OTIMIZAÇÃO DE PATHFINDING EM GPU

POR

ADÔNIS TAVARES DA SILVA

DISSERTAÇÃO DE MESTRADO

RECIFE

30/08/2013

Adônis Tavares da Silva

Otimização de Pathfinding em GPU

Dissertação apresentada como requisito parcial para a obtenção do título de Mestre, pelo Programa de Pós-Graduação em Ciência da Computação do Centro de Informática da Universidade Federal de Pernambuco

Orientador: Geber Lisba Ramalho, PhD

Co-Orientador: Veronica Teichrieb, PhD

RECIFE

30/08/2013

Adônis Tavares da Silva

Otimização de Pathfinding em GPU

Dissertação apresentada como requisito parcial para a obtenção do título de Mestre/Doutor, pelo Programa de Pós-Graduação em Ciência da Computação do Centro de Informática da Universidade Federal de Pernambuco

Aprovada em 30/08/2013

BANCA EXAMINADORA

Geber Lisboa Ramalho – Centro de Informática – CIn - UFPE

Charles Andryê Galvão Madeira – IMD - UFRN

Patricia Cabral de Azevedo Restello Tedesco – CIn - UFPE

RECIFE

Otimização de Pathfinding em GPU

RESUMO

Nos últimos anos, as unidades de processamento gráfico (GPU) têm apresentado um avanço significativo dos recursos computacionais disponíveis para o uso de aplicações não-gráficas. A capacidade de resolução de problemas envolvendo computação paralela, onde o mesmo programa é executado em diversos elementos de dados diferentes ao mesmo tempo, bem como o desenvolvimento de novas arquiteturas que suportem esse novo paradigma, como CUDA (Computed Unified Device Architecture), tem servido de motivação para a utilização da GPU em aplicações de propósito geral, especialmente em jogos. Em contrapartida, a performance das CPUs, mesmo com a presença de múltiplos núcleos (multi-core), tem diminuído, limitando o avanço tecnológico de diversas técnicas desenvolvidas na área de jogos e favorecendo a transição e o desenvolvimento das mesmas para a GPU.

Alguns algoritmos de Inteligência Artificial que podem ser decompostos e demonstram certo nível de paralelismo, como o pathfinding, utilizado na navegação de agentes através do jogo, têm sido desenvolvidos em GPU e demonstrado um desempenho melhor quando comparado à CPU. De modo semelhante, este trabalho tem como proposta a investigação e o desenvolvimento de possíveis otimizações ao algoritmo de pathfinding em GPU, por meio de CUDA, com ênfase em sua utilização na área de jogos, escalando a quantidade de agentes e nós de um mapa, possibilitando um comparativo com seu desempenho apresentado na CPU.

Palavras-chave: A-estrela, inteligência artificial, jogos, pathfinding, CUDA, GPU, GPGPU, agentes inteligentes

Otimização de Pathfinding em GPU

ABSTRACT

In recent years, graphics processing units (GPUs) have shown a significant advance of computational resources available for the use of non-graphical applications. The ability to solve problems involving parallel computing as well as the development of new architectures that supports this new paradigm, such as CUDA, has encouraged the use of GPU for general purpose applications, especially in games. Some parallel tasks which were CPU based are being ported over to the GPU due to theirs superior performance. One of these tasks is the pathfinding of an agent over a game map, which has already achieved a better performance on GPU, but is still limited. This paper describes some optimizations to a GPU pathfinding implementation, addressing a larger work set (agents and nodes) with good performance.

Keywords: pathfinding, artificial intelligence, games, CUDA, GPU, GPGPU, A-star, intelligent agents.

LISTA DE ILUSTRAÇÕES

LISTA DE TABELAS

SUMÁRIO

1 INTRODUÇÃO

Atualmente, a indústria de entretenimento movimenta bilhões de dólares; entretanto, ao contrário do que se esperava, não é o cinema, mas os jogos que estão no topo da lista. Jogos como *Call of Duty: Modern Warfare 3* ou Starcraft 2 atingiram níveis históricos de venda, assim como o montante gerado por filmes como *Avatar* [Humphries 2010]. Para que os jogos consigam movimentar essa quantidade exorbitante de capital financeiro, há muitos fatores envolvidos dentre os quais se encontram a jogabilidade (*gameplay*) e o realismo gráfico.

Apesar de o sucesso de um jogo estar popularmente atrelado à diversão proporcionada pelo mesmo, estabelecer o que o torna atrativo não é um conceito simples. Do ponto de vista do jogador, gráficos realistas não implica necessariamente em uma boa experiência de jogo. Entretanto, o conceito de imersão é algo que tem se tornado um senso comum na indústria de jogos para promover, avaliar e descrever a experiência vivenciada durante o jogo [Rouse 1999]. Existem diferentes níveis de imersão dentro de um jogo e os jogadores não devem ser impedidos de alcançar cada um desses diferentes níveis. Mesmo tendo a imersão como foco, os jogadores não esperam estar totalmente imersos durante todo o jogo. [Brown e Cairns 2004].

Com o intuito de melhorar a jogabilidade e, portanto, a imersão do jogador no ambiente do jogo, é necessária a utilização de técnicas de Inteligência Artificial (IA) [Mateas 2003]. Porém, o uso destas técnicas resulta em um maior custo de processamento, prejudicando o desempenho do jogo e obrigando que, muitas vezes, as melhores técnicas de IA sejam relegadas e não utilizadas. Assim, o produto final do jogo termina não tendo uma boa aceitação entre os jogadores pelo fato de não oferecer desafios em um nível de dificuldade suficiente [Csikszentmihalyi 2003].

Tentando introduzir novas experiências, diversos estudos têm sido realizados almejando um melhor desempenho, a fim de permitir a aplicação de técnicas de IA em jogos digitais. Muitas dessas técnicas são derivadas de técnicas de processamento gráfico, como Nível de Detalhe (*Level of Detail* – LOD) [Sery et.

Al. 2006] e processamento de multidões [Treuille et al. 2006]. Contudo, algumas limitações físicas tais como a alta frequência, o aumento excessivo de calor em uma pequena área, e a interferência eletromagnética, culminaram na necessidade de uma mudança na arquitetura de processadores, permitindo, assim, o surgimento de processadores comerciais com múltiplos núcleos, ou *multi-core* [AMD 2005]. Posteriormente, a ideia de utilizar múltiplos núcleos foi agregada a outros dispositivos, como as placas de vídeo.

A arquitetura atual das GPUs permite um alto poder de computação, atingindo mais de 20 vezes a potência de um processador de alta performance, como pode ser visto na comparação direta entre o processador Intel Core i7 e a placa de vídeo da série GTX400 [Owens et al. 2006]. Este alto poder computacional presente nas GPUs é capaz de ultrapassar inclusive a Lei de Moore. A principal diferença de desempenho entre CPUs e GPUs pode ser atribuída, principalmente às suas arquiteturas: as CPUs são otimizadas para alcançar alto desempenho na execução de códigos sequenciais, tendo, assim, muitas sub-tarefas dedicadas a dar suporte ao controle de fluxo e a dados em cache, enquanto que os processadores das GPUs são projetados para o processamento paralelo de instruções, seguindo o conceito de SIMD (*Single Instruction, Multiple Data*), possuindo, portanto, mais componentes dedicados ao processamento de instruções [Owens et al. 2006].

Aproveitando-se do desenvolvimento da arquitetura da GPU, diversas abordagens surgiram com o objetivo de aplicar o paralelismo destes processadores gráficos de alta potência para fins de programação de propósito geral. Apesar do avanço nesta área, nem todas as aplicações conseguem obter um ganho em desempenho quando migradas. Algumas características, tais como o grau de paralelismo de uma aplicação e o seu modelo de acesso à memória podem fazer com que uma aplicação demonstre um desempenho melhor na CPU. Além disso, a arquitetura das placas gráficas ainda apresenta limitações como a falta de uma hierarquia de cache, a presença de precisão de *double* apenas em placas mais recentes, e também os gargalos existentes na transferência de dados entre CPU e GPU.

Com a habilidade de utilizar as placas de vídeo para programação de propósito geral, aproveitando-se de seu paralelismo inerente, as técnicas de IA podem ser trabalhadas de forma menos onerosa na CPU, permitindo que os outros componentes de um jogo sejam processados sem sofrer grandes perdas de performance. A fim de que esse novo paradigma se desenvolvesse, fez-se necessária a criação de novas arquiteturas e plataformas, como CUDA (*Compute Unified Device Architecture* [CUDA 2010]). Desenvolvida pela NVIDIA, CUDA tem simplificado o gerenciamento da GPU como um dispositivo para computação paralela de aplicações de propósito geral, e permitido que estudos progridam, inclusive na área de Inteligência Artificial para Jogos.

É tomando como base o avanço da área de IA aplicada a Jogos na GPU que este trabalho tem como objetivo apresentar o ganho de performance alcançado com CUDA no processamento do algoritmo de pathfinding conhecido por A* (A-estrela) e também evidenciar como algumas pequenas alterações, fazendo um melhor uso da arquitetura de CUDA, podem proporcionar ganhos ainda maiores ao processamento de uma das atividades mais básicas de um jogo: calcular o melhor caminho entre dois pontos, conhecido como *pathfinding*.

O restante deste trabalho está organizado da seguinte forma: no Capítulo 2 são apresentados os conceitos gerais referentes ao problema da busca e busca pelo melhor caminho, incluindo sua aplicação em jogos.

O Capítulo 3 apresenta os conceitos básicos da utilização da GPU para programação de propósito geral e de CUDA, arquitetura desenvolvida para suportar esse tipo de aplicação.

No Capítulo 4 é exposta a metodologia empregada na realização deste trabalho, incluindo o estado da arte das metodologias utilizadas para adequação de técnicas de Inteligência Artificial aplicadas a Jogos em arquiteturas de processamento paralelo, o procedimento adotado para a análise da performance destas técnicas, além dos detalhes da implementação e dos experimentos realizados.

No Capítulo 5 são evidenciados os resultados obtidos com a implementação e as otimizações propostas, com seus julgamentos e comentários

e, então, é realizada uma comparação dos resultados alcançados com os relatados na literatura.

Por fim, o Capítulo 6 apresenta o fechamento do trabalho, ressaltando os resultados e as contribuições realizadas, e também propondo trabalhos futuros a serem realizados dentro da área abordada.

2 CARACTERIZAÇÃO DO PROBLEMA

O desenvolvimento de uma Inteligência Artificial que se assemelhasse ao comportamento humano nem sempre esteve entre as preocupações dos desenvolvedores de jogos. Restrita por muito tempo apenas ao meio acadêmico ou a pesquisas em outras áreas, principalmente na robótica, desde que foi inserida neste novo contexto, a IA tem fomentado diversas pesquisas nesta área e tem desempenhado um papel fundamental dentro dos jogos, influenciando-os desde o processo de concepção e design do jogo à etapa de marketing e divulgação. As seções seguintes descrevem todo o contexto associado a este trabalho, como a presença da Inteligência Artificial em jogos e o problema da busca por um melhor caminho.

2.1 INTELIGÊNCIA ARTIFICIAL EM JOGOS

Em um sentido mais amplo, muitos dos jogos eletrônicos existentes possuem alguma forma de inteligência artificial. Entretanto, a inclusão de técnicas de IA em jogos se deu praticamente por volta de 1970. Até então, os jogos tinham o propósito de entreter duas pessoas; tornar mais competitivo. Com a inclusão de um modo single player, objetivando atrair um maior público e, consequentemente, um aumento nos lucros das indústrias [Bourg e Seemann 2004], os jogos passaram a simular o comportamento humano, fazendo uso de técnicas já presentes no meio acadêmico.

Alguns jogos que se seguiram, mudaram o cenário mundial de jogos eletrônicos e introduziram, aos poucos, novos paradigmas e características que têm despertado o interesse de pesquisadores até hoje. Desde os mais antigos, como Space Invaders [Space Invaders 1978], um dos primeiros a apresentar entidades inteligentes, cujo objetivo do jogo consistia em destruir a maior quantidade de alienígenas possível e Pac-man [Pac-man 1980], que ficou mundialmente conhecido e introduziu a ideia de utilizar movimentos padronizados, diferenciando a forma como os inimigos caçavam o jogador, até os mais atuais, como Left 4 Dead 2 [Left 4 Dead 2 2009], em que a atmosfera sonora do jogo, as

hordas de zumbis, os companheiros do jogador entre outros aspectos são gerenciados pela IA, e FIFA Soccer 10 [FIFA Soccer 2010 2009], onde a maioria dos jogadores apresentam personalidade e habilidade diferentes, sendo também possível visualizar a evolução e a inclusão cada vez maior da Inteligência Artificial dentro dos jogos.

Embora a Inteligência Artificial seja comumente associada com a presença de personagens ou aspectos de inteligência humana, ela não precisa estar sempre personificada dentro do jogo. Além de proporcionar uma melhor jogabilidade com o uso de NPCs, a IA também possui outras aplicações. Em [Galstyan et al. 2003], a IA é utilizada na alocação de recursos e os agentes são recompensados quando utilizam um recurso sem exceder sua capacidade ou punidos caso contrário. Cada agente utiliza um conjunto de estratégias para decidir qual recurso escolher e um esquema de aprendizagem por reforço para atualizar as estratégias. Em [Darken and Paull 2006], é proposta uma solução para o problema da utilização efetiva de abrigos em um jogo de combate com um ambiente dinâmico, onde cada oportunidade de se proteger pode ser destruída ou criada no decorrer do jogo. Outra aplicação de IA em jogos é observada em [Andrade et al 2005], em que é proposta uma nova técnica baseada em aprendizagem por reforço para controlar automaticamente o nível do jogo, adaptando-o ao nível do jogador, garantindo um bom balanceamento do jogo.

Em jogos mais recentes, devido à demanda de mercado, existe uma grande riqueza nos detalhes das cenas, ainda que estas representem batalhas épicas, ambientes urbanos movimentados ou o espaço. Observando a cena na Figura 2-1, é possível perceber os avanços realizados na área de gráficos 3D em tempo real, complexidade do mundo e personagens e também em efeitos de pós-processamento. Entretanto, apesar da notável evolução dos gráficos dos jogos, a simulação desses ambientes utilizando técnicas confiáveis e sofisticadas de inteligência artificial ainda não é prioridade na indústria [Nereyek]

A inteligência artificial dentro de um jogo não está totalmente independente de outros aspectos como detecção de colisão, animação dos personagens e física, pelo fato de que a arquitetura do jogo em si é inerentemente sequencial, ou seja, cada frame depende diretamente do frame anterior. Por isso, normalmente ela é

incorporada aos jogos apenas no final do processo de desenvolvimento [Nereyek]. Outras razões favorecem isso: para as indústrias, como o fim principal de um jogo é a sua venda, do ponto de vista do marketing, é mais fácil vender um jogo que tenha bons gráficos (modelos humanos realistas, uma física coerente ou um pôr-do-sol bonito) do que um jogo que tenha inimigos com um comportamento razoavelmente inteligente. Entretanto, tipicamente, o sentimento do usuário é que quanto melhor a inteligência artificial, melhor o jogo. Seja ela um tanque em um jogo de estratégia em tempo real ou um alienígena num jogo de ação em primeira pessoa.

Figura 2-1: Demonstração de gameplay do jogo Battlefield 4 para o console PS4.

O impacto de uma IA ruim provoca uma série de problemas de game design [Mateas 2003] que terminam quebrando toda a imersão proporcionada pela criação de cenas graficamente perfeitas ou diminuindo o *replay-value* de um jogo por não apresentar desafios com um nível de dificuldade suficiente para o jogador [Csikszentmihalyi 1990]. O objetivo principal da IA em jogos e também seu grande desafio não é oferecer o melhor comportamento para vencer o jogador, mas se preocupar com duas propriedades dentro dos jogos [Champandard 2003]:

- Diversão: a Inteligência Artificial desperta as habilidades nos jogadores, proporcionando desafios e testando diferentes

habilidades à medida que aumenta o grau de dificuldade. Ela também pode se utilizar de aspectos emocionais para aumentar a diversão ou criar atmosferas assustadoras para provocar medo ou espanto no jogador.

- Realismo: é possível ampliar a imersão do jogador, provendo personagens que executem suas ações de forma imperceptível e logicamente plausível ao usuário.

Uma das áreas da IA em jogos que fomentam o estudo na academia diz respeito à movimentação dos NPCs. Para que os agentes se locomovam, é necessário que essa movimentação seja feita de forma inteligente – sem ficar preso em algum obstáculo ou bater em uma árvore pelo caminho, pegar o melhor caminho para alcançar o destino, etc. Dado os recursos computacionais existentes limitados pela CPU, fazer uma movimentação sofisticada é muito difícil. Além disso, alguns fatores como ambientes complexos com terrenos dinamicamente transformados, milhares de unidades que devem ter sua movimentação calculada em paralelo, entre outros, contribuem para uma IA não satisfatória. Na seção seguinte, a movimentação dos NPCs será discutida detalhadamente, dando ênfase no problema da busca pelo melhor caminho, que tem sido bastante discutido entre os pesquisadores e também é alvo deste trabalho.

2.2 BUSCA PELO MELHOR CAMINHO

A utilização de algoritmos de busca em jogos para sistema multiagentes tornou-se particularmente importante com a popularização de jogos estratégia em tempo real (RTS) [Tatarchuk 2005]. *Warcraft, Command & Conquer e Age of Empires* foram grandes sucessos mundiais por possibilitaram o controle de múltiplas unidades ao mesmo tempo. Esse tipo de jogo só se tornou computacionalmente possível com o avanço das técnicas de busca pelo melhor caminho. É o típico caso de game design focado em uma feature computacional

Normalmente, unidades controladas pelo computador se utilizam de uma navegação em *waypoints* baseada em uma máquina de estados simples, e não de algoritmos de busca pelo melhor caminho. Essa restrição já é utilizada exatamente

pelo fator performance de tais algoritmos e cria sérios limites ao design do jogo. Dessa maneira, ainda há uma necessidade grande na otimização desses algoritmos. Essa otimização pode ser feita também ao se utilizar recursos adicionais para a execução desses algoritmos, como as placas gráficas. Como o foco desse trabalho é exatamente essa abordagem, faz-se necessário uma apresentação geral dos algoritmos clássicos de busca do melhor caminho, para posterior debate sobre como os mesmos podem se beneficiar de uma eventual implementação para execução nas G*raphical Processing Units* (GPUs).

Os algoritmos de busca necessitam de um espaço de busca. No problema da busca pelo melhor caminho, a representação do espaço, seja ele bidimensional ou tridimensional, tem um impacto enorme na performance, nos requisitos de memória dos sistemas de pathfinding e na qualidade do caminho encontrado, determinando, muitas vezes, o sucesso ou o fracasso de um algoritmo de pathfinding [Tozour 2003].

Duas considerações que devem ser feitas na escolha da representação do espaço de busca são o desempenho e o *overhead* de memória. É preciso escolher uma representação que utilize uma quantidade razoável de memória e que permita realizar a busca o mais rápido possível. Além disso, deve-se verificar se ela suporta as diferentes necessidades dos agentes, visto que, em um jogo, é muito difícil ter todos os agentes navegando da mesma maneira. É preciso também levar em consideração a forma como a representação é inicializada: algumas delas são criadas derivando-se automaticamente a partir do mundo ou a partir de uma malha física de colisão.Em muitos casos, a fim de diminuir o espaço de busca no jogo, o mapa é particionado e simplificado [Tozour 2003]. O *pathfinder* utiliza, então, essa representação simplificada do mapa para determinar o melhor caminho. A representação do espaço de busca pode ser dividida da seguinte maneira: rotas claras que o agente pode fazer, ou superfícies livres de obstáculos (nas quais um agente pode andar livremente). As representações mais comuns são divisões de grafos com *waypoints*, malhas de navegação (*navigation meshes*) ou *grids* regulares [Tatarchuk, 2005]. Para facilitar a explicação dos mesmos utilizaremos uma parte do mundo virtal do jogo World of Warcraft mostrado na Figura 2-2.

Figura 2-2: Ilha do mundo virtual utilizada em World of Warcraft. Servirá de exemplo para as explicações desse capítulo.

Waypoints é uma coleção de nós com arestas os ligando, ou seja, um grafo. Os nós representam pontos de navegação e as arestas os caminhos livres por onde um agente pode passar. Qualquer um dos pontos de navegação desse grafo deve ser acessível diretamente por outro ponto ou indiretamente por um conjunto de pontos. Essa abordagem apesar de intuitiva e simples de entender cria problemas sérios na produção de um jogo. Quanto mais pontos o grafo tem, mais complexa a malha e, consequentemente, maior o custo na realização da busca. Por outro lado, grafos com poucos nós limitam os possíveis caminhos, já que os agentes andam exclusivamente pelas arestas de um ponto a outro, criando comportamentos pouco realistas para os padrões atuais de jogos. Na Figura 2-3 mostramos uma possível configuração de *waypoints* para a ilha de *World of Warcraft*

Figura 2-3: Possível configuração de *waypoints* na ilha de *World of Warcraft*.

Navigation meshes, por outro lado, são formados por um conjunto de polígonos convexos que representam áreas livres por onde o agente possa andar [Atanasov, 2010]. Dessa maneira, o agente fica livre para caminhar por um polígono. Os polígonos são convexos para garantir que o agente possa andar do centro de um deles para o centro de algum outro sem problemas. Ao mesmo tempo, o agente fica livre para caminhar por qualquer espaço dentro do polígono. A Figura 2-4 mostra uma malha de navegação criada a partir da ilha de *World of Warcraft*.

Há ainda os *grids* regulares, que são a forma mais simples de se representar um espaço de busca, no formato de quadrados, retângulos, hexágonos ou triângulos. Populares entre jogos 2D, especialmente em RTS, os *grids* não podem ser utilizados em um jogo 3D sem passar por algumas modificações. Uma de suas limitações está no fato de que são necessários muitos *grids* para representar um mundo de um jogo, o que requer mais memória e pode deixar o pathfinding significativamente lento. Mesmo assim, esse tipo de estrutura permite

determinar em tempo constante (O(1)) a qual *tile* determinada coordenada pertence. Algumas otimizações existentes propõem a criação de um caminho mais suave e curvo utilizando *splines* [Rabin 2000].

Figura 2-4: Malha de Navegação para a ilha de World of Warcraft.

Dada uma representação, um caminho é uma lista de nós que levam do ponto de partida até o ponto final da busca. Existem diversos algoritmos de busca, os mais usados são a busca por largura, busca por profundidade, Dijkstra e A* (pronuncia-se A estrela).

Todos os algoritmos apresentados podem usar como base qualquer uma das representações. Além disso, os mesmos possuem uma base em comum que é a utilização de duas listas: uma lista de nós abertos e uma lista de nós fechados, chamadas de lista aberta e lista fechada para facilitar a identificação. A lista aberta contém os nós promissores que ainda não foram avaliados, enquanto a lista fechada contém os nós já visitados. Cada nó na lista aberta é avaliado com relação ao objetivo. Caso não seja o nó objetivo é usado para inclusão de novos nós na

lista aberta e movido para a lista fechada. A estrutura geral pode ser descrita em alguns passos simples:

1. Criação do nó inicial e inclusão do mesmo na lista aberta.
2. Enquanto a lista aberta não estiver vazia fazemos:
    a) Tiramos um nó da lista aberta, chamaremos de nó atual.
    b) Se o nó atual é o objetivo, paramos a busca.
    c) Criamos novos nós que estão diretamente ligados, adjacentes, ao nó atual e incluímos os mesmos na lista aberta.
    d) Colocamos o nó atual na lista fechada.

Dado esse algoritmo base, a grande diferença entra as abordagens é como escolher o nó atual da lista aberta e também como avaliar os nós adjacentes em termos de potencial com relação ao objetivo (que possui a maior chance de ser o objetivo ou de estar mais perto do objetivo). Dessa maneira, classificamos os algoritmos entre direto e indireto. Os algoritmos indiretos – busca em largura e busca em profundidade - são aqueles que utilizam mecanismos triviais de avaliação dos nós abertos. Já os algoritmos indiretos utilizam o que chamamos de heurísticas para avaliação dos nós abertos.

A principal característica dos algoritmos indiretos é o fato de não levar em consideração o custo do caminho, mas são efetivos se não existe nenhum custo associado (como é comum nos casos que se escolhe a representação por grid). Além disso, esses algoritmos criam caminhos não realistas com muitas curvas desnecessárias. Sendo assim, só são utilizados em casos bem específicos de buscas para agentes que não estão no campo de visão do jogador.

Breadth First Search (Busca em Largura - BFS) trata o mundo virtual como um grande grafo conexo. A partir daí, cada nó conectado ao nó atual é expandido e, em seguida, cada nó conectado a esses novos nós são expandidos. Se existir um caminho, o BFS conseguirá encontrá-lo; se existirem vários, o caminho encontrado será o de menor profundidade.

Já o *Depth First Search* (Busca em Profundidade – DFS) é o contrário do BFS, de maneira que todos os filhos de cada nó são visitados antes de partir para os outros nós do mesmo nível, estabelecendo um caminho linear para o destino.

Apesar de ter uma grande chance de encontrar uma solução após percorrer um pequeno número de nós, esse tipo de busca, no pior caso, pode percorrer todos os nós do grafo até encontrar a solução.

Na abordagem direcionada, existe uma preocupação em avaliar o progresso da busca a partir de todos os nós adjacentes antes de escolher seguir algum, ou seja, o custo de chegar a um nó adjacente é avaliado a cada etapa. O custo nos mapas dos jogos normalmente está associado à distância entre os nós. Muitos dos algoritmos dessa categoria conseguirão encontrar uma solução para o problema, mas pode ser que não seja a mais eficiente, isto é, o menor caminho [Graham 2003].

As principais estratégias para os algoritmos de pathfinding direcionado:

- Um custo uniforme da busca, representado por g(n), onde n é o nó atual, modifica a busca de forma que o nó com menor custo seja sempre escolhido como o próximo nó a ser visitado. Apesar de minimizar o custo do caminho, em alguns casos pode se tornar bastante ineficiente;
- Uma busca heurística, representada por h(n), corresponde a uma estimativa do custo do próximo nó para o nó destino. Isto diminui o custo da busca consideravelmente, mesmo não sendo completo ou ótimo. Entretanto, quando a heurística não é consistente, o pathfinding pode se tornar altamente ineficiente [Björnsson 2006].

Os dois algoritmos mais conhecidos dentro da categoria de pathfinding direcionado normalmente utilizam uma ou mais das estratégias citadas. O Dijkstra [LaValle 2006] utiliza apenas a estratégia de custo uniforme para encontrar o caminho ótimo de um agente, enquanto que o A* se utiliza de ambas as estratégias, minimizando o custo total do caminho. Apresentando uma performance melhor que o Dijkstra e encontrando um caminho ótimo, o A* consagrou-se como o algoritmo mais utilizado no pathfinding, sendo o responsável por diversas pesquisas neste domínio. Quando comparados com os algoritmos de Busca em largura e Busca em Profundidade, o Dijkstra e o A*, apesar de necessitarem de mais iterações na busca, são capazes de encontrar um caminho ótimo, enquanto que o BFS e DFS não.

Em alguns sistemas de pathfinding, uma técnica conhecida como *Quick Paths* é caracterizada por executar dois algoritmos de busca: enquanto o agente se locomove utilizando um pathfinding mais rápido e menos preciso, outro pathfinding mais complexo é realizado para computar um caminho ótimo para o objetivo. Assim, ao passo que o caminho ótimo é encontrado, o caminho rápido é conectado a ele, criando a ilusão de que o agente encontrou o melhor caminho desde o começo.

Na maioria dos casos, jogos comerciais utilizam o algoritmo A* sobre a estrutura de dados *navigation mesh* (normalmente chamada de "*navmesh*"). Esse algoritmo de busca pelo melhor caminho é utilizado para as unidades controladas pelo jogador. Esse mesmo algoritmo também é um dos principais utilizados na academia para todo tipo de problema que envolve busca de melhor caminho. Isso não é por acaso. A* é o algoritmo com o melhor custo e melhor exatidão entre todos. Apesar de alguns experimentos realizados com o A* em GPU [Tatarchuk 2005], apresentarem um ganho em relação a CPU (speedup de 24x), existem pequenas melhorias que podem ser acrescentadas ao A* [3], como a utilização de uma heurística mais adequada ao contexto do jogo. É possível, também, utilizar algoritmos que levem em consideração informações obtidas pelos outros agentes e que diminuam o custo de calcular o caminho, como é observado nos sistemas multiagentes [Atanasov 2010].

2.3 PATHFINDING COM A*

O A* (A-estrela) é um algoritmo de pathfinding direcionado bastante popular na comunidade acadêmica que vem sendo utilizado, desde 1968, na resolução de diferentes problemas, entre eles o de *path-planning*. Tratando-se de um algoritmo direcionado, ao invés de realizar uma busca cega pelo caminho, ele acessa a melhor direção a ser explorada, inclusive algumas vezes voltando atrás para encontrar novas alternativas. Isto significa que o algoritmo não apenas encontra um caminho entre dois pontos, se ele existir, mas irá encontrar o menor caminho existente relativamente rápido.

Para que o A* possa ser executado, é necessário que o mapa do jogo esteja preparado ou pré-computado, ou seja, é preciso particionar o mapa em diferentes pontos chamados nós. Estes nós podem ser representados como pontos de visibilidade (*waypoints*) ou polígonos de uma malha de navegação (*navigation mashes*), e são utilizados para armazenar o progresso da busca. Além disso, cada nó possui três atributos que, somado a sua posição, conseguem representar o mapa do jogo: o custo total, o custo de se chegar ao nó atual partindo de uma origem e um custo associado a uma heurística, conhecidos por *f*, *g* e i respectivamente. O propósito desses três atributos é quantificar o quão promissor é um caminho até nó atual. Estes atributos são definidos a seguir:

- *g* é o custo de se chegar ao nó atual partindo de uma origem, ou seja, a soma de todos os valores no caminho entre a origem e o nó atual;
- *h* está associada à heurística, uma estimativa de custo do nó atual para o objetivo. Existem muitas heurísticas que estimam de maneira diferente o custo restante para o destino e a escolha da função heurística mais adequada é, hoje, motivo de diversas pesquisas na área. Algumas das mais conhecidas são: distância de Manhattan, Diagonal e Euclidiana, sendo esta última a distância em linha reta entre os nós envolvidos;
- *f* é a soma atual de *g* e *h* e é a melhor estimativa do custo do caminho se este for seguir pelo nó atual. Em geral, quanto menor o valor de f, mais eficiente será o caminho.

Em sua execução, o A* mantém duas listas de nós, chamadas de nós Abertos e Fechados, para os nós não visitados e visitados respectivamente [Stout 2000] No início do algoritmo, a lista de nós Fechados está vazia e a de nós Abertos possui apenas o nó origem. A cada iteração, o algoritmo remove o nó mais promissor de Abertos (aquele que apresenta menor custo total *f*) para ser examinado. Se este nó não for o nó destino, os nós vizinhos ao nó atual são examinados. Se forem novos e não tiverem sido visitados, estes nós são colocados na lista de nós Abertos. Se já estiverem na lista de Abertos, a informação contida nesse nó é atualizada com o novo custo se este for menor do que o já existente. Se já pertencerem à lista de nós Fechados, são ignorados, visto que ele e todos os seus vizinhos já foram visitados. Se a lista de nós não visitados ficar vazia antes do

nó destino ser encontrado, significa que não existe caminho entre a origem e destino passados. Para computar o caminho quando for encontrado, cada nó possui uma referência para seu pai, de onde ele veio. Dessa forma, é possível reconstruir o caminho de volta. O algoritmo em pseudocódigo é mostrado no Algoritmo 2-1.

Algoritmo 2-1. A* *Pathfinding*.

1. Let P = starting point.
2. Assign *f*, *g* and *h* values to P.
3. Add P to the Open list. At this point, P is the only node on the Open list.
4. Let B = the best node from the Open list (i.e. the node that has the lowest fvalue).
a. If B is the goal node, then quit – a path has been found.
b. If the Open list is empty, then quit – a path cannot be found
5. Let C = a valid node connected to B.
a. Assign *f*, *g*, and *h* values to C.
b. Check whether C is on the Open or Closed list.
i. If so, check whether the new path is more efficient (i.e. has a lower *f*-value).
1. If so update the path.
ii. Else, add C to the Open list.
c. Repeat step 5 for all valid children of B.

6. Repeat from step 4.

Apesar de ser amplamente utilizado e difundido entre o meio, existem algumas limitações impostas ao A* e muitas delas estão associadas ao pré-processamento do mapa, o que torna possível a implementação de um *pathfinding* mais complexo em tempo real [Trindade et al 2008].

Tais problemas incluem a incapacidade de algumas *engines* de jogos em lidar com ambientes dinâmicos e produzir movimentos com alto grau de realismo. Isto vem desde os estágios de pré-processamento, quando é feita a criação de uma representação estática do mapa e produzidos os nós por onde os agentes irão navegar. Muitas vezes um agente, ao encontrar um obstáculo dinâmico no meio do caminho computado, continua pensando que deve prosseguir atravessando o obstáculo.

Outro problema está ligado a suavização da movimentação do agente, isto é, uma movimentação mais condizente com a realidade, quando um agente se

move em linha reta de um nó para outro do caminho. Este tipo de problema acontece devido ao dilema existente entre ter um *tradeoff* entre velocidade (quanto menos nós na busca, melhor) e movimentos realistas (quanto mais nós, maior o realismo da movimentação). Uma solução proposta [Rabin 2000] consiste na utilização de *splines* entre os diferentes nós para suavizar o movimento ao longo do caminho.

Uma das maiores limitações impostas pelo algoritmo de *pathfinding* A* está relacionada com o consumo de grande quantidade de recursos da CPU, a partir do momento que o espaço de busca é muito grande (muitos nós para procurar) como é o caso de grande parte dos jogos atuais, especialmente os de RTS e FPS. Quando isso é expandido para a esfera de milhares de agentes ou quando um agente movimenta-se de uma extremidade a outra do mapa, pode haver um leve *delay* no jogo, o que definitivamente não é desejado. Isso pode se agravar ainda mais com a adição de obstáculos dinâmicos. Executar o A* toda vez que um agente encontrar-se com um obstáculo causaria um consumo ainda maior dos recursos disponíveis.

Diversos esforços [Holte et al 1996; Silver 2005; Wang and Botea 2008] têm sido realizados a fim de superar as limitações existentes no *pathfinding* com A*, inclusive na esfera de multiagentes. Em [Stentz 1994], por exemplo, Stentz propõe um novo algoritmo, o D* (abreviação de A* dinâmico – *Dynamic* A*), que lida com o fato de que o custo dos nós mudam ao passo que o agente se move no mapa e apresenta uma abordagem para modificar a estimativa dos custos em tempo real. Entretanto, essa abordagem consome ainda mais recursos da CPU e força um limite da quantidade de objetos dinâmicos que podem ser introduzidos no jogo.

A dependência de algoritmos de *pathfinding* como o A* impede o avanço da indústria de jogos em alguns aspectos. Este trabalho procura minimizar o custo de um A*, tornando-o mais eficiente com uma implementação em GPU que utilize os recursos disponíveis nessa nova plataforma de desenvolvimento.

3 UNIDADES DE PROCESSAMENTO GRÁFICO

A unidade de processamento gráfico (GPU), conhecida também como VPU ou unidade de processamento visual, é um tipo de multiprocessador especializado no processamento de gráficos. Este tipo de dispositivo pode ser encontrado em computadores pessoais, videogames, sistemas embarcados ou até em celulares, seja situado na placa de vídeo ou diretamente conectado à placa-mãe.

A arquitetura das atuais GPUs possibilita tanto um alto poder computacional quanto uma grande velocidade de barramento de dados, superando as CPUs comuns nestes aspectos [Owens et al 2005] , conforme pode ser observado na Figura 3-1.

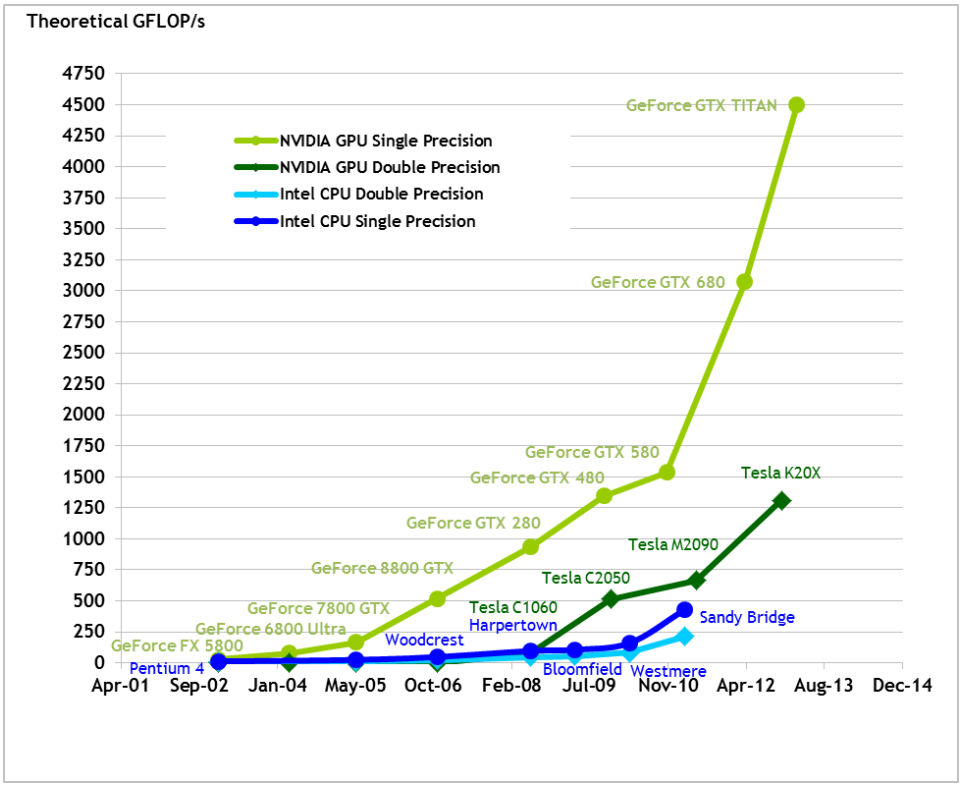

Figura 3-1: Comparação entre modelos de CPU e GPU em termos de operações com ponto flutuante por segundo (GFLOPS/s).

Em geral, a capacidade computacional das GPUs, medida pelas métricas de desempenho gráfico, tem alcançado uma taxa média de 1.7x (pixels/segundo) a 2.3x (vértices/segundo). Este crescimento supera a Lei de Moore aplicada aos microprocessadores tradicionais e mostra que o desempenho dos hardwares

gráficos tem praticamente dobrado a cada seis meses, como pode ser visto na Figura 3-2.

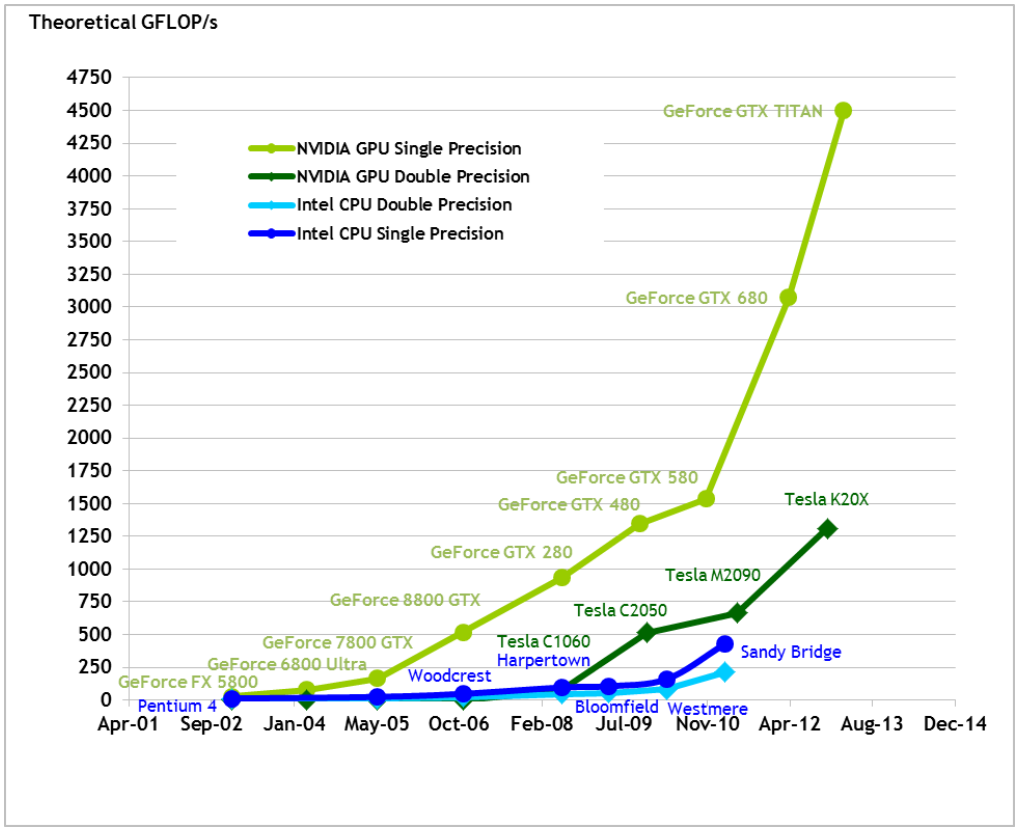

Figura 3-2: Evolução das GPUs em termos de largura de banda.

A grande diferença no desempenho de CPUs e GPUs pode ser atribuída às diferenças existentes na arquitetura de ambas: como as CPUs são otimizadas para obter alta performance na execução de código sequencial, muitos dos transistores são dedicados a suportar tarefas não computacionais, como controle de fluxo e cache de dados. Por outro lado, as GPUs são processadores orientados a execução paralela de instruções, sendo otimizados para processar operações sobre vetores SIMD (*Simple Instruction, Multiple Data*), permitindo a inclusão de mais transistores dedicados ao processamento de dados [Owens et al 2005]. Esta diferença pode ser mais bem visualizada na Figura 3-3.

Existem casos em que a performance em CPU é superior a de uma GPU. Isto se dá principalmente devido ao grau de paralelismo da aplicação e ao seu modelo de acesso a memória. Como a GPU consegue lidar melhor com problemas

que demandam uma grande quantidade de operações sobre dados, de modo que a mesma operação seja realizada sobre diferentes elementos, não existe um controle de fluxo mais sofisticado e as latências no acesso à memória são disfarçadas através da grande vazão de cálculos.

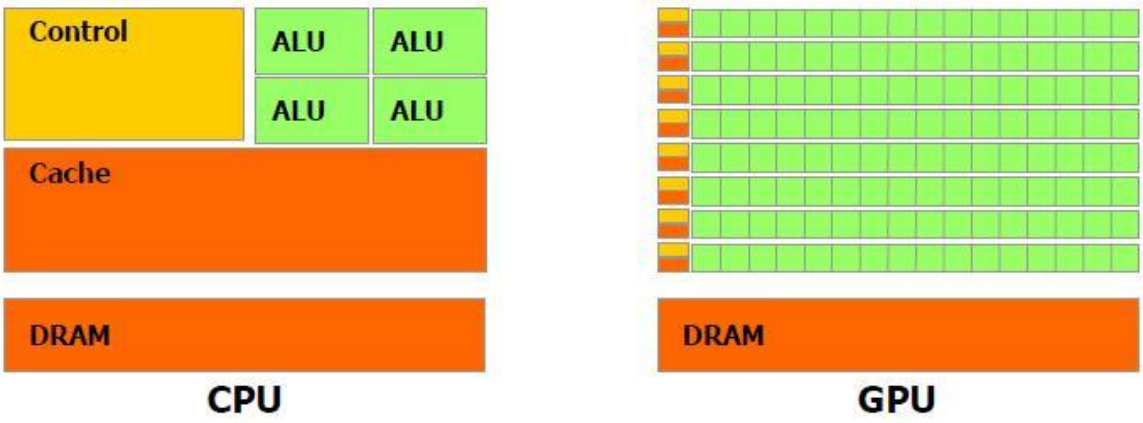

Figura 3-3: Arquitetura de uma CPU vs. de uma GPU

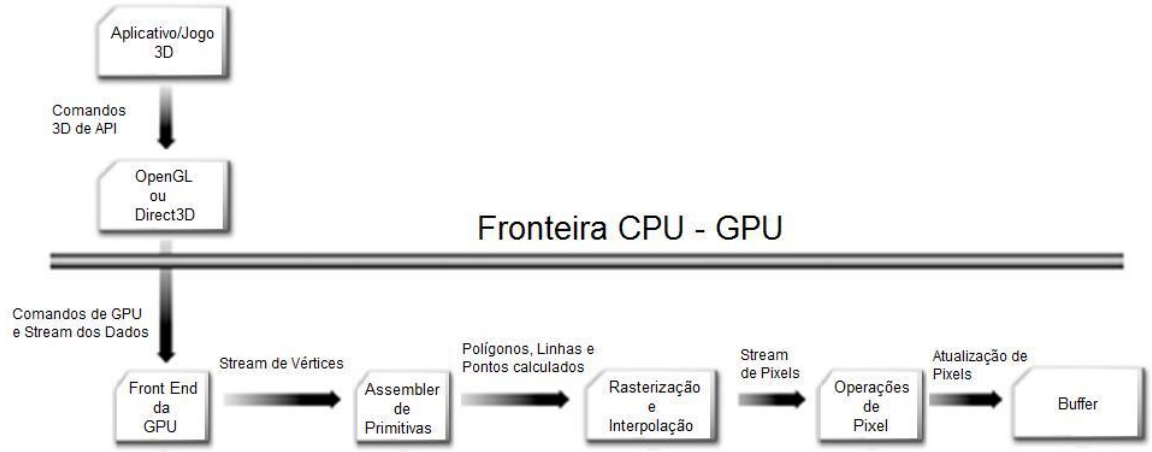

Figura 3-4: Pipeline Gráfico 3D.

Concebida com o intuito de realizar processamento gráfico, como o próprio nome sugere, a GPU possui uma sequência de estágios bem definidos, conhecida como *pipeline* gráfico. No *pipeline*, cada estágio é responsável pela execução de uma determinada tarefa, como cálculos de iluminação e transformação de coordenadas. A arquitetura da GPU foi projetada de forma que as operações contidas no pipeline sejam realizadas simultaneamente, segundo a Figura 3-4. Posteriormente, algumas novas funcionalidades foram adicionadas às GPUs,

possibilitando modificar o processo de renderização do *pipeline*, através dos *shaders* [Watt 2000], permitindo a adição de efeitos gráficos customizados e difíceis de programar utilizando o *pipeline* antigo. O novo *pipeline* com a adição dos componentes programáveis é mostrado na Figura 3-5.

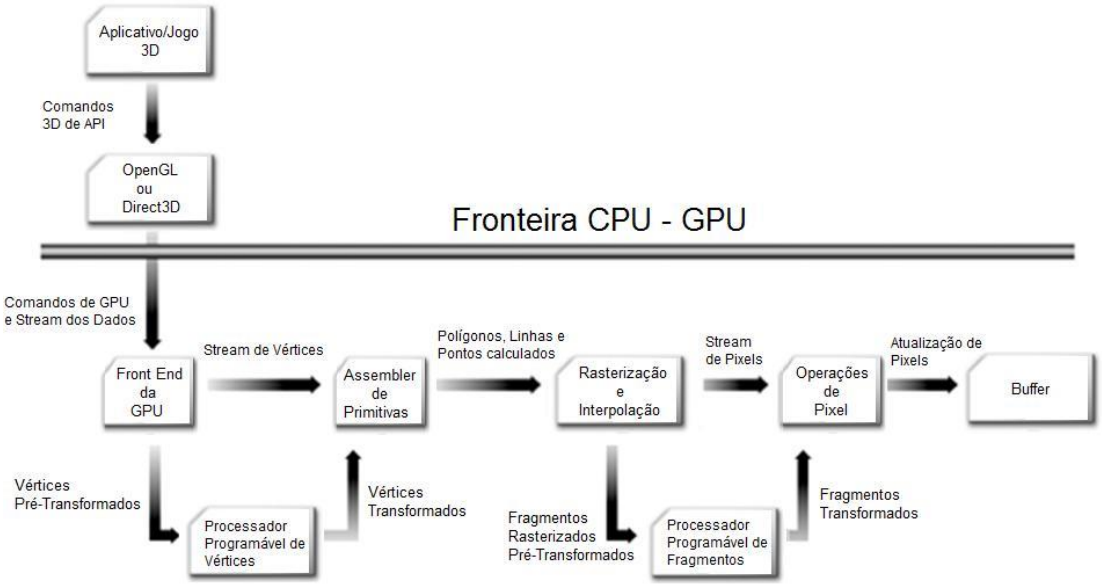

Figura 3-5: Pipeline Gráfico com adição dos componentes programáveis.

Atualmente, a NVIDIA e ATI/AMD, grandes empresas de dispositivos gráficos, desenvolveram os *frameworks* (SDK – *Software Development Kit*) CUDA e Stream respectivamente [ATI Stream Technology, 2010]**,** que possibilitam a utilização das GPUs para programação de propósito geral, mudaram completamente a forma como enxergamos a GPU: tornou-se viável converter um algoritmo de CPU para ser executado em GPU, obtendo uma performance melhor. Essas novas funcionalidades são detalhadas nas seções seguintes, afirmando a utilização das placas gráficas para aplicações de propósito geral e apresentando alguns dos *frameworks* existentes.

3.1 APLICAÇÕES DE GPGPU

Durante a última década, as placas aceleradoras gráficas passaram de um simples dispositivo de visualização 3D a um dispositivo *multi-core* de programação com propósito geral. Diversos trabalhos têm tentado explorar o enorme poder

computacional que as GPUs tem apresentado, baseado em OpenGL e DirectX [Bustos et al 2006; Chiara et al 2004; Micikevicius 2004]. Entretanto, observou-se que expressar algoritmos gerais em termos de texturas e operações 3D era bastante custoso. Além disso, a ausência de outros tipos de dados que não sejam números com ponto flutuante constituiu um empecilho para diversas aplicações.

Dessa forma, com o advento dos frameworks desenvolvidos pelas empresas de dispositivos gráficos, juntamente com o grande avanço das placas gráficas, surgiu uma nova forma de programar utilizando a arquitetura da GPU. Este modelo, conhecido como GPGPU (Programação de Propósito Geral em Unidades de Processamento Gráfico – *General-Purpose Computing on Graphics Processing Unit*), propõe a utilização da placa de vídeo não apenas para implementação de aplicativos gráficos, ampliando o alcance da GPU para programação de propósito geral abordando, praticamente, todas as áreas da computação.

Desde então, tem havido um aumento constante de pesquisas no domínio da utilização de hardwares gráficos para programação de propósito geral. Alguns algoritmos de segurança e de busca, caso demonstrem certo grau de paralelismo, podem ser implementados em GPGPU e otimizados, utilizando, ao máximo, o poder computacional paralelo da GPU. Yang e Welch em [2003] mostram como realizar uma rápida segmentação e suavização de imagens usando placas gráficas, implementando funções de erosão e dilatação na GPU. Eles afirmaram que sua implementação em GPU é cerca de 30% mais rápida que em CPU. Semelhantemente, Larsen and McAllister [2001] descrevem uma técnica para implementar multiplicação de matrizes muito grandes de forma eficiente utilizando o hardware gráfico.

Ainda que seja notório o avanço em GPGPU, existem alguns desafios mesmo para os problemas que conseguem ser bem mapeados para a GPU. A falta de uma hierarquia de cache evoluída, da maneira que existe na CPU, impede que determinadas aplicações alcancem um desempenho superior nas placas gráficas. Apesar de as versões mais recentes de CUDA já apresentarem precisão de double, isso é algo que até então não existia e que em aplicações de propósito geral é de extrema importância. Além disso, a transferência de dados entre CPU e

GPU não ocorre de forma tão eficiente, favorecendo o aparecimento de possíveis gargalos.

3.1.1. CUDA

CUDA [CUDA 2010], ou *Computed Unified Device Architecture*, diz respeito à arquitetura paralela de propósito geral lançada em Novembro de 2006 pela NVIDIA – com um novo modelo de programação e um conjunto de instruções – possibilitando, desde a série G80 de placas gráficas, a solução de diversos problemas computacionalmente complexos de maneira mais eficiente que na CPU. A arquitetura de CUDA é composta por quatro componentes principais conforme visto na Figura 3-6:

- *Engines* de computação paralela presente nas GPUs;
- Suporte ao SO para inicialização do hardware, configuração, etc.;
- *User-mode driver*, que provê uma API de desenvolvimento para GPU;
- Uma arquitetura de conjunto de instruções PTX, contendo os tipos de dados, instruções, registradores, modos de endereçamento, arquitetura de memória etc.

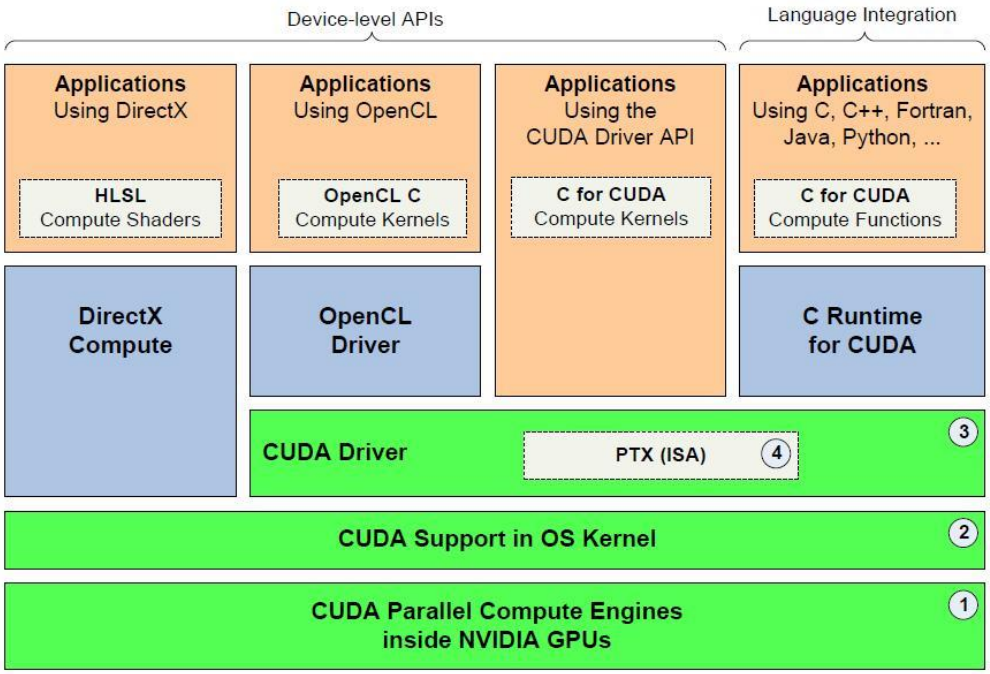

Figura 3-6: Arquitetura de CUDA.

Alguns componentes de software são disponibilizados para o desenvolvimento de aplicações em CUDA: CUDA SDK, que fornece exemplos de código e bibliotecas necessárias para a compilação de código escrito para CUDA, além de toda a documentação; CUDA *Toolkit*, que contém o compilador e bibliotecas adicionais, como o CUBLAS (CUDA *for Basic Linear Algebra Subprograms*) e o CUFFT (CUDA *implementation of Fast Fourier Transform*), além do CUDA *Profiler*, ferramenta bastante utilizada para análise de desempenho e uso da memória.

CUDA dá suporte a duas interfaces diferentes de programação. A primeira, mais baixo-nível, permite que o desenvolvedor utilize DirectX, OpenCL ou a API de CUDA *Driver* para configurar diretamente a GPU, executar *kernels* e ler os resultados. Diferentemente, a segunda interface, mais alto-nível, permite o uso de código C e partes de C++, com algumas extensões que indicam quais funções serão executadas em GPU ou em CPU. Neste nível, pode-se observar a possibilidade de integrar facilmente CUDA a aplicações já existentes desde que satisfaçam a sintaxe padrão.

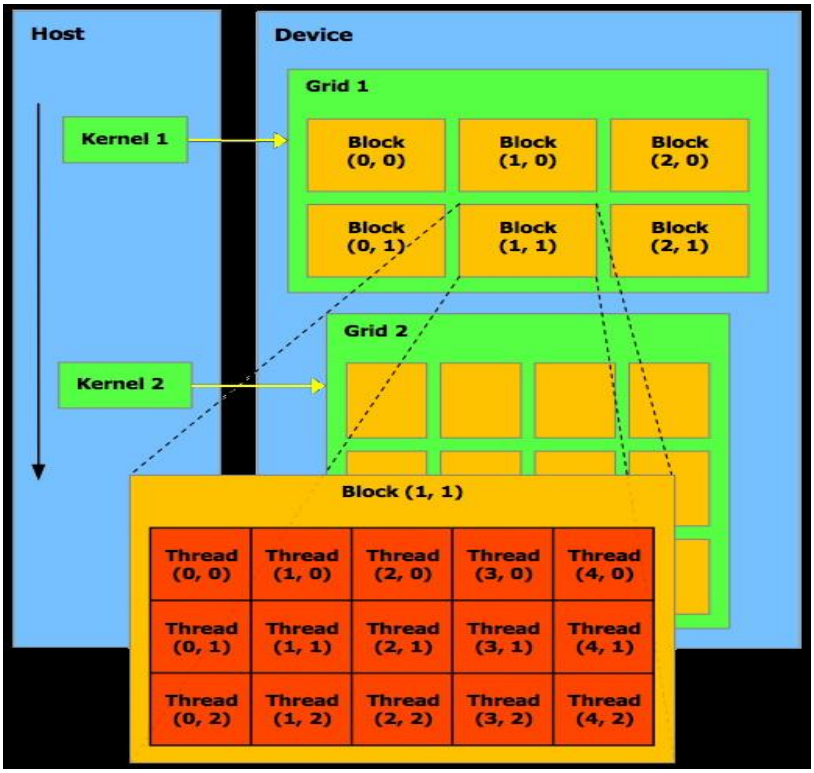

Figura 3-7: Elementos que integram CUDA

Alguns conceitos necessários a uma total compreensão de CUDA estão listados na Figura 3-7. *Threads* em CUDA podem ser consideradas as unidades básicas do processamento paralelo. Cada *thread* na GPU executa a mesma função do *kernel* e possui um ID e memória local. Elas são organizadas em blocos (*blocks*) e podem sincronizar sua execução e compartilhar um mesmo espaço de memória, conhecido por *shared memory*. Um conjunto de blocos representa um *grid*, que pode ser unidimensional ou bidimensional. O *grid* por sua vez, possui uma memória global, uma memória constante e uma memória de textura que podem ser acessadas por cada bloco que o compõe. Um núcleo ou *kernel* consiste no código que é executado por uma *thread* na GPU. Para cada chamada do *kernel* é necessário que seja especificada uma configuração contendo o número de blocos em cada *grid*, o número de *threads* por bloco e, opcionalmente, a quantidade de memória compartilhada a ser alocada e o *stream* associado ao *kernel*.

Em CUDA, algumas palavras são reservadas para qualificar e representar o escopo de uma função ou variável: __*host*__, __*device*__ e __*global*__. O __*global*__ identifica uma função como sendo o *kernel*. Essa função é executada na GPU (*device*), mas apenas pode ser chamada pelo CPU (*host*). O qualificador __*host*__ identifica uma função que é executada no *host* e só pode ser chamada a partir dele, ao contrário do __*device*__ que especifica uma função ou variável definida no *device* e chamada apenas por ele. A chamada de um *kernel* também é diferenciada em CUDA e é seguida pela sua configuração entre os caracteres “<<<” e “>>>”, como pode ser visto no exemplo abaixo:

funcao_kernel<<<dimensaoGrid, dimensaoBloco>>>(parametros);

Além dos qualificadores já descritos, outras palavras reservadas foram introduzidas: __*shared*__, indicando que uma variável será alocada no espaço de memória compartilhada, estando acessível apenas nas *threads* de um mesmo bloco, e __*constant*__, indicando que uma variável será alocada no espaço de memória constante, que possui cache.

Três bibliotecas podem ser utilizadas no desenvolvimento de aplicações em CUDA. A *Common Runtime Library* fornece alguns tipos de dados como vetores de duas, três ou quatro dimensões, e algumas funções matemáticas, como

floor, *ceil*, *sin*, entre outras. Outra biblioteca é a *Host Runtime Library*, que provê toda a parte de gerenciamento de dispositivos e da memória, como funções para alocar e desalocar memória, funções para transferir dados entre *host* e *device*, e funções de chamada de *kernels*. Por fim, CUDA disponibiliza a *Device Runtime Library* que fornece funções específicas da GPU como sincronização de *threads*, garantindo que todas as *threads* dentro de um mesmo bloco esperem pelo término da execução das outras.

4 METODOLOGIA

Este trabalho tem como objetivo explorar o paralelismo inerente à navegação de milhares de agentes dentro de um jogo. Essencialmente, o objetivo consiste em demonstrar o potencial de uma implementação de pathfinding em GPU quando comparada à implementação do mesmo algoritmo em CPU, evidenciando, portanto, o ganho alcançado em seu processamento. Apesar de existirem outras arquiteturas para programação de propósito geral na GPU, optamos por utilizar CUDA [CUDA 2010], desenvolvida pela NVIDIA, por esta se encontrar mais difundida entre a comunidade e, também, por permitir o acesso aos recursos oferecidos pelas placas gráficas que são essenciais no desempenho de aplicações paralelizadas.

Devido ao fato de que não existe um *benchmark* de testes para comparar implementações de algoritmos de busca entre GPU e CPU e tão pouco existe uma implementação única para o A*, nós primeiro investigamos como é realizada essa análise nos principais trabalhos desse tipo. Posteriormente, o método adotado é explicado em detalhes e, finalmente, expomos nossa implementação dos experimentos.

4.1 ESTADO DA ARTE NA GPU

No universo dos Jogos, um dos maiores desafios consiste em encontrar um caminho, muitas vezes, o melhor, entre dois pontos que representam a origem e o destino do movimento de um personagem. Este problema pode ser definido como pathfinding e tem como resultado da sua execução uma lista de pontos que representam o caminho do personagem entre os dois pontos definidos inicialmente.

O problema da navegação autônoma e planejamento de milhares de agentes constituem, atualmente, um desafio para os jogos em tempo real. Muitos tipos de jogos, especialmente os de ação em primeira pessoa e de estratégia em tempo real (RTS) são povoados por centenas ou milhares de atores sintéticos que se comportam como agentes autônomos. Para que esses agentes demonstrem um

comportamento inteligente, é necessário que sua movimentação, uma das ações básicas em um NPC, seja realizada de forma eficiente e satisfatória. Entretanto, levando em consideração que cada um desses NPCs precisa ter o caminho do seu movimento calculado de forma dinâmica, muitas vezes desviando de obstáculos estáticos e dinâmicos (o caso de outros NPCs presentes no ambiente) [La Valle 2006], vê-se que o custo do processamento do algoritmo de busca pode onerar bastante o jogo.

Com a evolução dos hardwares gráficos e o surgimento de uma arquitetura paralela que dê suporte ao desenvolvimento de aplicações de propósito geral, muitas implementações têm procurado viabilizar a diminuição do custo com o processamento global do jogo, adaptando para GPU soluções já existentes na CPU (como o pathfinding), a fim de utilizar ao máximo o poder computacional disponível nas placas gráficas.

Em 2004, Micikevicious [2004] em seu trabalho, descreveu sua implementação do pipeline gráfico na GPU do algoritmo de Warshall-Floyd para resolução do problema da busca pelo caminho mínimo para todos os pares (all-pairs shortest path – APSP). Ele reportou speedups da faixa de 3x sobre a implementação em CPU.

Harish e Narayanan [2007], por sua vez, apresentaram a implementação em CUDA do algoritmo de busca em largura (breadth-first search – BFS). Esse algoritmo pode ser entendido da seguinte maneira: dado um grafo G(V, E) não direcionado e sem pesos, e um vértice de origem S, deve-se encontrar o menor número de arestas necessárias para alcançar cada vértice de V de G, partindo de S, utilizando níveis de sincronização. BFS atravessa o grafo por níveis; uma vez que um nível é visitado, ele não será visitado novamente. A fronteira de uma busca em largura corresponde ao conjunto dos nós que estão sendo processados no nível atual. Ao invés de manter uma lista para cada vértice durante a execução da BFS, o que levaria a um *overhead* para manter os novos índices do array e mudar a configuração do grid a cada nível de execução da BFS, eles associam uma thread a cada vértice. Dois arrays de booleanos, F_a e X_a, de tamanho |V| são criados e armazenam os vértices da fronteira e os visitados, respectivamente. Outro array de inteiros, C_a, guarda o menor número de arestas de cada vértice

partindo do vértice S. A cada iteração, cada vértice observa sua entrada em F_a. Se for verdade, ele busca o seu custo em C_a e atualiza todos os custos dos seus vizinhos se esse for maior que o seu próprio custo acrescido de um. O vértice então remove sua própria entrada do array de fronteiras F_a e adiciona ao array de vértices visitados X_a, incluindo também os seus vizinhos em F_a, caso eles não tenham sido visitados ainda. Este processo se repete até que F_a esteja vazio. O pseudocódigo dos algoritmos descritos em CUDA pode ser visualizado no Algoritmos 4-1 e 4-2.

Algoritmo 4-1. Implementação de BFS em CUDA – Código da CPU.

CUDA BFS (Graph *G*(*V,E*); Source Vertex *S*)

1. Create vertex array V_a from all vertices and edge Array E_a from all edges in *G* (*V,E*),
2. Create frontier array F_a, visited array X_a and cost array C_a of size *V*.
3. Initialize F_a, X_a to false and C_a to ∞
4. F_a[S] **<-** true, C_a[*S*] **<-** 0
5. while F_a not Empty **do**
6. **for** each vertex *V* in parallel do
7. Invoke **CUDA BFS KERNEL(*V_a;E_a;Fa;X_a;C_a*)** on the grid.
8. **end for**
9. end while

Algoritmo 4-2. Implementação de BFS em CUDA – Código *kernel* da GPU.

CUDA BFS KERNEL (*V_a; E_a; F_a; X_a; C_a*)
1. *tid* **<-** getThreadID
2. if F_a[*tid*] **then**
3. F_a[*tid*] **<-** false, X_a[*tid*] **<-** true
4. **for all** neighbors *nid* of *tid* **do**
5. **if** NOT X_a[*nid*] **then**
6. C_a[*nid*] **<-** C_a[*tid*]+1
7. F_a[*nid]* **<-** true
8. **end if**
9. **end for**
10. end if

Neste mesmo trabalho, ainda são apresentados outros dois algoritmos: SSSP (*single-source shortest path*) e APSP. O método proposto por eles é capaz de lidar com grandes grafos, uma evolução quando comparado aos trabalhos anteriores relacionados a esses algoritmos. Eles relatam, ainda, que conseguem executar BFS em um grafo aleatório de 10 milhões de vértices com grau médio de 6, em 1 segundo, e SSP em 1.5 segundos, enquanto que o tempo para um grafo livre de escala de mesmo tamanho é praticamente o dobro. A execução do APSP é realizada em grafos com 30K vértices em cerca de 2 minutos. Entretanto, devido à restrição de memória dos modelos de GPUs da época, [Harish and Narayanan 2007] não conseguiram utilizar grafos com mais de 12 milhões de vértices com grau 6 por vértice. Além disso, as GPUs utilizadas ainda não suportavam precisão de *double*. As Figuras de 4-1 à 4-4 apresentam um resumo dos resultados alcançados por eles: as implementações tiveram uma performance similar ou superior quando comparadas com os mesmos algoritmos processados em um supercomputador que custa em torno de cinco a seis vezes a mais que o hardware gráfico utilizado no estudo.

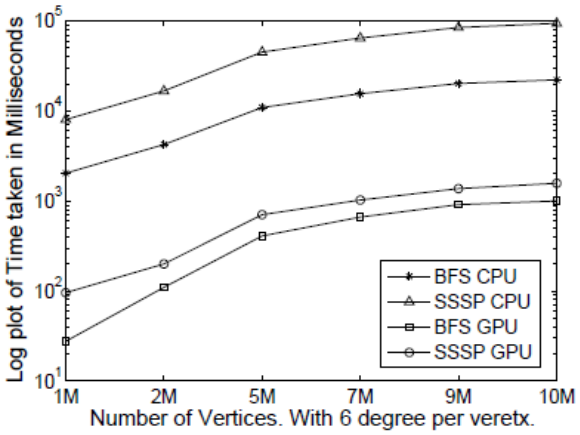

Figura 4-1: Tempos de BFS e SSSP com pesos de 1 a 10.

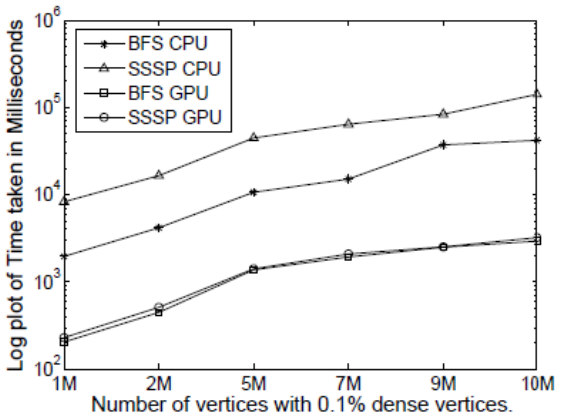

Figura 4-2: Tempos de BFS e SSSP para grafos livres de escala, pesos variando de 1 e 10.

Katz e Kider [2008] descreveram uma implementação em CUDA de cache de memória compartilhada para resolver a cláusula transitiva e o problema da APSP em grafos dirigidos para grandes conjuntos de dados. Eles também relatam bons speedups para dados sintéticos e reais. Diferentemente do trabalho de [Harish and Narayanan, 2007], o tamanho do grafo não é limitado pela memória do dispositivo.

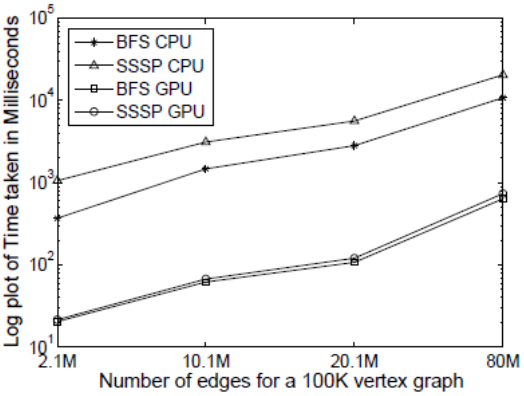

Figura 4-3: Tempos do APSP para vários grafos, pesos variando de 1 a 10.

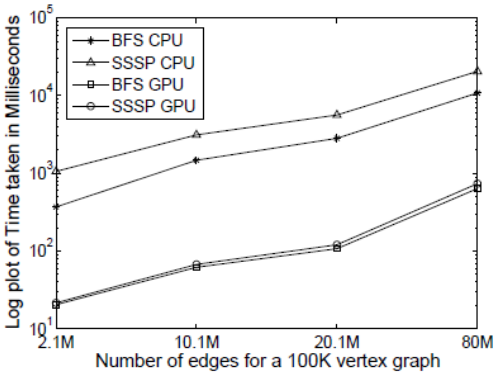

Figura 4-4: Grafos com 100k vértices, com variação de grau por vértice e pesos entre 1 e 10.

Outro trabalho relacionado com o problema APSP pode ser visto em [Buluç et al 2010], onde é descrita uma implementação (CUDA) do algoritmo APSP recursiva e particionada, em que quase todas as operações são expressas como multiplicação de matrizes em um semianel. Essa implementação tem um desempenho superior em mais de duas ordens de grandeza em uma placa de vídeo NVIDIA 8800 quando comparada à uma CPU Opteron. O número de vértices dos grafos utilizados nos experimentos variava entre 512 e 8192.

Com foco no mesmo problema, Tran [2010] apresenta a modelagem (CUDA) de dois algoritmos paralelos eficientes para dispositivos gráficos com múltiplos núcleos. Tran relata que estes modelos demonstram um paralelismo significativo enquanto mantêm uma comunicação global mínima. Utilizando-se do escopo da memória global da GPU, coalescendo as leituras e escritas dela e evitando conflitos na memória compartilhada, ele conseguiu atingir um speedup de 2500x comparada à execução em CPU single-core. A principal diferença em relação ao trabalho de Buluç está na escalabilidade da solução de Tran: ele consegue trabalhar com grafos de tamanho superior à memória disponível na GPU e em múltiplas GPUs, quando estas são adicionadas ao sistema.

Um dos trabalhos de Bleiweiss relacionados com o problema da navegação de agentes na GPU trata da adaptação do modelo de RVO (*Reciprocal Velocity Obstacles*, a fim de melhorar a escalabilidade da simulação de múltiplos agentes. Para isso, Bleiweiss substitui a busca pelos vizinhos mais próximos por uma mais eficiente (*hash based*) e reestrutura o algoritmo para que seu

paralelismo seja evidenciado. Para realizar a navegação multiagente, Bleiweiss assume que o destino dos agentes é estabelecido externamente e desconsiderando as condições do ambiente, que os agentes não podem colidir entre si e movimentam-se em direção ao seu destino com a maior velocidade possível; velocidade esta que pode ser modificada dependendo da presença de outros agentes em movimento. Ainda que os agentes procurem pela menor distância para o seu destino, eles escolhem consultar, constantemente, um caminho global mesmo quando seu caminho está desobstruído. Logo, o caminho global é calculado em uma única etapa de pré-processamento, envolvendo o método de redução da visibilidade do grafo [LaValle 2006] para obter o menor caminho que evita a colisão com obstáculos estáticos do mapa. Em seguida, o modelo de navegação proposto por Bleiweiss e composto por agentes, obstáculos e mapa é discretizado no tempo e a simulação avança todos os agentes passo a passo, concorrentemente [Bleiweiss 2009].

Outro trabalho bastante conhecido relacionados ao problema da busca pelo melhor caminho para dispositivos gráficos utilizando CUDA foi desenvolvido por Bleiweiss [Bleiweiss 2008], engenheiro de software da NVIDIA na época. Em seu trabalho, Bleiweiss propõe uma implementação eficiente de pathfinding global, adaptando os algoritmos de Dijkstra e A*, e explorando o paralelismo existente na navegação de milhares de agentes em um jogo. O foco de seu trabalho não está em executar um pathfinding com otimizações na busca ou detecção de colisão, entre outros fatores que aumentam a complexidade e tornam a movimentação mais suave e realista, mas em comprovar a portabilidade para GPU de algoritmos de IA, explorando o potencial da arquitetura paralela das placas gráficas. Algumas modificações aos algoritmos originais também foram realizadas de modo que as restrições inerentes à arquitetura de CUDA fossem satisfeitas.

O grafo que representa o mapa do jogo é encapsulado em uma lista de adjacências, pois, por limitações de memória, armazená-lo como uma matriz de adjacências se tornaria muito custoso. Necessitando apenas de leitura, o grafo é armazenado em regiões lineares em forma de referências para textura. Por estar organizada como uma cache, as texturas em CUDA possuem uma alta vazão para acessos localizados. Bleiweiss optou por utilizar texturas ao invés de CUDA arrays devido ao fato de texturas apresentarem uma extensão de endereçamento muito

maior se comparadas a CUDA arrays, sendo suficiente para representar grafos grandes. O mapa do jogo é representado, basicamente, por três estruturas: uma lista de nós, uma lista de arestas e um diretório de adjacências que contém um índice e contador para uma lista de adjacências específica de um nó. Apesar da organização do grafo escolhida por Bleiweiss implicar no aumento do consumo de memória (8*N bytes, onde N é o número de nós do grafo), se comparada à implementação em CPU, fez-se necessária a sua utilização para que o acesso à memória fosse feito corretamente na GPU.

O pseudocódigo do algoritmo de A* na GPU pode ser visualizado no Algoritmo 4-3. O algoritmo tem como entrada:

- Uma lista de caminhos definidos por uma origem e um destino, para cada agente;
- Uma lista de custos da posição de origem para cada nó do grafo;
- Uma lista de prioridade que mantém, para cada nó, o custo total do nó para o destino;
- Duas listas de ponteiros para arestas, equivalente, no A* original, à lista de nós abertos e fechados.

Após a execução do algoritmo, o *kernel* produz como saída:

- Uma lista com o custo acumulado para cada caminho encontrado;
- Uma lista de subárvores contendo os *waypoints* encontrados para o agente.

Algoritmo 4-3. Pseudocódigo de A* em GPU.

```
1: f = priority queue element {node index, cost}
2: F = priority queue containing initial f (0,0)
3. G = g cost set initialized to zero
4. P, S = pending and shortest nullified edge sets
5. n = closest node index
6. E = node adjacency list
7. while F not empty do
8.     n ← F.Extract()
9.     S[n] ← P[n]
10.    if n is goal then return SUCCESS
```

```
11.   foreach edge e in E[n] do
12.         h ← heuristic(e.to, goal)
13.         g ← G[n] + e.cost
14.         f ← {e.to, g + h}
15.         if not in P or g < G[e.to] and not in S then
16.               F.Insert(f)
17.               G[e.to] ← g
18.               P[e.to] ← e
19. return FAILURE
```

Embora Bleiweiss, em seu trabalho, tenha alcançado um ganho considerável (speedup em torno de 24x), tanto no Dijkstra quanto no A*, podemos identificar algumas limitações: a) mapas pequenos – um máximo de 340 nós, equivalente a um mapa navegável de 18x18, enquanto em alguns jogos de RTS esse número ultrapassa os milhares; b) uma quantidade reduzida de agentes, se comparada à realidade dos jogos como GTA IV [GTA – *Grand Theft Auto* IV], onde o número de agentes pode ser equivalente à população de toda uma cidade; c) um alto consumo de memória alocada estaticamente ou pré-alocada.

Reynolds [2006] fez o seu trabalho de forma semelhante a Bleiweiss, mas utilizando o poder computacional do hardware do Playstation 3® para melhorar a performance. Para isso, Reynolds particionou o problema em tarefas menores, onde cada uma delas é processada por uma Unidade de Processamento Sinergético (*Synergistic Processor Units – SPUs*) diferente, que compõem o hardware do console [Pham et al. 2005]. Fisher, em seu trabalho [Fisher 2009], utiliza a arquitetura de CUDA para adaptar a implementação de Reynolds para a GPU. Com um speedup de 56x comparado ao mesmo algoritmo na CPU, Fisher utiliza algumas modificações que reduzem o custo das transações de memória entre CPU e GPU.

Na área de robótica, o trabalho desenvolvido por Kider e companheiros [Kider et al. 2010] apresenta uma adaptação do algoritmo R* [Stentz 2008] para a GPU. Como os algoritmos de busca heurística (como o A*), são bastante utilizados para planejamento em espaços de baixa dimensão (2D), eles normalmente não escalam bem para problemas de planejamento em espaços de alta dimensão, por exemplo, no planejamento do movimento de um braço robótico. Os experimentos

realizados demonstraram um speedup em relação à versão do R* para CPU, tanto para o planejamento de um movimento de braço robótico com 6 graus de liberdade (*Degree of Freedom – DOF*) quando para um pathfinding num espaço bidimensional.

Do exposto acima, tem-se uma clara ideia de que a implementação de técnicas de Inteligência Artificial na GPU vem acontecendo de forma experimental, seja para a comunidade de Jogos, ou para outras áreas aplicadas. Isto se deve, principalmente, ao fato de não existir um *benchmark* de testes definido para esse tipo de problema na GPU. A forma como são calculados os ganhos, parte do princípio de que é necessário, primeiramente, implementar o algoritmo em CPU, não necessariamente otimizado. A partir dessa implementação em CPU, é preciso realizar a portabilidade para GPU, tentando manter o mais próximo possível da original. Além disso, a maioria dos estudos não explora todos os recursos disponíveis nas novas gerações de placas de vídeo. Alguns desses recursos podem, inclusive, influenciar positivamente na performance do algoritmo, aumentando ainda mais o speedup em relação às versões simplificadas processadas na CPU.

4.2 IMPLEMENTAÇÃO

O objetivo principal deste trabalho é explorar o paralelismo na realização de um algoritmo de navegação global para milhares de agentes, possibilitando otimizações no pathfinding através dos recursos oferecidos nas placas gráficas atuais. O algoritmo de pathfinding escolhido, o A*, impõe alguns desafios e restrições, principalmente no que diz respeito a buscar uma aceleração em dispositivos com um grupo de threads SIMD (*Single Instruction, Multiple Data*) relativamente grande. Para alcançar tais objetivos, serão demonstrados os ganhos materiais alcançados em GPU comparados com a implementação em CPU e algumas possíveis otimizações a essa implementação em GPU.

Com o desenvolvimento de *frameworks* que ofereçam suporte à programação de propósito geral na GPU, foi escolhido CUDA como plataforma de implementação, dado que ele possibilita o acesso a determinadas características

de hardware que possuem impacto direto na performance da computação de dados paralelos.

Algumas implementações em GPU foram realizadas, ajudando na construção, aos poucos, de uma solução que apresentasse um ganho superior, a fim de perceber a evolução gradativa, à medida que novos recursos das GPUs iam sendo integrados. Os detalhes das implementações em CPU e GPU são especificados nas seções seguintes.

4.2.1. PATHFINDING NA CPU

A solução proposta para CPU foi desenvolvida para fins de comparação e também para um melhor entendimento do algoritmo A*. A utilização de otimizações em CPU, no que diz respeito a múltiplos núcleos ou chamadas de SIMD intrínseco estão fora do escopo deste trabalho. Apesar disso, algumas modificações propostas no trabalho de [Rabin 2000] foram realizadas para oferecer uma implementação do A* consistente ainda que simplificada.

Inicialmente, o mapa do jogo foi particionado em *tiles* quadrados (de lado igual a 20), organizado como um grid regular e definido em um arquivo de texto. Em seguida, foi gerado o grafo como representação do mapa, estruturado em uma lista de adjacências [Icormen et al. 2001]. Computados os nós e arestas do mapa, geramos aleatoriamente os caminhos como um par de nós (origem e destino) para cada agente especificado. Pré-processados os conjuntos que caracterizam o mapa do jogo, é executado, para cada agente, o algoritmo A*, encontrando um caminho para o seu destino. Apesar de ser possível especificar na criação do mapa alguns nós bloqueados, não navegáveis, não foi implementada nenhuma detecção de colisão. Dessa forma, os agentes podem se mover livremente no mapa, sendo possível, inclusive, ocuparem o mesmo nó no grafo.

A heurística utilizada na estimativa do custo entre dois nós avaliados no A* foi a distância Euclidiana. A escolha de uma heurística requer um pouco mais de atenção. No trabalho de Rabin [Rabin, 2000], ele relata que se utilizarmos uma heurística que rotineiramente superestime um pouco, normalmente a busca é mais rápida e os caminhos são razoáveis: se a parta da heurística referente ao custo

total (*f(n) = g(n) + h(n)*) é superior ao que deveria ser, ela distorce a lógica com que os nós pertencentes à lista de nós abertos são escolhidos. Como o A* sempre escolhe o nó com menor custo total, essa distorção faz com que os nós estejam mais próximos do nó que será escolhido. Em espaços de busca que não sejam *grids*, a escolha por uma heurística coerente é um processo de experimentação, pois de torna difícil escolher a melhor estimativa.

Algumas modificações no A* original foram realizadas, dentre elas restrições que adotamos em nossa solução:

- Cada nó do mapa armazena dois booleanos que indicam sua presença ou ausência na lista de nós abertos e na de nós fechados. Como o A* trabalha com duas listas de nós, uma de nós visitados e outra de nós ainda não visitados, dependendo do número de nós, essa estrutura pode se tornar muito grande, ocupando um espaço desnecessário.
- A lista de nós abertos é implementada como uma lista de prioridades (*priority queue*). Como o nó a ser extraído é sempre o que possui o menor custo total, a melhor maneira de armazenar essa lista é mantendo-a como uma *priority queue*, que também pode ser implementada como um *heap* binário (estrutura de árvore ordenada que tem a propriedade de cada nó pai sempre ter um valor menor que o de seus filhos).
- A movimentação dos agentes só é permitida em quatro direções: norte, sul, leste e oeste. Por motivo de simplificação, transições diagonais entre nós não foram incluídas.
- Todos os agentes movimentam-se da mesma forma, não havendo diferenciação entre tipos de agentes.

Ao final da execução do A* na CPU, temos como saída o caminho encontrado para cada agente e o tempo de execução do algoritmo para o conjunto de entrada especificado, que será utilizado para comparar o ganho alcançado na implementação para GPU.

4.2.2. PATHFINDING NA GPU

Um dos maiores desafios da área de Inteligência Artificial em Jogos é conseguir desenvolver soluções com alto grau de realismo. Lake [Gabb e Lake 2005], em seu trabalho, chamou atenção para o fato de que a paralelização funcional é complexa e a adaptação de algoritmos em si, projetados para CPU, para novas arquiteturas de hardware é impraticável ou simplesmente não vale o esforço, pois a interdependência dos componentes de um jogo impossibilita um total paralelismo. Essa interação entre os componentes num motor de jogos pode ser visualizada na Figura 4-5. Dessa maneira, Lake conclui que a melhor solução é a paralelização dos agentes, com a execução do algoritmo de pathfinding em paralelo e cada processo sendo responsável por uma das entidades da simulação.

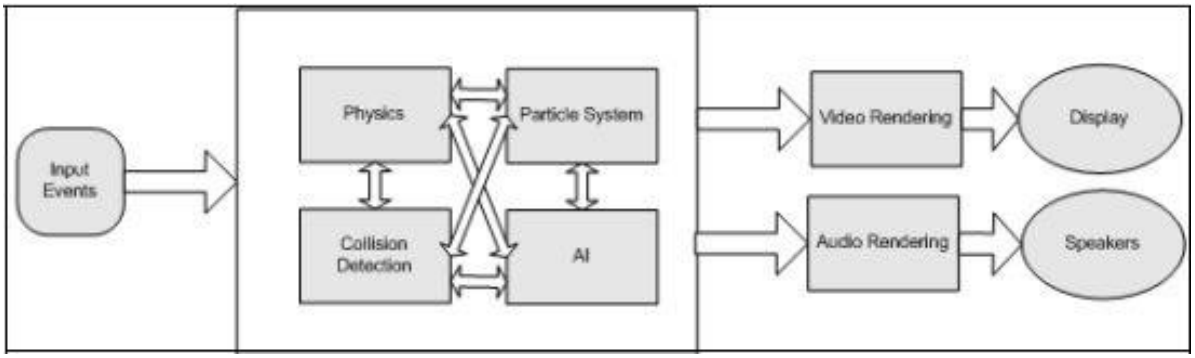

Figura 4-5: Dependência entre componentes de uma game engine.

Foi seguindo essa linha que alguns algoritmos de pathfinding forma implementados em GPU, como o apresentado por [Bleiweiss 2008], é levantada a questão da latência na operação de transferência de dados entre CPU e GPU, apontando para a necessidade do desenvolvimento de técnicas de manipulação dos dados diretamente na GPU, ou seja, quando outros aspectos como jogabilidade e física são simuladas em CPU, existe um gargalo adicional via PCI-E (padrão de *slots* para placas de expansão utilizadas em PCs) para transferência da posição e estados de um personagem entre CPU e GPU. Apesar de ter diminuído nas novas gerações de placas de vídeo, esse gargalo ainda é alvo de diversos estudos.

ASPECTOS DA IMPLEMENTAÇÃO

A implementação em GPU proposta neste trabalho tem como objetivo investigar os aspectos da paralelização dos agentes, apresentando uma solução para o problema do pathfinding através da implementação do algoritmo A* em CUDA. Os aspectos relevantes desta solução e as otimizações propostas serão descritos no decorrer desta seção.

Semelhantemente à implementação em CPU, o grafo que representa o mapa do jogo é encapsulado em listas de adjacências. Entretanto, devido a algumas restrições da arquitetura da GPU (alinhamento de memória, alocação de texturas e uso da lista de prioridades), algumas modificações foram necessárias. Além disso, como tratamos da paralelização de agentes, cada agente é representado por uma *thread*, onde cada *thread* executa um A*. A partir deste ponto, trataremos o termo *thread* e agente com o mesmo significado.

O grafo foi dividido em três estruturas principais, de forma que a memória permanecesse alinhada:

- Nós: representados por quatro floats (totalizando 16 bytes) – um para o id correspondente ao nó e três para armazenar a posição do nó (x, y, z) no mundo do jogo. Apesar de dar suporte a um espaço 3D, optamos por trabalhar apenas com um espaço bidimensional.
- Arestas: representados por quatro floats – dois deles para armazenar os ids das conexões do nó (origem, destino), um pra armazenar o custo e outro reservado apenas para coalescer a memória.
- Diretório de Adjacências: representa o conjunto de arestas de um nó e é composto por dois inteiros não negativos (totalizando 8 bytes) – um indica o deslocamento inicial na lista de arestas e o outro armazena o deslocamento mais a quantidade de arestas do nó, constituindo, assim, as arestas do nó. Apesar de o diretório de adjacências onerar em 8*N bytes de memória se comparada com a implementação em CPU, essa estrutura contribui para uma navegação mais eficiente na GPU.

Como o grafo criado é basicamente uma estrutura de consulta, é mais adequado que ele esteja armazenado em uma região de memória que tenha um acesso rápido. Dessa forma, todas as estruturas que representam o grafo foram mapeadas para a memória de textura na GPU. A principal vantagem em utilizar esse tipo de memória está no seu funcionamento semelhante a uma cache, permitindo taxas de transferência altas para acessos localizados. Além disso, a utilização da memória de textura permitiu que nossa solução desse suporte a grafos maiores, visto que a quantidade de memória de textura disponível é proporcional ao tamanho da RAM da GPU.

A implementação da lista de prioridades, que representa os nós abertos do algoritmo, é um dos aspectos mais importantes na execução do A*. Dentro do loop principal do A*, ela é a estrutura mais acessada e impacta diretamente na performance do algoritmo. Por esse motivo, nós a restringimos a cada agente e a alocamos na memória local da GPU, podendo alcançar, no máximo 16KB por *thread*. Nós optamos por implementar a lista de prioridades como um *heap* binário [Rabin 2000], previamente alocado com o tamanho igual ao número de nós do grafo, enquanto mantém um conjunto de pares compostos por um float que representa o custo e um inteiro que representa o id do nó. Os elementos com o menor custo são colocados no topo da lista. As operações de *insert* e *extract*, essenciais à execução do A*, forma implementadas com um custo logarítmico, evitando recursões. O código abaixo, na Figura 4-6, apresenta o método de extração no *heap*.

```
__device__ CUCost extractFromQueue(CUPriorityQ* pq,unsigned int* qSize) {
   CUCost cost;
   if((*qSize) >= 1) {
      cost = pq->costs[0];
      pq->costs[0] = pq->costs[(*qSize)- 1];
      (*qSize)--;
      heapify(pq, qSize);
   }
   return cost;
}
```

Figura 4-6: Implementação em CUDA do método para extração no *heap*.

CONJUNTO DE TRABALHO PARA EXECUÇÃO DO ALGORITMO

Para executar o algoritmo A* na GPU, nós definimos um conjunto de trabalho composto de quatro entradas e duas saídas. As entradas foram implementadas como um array, da seguinte maneira:

- Um array contendo a origem e o destino de cada agente, em forma de par.
- Um array de custos (float) inicializado com zero, representando o custo de cada nó, a partir do nó inicial.
- Dois arrays de inteiros representando a lista de nós/arestas abertas e fechadas.

O tamanho do array de custos e das listas de nós/arestas abertas e fechadas é A*N, onde A é o número de agentes do jogo e N corresponde ao número de nós do grafo. O conjunto de saída compreende:

- Um array de custos acumulados, armazenados como float, contendo a soma dos custos de cada caminho, para cada agente.
- Um array de posições de nós (float3) contendo os caminhos encontrados para cada agente, tendo, no pior caso, o tamanho de A*N.

As estruturas citadas acima são alinhadas com a memória no tamanho de 4, 8 ou no máximo 16 bytes para limitar a quantidade de múltiplas instruções de leitura e escrita por operação de transferência de memória. O acesso coalescido de 4 bytes apresenta a melhor taxa de transferência, enquanto que as de 8 ou 16 bytes têm uma considerável redução na taxa de transferência.

EXECUÇÃO

Para a execução do algoritmo, a configuração do *kernel* é calculada com base no número de agentes do jogo. O número de *threads* por bloco é fixo (384) e foi calculado utilizando o CUDA Occupancy Calculator **[CUDA]**, que permite visualizar qual a melhor configuração para o tamanho do bloco, baseado em alguns valores como a *capability* e o número de registradores por bloco, a fim de alcançar uma alta ocupância. O conceito de ocupância está atrelado à taxa de

warps ativos e à quantidade máxima de *warps* suportado por um multiprocessdor na GPU. Basicamente, ela ajuda na compreensão de quão eficiente um *kernel* será na GPU: quanto maior a ocupância, normalmente a performance também o é. Para aplicações que não dependem de muita computação aritmética, como o pathfinding, a ocupância alcança picos de até 75%. Na Tabela 4-1 podemos observar o resultado alcançado para blocos de 384 *threads*, com 21 registradores e 44 bytes de memória compartilhada.

Tabela 4-1: Configuração gerada pelo CUDA Occupancy Calculator para um bloco de 384 threads do pathfinding

Threads por bloco	384
Registradores por bloco	8192
Warps por bloco	12
Threads por Multiprocessador	768
Blocos de *Thread* por Multiprocessador	2

Inicialmente, origem e destino são calculados de forma aleatória para o agente e um A* é executado para esse par. Entretanto, como era de se esperar, o algoritmo de pathfinding consome uma grande quantidade de recursos, especialmente de memória. No algoritmo implementado em GPU, se calcularmos a quantidade de memória que está sendo utilizada devido à alocação estática de um conjunto muito grande para execução do *kernel*, podemos verificar que ela pode ultrapassar a memória RAM disponível na placa de vídeo. Por isso, dividimos o conjunto de trabalho, para executar o algoritmo em loops do *kernel*.

Esse mecanismo de dividir o conjunto de trabalho, adequando-o ao limite de memória disponível é possível através de uma consulta às propriedades da placa de vídeo utilizando CUDA **[CUDA]**. Dessa forma, o *kernel* é executado em loop, e os resultados parciais são copiados de volta para a CPU a cada iteração, de maneira que, se calculada a quantidade de memória que cada agente irá precisar dentro do *kernel* (lembrando-se que as estruturas que representam o grafo são comuns a todos os agentes) e dado o número total de agentes e a memória livre disponível, podemos alocar menos memória para os arrays que compõem o conjunto de trabalho. Isso reduz o consumo de memória por execução e torna possível encontrar o caminho de milhares de agentes, em um mapa com mais de 400 nós, inclusive em GPUs que não possuam um grande poder computacional (as

primeiras séries de placas da NVIDIA que dão suporte a CUDA), aproximando-se de uma situação mais real no mundo dos jogos. Como o índice das *threads* de um bloco pode vir a ultrapassar o número total de agentes para aquela chamada, é feita uma verificação para não permitir que a GPU compute algo além do que deve ser feito.

Ao final da execução do pathfinding para cada *thread*, o algoritmo retorna os caminhos encontrados para cada agente juntamente com o custo acumulado calculado pelo A*. Durante o desenvolvimento do algoritmo de navegação na GPU, identificamos alguns gargalos e possíveis otimizações que proporcionam um ganho ainda maior com a implementação do A* nas placas gráficas. Essas mudanças são apresentadas na seção seguinte.

OTIMIZAÇÕES

Partindo da implementação básica do A* na GPU, propomos algumas modificações que nos permitiriam alcançar um ganho superior à versão para CPU, bem como um conjunto maior de teste (mais agentes e mapas maiores).

Com a execução do *kernel* em loop, tornou-se necessária a transferência dos resultados parciais referentes aos agentes computados a cada iteração, isto é, a cada chamada do *kernel*, o conjunto de saída era copiado para uma região de memória alocada na CPU. Entretanto, como o array de saída que armazena as posições dos nós, representando o caminho de cada agente, é alocado estaticamente com o tamanho do número de nós (maior caminho que um agente pode ter), seu caminho pode ser menor que a região alocada. Por exemplo: para o agente A é alocado um array de N nós para armazenar seu caminho, inicializado com 0. Se o caminho encontrado para o agente A possui apenas 2 nós (origem e destino estão próximos), quando terminar a execução do *kernel*, um array grande será copiado com “lixo”, quando deveria copiar apenas um array de tamanho igual a 2. Se N for muito grande, essa taxa de transferência de dados da GPU para a CPU consome muito tempo e, consequentemente, aumenta o tempo de execução do pathfinding.

Para compensar esse gargalo existente na transferência dos resultados parciais do pathfinding, procuramos diminuir o volume de dados transferidos,

armazenando apenas o Id do nó correspondente no array que guarda o caminho encontrado para o agente, ao invés da sua posição. Dessa forma, continuamos a copiar “lixo”, mas diminuímos o volume de *float3* para *unsigned int*. Apesar de ser uma modificação simples, o impacto que se tem é grande e, ainda assim, é possível ter acesso às posições dos nós no mapa, acessando pela CPU, através do Id do nó.

Além disso, esse mesmo array de saída foi alocado na região de memória não paginada da CPU, quando disponível, onde há menos controle por parte do Sistema Operacional e, consequentemente, uma melhora na comunicação entre a placa de vídeo e o barramento de memória RAM da placa-mãe. Essas modificações no conjunto de saída proporcionaram um ganho de 3x sobre a taxa de transferência dos dados parciais.

Outra otimização realizada está relacionada com um conceito relativamente recente, primeiramente utilizado por Aila [Aila et al. 2009] na área de *Ray-Tracing*, mas que tem sido amplamente estudado atualmente: *threads* persistentes. A ideia consiste, basicamente, em tornar os *warps* mais independentes e mais eficientes. Numa placa gráfica, um bloco só termina de executar quando todos os *warps* terminam seu trabalho. Nesse caso, no contexto do pathfinding, é possível que *warps* dentro do mesmo bloco possuam caminhos bem diferentes. Logo, pelo menos um dos *warps* gastará mais tempo que outro calculando o caminho, deixando o bloco, de certa forma, ineficiente. Com a inclusão de *threads* persistentes, cada *warp* funciona praticamente independente e não espera o término do outro para começar, aumentando a performance do algoritmo. Apesar do *overhead* que esta técnica agrega à solução, nós implementamos *threads* persistentes utilizando operações atômicas na memória compartilhada, ao invés da memória global, devido ao rápido acesso da memória compartilhada.

Com um melhor entendimento da arquitetura de CUDA, pudemos realizar outras modificações mais simples, mas que contribuam, de forma geral, para o desempenho do algoritmo: para calcular a distância Euclidiana (heurística utilizada no nosso A*) não utilizamos a função *sqrt*, que é uma operação custosa em CUDA; realizamos um tuning para reduzir a quantidade de registradores utilizados e a

transferência de dados para a memória global, que possui uma baixa taxa de transferência.

Baseado nessas otimizações realizadas, foi possível visualizar ganhos reais sobre a implementação em GPU e CPU, fortalecendo ainda mais a ideia de adaptar algoritmos de pathfinding para utilizar o potencial computacional dos hardwares gráficos. Os resultados e suas implicações são detalhados no capítulo seguinte.

5 ANÁLISE DOS RESULTADOS

Os algoritmos de pathfinding A* implementados em CPU e GPU e descritos no capítulo 4 são analisados e comparados neste capítulo. Avaliados sob a perspectiva do ganho de desempenho (speedup) e consumo de memória, esta análise gerou resultados significativos, tornando possível visualizar que a paralelização (em nível de agentes) para o algoritmo implementado demonstrou ganhos de performance, fortalecendo ainda mais a ideia de utilizar a GPU como plataforma para o desenvolvimento de aplicações de propósito geral, com ênfase na adaptação de algoritmos de Inteligência Artificial em Jogos.

Nós utilizamos para os experimentos grafos não direcionados, que foram gerados automaticamente com uma topologia de complexidade baixa. O número de agentes usado foi, em média, o quadrado da quantidade de nós do grafo, exceto em alguns testes focados em executar o pathfinding para um conjunto grande de agentes e nós. As extremidades do caminho de cada agente (origem, destino) foram geradas randomicamente. Optamos por utilizar a *capability* de 1.2, compatível com todas as placas gráficas atualmente no mercado. Todos os testes descritos neste capítulo forma realizados em um processador Intel Core i7 1.6GHz com 4GB de memória RAM, para ambas implementações em CPU e GPU, e uma placa de vídeo NVIDIA GeForce GTS 360M com 1782MHz de *shader clock*, 1GB de memória global e 12 multiprocessadores.

O speedup foi medido com base na comparação entre o algoritmo implementado na CPU (*single-threaded*) e as implementações em GPU (com e sem otimizações). Como o algoritmo foi executado no Windows 7, utilizamos o *timer* da API do Windows para medir o tempo de execução.

Como os caminhos são gerados de modo aleatório, cada *benchmark* é executado três vezes e a média do tempo de execução e da memória utilizada são computados para manter um resultado mais consistente. Assim, nós realizamos a comparação de performances entre a implementação para CPU, a implementação básica para GPU (sem otimizações), a implementação para GPU com otimizações mas sem *threads* persistentes e a implementação para GPU com otimizações e

threads persistentes. Na Tabela 5-1 é possível visualizar todos os *benchmarks* utilizados nos testes.

Tabela 5-1: Lista de *Benchmarks*; Número de arestas e nós, quantidade de agentes (*threads*) e número de blocos.

Grafo	Nós	Arestas	Agentes	Blocos
G0	16	48	64	1
G1	64	224	1024	3
G2	144	528	20736	54
G3	256	960	65536	171
G4	324	1224	115600	302
G5	400	1520	300000	784
G6	900	3480	20736	54
G7	2025	7920	1024	3
G8	2025	7920	65536	150

A partir dos valores obtidos com a média dos tempos de execução, é calculado o speedup que o algoritmo teve em relação à CPU, segundo a fórmula:

$$speedup = TCPU / TGPU,$$

onde TCPU é o tempo de execução do algoritmo na CPU e TGPU o tempo de execução alcançado pela implementação em GPU.

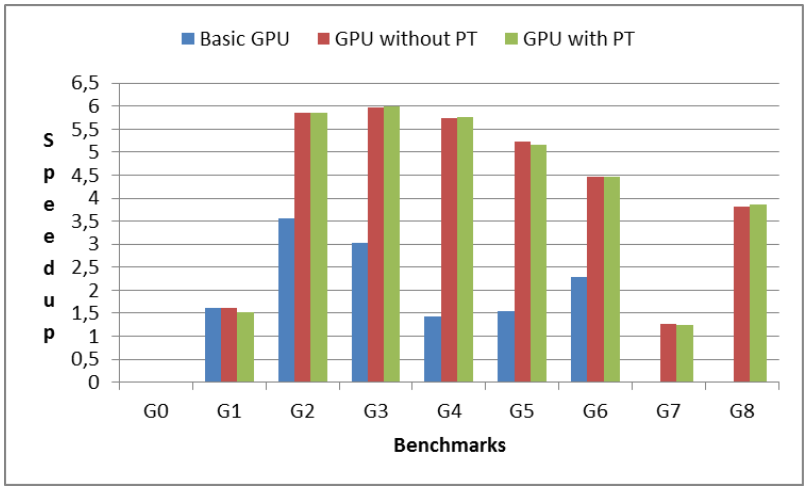

Figura 5-1: Comparação da performance entre implementação básica na GPU, com e sem *threads* persistentes (PT) vs. implementação em CPU.

O speedup calculado para cada conjunto de teste é apresentado na Figura 5-1. Observamos que o ganho alcançado é cresce à medida que a complexidade do mapa aumenta (maior número de nós) e a quantidade de agentes também,

atingindo o seu topo em G3. É possível observar, também, que para um conjunto de trabalho pequeno, como visto em G0, a implementação em CPU ainda representa uma boa solução. Por outro lado, com um conjunto de trabalho grande, a GPU obteve um speedup de cerca de 6x. Quanto à utilização de *threads* persistentes, observamos que o speedup alcançado está muito próximo da implementação sem este recurso, o que, de certa forma, nos surpreendeu.

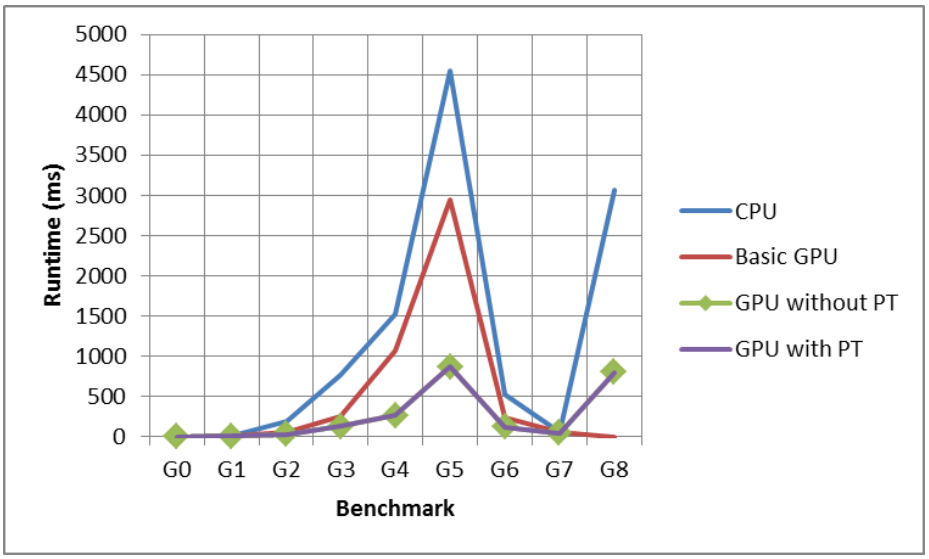

Figura 5-2: Tempo absoluto de execução da implementação do A* na CPU, básica na GPU e implementação na GPU com e sem *threads* persistentes (PT).

Na Figura 5-2, vemos que na CPU o custo é mais influenciado pelo aumento do número de agentes no mapa: do conjunto G5 para o G6, mesmo aumentando a quantidade de nós do mapa, o speedup diminuiu ao aumentarmos o número de agentes. Com um número reduzido de nós, o conjunto de trabalho não se torna muito extenso e custoso, mesmo com milhares de agentes. Entretanto, com o aumento do número de nós, à medida que o número de agentes também aumenta, o conjunto de trabalho se expande muito, consumindo mais recurso da GPU e aumentando o gargalo na transferência de dados entre CPU e GPU. Ainda assim, observando o tempo de execução em G8, vemos que a implementação em GPU representa uma boa solução.

Muito embora o speedup obtido seja inferior ao apresentado por Bleiweiss, a principal contribuição deste trabalho está na quantidade de agentes e no

tamanho do mapa utilizado. Enquanto [Bleiweiss 2008] utiliza um mapa de no máximo 340 nós, nós conseguimos executar o A* na GPU para um máximo de 2025 nós com 65536 agentes, e um máximo de 300.000 agentes em um mapa de jogo com 400 nós, o que representa números mais próximos de um jogo real. Visto que a memória das placas gráficas continua limitada, executar um pathfinding de alta escalabilidade e grau de realismo, próximo ao encontrado nos jogos mais atuais, é um desafio constante comum e área de pesquisa crescente.

6 CONSIDERAÇÕES FINAIS

Este trabalho apresentou um estudo de técnicas aplicadas à resolução do problema da navegação de um agente no terreno, tendo em vista seu emprego direto no desenvolvimento de jogos. O principal objetivo deste trabalho foi desenvolver um algoritmo de pathfinding em GPU e explorar possíveis e efetivas otimizações, utilizando o potencial das placas gráficas e da arquitetura de CUDA para permitir o uso dos multiprocessadores da GPU para o problema da navegação.

Por esta razão, é possível visualizar o potencial das GPUs na execução do pathfinding. Atualmente, os jogos seguem a tendência de trazer, cada vez mais, ambientes complexos, com a simulação de milhares de agentes em tempo real. Com as limitações impostas pela arquitetura e recursos da CPU, algumas técnicas de IA vêm apresentando dificuldades para executar em CPU, ao passo que o poder computacional dos processadores gráficos tem crescido constante e rapidamente. Observando essa conjuntura, vê-se que a implementação dessas técnicas de Inteligência Artificial, principalmente as que demonstram um paralelismo inerente, tornam-se bastante promissoras na GPU, estabelecendo um link para uma série de possibilidades nos jogos.

6.1 LIÇÕES APRENDIDAS

Embora tenham surgido algumas dificuldades com relação ao entendimento de uma arquitetura diferenciada, que acarreta em uma série de fatores no desenvolvimento da solução, e mesmo que os resultados alcançados não sejam superiores ao de outros trabalhos, foi possível visualizar o potencial das GPUs na execução do pathfinding.

Quando atingimos um nível razoável de aceitação de speedup, inicialmente pensamos em adaptar a solução para múltiplas GPUs. Entretanto, essa adaptação gerou uma série de problemas e questionamentos ligados ao mapa do jogo, à transferência de dados entre as GPUs e ao entendimento do funcionamento das regiões de memória, além de ser um recurso um pouco complicado de se

administrar. Em relação ao mapa, a solução mais básica que encontramos foi replicar o mapa para cada GPU envolvida, gerando uma quantidade maior de recursos para armazenamento. Apesar de nas versões mais recentes de CUDA ser possível realizar a transferência de dados entre GPUs de forma otimizada, esse é um recurso ainda escasso. Por fim, a forma como o trabalho seria escalonado entre as GPUs também se tornou motivo de estudo.

Outro ponto que sentimos dificuldade foi com relação aos *benchmarks* de teste. Como não existe nenhum trabalho relacionado a isso, a comparação entre soluções de diversos trabalhos se tornam complicadas, pois existem muitos fatores que podem contribuir para que um speedup seja maior ou não.

6.2 RECOMENDAÇÕES PARA TRABALHOS FUTUROS

Como resultado deste trabalho, nós acreditamos que as seguintes modificações merecem uma investigação mais profunda e podem vir a contribuir bastante com este trabalho e com a área:

1. Reduzir o conjunto de trabalho, principalmente utilizando alocação dinâmica. Atualmente o conjunto de trabalho é todo previamente alocado e, a depender do número de agentes e nós, pode se tornar muito grande.
2. Investigar a transferência de dados via *Peer-to-Peer* entre GPUs, presente na versão mais nova de CUDA. Deve ser levada em consideração a replicação do mapa para cada GPU bem como o escalonamento do trabalho entre elas.
3. Investigar a possibilidade de abordagens multiagente, em que cada agente pode reusar o caminho previamente calculado por outro agente. Esse tipo de abordagem diminuiria o custo com o cálculo de caminhos que já tenham sido computados.
4. Investigar se existe a possibilidade de utilizar outra abordagem de paralelização. A mais utilizada atualmente está relacionada com o mapeamento de 1 agente por *thread*.

5. Utilizar-se de ferramentas que permitam a visualização dos caminhos calculados para cada agente, assim como a simulação da navegação em tempo real.
6. O principal trabalho futuro está no desenvolvimento de um *benchmark* para testes. Com isso, seria possível padronizar os testes, utilizando os mesmos parâmetros e recursos (número de agentes, tamanho do mapa, placa de vídeo utilizada, etc.), com diferentes algoritmos.

REFERÊNCIAS

Aila, T. and Laine, S. Understanding the Efficiency of Ray Traversal on GPUs. (2009).

Andrade, G., Ramalho, G., Santana H. and Corruble V. Automatic computer game balancing: a reinforcement learning approach. In Proceedings of the Fourth international Joint Conference on Autonomous Agents and Multiagent Systems (2005), July 25 - 29, AAMAS '05. ACM, New York.

Atanasov D. General Purpose GPU programming. In International Conference on Computer Systems and Technologies. CompSysTech (2005).

"ATI Stream Technology": http://www.amd.com/stream 22/07/2010. [Online]. Available: http://www.amd.com/stream

Björnsson Y. and Halldórsson K. Improved heuristics for optimal pathfinding on game maps. In American Association for Artificial Intelligence (2006).

Bleiweiss A. GPU accelerated pathfinding. In Proceedings of the 23rd ACM SIGGRAPH/EUROGRAPHICS Symposium on Graphics Hardware (Sarajevo, Bosnia and Herzegovina, June 20 - 21, 2008). SIGGRAPH/EUROGRAPHICS Conference On Graphics Hardware. Eurographics Association, Aire-la-Ville, Switzerland (2008), 65-74.

Bleiweiss, A. Multi Agent Navigation on the GPU. In NVIDIA Corporation. (2009).

Bourg, D.M., and Seemann, G. AI for Game Developers. O'Reilly (2004).

Brown, Emily, Cairns, Paul. A Grounded Investigation of Immersion. In Proc. CHI2004, ACM Press, Vienna, Austria (2004), p. 1297-1300

Bustos, B., Deussen O., Hiller S. and Keim, D. A graphics hardware accelerated algorithm for nearest neighbor search (2006).

Buluç, A., Gilbert, J.R., Budak, C.: Solving path problems on the GPU. In: Parallel Computing, The Netherlands, 2010, vol 36 (5-6), p.241-253. doi>10.1016/j.parco.2009.12.002

Csikszentmihalyi, M. Flow: The Psychology of Optimal Experience. New York: Harper & Row (1990).

"CUDA", http://www.nvidia.com/object/cuda_home_new.html , 20/07/2010. [Online]. Available: http://www.nvidia.com/object/cuda_home_new.html

Chiara, R.C., Erra, U., Scarano, V. and Tatafiore, M. Massive simulation using GPU of a distributed behavioral model of a flock with obstacle avoidance. In B. Girod, M. A. Magnor, and H.-P. Seidel, editors, VMV, pages 233–240. Aka GmbH (2004).

Champandard, A.J.. AI Game Development: Synthetic Creatures with Learning and Reactive Behaviors. (2003).

Darken, C. J. and Paull, G. H. Finding Cover in Dynamic Environments. In AI Game Programming Wisdom 3, editors, Charles River Media, pages 405-415. Boston, Massachusetts (2006).

"FIFA Soccer 2010": http://www.ea.com/games/fifa-soccer-10 21/07/2010. [Online]. Available: http://www.ea.com/games/fifa-soccer-10

Fisher, L. G., Silveira, R., Nedel, L."GPU accelerated path-planning for multi-agents in virtual environments", In SBGames, 2009.

Galstyan, A., Kolar, S.,and Lerman, K. Resource Allocation Games with Changing Resource Capacities. In Proccedings of the International Conference on Autonomous Agents ant Multi-Agent Systems (AAMAS-2003), Melbourne, Autralia (2003).

Gabb, H., Lake, A. Threading 3D Game Engine Basics. [Online]. Available: http://www.gamasutra.com/features/20051117/gabb_01.shtml

"GTA – *Grand Theft Auto* IV" http://www.rockstargames.com/IV/ 20/07/2010 [Online]. Available: http://www.rockstargames.com/IV/

Graham, R., McCabe, H., Sheridan, S. Pathfinding in Computer Games. In ITB Journal. Issue Number 8 (2003).

Holte, R. C., Perez, M. B., Zimmer, R. M., MacDonald, A. J. Hierarchical A*: Searching Abstraction Hierarchies Efficiently. (1996).

Icormen, T.H., Leiserson, C. E., Rivest, R. L., Stein, C. Introduction to Algorithms. MIT Press, Cambridge, MA (2001).

Harish, P. and Narayanan, P. J. Accelerating large graphs algorithms on the GPU using CUDA. (2007).

Katz, G.J., Kider Jr, J.T.: All-Pairs Shortest-Paths for Large Graphs on the GPU. In Graphics Hardware, editors David Luebke and John D. Owens, Pennsylvania, p. 47-55 2008.

Kider, J., Henderson, M., Likhachev, M., Safonova, A.: High-dimensional planning on the GPU. In: Robotics and Automation (ICRA), 2010 IEEE International Conference on, pp. 2515{2522 (2010)

Lavalle, S.M. Planning Algorithms. Cambridge University Press. http://msl.cs.uiuc.edu/planning/, (2006).

"Left 4 Dead 2": http://www.l4d.com/, 21/07/2010. [Online]. Available: http://www.l4d.com/

Larsen, E. S. and McAllister, D. Fast matrix multiplies using graphics hardware. Supercomputing 2001 , November (2001).

Mateas, M.: Expressive AI: Games and Artificial Intelligence. (2003).

Micikevicius, Paulius. General parallel computation on commodity graphics hardware: Case study with the all-pairs shortest paths problem. H. R. Arabnia, editor, PDPTA, 2004, pp. 1359–1365. CSREA Press.

Nereyek, A. AI in Computer Games. In Guest Researcher. Carnegie Mellon University.

Owens, J. D., Luebke, D., Govindaraju, N., Harris, M., Krüger, J., Lefohn, A. E.and Purcell, T. J. A Survey of General-Purpose Computation on Graphics Hardware. In Start of The Art Report. Eurographics (2005).

"*Pac-man*":http://home.comcast.net/~jpittman2/pacman/pacmandossier.html 21/07/2010.[Online].Available:http://home.comcast.net/~jpittman2/pacman/pacmandossier.html

Pham, D. et al., “The design and implementation of a first-generation CELL processor”, in Solid-State Circuit Conference, 2005, ISSCC IEEE International, 2005.

Rabin, S. A* Aesthetic Optimizations. In Game Programming Gems. pages 264-271. (2000).

Rabin, S. A* Speed Optimizatios. In Game Programming Gems. pages 272-287. (2000).

Reynolds, C. “Big fast crowds on PS3”, in sandbox’06: Proc. ACM SIGGRAPH symposium on Videogames, ACM Press, New York, USA, 2006, pp.113–121.

Rouse 3rd R. What’s Your Perspective? In Gaming and Graphics Aug, New York, 1999, vol 33 (3),p. 9-12. doi>10.1145/330572.330575

“*Space Invaders*”: http://www.spaceinvaders.de/ 21/07/2010. [Online]. Available: http://www.spaceinvaders.de/

Silver, D. Cooperative Pathfinding. In American Association for Artificial Intelligence (2005).

Stout, B. The Basics for A* Path Planning. In Game Programming Gems. pages 254-263. (2000).

Stentz, A. Optimal and Efficient Path Planning for Partially-known Environments. In proceedings of the IEEE International Conference on Robotics and Automation, May (1994).

Tatarchuk, N. Richer Worlds for Next Gen Games: Data Amplification Techniques Survey. GDC Europe (2005).

Tran, Q.N. Designing Ecient Many-Core Parallel Algorithms for All-Pairs Shortest-Paths Using CUDA. In: Proceedings of the 2010 SeventhInternational Conference on Information Technology: New Generations, ITNG '10, pp. 7{12. IEEE Computer Society, Washington, DC, USA (2010)

Tozour, P. Search Space Representations. In AI Game Programming Wisdom 2, editors, Charles River Media, pages 405-415. Hingham, Massachusetts. (2003).

Trindade, E., Ferreira, R. M. , Fantini, E., de Paula, H. Algoritmos de Busca em Tempo Real aplicados a jogos digitais. *In Proceedings* SBGames 2008. (2008).

Wang, K.C. and Botea, A. Fast and Memory-Efficient Multi-Agent Pathfinding. Proceedings of the Eigthteenth International Conference on Automated Planning and Scheduling (ICAPS 2008). In Association for the Advancement of Artifical Intelligence (2008).

Watt, A. 3D Computer Graphics, Pearson – Addison Wesley, New York (2000).

Yang, R. and Welch, G. Fast image segmentation and smoothing using commodity graphics hardware. To appear in the journal of graphics tools, special issue on Hardware-Accelerated Rendering Techniques (2003).