2006), 61–85.

[42] van den Berg, J.; Patil, S.; Sewall, J.; Manocha, D.; Lin, M. Interactive navigation of multiple agents in crowded environments. In *Proceedings of the 2008 symposium on Interactive 3D graphics and games* (2008), I3D '08, ACM, pp. 139–147.

[43] Vigueras, G.; Orduna, J. M.; Lozano, M. A gpu-based multi-agent system for real-time simulations. In *Advances in Intelligent and Soft Computing* (2010), vol. 70-2010, pp. 15–24.

[44] Walsh, K.; Banerjee, B. Fast A* with iterative resolution for navigation. *International Journal on Artificial Intelligence Tools 19* (February 2010), 101–119.

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

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

[47] Yersin, B.; J, M.; Morini, F.; Thalmann, D. Real-time crowd motion planning: Scalable avoidance and group behavior. *Vis. Comput. 24*, 10 (Sept. 2008), 859–870.

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`.