UNIVERSIDADE ESTADUAL DE CAMPINAS
Faculdade de Engenharia Elétrica e de Computação

Mario Santos Camillo

Estudo Comparativo entre OpenGL e CUDA em Acessos Aleatórios à Vizinhança de Primeira Ordem

CAMPINAS

2018

Mario Santos Camillo

Estudo Comparativo entre OpenGL e CUDA em Acessos Aleatórios à Vizinhança de Primeira Ordem

Dissertação apresentada à Faculdade de Engenharia Elétrica e de Computação da Universidade Estadual de Campinas como parte dos requisitos exigidos para a obtenção do título de Mestre em Engenharia Elétrica, na Área de Engenharia de Computação.

Orientador: Profª. Dr.-Ing. Wu Shin-Ting

Este exemplar corresponde à versão final da dissertação defendida pelo aluno Mario Santos Camillo, e orientada pela Profª. Dr.-Ing. Wu Shin-Ting

CAMPINAS

2018

Aos meus pais e à minha esposa

Agradecimentos

À Profª. Dr.-Ing. Wu Shin-Ting, por sua orientação e conselhos, por me aguentar e perseverar comigo durante todos esse anos e pelas várias horas fora de expediente revisando meus trabalhos em prazos apertados.

Aos meus familiares, principalmente minha mãe, Raquel, e minha esposa, Yumi, por nunca pararem de me incentivar para que perseverasse nesse caminho e concluísse esse trabalho.

Ao meu amigo, Edgar, pela sua disponibilidade em ser um revisor de última hora desse trabalho.

Ao Instituto de Pesquisas Eldorado, pelo incentivo aos meus estudos enquanto trabalhava para eles e pela permissão para o uso de equipamentos da empresa para execução de alguns dos testes aqui apresentados.

Aos meus colegas do LCA pela companhia, ajuda com revisões de artigos e por compartilharem seu conhecimento comigo.

Aos professores e funcionários da pós graduação pelo seu trabalho e dedicação em manter a alta qualidade do curso de mestrado da FEEC.

Aos professores Drs. Luiz Otávio Saraiva Ferreira e Márcio Machado Pereira por aceitarem o convite para a banca examinadora desse trabalho e pelas sugestões construtivas.

Resumo

Os avanços da arquitetura das GPUs e sua rápida evolução em direção a GPUs de propósito geral (GPGPU) tem feito com que uma série de aplicações adotem soluções de interoperabilidade entre GPGPU e computação gráfica, com o primeiro usado para cálculos pesados e o segundo para a renderização de modelos gráficos 3D em imagens de alta resolução em tempo interativo. APIs de GPGPU como CUDA, por exemplo, expõe explicitamente diversos recursos de *hardware*, tais como memória compartilhada e mecanismos de sincronização de *threads*. Isso é particularmente atraente para aplicações com acessos aleatórios (não-contíguos) à memória, pois permite que o desenvolvedor escreva código mais eficiente para gerenciamento desses acessos. Ao mesmo tempo, diversos trabalhos mostram que, para aplicações gráficas ou que demandam visualizações gráficas dos resultados, a implementação somente em OpenGL tem melhor desempenho que implementações com interoperabilidade entre CUDA e OpenGL.

Nesta dissertação nós apresentamos uma comparação entre o uso de OpenGL e CUDA para acessos aleatórios à vizinhança de primeira ordem. Nós investigamos duas aplicações representativas desse problema: estimativa de tensores de curvatura e simulação de N-corpos. Em ambos os casos nós implementamos um mesmo algoritmo com uso das APIs CUDA e OpenGL. Conduzimos uma análise dos recursos de *hardware* expostos por cada API e a equivalência entre eles, inclusive através de usos não-gráficos do *pipeline* da OpenGL. Medições de desempenho são apresentadas para validar nossos resultados.

Nós acreditamos que os resultados deste estudo provêem um melhor entendimento dos recursos gráficos necessários para diminuir a diferença de desempenho de OpenGL e CUDA, e abrem novas perspectivas para implementações somente em OpenGL das aplicações que requerem acessos pré-determinados e intensos à memória e renderização de gráficos 3D.

Palavras-chaves: GPU, Tesselação, Combinação, Tensores de Curvatura, N-corpos, GPGPU, OpenGL, CUDA.

Abstract

Advances in GPU architecture and its rapidly evolving towards general purpose GPU (GPGPU) make a series of applications adopt a GPGPU and graphics computing interoperability approach in which the first is used for heavy computations and the second for 3D graphics rendering. GPGPU APIs, like CUDA, explicitly expose several hardware features, such as shared memory and thread synchronization mechanisms. This is particularly attractive to applications with random (non-coalesced) memory accesses, as it allows developers to write more efficient code for handling these accesses. At the same time, several studies show that, for graphics applications or programs that require graphical visualization of their results, an OpenGL implementation has better performance than implementations based on OpenGL and CUDA interoperation.

In this dissertation we draw a comparison of the use of OpenGL and CUDA for random (non-coalesced) accesses to the 1-ring neighborhood of a vertex. We study two examples of this problem: curvature tensor estimatives and N-body simulation. In both cases we implement the same algorithm with the CUDA and OpenGL APIs. We analyzed the hardware features exposed by each API and their equivalence, including non-graphics usage of the OpenGL pipeline. Comparative timing analysis is provided to validate our results.

We believe that our study provides a better understanding of the graphics features that are useful for closing the performance gap between OpenGL and CUDA based implementations, and open new perspectives on implementing, solely with the OpenGL, graphics applications that require both intense but pre-specified memory access and 3D graphics rendering.

Keywords: GPU, Tessellation, Blending, Curvature Tensors, N-body, GPGPU, OpenGL, CUDA.

Lista de ilustrações

Lista de tabelas

Sumário

1 Introdução

As GPUs (*Graphical Processing Unit*) vêm evoluindo de maneira considerável desde sua primeira aparição, partindo de um processador de núcleo único com um *hardware* com *pipeline* fixo, para um conjunto de núcleos programáveis e altamente paralelos (MCCLANAHAN, 2010). Com o advento do *pipeline* programável em GPUs, diversos trabalhos de adaptação de algoritmos para GPGPU (*General Purpose GPU*) passaram a ser desenvolvidos (OWENS *et al.*, 2007). A maioria dos trabalhos desenvolvidos trata de cômputo altamente paralelizável de dados maciços. O acesso a esses dados é um dos principais gargalos de desempenho (LOBEIRAS *et al.*, 2012; CAMILLO; WU, 2013; SHIUE *et al.*, 2005; HJELMERVIK; LEON, 2007) para esse tipo de aplicação.

Em aplicações de finalidade gráfica ou que demandam visualização gráfica dos resultados, a representação dos dados do problema nos permite identificar duas classes distintas de problemas que podem se beneficiar do poder de processamento paralelo das GPUs.

- problemas descritíveis com uma topologia estática, como cômputo de tensores de curvatura, simulação de tecido e simulação de n-corpos.
- problemas representáveis por uma topologia dinâmica, como refinamento de malhas pela técnica de Catmull-Clark.

Problemas de topologia dinâmica precisam armazenar as informações de uma malha 3D e sua topologia em estruturas de dados bastante flexíveis, tal como a estrutura *half-edge* bastante usada no modelo de *Boundary Representation*. Esse tipo de estrutura costuma ser altamente dependente de ponteiros e não são apropriadas para a arquitetura da GPU, normalmente apresentando desempenho insatisfatório devido ao padrão de acessos totalmente irregular à memória global. Diversos trabalhos tentam atacar esses problemas através de mapeamentos dos dados da malha para GPU (HJELMERVIK; LEON, 2007; SHIUE *et al.*, 2005) e criação de novas estruturas ou organização desses dados na memória (ALUMBAUGH; JIAO, 2005; YOON; LINDSTROM, 2006). Essa categoria de problemas não é o foco deste trabalho.

Dentre os problemas que podem ser modelados por uma topologia estática diferenciamos ainda duas sub-classes de problemas em função da limitação das GPUs na implementação de ponteiros:

- vértices com um número de valência limitado: tensores de curvatura, simulação de tecidos.
- vértices com um número de valência arbitrário: simulação de n-corpos, dinâmica de fluidos.

A acelerada evolução da GPU de propósito gráfico para propósito genérico, oferecendo uma alternativa de acesso direto aos recursos de *hardware* disponíveis, tem propiciado a migração de muitas aplicações da arquitetura OpenGL (*Open Graphics Library*) para as arquiteturas CUDA (*Compute Unified Device Architecture*) e OpenCL (*Open Computing Language*). Muitas dessas aplicações são de natureza gráfica e se encaixam na classe de topologia estática. Nesse caso, elas procuram explorar a interoperabilidade entre as duas arquiteturas para tirar máximo proveito de cada uma. Essa interoperabilidade entretanto, adiciona um *overhead* nas aplicações e diversos estudos mostram que o uso exclusivo de OpenGL, nesses casos, tem um melhor desempenho (VASSILEV, 2011; FRATARCANGELI, 2011; GRIFFIN *et al.*, 2012; GORDON *et al.*, 2010; SANS; CARMONA, 2016). E, quando mapeamos, de forma apropriada, algum processamento numérico pesado em circuito dedicado do *pipeline* gráfico como em (GRIFFIN *et al.*, 2012), a superioridade pode ser imbatível por implementações usando as funcionalidades expostas por CUDA.

Baseado nesses estudos, este trabalho investiga como é possível acessar os mesmos recursos de *hardware* disponíveis em *frameworks* de mais "baixo nível" através da biblioteca gráfica OpenGL com o intuito de evitar o *overhead* gerado pela interoperabilidade de CUDA com OpenGL e comparar o desempenho em ambos os casos.

1.1 Motivação

Trabalhos como o de Griffin et al. (GRIFFIN *et al.*, 2012) mostram como o uso engenhoso de recursos indisponíveis em CUDA mas presentes em OpenGL, como *pixel blending*, podem resultar em melhor desempenho, enquanto outros trabalhos, como o de Nießner et al. (NIESSNER *et al.*, 2015) mostram como alguns recursos gráficos, como *tessellation shaders*, permitem acessar recursos que não são explicitamente expostos em OpenGL, como memória compartilhada. Aliados a trabalhos comparativos como o de Vassilev (VASSILEV, 2011) e Fratarcangeli (FRATARCANGELI, 2011) que demonstram a superioridade de desempenho de aplicações baseadas somente em OpenGL sobre aquelas com interoperabilidade entre OpenGL e CUDA, esses estudos nos motivaram a questionarmos se o uso de CUDA é necessário para obter o melhor desempenho em problemas gráficos com topologia estática.

1.2 Objetivos

Diante dos trabalhos existentes, GPU é bastante limitada na solução de problemas de topologia dinâmica. Portanto, limitamos o nosso estudo comparativo em duas classes de problemas de topologia estática:

- vértices com valência limitada; e

- vértices com valência arbitrária.

O acesso aleatório à memória, intrínseco ao acesso à vizinhança de primeira ordem, ou em inglês *1-ring*, de um vértice, e a sincronização entre os diferentes passos necessários para os algoritmos existentes de ambos os problemas apresentados é um grande desafio para os quais existem algumas soluções implementadas em CUDA. Porém, sabemos que diferentes estudos mostram que implementações baseadas somente em OpenGL podem apresentar melhor desempenho quando comparadas com implementações com interoperabilidade entre CUDA e OpenGL, desde que os aceleradores gráficos possam ser adequadamente aproveitados.

O objetivo deste trabalho é implementar aplicações das duas classes de problemas para entender como otimizar esses acessos e sincronizações utilizando apenas as ferramentas disponíveis no *pipeline* gráfico da OpenGL, e obter um desempenho competitivo com as implementações existentes em CUDA. Com isso esperamos também entender melhor os recursos de *hardware* usados implicitamente pela OpenGL e achar as correspondências entre esses recursos nas duas APIs.

1.3 Problemas

Diversos problemas de simulação e computação gráfica são dependentes da vizinhança de um vértice. Nós focamos naqueles de topologia estática e os subdividimos em duas classes. O primeiro passo para alcançar nossos objetivos é a seleção de um aplicativo representativo de cada classe para estudo.

Selecionados os aplicativos, temos que avaliar quais recursos de *hardware* e quais técnicas de otimização são mais promissores para investigarmos. O principal empecilho é o fato de que, como OpenGL é uma API independente de plataforma, os padrões que ela define não entram em detalhes de implementação e uso de recursos de *hardware*. Alguns desses detalhes são amplamente documentados, como as otimizações de *cache* para texturas, enquanto outros detalhes precisam ser conjecturados e validados através de testes empíricos.

1.4 Contribuições

Este trabalho categoriza aplicações gráficas em duas classes, de topologia estática e de topologia dinâmica, apresentando para a classe de topologia estática comparações entre diferentes implementações: baseadas em OpenGL; e em CUDA/OpenGL, comprovando resultados anteriores de que a adaptação de algoritmos que demandam visualização para utilizarem somente OpenGL possibilita a obtenção de desempenho tão bom quanto ou melhor que CUDA.

Para tanto, apresentamos um novo uso não-gráfico de *tessellation shaders* e renderização instanciada. Ao invés de refinar uma malha, nós usamos um *patch* abstrato para ganhar

acesso direto à memória compartilhada. No lugar de desenhar múltiplos objetos em uma chamada nós utilizamos a tecnologia de renderização instanciada para melhorar o acesso sequencial aos dados. Ambos os resultados são diretamente aplicáveis à otimização do desempenho de *shaders*.

1.5 Organização

Esta dissertação está organizada da seguinte forma. O Capítulo 2 faz uma revisão da arquitetura das GPGPUs atuais e dois dos *frameworks* usados para programá-las: OpenGL e CUDA. O Capítulo 3 apresenta uma breve revisão dos trabalhos anteriores em problemas de topologia estática com valência limitada, explica o problema de estimativa de tensores de curvatura em malhas triangulares, as implementações realizadas, os resultados comparativos entre elas e, por fim, discussões sobre esses resultados. O Capítulo 4 segue a mesma estrutura, mas está relacionado a problemas de topologia estática com valência ilimitada, focando no problema de simulação de N-corpos. No Capítulo 5 apresentamos nossas conclusões e propomos trabalhos futuros, dados os resultados obtidos.

2 Arquitetura da GPU

A arquitetura geral das GPUs segue o padrão definido por von Neuman, sendo dividida em processadores e memórias ligados através de um barramento de comunicação. Porém, devido à sua origem como processadores gráficos, a arquitetura das GPUs tem certas características específicas. Nesta seção apresentamos as especificidades da arquitetura das GPUs da geração Maxwell. Focamos nas fabricadas pela empresa NVidia, com a interface de programação CUDA (*Compute Unified Device Architecture*), destinada a computação paralela de propósito geral nessas GPUs. Em (ZHANG *et al.*, 2011), Zhang et al. discorrem sobre as diferenças nas arquiteturas das GPUs NVidia e ATI/AMD para as quais o CUDA pode apresentar desempenho diferente. A principal diferença se dá no modelo de execução, mas a arquitetura de memória (ponto mais revelante neste trabalho) é equivalente e os pontos levantados aqui podem ser generalizados para as GPUs fabricadas por ambos os fabricantes.

2.1 Arquitetura dos Processadores

As GPUs, como o próprio nome diz, começaram como *chips* específicos para processamento gráfico. Ou seja, elas foram criadas para processar imagens *pixel* a *pixel* em tempo real. Isso foi conseguido paralelizando o máximo possível o processamento fazendo os cálculos para cada *pixel* de forma independente e concorrente. Com o tempo esses processadores foram sendo otimizados para processar não só *pixels*, mas malhas poligonais, texturas e outros elementos gráficos.

Com o advento das GPGPUs esses processadores começaram a se tornar mais genéricos mas sem perder a sua característica mais marcante de paralelismo em dados e simplicidade. A versão mais atual dos *chips* da fabricante NVidia são os da família Volta (GV100) (NVIDIA, 2017) que, durante o desenvolvimento desse trabalho, estão disponíveis apenas em poucas placas para uso profissional: Titan V, Quadro GV100 e gDX (para servidores). Sua antecessora é a família Pascal (GP104), que implementa a versão (*compute capability*) 6.0 do CUDA (NVIDIA, 2016). Uma visão geral de um processador GP104 é dada na Figura 1. O Apêndice A descreve brevemente esta geração de GPU Nvidia como outras três, Fermi, Kepler e Maxwell, utilizadas neste trabalho.

Figura 1 – Visão geral da arquitetura de um *chip* Pascal GP104 (NVIDIA, 2016)

As GPUs são compostas por centenas de processadores mais simples numa arquitetura chamada de massivamente paralela. Esses processadores são chamados de *Stream Multiprocessors* (SM) e cada um deles pode executar várias centenas de *threads*. Uma coleção dessas *threads* executando um mesmo *kernel* (uma subrotina implicitamente paralela, análoga aos *shaders* dos *pipelines* programáveis de GPUs) é chamada de *grid*. Um *grid* é dividido em blocos de *threads*, que são conjuntos de *threads* executando dentro de um mesmo SM, de forma que todas as *threads* do bloco têm acesso a uma memória compartilhada e podem ser sincronizadas de forma explícita. Por último, as *threads* de um bloco são agrupadas em *warps* de 32 *threads*, cada um dos quais é executado de forma SIMT (*single instruction multiple threads*) em cada SM, e um SM pode executar múltiplos *warps* ao mesmo tempo. Ou seja, dentro de um mesmo *warp* todas as *threads* executam a mesma instrução, cada uma com seu conjunto de dados, mas diferentes *warps* podem estar executando diferentes instruções, inclusive podendo estar executando *kernels* diferentes (NVIDIA, 2012b).

A Figura 2 mostra a relação entre *threads* e SMs de forma clara e sucinta. Uma *thread* é executada por um *core*. Cada bloco de *threads* é mapeado para um SM e, por fim, um *kernel grid* (conjunto de blocos de *threads* executando o mesmo *kernel*) é executado por uma GPU.

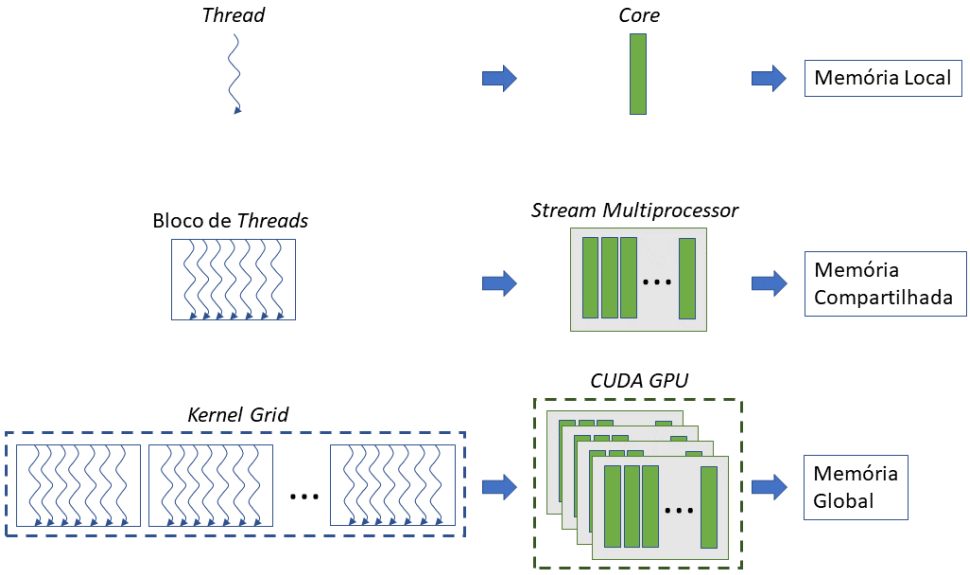

Figura 2 – Mapeamento entre *threads* e SMs

O escalonador de *warps* é o responsável por escalonar os diferentes *warps* para execução das instruções de um *kernel*, como ilustra a Figura 3. Nesse caso temos duas unidades de despacho de instrução em cada SM. Cada unidade é responsável por despachar as instruções a serem executadas por cada *warp* que, por sua vez, são executados em um simples esquema *round robin*, atribuindo frações de tempo iguais para cada *warp* de forma cíclica.

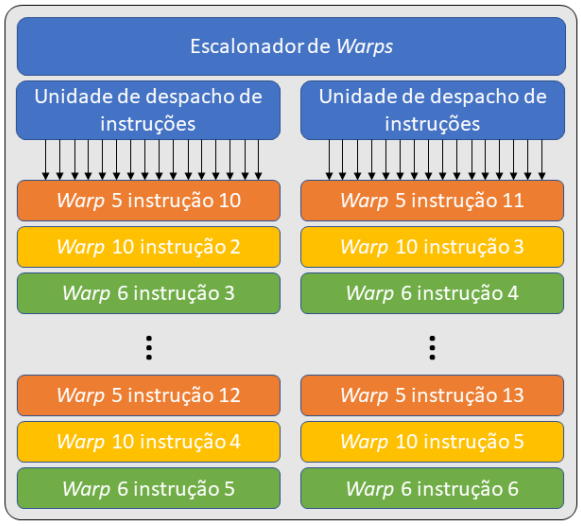

Figura 3 – Exemplo de escalonamento de *warps* em um SM

Essas características dos processadores de uma GPU precisam ser conhecidas por um programador para que o *kernel* desenvolvido possa ser otimizado, aproveitando ao máximo as vantagens dessa arquitetura. Algumas estratégias de otimização são apresentadas na seção 2.3.

2.2 Arquitetura da Memória

Assim como a memória da CPU, nas GPUs a memória também é organizada de forma hierárquica, com uma memória global, maior e mais lenta, *caches* (L1 e L2) para cada SM que se comportam como os *caches* de mesmo nível da CPU e memórias locais (registradores) para cada *thread*. Um exemplo dessa hierarquia pode ser visto na Figura 4. Porém, como ocorre com os processadores, a memória das GPUs também apresenta algumas especificidades que precisam ser observadas pelo desenvolvedor para que os *kernels* implementados possam rodar da forma mais otimizada possível.

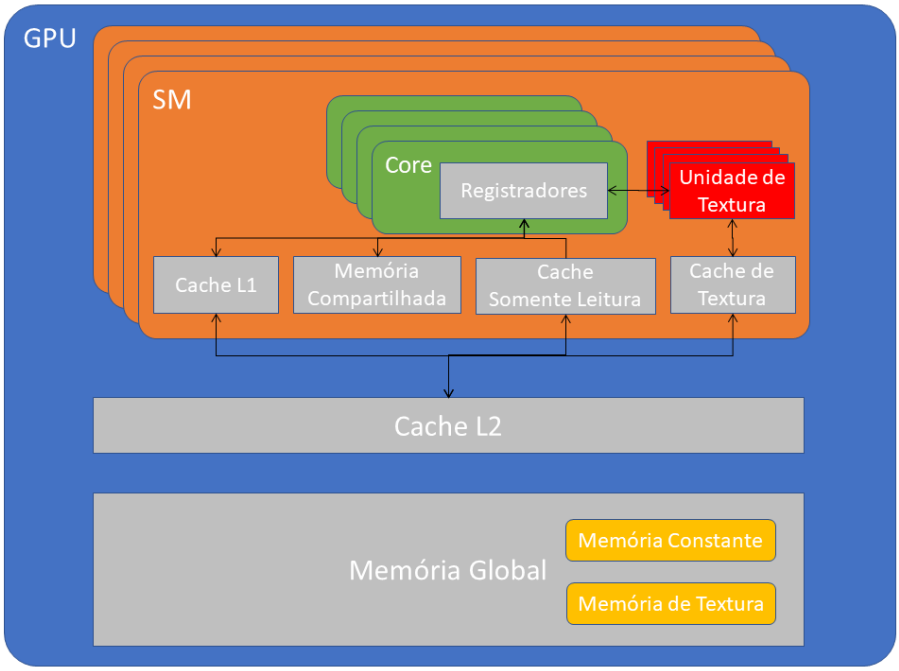

Figura 4 – Organização hierárquica das memórias de uma GPU.

Uma característica marcante da memória global da GPU é a sua grande largura de banda e alta latência. Ou seja, poucos acessos que carreguem grande quantidade de dados para as memórias mais rápidas são preferidos. Para tirar proveito dessa capacidade de acessar um maior número de dados da memória global por requisição, os SMs têm a capacidade de organizar os acessos feitos em uma instrução executada por um *warp* de forma que eles sejam feitos no menor número de requisições possível. Esses acessos são agrupados em acessos contíguos de 32, 64 ou 128 *bytes* da melhor forma possível (NVIDIA, 2012a). A Figura 5 mostra alguns exemplos de como esse agrupamento de acessos é realizado. Observe que no primeiro caso os

acessos são feitos de forma aleatória, mas todos os dados acessados pelo *warp* estão dentro de um bloco alinhado de 64 *bytes* e podem ser carregados em apenas uma transação. No segundo exemplo, apesar das *threads* do *warp* acessarem dados contíguos na memória os mesmos não estão alinhados em 64 *bytes* e nem em 128 *bytes*. Dessa forma são necessárias duas transações para ler os dados necessários à execução das *threads*: um bloco de 64 *bytes* e um bloco de 32 *bytes*. O último exemplo é igual ao anterior mas, nesse caso, os dados sendo acessados estão dentro de um bloco alinhado em 128 *bytes* e podem ser lidos com apenas uma transação.

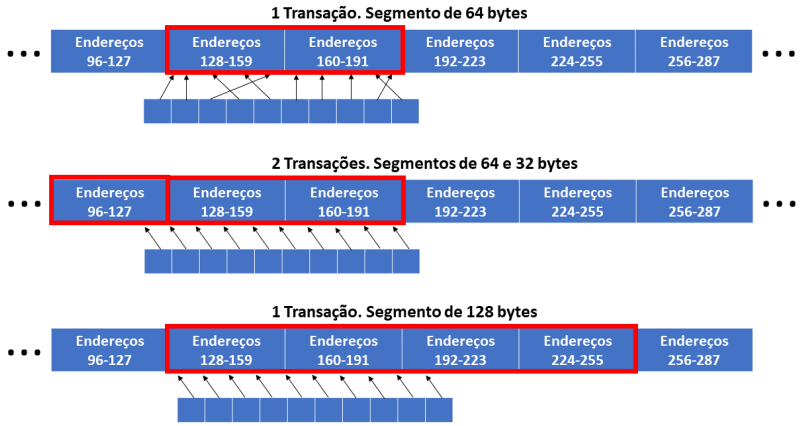

Figura 5 – Exemplo de acesso à memória global agrupado no menor número de segmentos possível.

As GPUs tem mais dois tipos de memória além das já citadas. A primeira é a memória compartilhada. Essa é uma memória rápida e pequena que é acessível por todas as *threads* de um *warp*. Porém, o acesso a esse recurso não tem controle de concorrência por *hardware*, cabendo ao usuário garantir um acesso seguro para evitar condições de disputa. A memória de textura é o último tipo presente nas GPUs e é apenas uma diferenciação lógica da memória global que permite o uso de dois *hardwares* periféricos, herança direta das origens gráficas desses *chips*, que podem oferecer algumas vantagens. O primeiro *hardware* específico é uma memória *cache* destinada apenas a esse tipo de memória. Ele é um *cache* global otimizado para tirar proveito da localidade de dados bi-dimensionais. O outro *hardware* específico disponibilizado para esse tipo de memória são as unidades de textura. Essas unidades permitem que os acessos a uma textura possam ser feitos no espaço real, com interpolação feita por *hardware* (NVIDIA, 2012a). Esses dois tipos de memória também são mostrados na Figura 4.

2.3 Estratégias de Otimização para GPGPU

Conforme mostrado anteriormente, a GPU tem diversas características únicas quando comparada à já conhecida arquitetura de uma CPU. Nas próximas seções discorremos sobre algumas estratégias de otimização advindas das peculiaridades de uma GPU e que são levadas em

consideração durante o desenvolvimento da solução do problema pesquisado no contexto deste projeto de Mestrado.

2.3.1 Simplificação dos *Kernels*

A primeira estratégia que podemos extrair vem da característica SIMT dos processadores da GPU. Devido ao agrupamento de *threads* em *warps* e a execução desses *warps* de forma que todas as *threads* estejam sempre processando a mesma instrução, criar *kernels* rebuscados e com vários caminhos de execução (laços e condicionais) diminui a eficiência de todo o *warp*. Isso ocorre porque como todas as *threads* do *warp* precisam executar a mesma instrução, se apenas uma *thread* entrar em um caminho diferente todas as outras deverão esperar (executar instruções que serão jogadas fora) até que o fluxo de execução volte para um caminho válido para elas. Dessa forma, mesmo que dois *kernels* sejam muito parecidos, é mais eficiente mantê-los separados, ao invés de criar um único *kernel* com uma condicional (NVIDIA, 2018).

2.3.2 Maximização da Ocupação da GPU

Como mostrado anteriormente, a arquitetura das GPUs se baseia nos SMs e esses tem um limite de *threads* e *warps* que podem executar simultaneamente. Além disso, cada SM tem um limite de registradores que cada *thread* pode usar, assim como um limite de memória compartilhada (NVIDIA, 2012b). Essas limitações podem fazer com que um *kernel* mal implementado tenha uma ocupação pobre da GPU. Ou seja, cada SM estará utilizando apenas uma pequena porcentagem da sua capacidade ótima de *threads* simultâneas. Isso pode causar um grande impacto no desempenho de algoritmos para GPGPU (LOBEIRAS *et al.*, 2012).

A estratégia de simplificação de *kernels* descrita na seção 2.3.1 ajuda na melhoria da ocupação dos SMs. Outras formas de tentar melhorar esse quesito é a reutilização de variáveis locais, ou a correta utilização de escopos locais no código. Declarando variáveis dentro de escopos locais faz com que os recursos utilizados por elas sejam liberadas quando o escopo terminar, possibilitando seu uso em um novo escopo local mais adiante no programa. A correta chamada de execução dos *kernels* (com a dimensão correta dos blocos sendo lançados) também melhora a ocupação da GPU. O *kit* de desenvolvimento distribuído pela NVidia contém uma planilha para o cálculo dos parâmetros de ocupação ótima para cada *kernel* (NVIDIA, 2012a).

2.3.3 Minimização de Acessos à Memória Global

Conforme discutido na seção 2.2, o acesso à memória global da GPU é o mais custoso das memórias disponíveis, mas é o único meio de comunicação com a CPU, além de ser o recurso mais abundante. Sendo assim, é de se esperar que uma das estratégias de otimização seja a diminuição das requisições feitas a esse recurso. Existem duas formas de se alcançar esse objetivo.

A primeira forma de minimizar os acessos à memória global é o agrupamento das porções de memória a serem acessadas pelas *threads* de um *warp*. Esse agrupamento deve ser feito de forma que os *warps* acessem grupos contíguos de dados para que possamos tirar proveito da alta largura de banda dessa memória, maximizando a quantidade de dados obtidos em cada acesso. Lobeiras et al. (LOBEIRAS *et al.*, 2012) mostram o impacto dos acessos não agrupados no desempenho de um algoritmo sendo executado na GPU.

Outra forma de limitar o número de requisições feitas é utilizar a memória compartilhada como um *cache* gerenciado pelo desenvolvedor. Ou seja, utiliza-se esse recurso como uma memória mais barata, fazendo com que cada *thread* copie uma certa parte dos dados da memória global no início da execução e transfira os resultados para ela no final. Porém, essa estratégia oferece alguns riscos. Um deles é a necessidade de sincronizar as *threads* para se certificar que todos os dados estejam presentes ou que toda a computação tenha terminado. Outro possível risco é a diminuição da porcentagem de ocupação da GPU devido a um aumento do uso da memória compartilhada que, como visto na seção 2.3.2, influencia em quantas *threads* cada SM será capaz de executar. É necessário, portanto, fazer uma análise dos ganhos e perdas ao se utilizar essa estratégia.

2.4 Interface de Programação de Aplicativos

As GPUs foram originalmente projetadas para renderização em tempo real de gráficos 3D de alta resolução, em conformidade com o padrão OpenGL, uma API (*Application Programming Interface*) mantida pelo consórcio ARB (*Architecture Review Board*). Esta API expõe o procedimento de renderização como um *pipeline* gráfico. As primeiras versões do *pipeline* gráfico eram compostas de estágios não programáveis, ou funções fixas. Cada estágio continha uma otimização em *hardware* de uma parte do *pipeline*: transformações e iluminação dos vértices (T&L); rasterização de facetas triangulares espaciais; operações lógicas em *pixels*; *pixel blending*; cômputo de mapa de profundidade; e aplicação de máscaras nos *pixels* (KHRONOS GROUP, 1994).

Com a evolução das GPUs o *pipeline* gráfico foi se tornando cada vez mais programável, misturando funções fixas com estágios programáveis, também chamados de *shaders*. Isso motivou os pesquisadores a aplicarem as GPUs na computação paralela de propósito geral, que, por sua vez, impulsionou a proposta de novas APIs específicas para o desenvolvimento de programas massivamente paralelos na GPU, como a prioneira Brook (BUCK *et al.*, 2004). O CUDA foi a primeira solução apresentada pela NVidia em 2006.

Apesar de serem APIs para programação de um mesmo *hardware*, a forma como cada uma expõe seus recursos é diferenciada. Nessa seção nós apresentamos as APIs CUDA e OpenGL, seu modelo de programação e como os recursos de acesso à memória da GPU – o foco deste trabalho – são expostos.

2.4.1 CUDA

CUDA, além de definir uma plataforma de programação paralela, é também um modelo de API, criado pela empresa NVidia para permitir que desenvolvedores acessem de forma explícita os recursos das suas GPUs.

Existem diferentes formas de um programador interagir com CUDA (NVIDIA, 2018):

- através de uma extensão para a linguagem de programação C. Programas escritos nessa linguagem devem ser compilados utilizando um compilador proprietário baseado em LLVM chamado *nvcc*;
- via *Frameworks* de programação paralela tais como OpenACC e OpenMP;
- por meio de bibliotecas e *wrappers* de terceiros seja para outras linguagens de programação como Python, Fortran, quanto para aplicativos como Matlab; entre outras.

Para implementar um aplicativo em cima de CUDA, utiliza-se um modelo de programação de computação heterogênea. Nesse modelo é assumido que o sistema está dividido em *host* e *device*, cada um com sua memória e fluxo de execução. Ou seja, funções podem ser compiladas e executadas em um ou em ambos, e o programador é o responsável por gerenciar a memória de ambos.

A extensão da linguagem C possibilita a declaração de onde uma função será executada através de especificadores de localidade de sua execução. A palavra-chave *__global__* especifica que a função será executada no *device* e pode ser chamada tanto pelo *host* quanto pelo *device*. Se a função é declarada como *__device__* ela só pode ser executada e chamada pelo *device*, enquanto se declarada como *__host__* só é executada no *host*. Por fim, também é definida uma nova sintaxe (*<<<GridDimension, BlockDimension, SharedMemSize, CudaStream>>>*) para especificar a configuração de execução de uma função no *device*.

Como o programador deve gerenciar a memória do *device*, CUDA expõe diretamente vários recursos do *hardware* para acesso aos diferentes níveis hierárquicos de memória apresentados na Seção 2.2. A memória global é acessada através de simples ponteiros, sendo alocada e desalocada pelas funções *cudaMalloc* e *cudaFree*, respectivamente, enquanto a transferência de dados entre *host* e *device* é feita pela função *cudaMemcpy*. CUDA ainda permite alocar memórias travadas no host, *locked* ou *pinned* em inglês. Essas memórias não podem ser paginadas pelo sistema operacional e se beneficiam de melhores taxas de transferência com o *device* usando acesso direto à memória (DMA).

O acesso à memória compartilhada é realizado através do uso da palavra-chave *__shared__* quando uma variável é declarada, devendo o tamanho das variáveis nesse caso ser definido em tempo de compilação. Para alocar memória compartilhada em tempo de execução,

pode-se definir uma única variável (ponteiro) como *extern __shared__* e o tamanho da memória compartilhada alocada é definido como o terceiro parâmetro da configuração de execução do *kernel*.

O CUDA também expõe um sub-conjunto dos recursos de textura da GPU. Para acessar esses recursos o *kernel* precisa acessar a memória utilizando uma referência de textura através de funções específicas *tex*N*D*, com $N \in \{1,2,3\}$. A referência de textura define o tipo, a dimensionalidade e o modo de leitura da textura, e precisa ser ligada a uma memória do *device* no *host* antes da execução do *kernel* com a função *cudaBindTexture*N*D*.

2.4.2 OpenGL – *Pipeline* Gráfico

OpenGL é uma API gráfica cujo objetivo é expor diversos recursos de uma *GPU* de forma simplificada, independente do fabricante do *hardware* e do sistema operacional e do sistema de janelas. Para tanto a OpenGL define um modelo de programação baseado em estados, com um típico programa composto por:

- inicialização dos estados globais;
- configuração do estado corrente para renderização;
- chamada de renderização e execução do *pipeline* gráfico (Figura 6) na GPU.

A inicialização normalmente começa com a criação de um contexto OpenGL. Dentro desse contexto são criados diferentes objetos de memória (*buffer objects*, texturas), definindo-se os tipos e a estrutura dos dados que serão guardados em cada objeto. É nesse momento também que programas (conjuntos de *shaders* implementados na linguagem GLSL) são criados e compilados.

A configuração do estado corrente define qual programa será executado no *pipeline* gráfico e quais objetos de memória serão utilizados na próxima chamada de renderização. O estado corrente também define como serão executados os estágios fixos do *pipeline* após o *fragment shader* como, por exemplo, se o *blending* será ativado e qual será a função aplicada no mesmo. Ainda uma outra configuração possível é a ativação do *transform feedback*. Esse mecanismo permite que os resultados de qualquer estágio antes da rasterização (*vertex*, *tessellation* e *geometry*) sejam gravados diretamente em qualquer *buffer* conectado permitindo a ressubmissão desses dados diretamente ao *pipeline* gráfico, sem a necessidade de passar pela CPU.

Depois que o estado corrente foi corretamente atualizado, o próximo passo é realizar a execução da renderização na GPU. Esse processo é disparado através de chamadas de funções de *draw*, ou de *dispatch* no caso de *compute shaders*. As chamadas de *draw* são realizadas sobre um conjunto de primitivas definido em um objeto de memória chamado *Vertex Buffer Object*.

A partir da versão 3.1 da OpenGL o recurso de renderização instanciada foi introduzido. Esse recurso permite fazer uma chamada de *draw* que é executada *n* vezes para um mesmo objeto, diminuindo a necessidade de cópia de dados entre CPU e GPU.

A Figura 6 mostra o *pipeline* de renderização da versão 4.5 da OpenGL (KHRONOS GROUP, 2014). Nela podemos ver o *vertex* e o *fragment shader*, que foram os primeiros estágios a se tornarem programáveis. Também se notam os novos recursos introduzidos ao longo dos anos: o *geometry*, *tessellation* e *compute shaders*. O *geometry shader* foi incluído na versão 3.0 e é responsável por mudar a geometria das primitivas sendo renderizadas. O *tessellation shader*, introduzido na versão 4.0, serve para refinar malhas arbitrárias. Ele é composto por três estágios: o *tessellation control* e *evaluation* que são programáveis e o *tessellation primitive generator* que é fixo. Nesse trabalho apenas o *tessellation evaluation shader* (TES) foi utilizado. Por fim, o *compute shader* foi o último recurso a ser adicionado e permite que o desenvolvedor despache programas não-gráficos numa maneira similar ao CUDA, mas ainda mantendo acesso a alguns mecanismos do *pipeline* gráfico, tais como *texture fetches*, *uniforms* e carregamento/armazenamento de imagens.

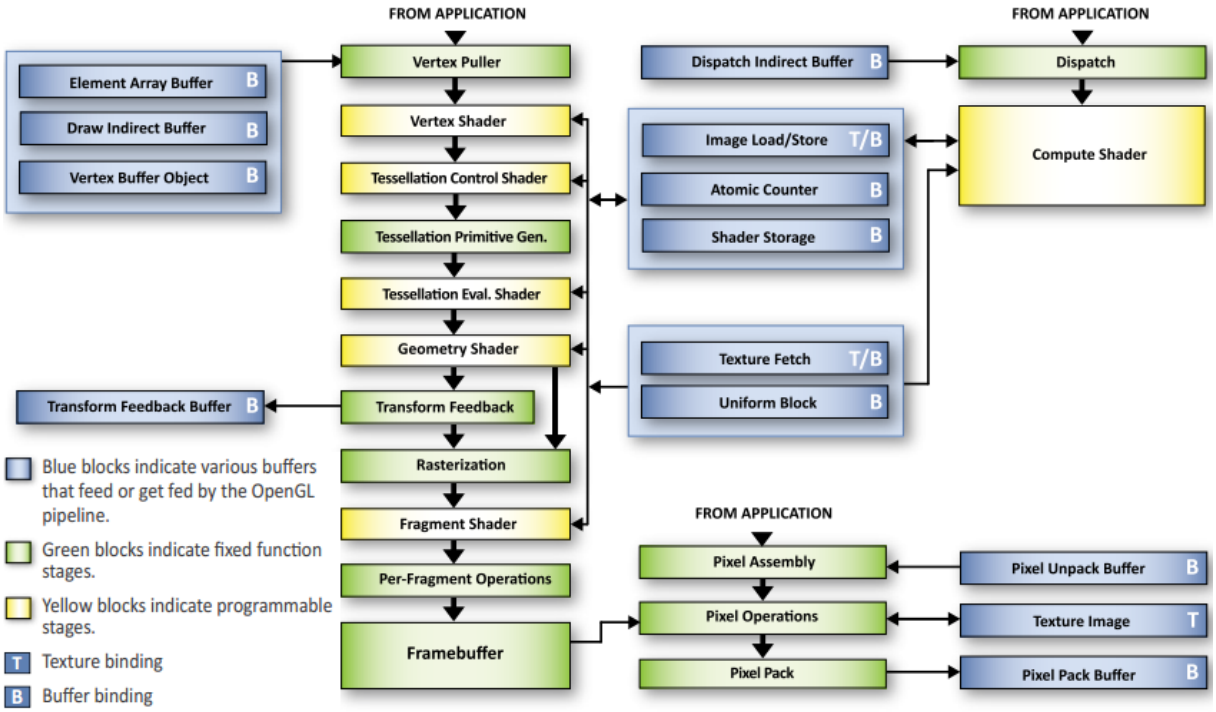

Figura 6 – *Pipeline* gráfico da versão 4.5 do OpenGL (KHRONOS GROUP, 2014).

O acesso aos diferentes tipos de memória descritos na Seção 2.2 não é tão aparente em OpenGL, como em CUDA. Ele é abstraído em funções gráficas do *pipeline*. O uso das otimizações de *hardware* para acesso, *caching* e interpolação de texturas é explícito através da criação de objetos de textura e do seu acesso através de funções específicas nos *shaders*. Já o uso das otimizações de *caching* para memórias constantes é definido pelo parâmetro *usage* quando se aloca a memória de um objeto com a função *glBufferData*. Por exemplo, definir

um objeto de memória como *GL_STATIC_DRAW* comunica à OpenGL que essa memória é constante e será utilizada várias vezes em chamadas de renderização, sendo uma boa candidata para *caching* de memórias constantes (KHRONOS GROUP, 2014). A memória compartilhada, por sua vez, é acessível de forma explícita por *compute shaders* através do uso da palavra-chave *shared* na declaração de uma variável. Esse comportamento é parecido com o de CUDA, mas é limitado por não permitir a definição do tamanho da memória compartilhada em tempo de execução (KHRONOS GROUP, 2014). Para outros *shaders* o acesso à memória compartilhada não é aparente. Segundo Niessner et al. (NIESSNER *et al.*, 2015) ela é utilizada para armazenar os dados de um *patch*. Outras formas de abstrações dos acessos otimizados à memória não foram encontradas na literatura.

2.4.3 Interoperabilidade

A interoperabilidade entre CUDA e OpenGL é feita através de regiões de memória alocadas por funções OpenGL e registradas em um contexto CUDA para acesso de leitura e/ou escrita (NVIDIA, 2018). As memórias registradas precisam então ser mapeadas para uso nos *kernels* CUDA e posteriormente desmapeadas para que possam ser novamente utilizadas pelo programa OpenGL. Três tipos de memória OpenGL são suportadas:

- *Buffer*: Representada em CUDA como um ponteiro para memória do dispositivo;
- *Renderbuffer* e Textura: São tratadas como imagens e, por isso, representadas como um *CUDA array*. Elas precisam ser atreladas a uma referência de textura ou superfície antes de poderem ser acessadas.

Para poder mapear e compartilhar as memórias de um contexto OpenGL o mesmo deve pertencer à mesma *thread* que está realizando as chamadas CUDA de interoperabilidade.

2.5 Discussões

A interoperabilidade entre CUDA e OpenGL permite aos desenvolvedores utilizar os pontos fortes, executando algoritmos massivamente paralelos na GPU sem precisar de "truques" para enganar o *pipeline* gráfico, mas ainda assim tendo acesso a esse para realizar as operações de renderização. O grande ganho da interoperabilidade vem do fato de ambos os contextos compartilharem a mesma área de memória dentro da GPU, evitando o alto custo de transferências com a memória da CPU.

Apesar do ganho proporcionado pelos dados não saírem da GPU, a interoperabilidade entre CUDA e OpenGL ainda impõe algumas barreiras que podem atrapalhar o desempenho de algumas aplicações. A sincronia necessária entre a execução dos dois contextos, computação e renderização, faz com que essas etapas não possam ser executadas simultaneamente

na GPU. O constante mapeamento e desmapeamento das áreas de memória entre as mudanças de contexto também trazem um *overhead* e podem afetar o desempenho.

Diferentes pesquisadores buscaram avaliar o desempenho de aplicações com etapas de computação e renderização tanto utilizando interoperabilidade entre CUDA e OpenGL como utilizando apenas OpenGL, como veremos nos próximos capítulos. Esses trabalhos reforçam o fato de que, quando cálculos de propósito geral são necessários em uma aplicação de renderização, realizar o seu cômputo utilizando somente a API OpenGL pode resultar em maior eficiência.

Embora aplicativos construídos em ambas as APIs sejam executados sobre o mesmo *hardware*, estudos anteriores mostram que o desempenho obtido pode ser bem diferenciado. Cabe a pergunta se isso é devido somente à diferença nos modelos de programação ou se decorre do uso ineficiente dos recursos expostos por ambas as APIs. Nos Capítulos 3 e 4 nós investigamos a equivalência entre as funcionalidades das APIs para 2 classes de aplicações que envolvem padrão de acessos de memória irregular dos dados da vizinhança de primeira ordem: valência limitada e valência arbitrária.

3 Aplicações com Vértices de Valência Limitada

A primeira sub-classe de problemas de topologia estática são os problemas onde o número de valência de cada vértice é limitado. Nessa categoria os vértices tem um limitante superior no número de vizinhos de primeira ordem independente da quantidade de vértices presentes na estrutura do objeto de interesse. A simulação de tecidos (*cloth*) e a estimativa de tensores métricos e de curvatura de uma superfície a partir de uma malha triangular são dois exemplos dessa classe de problemas. Por serem aplicações de processamento numérico pesado, são bons candidatos para a comparação entre o desempenho de aplicações implementadas com OpenGL e o de aplicações que exploram a sua interoperabilidade com CUDA. O principal desafio é como mapear, através da API OpenGL e através do CUDA, a natureza variável da conectividade dos vértices envolvidos nestas aplicações num padrão de processamento para as quais as GPUs são otimizadas. Na Seção 3.1 mostramos os trabalhos mais relevantes neste tópico em que os seus autores foram unânimes na superioridade do desempenho da API OpenGL aplicando somente os recursos disponíveis nas versões da OpenGL inferiores à OpenGL 4.0.

O recurso *tesselation shader* foi introduzido na versão 4.0 da OpenGL. Sabendo que nos estágios do *tessellation shader* tem-se acesso a todos os vértices de um *patch* (KHRONOS GROUP, 2014), levantamos a hipótese de que o *hardware* que está por baixo disponha de algum mecanismo de acessos mais eficientes aos dados não contíguos da memória global e conjecturamos que é possível utilizar o circuito de *tessellation* como uma forma de aprimorar ainda mais os acessos à vizinhança de primeira ordem, isto é, aos vértices adjacentes, de um vértice. Além disso, acreditamos que com o uso do *tessellation shader* nós nos beneficiaríamos de novas otimizações de *driver* e *hardware* de forma mais acentuada que outros recursos mais maduros como *geometry shader* e *blend*. Para demonstrar essa hipótese optamos pelo problema de estimativa de tensores de curvatura por ser um problema de escopo mais fechado.

Na Seção 3.2 apresentamos brevemente os conceitos básicos de tensores métrico e de curvatura e o método algébrico envolvido. Em seguida, na Seção 3.3, descrevemos sucintamente um procedimento de estimação proposto pelo Rusinkiewicz que é altamente paralelizável. Na sequência, mostramos na Seção 3.4 uma implementação do algoritmo em GLSL (*OpenGL Shading Language*) por Griffin et al. (originalmente em HLSL, a linguagem de *shaders* do DirectX) que explora o recurso de *blending* disponível no *pipeline* gráfico. Na Seção 3.5 apresentamos nossa implementação utilizando *tessellation shaders*, assim como uma implementação em CUDA com otimização de acesso à memória utilizando memória compartilhada para servir de base de comparação. Por fim, na Seção 3.5.3, apresentamos os resultados de desempenho obtidos com execuções em três GPUs NVidia de diferentes gerações, Fermi, Kepler e

Maxwell, discutindo em seguida o porquê de nosso método não apresentar o resultado esperado, mas ainda ter o maior ganho de desempenho nas gerações mais novas de GPU NVidia.

3.1 Trabalhos Relacionados

Fratarcangeli (FRATARCANGELI, 2011) implementou um programa de simulação de tecidos interagindo com primitivas 3D simples, como planos e esferas. A dinâmica da deformação dos tecidos é baseada na física do modelo massa–mola, em que as amostras ("pontos" de massas) de um tecido são conectadas por molas de diferentes constantes de elasticidade e os movimentos destas amostras são regidos pela Lei de Hooke (BAYRAKTAR, 2002). Ele utilizou OpenGL para a renderização das cenas, e CPU, GLSL, CUDA e OpenCL para a simulação da dinâmica do tecido.

O tecido é representado como um reticulado 2D, $n \times n$, onde cada ponto corresponde a uma partícula. Para renderização, a posição atual, posição anterior e velocidade de cada partícula são armazenadas em três distintas texturas 2D com uma simples transformação entre as coordenadas (i, j) no reticulado e as coordenadas $(\frac{i}{n}, i \mod n)$ na textura. O uso de texturas nesse caso tira proveito da localidade geométrica dos vizinhos de primeira ordem na representação do tecido como um reticulado 2D. Entretanto, para simulação da dinâmica do tecido, nenhuma tentativa de otimização dos acessos à vizinhança de primeira ordem foi feita, com todos os dados do tecido armazenados como *buffers* na memória global.

A simulação foi realizada usando o método de *dual buffers*, onde o resultado de um passo é usado como entrada para o passo seguinte. Seus resultados mostraram claramente o ganho ao usar a GPU para os cálculos devido à alta paralelizabilidade do algoritmo massa–mola. Além disso, os resultados também mostraram que a implementação em GLSL teve melhores resultados que CUDA e essa, por sua vez, teve melhores resultados que OpenCL. A explicação dada por Fratarcangeli é que GLSL usa memórias de textura para acesso aos dados da simulação, ganhando desempenho com a memória *cache* especializada e as otimizações para acesso localizado em 2D. Ele aponta a necessidade de cópias extras de memória dentro da GPU para interoperabilidade entre CUDA e OpenGL como uma outra possível causa. Não fica claro porque as capacidades de acesso ao *hardware* de textura através do CUDA não foram utilizadas. Já no caso de OpenCL ele afirma ser difícil equilibrar o número de itens de trabalho locais e globais devido à causas desconhecidas, mas teoriza que um amadurecimento do padrão e das implementações dos *drivers* poderia melhorar o desempenho.

Em (VASSILEV, 2011) Vassilev também apresenta um estudo comparativo das diferentes implementações de simulação de tecidos baseada nos princípios de massa–mola com renderização em OpenGL e cômputos da dinâmica de deformação do tecido nas três APIs de GPGPU: GLSL, CUDA e OpenCL. Diferente de Fratarcangeli, ele implementa dois algoritmos. O primeiro, chamado EPA (*Edge-Point Algorithm*), divide o cômputo em duas fases, cada uma

num *kernel*: a primeira fase calcula as forças internas da malha, exercidas por cada mola, e a fase seguinte calcula a nova posição de cada massa levando em consideração o estado anterior, as forças internas, externas e colisões. O segundo algoritmo, PPA (*Pure Point Algorithm*) faz todos os cálculos em apenas um *kernel*. Em ambos os algoritmos os dados das partículas são armazenados em três *buffers*: posição, velocidade e normal. No PPA um quarto *buffer* de conectividade guarda os 16 vizinhos de uma partícula, assim como as constantes físicas da mola que os liga. Já no EPA mais dois *buffers* são necessários: um armazena as constantes de uma mola e os índices das partículas ligadas a ela; e outro armazena para cada partícula as 16 molas que a ligam a seus vizinhos.

Em seus resultados com todas as plataformas, Vassilev demonstra que o algoritmo PPA tem melhor desempenho que o algoritmo EPA. Isso se dá porque, apesar de duplicar os cálculos necessários para as forças internas, o PPA elimina a sincronização explícita entre os dois *kernels*, podendo os cálculos serem realizados de forma paralela e independente. Em ambos os algoritmos os resultados obtidos por Vassilev são similares aos obtidos por Fratarcangeli: GLSL tem o melhor desempenho, seguido por CUDA e, por último, OpenCL. O gargalo identificado também é o mesmo: a interoperabilidade entre visualização (OpenGL) e cômputos (CUDA/OpenCL). Entretanto, Vassilev dá mais ênfase à troca de contexto como causa para a perda de desempenho e deixa a análise do baixo desempenho de OpenCL com relação à CUDA para trabalhos futuros.

Griffin et al. exploram em (GRIFFIN *et al.*, 2012) um outro problema da classe de vértices com valência limitada: a implementação do procedimento de estimativa de tensores de curvatura em cada vértice de uma malha triangular proposto por Rusinkiewicz. Como veremos na Seção 3.4, esta estimativa requer, além das informações do vértice de interesse, as informações de todos os seus vértices adjacentes. Eles propõem uma nova solução para a implementação da estimativa de tensores de curvatura em GPU, usando apenas o *pipeline* gráfico ou interoperando com CUDA. O DirectX é utilizado para a renderização dos tensores de curvatura estimados, enquanto as estimativas são realizadas em HLSL (equivalente de GLSL para DirectX) e CUDA.

O algoritmo em HLSL utiliza *geometry shaders* e o *hardware* otimizado para *blending* como ferramentas para calcular o tensor de curvatura em cinco passos de renderização. O mecanismo de *blending* é explorado para contornar acessos à vizinhança de primeira ordem (Seção 3.4.1). Já o algoritmo em CUDA utiliza operações atômicas e memória compartilhada no lugar de *blending*, mantendo os cinco passos como cinco *kernels* separados, mais um *kernel* extra para pré-processamento dos dados de entrada. Os resultados obtidos por Griffin et al. mostram, novamente, que a implementação somente em *pipeline* gráfico tem desempenho melhor que a implementação interoperando com CUDA. Nesse caso eles argumentam que a perda de desempenho no algoritmo de CUDA se dá não por causa da interoperabilidade, mas pela soma atômica em ponto flutuante utilizada e pelo passo extra de pré-processamento necessário.

Dessa forma eles conjecturam que seria possível um melhor desempenho em CUDA do que em *pipeline* gráfico no futuro.

Sintetizando, os estudos conduzidos pelos trabalhos citados mostram que o ganho do desempenho pode ser alcançado tanto com a otimização de acessos aos diferentes tipos de memória quanto com o uso adequado do *hardware* disponível. Pensando nisso nós nos propusemos a investigar o uso dos mecanismos de otimização introduzidos com o *tessellation shader*, que provê suporte ao refinamento de malhas, para melhorar o desempenho de acessos não-contíguos à memória no problema de estimativa de tensores de curvatura.

3.2 Tensores métrico e de curvatura

Em Geometria Diferencial, o tensor métrico e o tensor de curvatura são duas noções métricas fundamentais. O tensor métrico nos permite fazer medidas sobre uma superfície (comprimento de arcos, ângulo entre duas direções e área de uma região) sem recorrer ao espaço ambiente em que a superfície está imersa. O tensor de curvatura, por sua vez, nos permite medir o curvamento de uma superfície na vizinhança de um ponto dentro de um espaço ambiente. Nesta seção apresentamos as principais equações envolvidas com estas medidas.

Seja $\mathbf{r}\colon \Omega \to R^3$ uma superfície regular $\mathscr{S}$ dada por

$$\mathbf{r}(x^1,x^2) = (x(x^1,x^2), y(x^1,x^2), z(x^1,x^2)), \tag{3.1}$$

com $x^1, x^2 \in \Omega$. Como temos

$$d\mathbf{r} = \frac{\partial \mathbf{r}}{\partial x^1} dx^1 + \frac{\partial \mathbf{r}}{\partial x^2} dx^2, \tag{3.2}$$

o comprimento ao quadrado $I(\alpha(t))$ de um arco de uma curva parametrizada $\alpha(t) = \mathbf{r}(x^1(t), x^2(t))$ pode ser expresso em

$$I(\alpha(t)) \;=\; d\mathbf{r}\cdot d\mathbf{r} = a_{\alpha\beta} dx^\alpha dx^\beta = \begin{bmatrix} dx^1 & dx^2 \end{bmatrix} \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \begin{bmatrix} dx^1 \\ dx^2 \end{bmatrix}, \tag{3.3}$$

onde

$$a_{\alpha\beta}(\mathbf{r}(a)) = \frac{\partial \mathbf{r}}{\partial x^\alpha} \cdot \frac{\partial \mathbf{r}}{\partial x^\beta}. \tag{3.4}$$

A Equação (3.3) é a *primeira forma fundamental* da superfície $\mathscr{S}$, enquanto $a_{\alpha\beta}$ são chamados de componentes de *tensor métrico*.

A *segunda forma fundamental* da superfície $\mathscr{S}$, que nos dá uma medida da variação do vetor normal $\mathbf{n}$ ao longo da curva $\alpha(t)$, pode ser expressa como

$$II(\alpha(t)) \;=\; b_{\alpha\beta}dx^{\alpha}dx^{\beta} = \begin{bmatrix} dx^1 & dx^2 \end{bmatrix} \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix} \begin{bmatrix} dx^1 \\ dx^2 \end{bmatrix}, \tag{3.5}$$

onde

$$b_{\alpha\beta} = \mathbf{n} \cdot \frac{\partial^2 \mathbf{r}}{\partial x^{\alpha} \partial x^{\beta}}, \tag{3.6}$$

com

$$\mathbf{n} = \left(\frac{\partial \mathbf{r}}{\partial x^1} \times \frac{\partial \mathbf{r}}{\partial x^2}\right) \Big/ \left\| \frac{\partial \mathbf{r}}{\partial x^1} \times \frac{\partial \mathbf{r}}{\partial x^2} \right\|. \tag{3.7}$$

Os componentes $b_{\alpha\beta}$ formam o *tensor de curvatura*.

De forma simplificada, ao consideramos um ponto P pertencente a superfície $\mathscr{S}$, $a_{\alpha\beta}$ está relacionado às propriedades métricas (comprimento e área), enquanto $b_{\alpha\beta}$ descreve o quão curva é a superfície na vizinhança de P. Equações de Weingarten relacionam, por sua vez, por meio de componentes de tensores de curvatura as derivadas da normal (unitária) em P em relação às curvas coordenadas com os seus vetores tangentes a $\mathscr{S}$, $\frac{\partial r}{\partial x^1}$ e $\frac{\partial r}{\partial x^2}$,

$$\begin{aligned} \frac{\partial \mathbf{n}}{\partial x^1} &= b_{11}\frac{\partial \mathbf{r}}{\partial x^1} + b_{12}\frac{\partial \mathbf{r}}{\partial x^2} \\ \frac{\partial \mathbf{n}}{\partial x^2} &= b_{21}\frac{\partial \mathbf{r}}{\partial x^1} + b_{22}\frac{\partial \mathbf{r}}{\partial x^2}. \end{aligned} \tag{3.8}$$

O tensor de curvatura corresponde ao que chamamos de matriz de Weingarten que possui duas propriedades que nos interessam: (1) os autovalores representam as curvaturas principais da superfície no ponto; e (2) os autovetores representam as direções principais no ponto.

3.3 Algoritmo de Rusinkiewicz

A partir da Equação (3.8) Rusinkiewicz (RUSINKIEWICZ, 2004) propôs um algoritmo para CPU baseado no método de cálculo das normais de um vértice através da média das normais de suas faces adjacentes. Para isso ele divide o problema em dois passos: (1) calcular o tensor de curvatura em cada face da malha, e (2) calcular o tensor de curvatura em cada vértice através da soma dos tensores de suas faces adjacentes ponderados pela área de Voronoi das respectivas faces em relação ao vértice. O procedimento contendo esses passos pode ser visto no Pseudocódigo 3.1.

Esse algoritmo é totalmente paralelizável. Tanto a estimativa dos tensores de curvatura em cada face triangular (Seção 3.3.1) quanto o cômputo da área de Voronoi de cada vértice

Pseudocódigo 3.1 Algoritmo de Rusinkiewicz

```
1: computa as normais dos vértices
2: constrói um sistema de coordenadas inicial em cada vértice (u_p, V_p)
3: for face ∈ malha do
4:     calcula tensor de curvatura
5:     for vértice ∈ face do
6:         calcula área de Voronoi
7:         transforma tensor de curvatura para base (u_p, V_p)
8:         soma de forma ponderada esse tensor à curvatura do vértice
9:     end for
10: end for
11: for vértice ∈ malha do
12:     Divide tensor resultante pela somatória de todas as áreas de Voronoi
13:     diagonaliza o tensor de curvatura para achar direções e curvaturas principais
14: end for
```

dentro da face (Seção 3.3.2) podem ser processados de forma independente para cada face. A estimativa das curvaturas e direções principais em cada vértice pode também ser computada de forma independente para cada vértice (Seção 3.3.3).

3.3.1 Tensor de Curvatura Aproximado por Face

Para calcular o tensor de curvatura numa face, Rusinkiewicz utiliza a Equação (3.8) para construir um sistema de equações usando as diferenças das normais dos vértices que compõem a face. Tomando como exemplo um triângulo como o descrito na Figura 7, podemos obter o seguinte sistema de equações:

$$\begin{aligned}
(n_2 - n_1)u &= b_{11}u^2 + b_{12}uv \\
(n_2 - n_1)v &= b_{12}vu + b_{22}v^2 \\
(n_1 - n_0)u &= b_{11}u^2 + b_{12}uv \\
(n_1 - n_0)v &= b_{12}vu + b_{22}v^2 \\
(n_0 - n_2)u &= b_{11}u^2 + b_{12}uv \\
(n_0 - n_2)v &= b_{12}vu + b_{22}v^2.
\end{aligned} \tag{3.9}$$

A solução desse sistema, usando o método de mínimos quadrados, nos dá os componentes b_{ij} do tensor de curvatura para a face.

3.3.2 Área de Voronoi

Para a ponderação da contribuição de cada faceta adjacente a um vértice, Rusinkiewicz (RUSINKIEWICZ, 2004) utiliza a área de Voronoi do vértice naquela face. O Pseudocó-

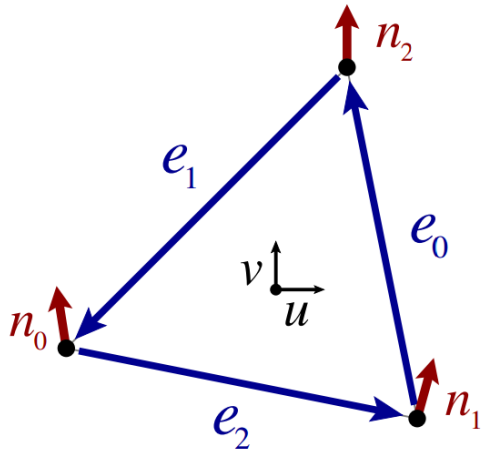

Figura 7 – Arestas e_i, vértices (pontos pretos) e vetores normais n_i de um triângulo para cálculo do tensor de curvatura

digo 3.2 descreve uma forma de cálculo das áreas de Voronoi A_0, A_1 e A_2, respectivamente, de cada vértice V_0, V_1 e V_2, de um triângulo arbitrário somente em termos dos seus lados e_0, e_1 e e_2. No Apêndice B encontram-se os detalhes da forma como se obtém as equações apresentadas no Pseudocódigo quando o triângulo é obtuso (linhas 6, 9 e 13) e não é obtuso (linha 18).

Pseudocódigo 3.2 Cálculo da área de Voronoi dos vértices de um triângulo

1: $vec\ e[3] = \{V_2 - V_1, V_0 - V_2, V_1 - V_0\}$;
2: $float\ triangleArea = cross(e[0], e[1])/2$;
3: $float\ l2[3] = \{len2(e[0]), len2(e[1]), len2(e[2])\}$;
4: $float\ bcw[3] = \{l2[0]*(l2[1]+l2[2]-l2[0]), l2[1]*(l2[0]+l2[2]-l2[1]), l2[2]*(l2[1]+l2[0]-l2[2])\}$;
5: **if** $bcw[0] \leq 0$ **then**
6: $\quad A[1] = -0.25*triangleArea*l2[2]/dot(e[0], e[2])$;
7: $\quad A[2] = -0.25*triangleArea*l2[1]/dot(e[0], e[1])$;
8: $\quad A[0] = triangleArea - A[1] - A[2]$;
9: **else if** $bcw[1] \leq 0$ **then**
10: $\quad A[0] = -0.25*triangleArea*l2[2]/dot(e[1], e[2])$;
11: $\quad A[2] = -0.25*triangleArea*l2[0]/dot(e[1], e[0])$;
12: $\quad A[1] = triangleArea - A[0] - A[2]$;
13: **else if** $bcw[2] \leq 0$ **then**
14: $\quad A[0] = -0.25*triangleArea*l2[1]/dot(e[2], e[1])$;
15: $\quad A[1] = -0.25*triangleArea*l2[0]/dot(e[2], e[0])$;
16: $\quad A[2] = triangleArea - A[0] - A[1]$;
17: **else**
18: $\quad scale = 0.5*triangleArea/(bcw[0]+bcw[1]+bcw[2])$;
19: $\quad A[0] = scale*(bcw[1]+bcw[2])$;
20: $\quad A[1] = scale*(bcw[0]+bcw[2])$;
21: $\quad A[2] = scale*(bcw[1]+bcw[0])$;
22: **end if**

3.3.3 Tensor de Curvatura Aproximado por Vértice

A estimativa do tensor de curvatura de um vértice é, por fim, calculada como uma média ponderada das estimativas dos tensores de cada face adjacente a ele. Como cada tensor é obtido em uma base diferente, todos os tensores devem ser girados para uma base comum associada ao vértice antes de suas contribuições serem somadas de forma ponderada. O pseudocódigo 3.3, proposto por Rusinkiewicz, obtem as transformações necessárias para levar o referencial antigo ao novo. Os detalhes da forma como são obtidas as fórmulas usadas no Pseudocódigo são dados no Apêndice C.

Pseudocódigo 3.3 Rotação de um sistema de coordenadas para ser perpendicular a uma nova normal

```
1:  u_new = u_old;
2:  v_new = v_old;
3:  vec norm_old = u_old CROSS v_old;
4:  float ndot = norm_old DOT norm_new;
5:  if ndot ≤ -1.0 then
6:      u_new = -u_new;
7:      v_new = -v_new;
8:  else
9:      vec perp_old = norm_new - ndot * norm_old;
10:     vec dperp = 1.0f / (1 + ndot) * (norm_old + norm_new);
11:     u_new -= dperp * (u_new DOT perp_old);
12:     v_new -= dperp * (v_new DOT perp_old);
13: end if
```

Após girado, o tensor de curvatura de cada face adjacente ao vértice é somado com peso igual a área de Voronoi do vértice naquela face dividida pelo somatório das áreas de Voronoi em todas as faces adjacentes: $\frac{A_{voronoi}}{\sum A_{voronoi}}$. Para obter as curvaturas e direções principais em cada vértice, basta aplicarmos um procedimento de diagonalização (PRESS *et al.*, 1992) dos tensores.

3.4 Implementação baseada em *Blending*

Griffin e seus colegas elaboraram uma forma de implementar o algoritmo definido por (RUSINKIEWICZ, 2004) com uso de *shaders* (GRIFFIN *et al.*, 2012). O principal gargalo, nesse caso, é o acesso à vizinhança de primeira ordem do vértice para fazer os cálculos necessários. Eles atacam o problema do acesso de forma implícita utilizando a capacidade de realizar somatórios por meio de operação de *blending* sobre um mesmo *pixel* no *fragment shader*. Ou seja, eles contornaram acessos não contíguos no cômputo das médias ponderadas dos tensores e dos vetores normais nos vértices usando o mecanismo de *blending* para acumularem os valores parciais à medida que eles são computados.

Para poder tirar proveito deste recurso, o algoritmo de Rusinkiewicz é implementado em 4 passos de renderização:

- Estimativa do vetor normal, da área de Voronoi de cada vértice e da percentagem de contribuição ao seu tensor de curvatura por cada face adjacente ao vértice.
- Cômputo das bases do sistema de coordenadas de cada vértice.
- Obtenção dos tensores de curvatura de cada vértice.
- Diagonalização dos tensores de curvatura e obtenção das direções principais por vértice.

Antes de detalharmos cada passo do algoritmo proposto por (GRIFFIN *et al.*, 2012), é necessário explicar a técnica de computação de vizinhança de primeira ordem proposta por eles.

3.4.1 Técnica de Computação de Vizinhança de Primeira Ordem

As partes do algoritmo que dependem de cálculo por face adjacente a um vértice são implementadas num *geometry shader* que tem como saída um conjunto de pontos, um para cada vértice da face. Nesse ponto não há nenhum acesso explícito à vizinhança de primeira ordem de um vértice. Porém, como todas as operações sobre a vizinhança de primeira ordem são somadas ou ponderadas, essa computação pode ser realizada com o uso de *blending* na saída da renderização.

Para que todos os valores computados por face no *geometry shader* sejam usados na estimativa dos valores de cada vértice por meio do mecanismo de *blending*, cada vértice renderizado na posição $(x,y) \in [-1,1,-1,1]$ do espaço de *viewport* deve ser mapeado de forma determinística a um ponto discreto no espaço $[0,1,0,1]$ de uma textura, conforme mostra a Figura 8. A função *idx2tc* (linhas 1–4 do Pseudocódigo 3.4) transforma o índice do vértice, *index*, no *Vertex Buffer Object* (VBO) na coordenada correspondente na textura de saída de acordo com o tamanho, *size*, da mesma. Com essa coordenada podemos fazer o caminho inverso do mapeamento do *viewport* para a textura na função *tc2pos* (linhas 5–9 do Pseudocódigo 3.4), definindo de forma determinística a coordenada de um vértice no *viewport*.

Por fim, o mecanismo de *blending* é configurado com o fator de ponderação GL_ONE e o modo de mistura GL_FUNC_ADD ($O = Src + Dst$). Ou seja, o novo valor a ser escrito na textura, Src, é adicionado ao valor existente, Dst, e esse valor é atualizado com o resultado da adição O.

3.4.2 Estimativa de Vetores Normais e Áreas de Voronoi por Vértice

A primeira etapa de renderização do método proposto por (GRIFFIN *et al.*, 2012) faz o cálculo da área de Voronoi e das normais de um vértice. Ambos são cômputos por face,

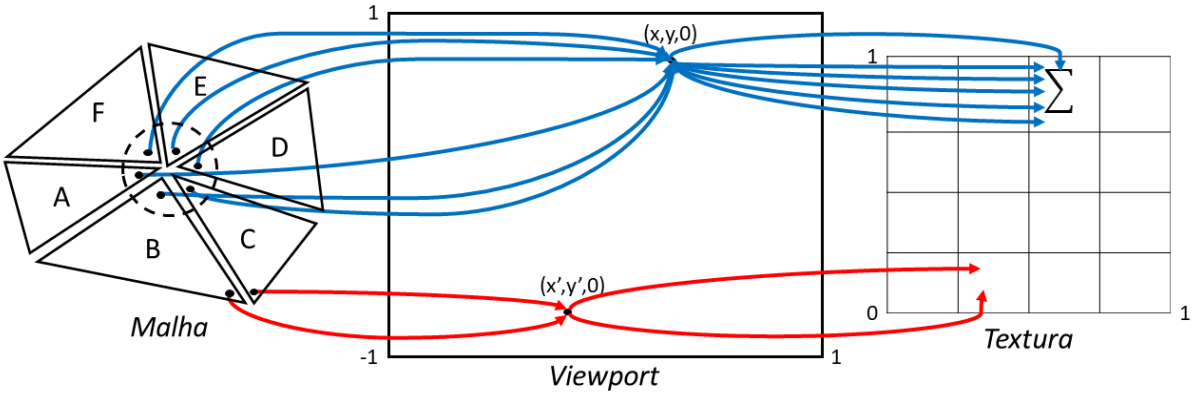

Figura 8 – Mapeamento de um vértice para a mesma posição de textura em toda sua vizinhança de primeira ordem.

Pseudocódigo 3.4 Funções para mapeamento de um vértice para sua posição de textura (idx2tc) e da posição de textura para a coordenada do *viewport* (tc2pos)

```
1: idx2tc(index){
2:   pos = vec2(index%size.x, index/size.x);
3:   return vec2(pos.x/size.x, pos.y/size.y);
4: }
5: tc2pos(texCoord){
6:   x = -1 + 2texCoord.x;
7:   y = -1 + 2texCoord.y;
8:   return vec4(x, y, 0, 1);
9: }
```

sendo realizados no *geometry shader*, que emite os resultados por vértice que compõe a face. Esses valores são então somados e ponderados nas texturas de saída utilizando o mecanismo descrito na Seção 3.4.1.

Os resultados dessa etapa são armazenados em duas texturas. A primeira textura contém, para cada vértice, o somatório das áreas de Voronoi de todas as faces adjacentes a ele. Na segunda textura é salvo o vetor normal de cada vértice: uma média dos vetores normais das faces adjacentes a ele.

3.4.3 Cômputo das Bases dos Vértices

O segundo passo é relativamente simples e constitui o cômputo de uma base para cada vértice da malha. Essa base é calculada de forma simples com um produto vetorial entre a normal do vértice e uma de suas arestas (unitárias) adjacentes e o produto vetorial do vetor resultante com a normal do vértice:

$$b_0 = cross(V_1 - V_0, n_0),$$

$$b_1 = cross(n_0, b_0).$$

Esse passo é computado por face utilizando um *geometry shader*, que emite os dois vetores da base resultante para cada vértice que compõe a face. Os dois vetores são então escritos em duas texturas diferentes, uma para cada vetor da base. Conforme explicado na Seção 3.4.1 isso gera mais de um resultado por vértice (um para cada face adjacente). Como neste caso não nos interessa agregar os múltiplos resultados calculados para um mesmo vértice, não é utilizado nenhum *blending* e a última base a ser escrita nas texturas de saída é o resultado final desse passo.

3.4.4 Cálculo do Tensor de Curvatura

Nesse passo o *geometry shader* é utilizado para calcular o tensor de curvatura da face através das Equações (3.9). Novamente a saída do *geometry shader* é configurada para ser três pontos, um para cada vértice do triângulo, que são emitidos contendo em sua estrutura de dados o valor da contribuição dessa face para o tensor de curvatura do vértice. As informações de área de Voronoi e de bases calculadas nos dois passos anteriores são lidas de suas respectivas texturas no *vertex shader*. Com isso, após a resolução do sistema de equações (Equação (3.9)) usando o método dos mínimos quadrados, o peso da contribuição do tensor de curvatura da face para o tensor dos vértices é calculado com uso da fração $\frac{area\ de\ Voronoi}{area\ da\ face}$. O tensor de curvatura da face estimado numa base arbitrária da face é então transformado para a base comum do vértice e multiplicado por esse peso. Finalmente, a contribuição de cada face adjacente a um vértice é somada na textura de saída usando o mecanismo de *blending*, conforme explicado na Seção 3.4.1.

3.4.5 Diagonalização e Cálculo das Direções Principais

O último estágio do método de renderização proposto por (GRIFFIN *et al.*, 2012) é o único que não depende do *geometry shader* por não depender de nenhuma vizinhança dos vértices. Nesse passo o tensor de curvatura, assim como os vetores da base, para cada vértice já foi calculado. A diagonalização do tensor e a estimativa das direções principais são feitas diretamente no *vertex shader*.

Como há apenas uma saída para cada vértice, diferente dos outros passos do algoritmo, o *blending* não é utilizado, e o resultado calculado no *vertex shader* é escrito diretamente na textura de saída após passar inalterado pelo *fragment shader*.

3.5 Avaliação do Potencial de *Tesselation Shaders*

Para testar nossa conjetura de que o *tessellation shader* se beneficia de otimizações de *hardware* ou *driver* no acesso aos dados dos vértices pertencentes a um *patch*, nós implementamos uma versão do algoritmo de Rusinkiewicz, com algumas modificações baseadas no

trabalho de Costa e Wu (COSTA; WU, 2012). Nós também implementamos esse mesmo algoritmo em CUDA, utilizando técnicas de otimização com memória compartilhada, conforme descrito por Wu et al. (WU *et al.*, 2013).

3.5.1 Implementação com *Tessellation Shaders*

Diferente de (GRIFFIN *et al.*, 2012), nossa implementação não tem múltiplos passos de renderização. Utilizamos uma versão modificada do algoritmo descrito por (RUSINKIEWICZ, 2004), que calcula diretamente o tensor de curvatura do vértice a partir dos seus vértices adjacentes (COSTA; WU, 2012), ao invés de calculá-lo pela soma dos tensores de curvatura das suas faces adjacentes ponderados pelas áreas de Voronoi. Portanto, a malha é renderizada apenas uma vez, com os tensores de curvatura e as direções principais computados em cada vértice e escritos em duas texturas de saída.

Para calcular os tensores utilizando o *tessellation shader* nós precisamos preparar a malha e configurar o *pipeline* OpenGL. Primeiramente o tamanho dos *patches* que serão renderizados é configurado para $\textbf{size} = \textbf{max\_adj} + 1$, sendo max_adj o número máximo de vizinhos de um vértice na malha. Com isso cada *patch* irá conter o próprio vértice e todos os seus vizinhos de primeira ordem. Um *patch* resultante no *IBO* (Index Buffer Object) seria $\{\mathbf{v}, \mathbf{v}_1, \mathbf{v}_2, ..., \mathbf{v}_{max\_adj}\}$. Nos casos onde um vértice tem menos vizinhos que o máximo, nós completamos o *patch* com o índice do vértice principal de forma que possamos identificar isso no *shader*. Por exemplo, um *patch* com n ($< max\_adj$) vértices seria representado como $\{\mathbf{v}, \mathbf{v}_1, \mathbf{v}_2, ..., \mathbf{v}_n, \mathbf{v}, ..., \mathbf{v}\}$.

Nossa implementação faz o uso dos seguintes *shaders*: *vertex shader*, *tessellation control shader*, *tessellation evaluation shader* e *fragment shader*. O *vertex shader* é utilizado apenas por ser obrigatório no *pipeline* gráfico da OpenGL, não realizando nenhuma operação sobre os dados dos vértices. Após o *vertex shader*, o *tessellation control shader* é utilizado para o cômputo de uma base local para o vértice como em (GRIFFIN *et al.*, 2012). Ainda no mesmo *shader* as diferenças das normais e os comprimentos das arestas adjacentes ao vértice são calculados e o sistema de Equações 3.9 é montado. O *tessellation control shader* emite então apenas um vértice (o principal), junto com os dados de base e as matrizes do sistema de equações, e configura os níveis de tesselação para {1, 1, 1} (externo) e {1} (interno) de forma que apenas um ponto de tesselação é gerado pelo estágio fixo do tesselador. Com isso, uma instância do *tessellation evaluation shader* é gerada por *patch*, ou seja, uma instância por vértice da malha. Durante essa etapa do *pipeline* o sistema (Equação 3.9) montado no estágio anterior é resolvido utilizando o método dos mínimos quadrados. O resultado, como explicado em (COSTA; WU, 2012), é o tensor de curvatura do vértice. Além de estimar o tensor em cada vértice, calculam-se as direções principais no *tessellation evaluation shader*. Por fim, esses dados são enviados ao *fragment shader* que os redireciona para a unidade de textura correta.

3.5.2 Implementação em CUDA com Memória Compartilhada

Nós propusemos um método genérico de diminuir o acesso não-contíguo à memória principal da GPU através do uso da memória compartilhada de um bloco de *threads* (WU *et al.*, 2013). Nesse método todos os dados acessados por um bloco de *threads* é copiado de uma só vez da memória global para a memória compartilhada da GPU. Com isso grandes quantidades de dados podem ser lidas de uma vez, aproveitando a maior largura de banda oferecida pela GPU e os acessos subsequentes a esses dados se aproveitam da baixa latência apresentada pela memória compartilhada.

Para que o acesso durante a cópia para a memória compartilhada seja contíguo na memória principal, um novo vetor de dados e um novo vetor de índices são necessários. Essas novas estruturas contém, para cada bloco, todos os dados necessários para a execução dos *kernels* daquele bloco. No nosso caso isso significa que o novo vetor de dados conterá todas as posições e normais dos vértices vizinhos, assim como as faces adjacentes para cada vértice que faça parte daquele bloco de *threads*. Apesar de poder haver duplicação dos dados devido à sobreposição de vizinhos entre blocos diferentes, a quantidade de duplicação costuma se manter em um nível aceitável.

3.5.3 Resultados

Para avaliar o desempenho das implementações apresentadas, nós fizemos a estimativa do tensor de curvatura das amostras de uma malha de sela tesselada numa quantidade de faces que varia entre 200 e 500 mil. Esse teste foi executado em três diferentes máquinas (ver Tabela 1), cada uma contendo uma GPU diferente, contemplando 3 das 4 arquiteturas mais recentes da NVidia (ver Apêndice A).

Nome da Máquina	Processador	Tamanho da RAM	GPU	# de CUDA cores
Fermi	Intel core i5	8 GB	NVidia GeForce 540M 1 GB VRAM	96
Kepler	Intel core i7	16 GB	NVidia GeForce GTX Titan 6 GB VRAM	2688
Maxwell	Intel Xeon	16 GB	NVidia GeForce GTX Titan X 12 GB VRAM	3072

Tabela 1 – Especificação das máquinas usadas nos testes descritos na Seção 3.5.3

As Figuras 9, 10 e 11 mostram o desempenho de cada algoritmo, em μs, para diferentes resoluções de uma malha discreta de uma superfície de sela.

Griffin et al. (GRIFFIN *et al.*, 2012) compararam em seu trabalho duas implementações de seu algoritmo, uma em DirectX e outra em CUDA, e demonstraram pelos resultados obtidos que a versão utilizando o *pipeline* gráfico era mais rápida que a versão em CUDA. Em nossos experimentos (Figura 11) nós demonstramos que o mesmo algoritmo de Griffin et al. implementado em OpenGL é mais rápido do que outra versão de estimação de tensores de curvatura implementada em CUDA usando otimizações baseadas em memória compartilhada. Isso fortalece a hipótese levantada de que algoritmos computacionalmente intensos, quando corretamente adaptados ao *pipeline* gráfico se beneficiam das otimizações realizadas para os mesmos

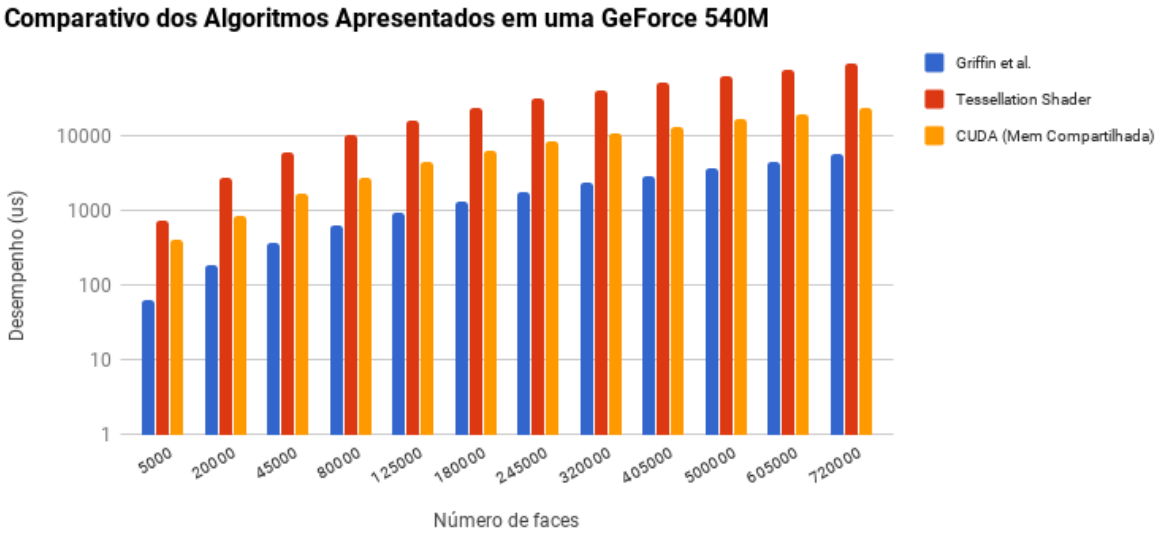

Figura 9 – Comparação do desempenho dos algoritmos apresentados em uma placa NVidia GeForce 540m (arquitetura Fermi)

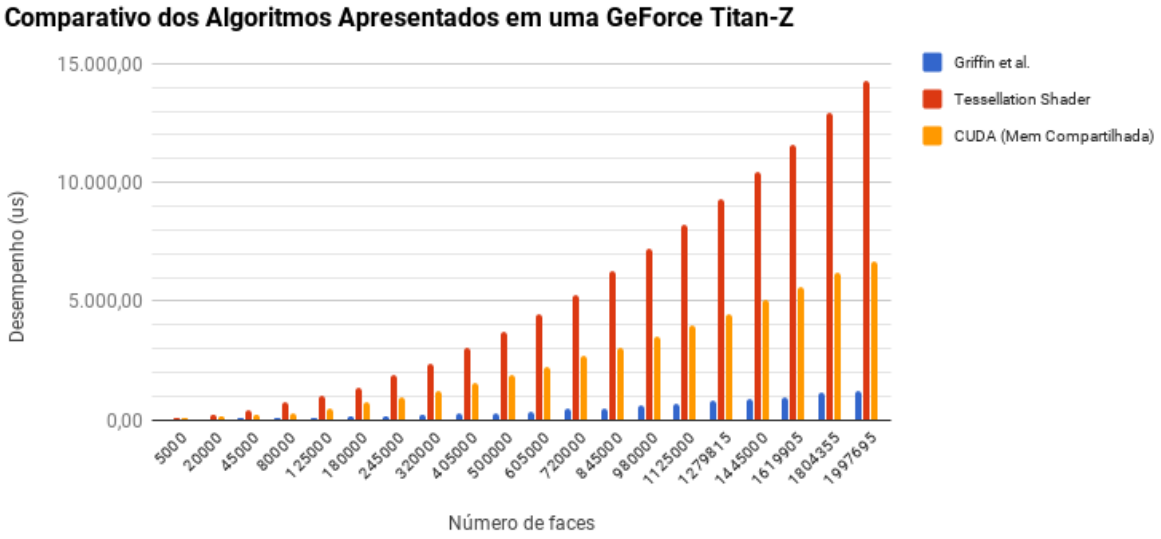

Figura 10 – Comparação do desempenho dos algoritmos apresentados em uma placa NVidia GeForce Titan-Z (arquitetura Kepler)

e são capazes de ter um desempenho melhor ou equivalente a uma versão otimizada do mesmo algoritmo em CUDA.

Já a versão proposta por nós, que utiliza o *tessellation shader*, não corrobora nossa hipótese de que as otimizações de *hardware*/*driver* criadas para o mesmo são capazes de melhorar o desempenho a níveis próximos das versões em CUDA. Mesmo assim nosso algoritmo requer menos passos de renderização que o de Griffin et al. e é capaz de realizar o cálculo dos tensores de curvatura em malhas arbitrárias com mais de 1 milhão de polígonos em tempo real (mais de 120 quadros por segundo de acordo com o gráfico da Figura 11), com pouca adaptação

Figura 11 – Comparação do desempenho dos algoritmos apresentados em uma placa NVidia GeForce Titan X (arquitetura Maxwell)

do código originalmente pensado para a CPU.

Os resultados obtidos durante esses testes nos permitem também traçar as curvas de ganho de desempenho para cada algoritmo entre diferentes versões de arquiteturas de GPU. Nós apresentamos na Figura 12 o gráfico de ganho de desempenho quando comparamos os resultados entre GPUs da arquitetura Fermi e Kepler. Já na Figura 13 a comparação é entre GPUs da arquitetura Kepler e Maxwell. Vale notar que o ganho de desempenho demonstrado pela mudança da arquitetura nas GPUs NVidia é significativamente maior para o nosso algoritmo, com os ganhos obtidos pelo mesmo constantemente acima da média dos ganhos dos três algoritmos analisados segundo os gráficos nas Figuras 12 e 13. Isso mostra uma evolução nos mecanismos de otimização pensados para o *tessellation shader*, incluindo aí uma melhoria no *Polymorph Engine 2.0* entre as arquiteturas Fermi e Kepler que viu a capacidade computacional do mesmo dobrar, e outra melhoria com a atualização para o *Polymorph Engine 3.0* entre as arquiteturas Kepler e Maxwell.

3.6 Discussões

Mapear conceitos não-gráficos tal como soma ponderada numa função de *blending* com aceleração por *hardware* para maximizar desempenho não é um conceito novo. Brook (BUCK *et al.*, 2004) foi um projeto pioneiro que representa os esforços realizados para se usar *hardware* gráfico para resolver aplicações de propósito geral computacionalmente intensivas. Hoje em dia padrões abertos como OpenCL e APIs específicas de fabricantes como CUDA são interfaces mundialmente conhecidas para programação de propósito geral em GPUs. Ainda assim, vários estudos mostram que, devido ao *design* inicial de GPUs para programas gráficos, algorit-

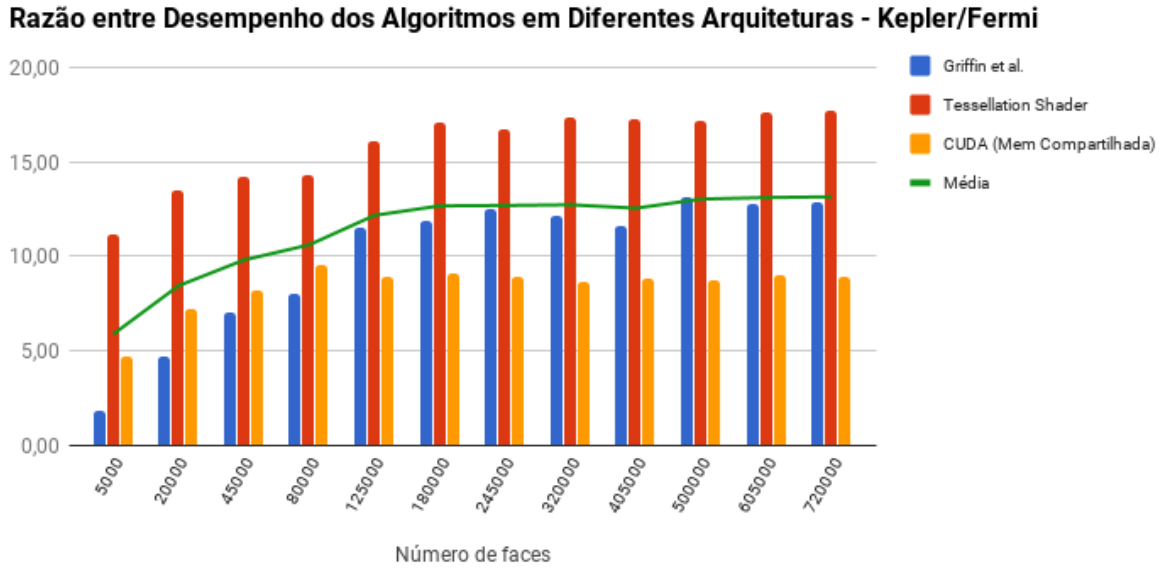

Figura 12 – Ganho de desempenho nos algoritmos quando passamos de uma placa NVidia GeForce 540m (arquitetura Fermi) para uma NVidia GeForce Titan Z (arquitetura Kepler)

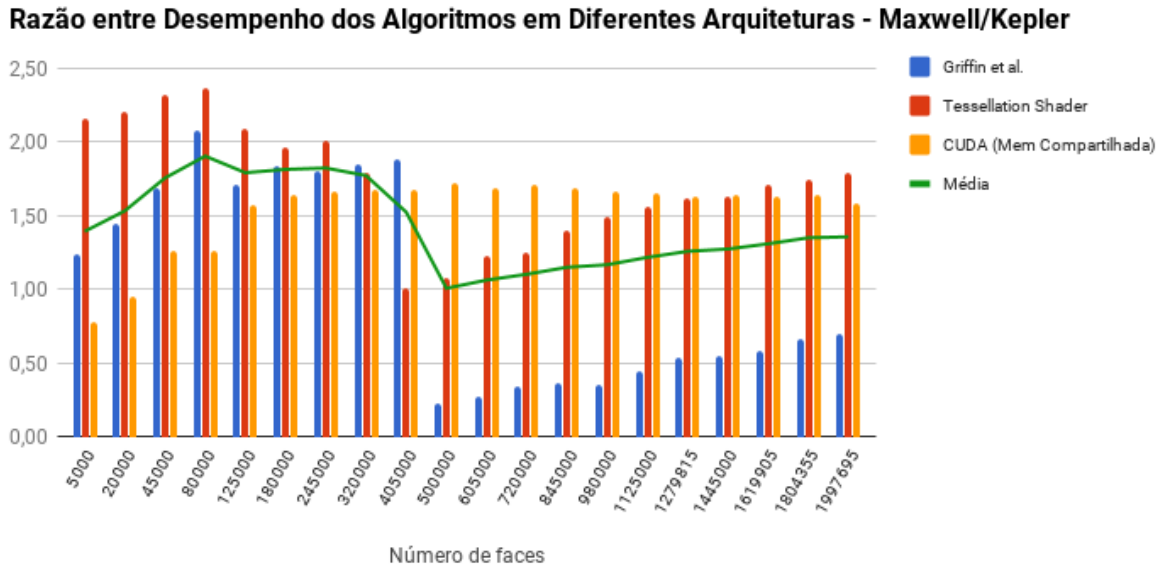

Figura 13 – Ganho de desempenho nos algoritmos quando passamos de uma placa NVidia GeForce Titan Z (arquitetura Kepler) para uma NVidia GeForce Titan X (arquitetura Maxwell)

mos que fazem um bom uso do *pipeline* gráfico costumam apresentar melhor desempenho que implementações usando APIs de propósito geral.

Em seu trabalho, Griffin et al. mostraram que, remodelando o problema de acesso à vizinhança de primeira ordem como um problema de *blending* de valores computados independentemente, foi capaz de alcançar um altíssimo desempenho, em parte devido ao *hardware*

otimizado de *blending* presente nas GPUs. Nós nos perguntamos se outros problemas de acesso à vizinhança de primeira ordem podem ser remodelados para tirarem proveito de *blending* e do acesso regular em *geometry shaders* e terem ganhos de desempenho no mesmo nível dos apresentados por Griffin et al.

Apesar de nossa conjetura de que *tessellation shaders* teriam algum *hardware* otimizado para acesso não coalescente aos dados de um *patch* não ter sido corroborada pelos resultados, nós notamos que o desempenho apresentado por ele acompanha o desempenho da implementação em CUDA com memória compartilhada. Um estudo aprofundado desse comportamento nos levou ao trabalho de Niessner et al. (NIESSNER *et al.*, 2015) onde eles afirmam que o *tessellation shader* utiliza, de forma implícita, memória compartilhada para otimizar o acesso aos dados de um *patch*. Sendo assim, seria possível utilizar outros recursos expostos pelo OpenGL para fazer com que o desempenho em OpenGL seja comparável, ou melhor, que o desempenho em CUDA com memória compartilhada?

No Capítulo 4 nós tentamos responder essas perguntas, aplicando os conhecimentos adquiridos nessa investigação num conhecido problema de acesso à vizinhança de primeira ordem cujo desempenho é altamente beneficiado pelo uso de memória compartilhada em CUDA.

4 Aplicações com Vértices de Valência Arbitrária

Dentro da classe de problemas com topologia estática a sub-classe de problemas com vértices de valência arbitrária engloba aqueles onde a vizinhança de primeira ordem de um elemento não tem um limitante superior e, normalmente, está relacionada à quantidade total de elementos sendo analisados. Problemas de simulação de fenômenos físicos como N-corpos ou mecânica dos fluídos são exemplos clássicos dessa sub-classe. Como nesses casos não há uma topologia explícita e os resultados numéricos são tão ou mais importantes que a visualização, o uso de CUDA ou OpenCL é mais difundido como veremos na Seção 4.1.

Baseado nos resultados obtidos para algoritmos de valência limitada apresentados no Capítulo 3, conjeturamos que no caso de problemas de valência ilimitada, onde a renderização dos resultados é necessária, é possível obter desempenho melhor, ou equivalente, à interoperabilidade com CUDA usando somente OpenGL. Para testarmos a nossa hipótese, utilizamos o algoritmo de simulação de N-corpos descrito em (NYLAND *et al.*, 2007) como *benchmark*, pois uma implementação deste algoritmo faz parte dos exemplos do SDK da plataforma CUDA. As diversas otimizações aplicadas nesta implementação são apresentadas de forma bem didática.

Nas Seções 4.2 e 4.3 são apresentadas, respectivamente, uma descrição sucinta do modelo de simulação de N-corpos e uma eficiente implementação com CUDA proposta por Nyland et al. (NYLAND *et al.*, 2007). Nossas experimentações em usar diferentes recursos otimizados do *pipeline* gráfico para chegar no nível de desempenho equiparável com o da CUDA são detalhadas na Seção 4.4. Tentamos aplicar as técnicas de otimização utilizadas por Griffin et al. (GRIFFIN *et al.*, 2012), assim como propusemos uma nova forma de utilizar o *tessellation shader* para replicar o padrão de execução obtido pela versão em CUDA e utilizar circuitos dedicados de *transform feedback* e *instancing* para melhorar a fluidez de dados ao longo do *pipeline* gráfico. Desempenhos alcançados nas quatro arquiteturas mais recentes da GPU, Fermi, Kepler, Maxwell e Pascal (ver Apêndice A), são apresentados na Seção 4.5. Por fim, na Seção 4.6, apresentamos resultados que corroboram nossa conjetura e discutimos as vantagens e desvantagens das implementações utilizadas.

4.1 Trabalhos Relacionados

Walters et al. (WALTERS *et al.*, 2008) apresentam um algoritmo de dinâmica molecular em GPU utilizando CUDA. Eles reportam duas grandes otimizações em seu código. A primeira é a reestruturação do vetor de posições dos átomos para maximizar o acesso coales-

cente à memória global. Ao invés de guardarem as posições (x, y, z) interpoladas em um vetor, eles armazenam a coordenada x de todos os átomos, seguida pela coordenada y de todos os átomos e, por fim, a coordenada z de todos os átomos. A outra otimização é o uso explícito de registradores para armazenar dados do átomo principal de uma *thread*. Isso diminui a ocupação da GPU de 100% para 67%, mas o uso intenso dos dados do átomo principal nos registradores ao invés de acessos à memória (global ou *cache*) resultou em ganho de desempenho chegando a até 3.71x com relação à versão sem essa otimização.

Mais recentemente, Niemeyer e Sung (NIEMEYER; SUNG, 2014) fizeram um estudo extensivo sobre soluções de dinâmica de fluidos utilizando GPGPU. Analisando os ganhos de desempenho com diferentes técnicas de otimização em GPU reportados em diversos trabalhos, eles concluem que as seguintes otimizações de acesso à memória costumam ser bem sucedidas:

- Reestruturação dos dados ou agrupamento de *threads* nos blocos para maximizar o acesso coalescente à memória global.
- Uso de memória compartilhada para armazenar resultados de operações de redução, como somatórios e busca de máximos, de forma que apenas um resultado por bloco seja escrito de volta na memória global.

É claramente visível no estudo de Niemeyer e Sung que, após o advento de CUDA, pesquisadores pararam de desenvolver soluções utilizando o *pipeline* gráfico, preferindo focar em CUDA e soluções multi-GPU/multi-CPU. No melhor do nosso conhecimento não há, para o problema de dinâmica de fluídos, nenhum estudo que compare as diferentes implementações em CUDA com suas traduções para o *pipeline* gráfico, mesmo para casos de simulações em tempo real.

Hunz (HUNZ, 2013) realizou um estudo sobre a aplicabilidade do estágio de *compute shader* do *pipeline* gráfico introduzido no OpenGL na versão 4.3. Ele implementa três diferentes problemas usando *compute shader*: simulação n-corpos, simulação de tecidos e detecção de linhas. Para a simulação de n-corpos ele demonstrou a capacidade do *compute shader* de realizar o mesmo trabalho que CUDA, traduzindo a implementação de Nyland et al. (NYLAND *et al.*, 2007). Infelizmente não é apresentada uma comparação extensiva entre as duas implementações. Vale ainda lembrar que o *compute shader* não faz parte do *pipeline* gráfico (KHRONOS GROUP, 2014). Ou seja, quando um comando de renderização é executado, mesmo que o *compute shader* esteja definido ele não é executado. Por isso ele não tem acesso a todos os recursos do *pipeline* gráfico como, por exemplo, *blending* e *mipmapping*. Os outros problemas estudados por Hunz operam em reticulados regulares e otimizam seus acessos à vizinhança de primeira ordem através do uso de texturas. As soluções em GPU são comparadas com suas versões em CPU e, conforme esperado, apresentam um grande ganho de desempenho em sua versão em GPU.

Sumarizando, há uma preferência por CUDA quando se trata de aplicações que demandam processamentos numéricos pesados, porém paralelizáveis. Entretanto ambas as APIs gráfica e não gráfica executam sobre o mesmo *hardware* e, conforme estudos e experimentos explicados no Capítulo 3, quando os recursos de *hardware* são corretamente utilizados através da API gráfica essa apresenta desempenho tão bom quanto ou melhor que APIs não gráficas.

4.2 Simulação de n-corpos

Uma simulação de n-corpos consiste em simular as interações entre n corpos e calcular suas posições e velocidades resultantes a cada iteração. Esse tipo de simulação tem ampla aplicabilidade em astrofísica, servindo tanto para problemas mais simples, como a interação entre terra, lua e sol, como também para problemas mais complexos, como entender a evolução de grandes estruturas do universo. Alterando-se as leis de interação entre os corpos, a mesma simulação pode ser utilizada também em problemas de dinâmica molecular, mecânica dos fluídos, simulação de materiais granulados ou efeitos de partículas em jogos eletrônicos.

Nesse trabalho nós implementamos a lei da gravidade como exemplo da interação entre os corpos de um sistema. A força entre dois corpos, F_{ij}, devido à atração gravitacional entre eles é definida pela equação

$$F_{ij} = \frac{Gm_i m_j}{|r_{ij}|^2} \cdot \frac{r_{ij}}{|r_{ij}|}, \tag{4.1}$$

onde m_i é a massa do corpo i e $|r_{ij}|$ é a distância entre os corpos i e j. Para evitar valores ilimitados conforme dois corpos se aproximam muito ($|r_{ij}|$ tende a zero), é comum adicionar um termo de amortecimento ε, resultando em

$$F_{ij} \approx \frac{Gm_i m_j r_{ij}}{(|r_{ij}|^2 + \varepsilon)^{\frac{3}{2}}}. \tag{4.2}$$

Sendo $a_{ij} = \frac{F_{ij}}{m_i}$ a aceleração resultante dessa força, a aceleração final de um corpo num instante de tempo devido à interação com todos os outros corpos é calculada por

$$a_i \approx G \sum_{0 \leq j \leq N} \frac{m_j r_{ij}}{(|r_{ij}|^2 + \varepsilon)^{\frac{3}{2}}}. \tag{4.3}$$

Como mostrado no Pseudocódigo 4.1 (NYLAND *et al.*, 2007), esse tipo de simulação é altamente paralelizável já que o cálculo da posição (linha 12) e da velocidade (linha 11) de cada corpo i é independente dos outros, apesar de requerer a posição de todos os outros corpos j, $other\_body$, para calcular o deslocamento relativo r_{ij} (linha 2) a cada iteração. Usando a norma do deslocamento ao quadrado, $||r_{ij}||^2$, as interações gravitacionais entre os corpos podem ser calculadas em pares (linhas 1–4). Note que um fator de amortecimento s é adicionado ao quadrado da distância euclidiana para fazer com que os corpos se comportem como galáxias

esféricas (AARSETH, 2009; DYER; IP, 1993). Quando multiplicamos essa interação gravitacional pela massa do outro corpo j, *other_body.m*, o resultado é a aceleração parcial do corpo i devido à sua interação com o corpo j. Essa aceleração é acumulada na variável a, de forma que, após realizar o cálculo para todos os corpos *other_body* adjacentes a i, nós obtemos a aceleração total do corpo i (linha 9). A partir da posição *body.pos* e da velocidade *body.vel* do corpo no instante t e da aceleração a, a nova velocidade *body.new_vel* (linha 11) e a nova posição *body.new_pos* (linha 12) e do corpo i no instante de tempo $t + \Delta t$ são estimadas. A variável d na linha 11 é utilizada para diminuir os efeitos do crescimento exponencial da velocidade dos corpos.

Pseudocódigo 4.1 Simulação de N-corpos

```
1:  interaction(pos_i, pos_j){
2:    r_ij = pos_j − pos_i
3:    return r_ij / (||r_ij||^2 + s)^(3/2)
4:  }
5:
6:  for body i in bodies do
7:     a = 0
8:     for other_body j in bodies do
9:        a = a + interaction(body.pos, other_body.pos) * other_body.m
10:    end for
11:    body.new_vel = (body.vel + a * Δt) * d
12:    body.new_pos = body.pos + body.vel * Δt
13: end for
```

4.3 Uma implementação Eficiente com CUDA

Uma implementação altamente otimizada de uma simulação N-corpos em CUDA é descrita no Pseudocódigo 4.2 (NYLAND *et al.*, 2007). A principal característica dessa implementação é o melhor uso possível da memória e hierarquia da memória *cache* da GPU. A posição e velocidade inicial de todos os n corpos são transferidas para um *buffer* na memória principal da GPU. O algoritmo executa em blocos de p *threads*, sendo lançado um total de n/p blocos, como ilustrado pela Figura 14. Cada *thread threadIdx* em cada bloco *blockIdx* é responsável por calcular a nova velocidade e posição de um único corpo *mydata* (linha 2).

Para evitar o gargalo de desempenho criado pelo acesso não-contíguo à memória, o cálculo ocorre em duas etapas. Primeiro, cada *thread* carrega os dados de um único novo corpo da memória principal para um vetor na memória compartilhada (linha 5) e espera em uma barreira pela sincronização. Isso garante que um total de p *other_body* seja transferido para a memória compartilhada. Após isso, cada *thread* pode iterar sobre esses p *other_body* acumulando a contribuição de cada um para a aceleração do corpo *mydata* na variável a (linha 8).

Esses passos são repetidos $\frac{n}{p}$ vezes de forma que ao final a variável a contém a aceleração total do corpo $mydata$ no instante t. Utilizando o valor atualizado de a, a velocidade e a posição de $mydata$ no instante $t + \Delta t$ são atualizados (linha 11). O número de *threads*, p, é um dos parâmetros de ajuste para esse algoritmo e nós usamos $p = 256$ em nossos experimentos já que essa é a melhor opção para $n \geq 4096$ de acordo com (NYLAND *et al.*, 2007).

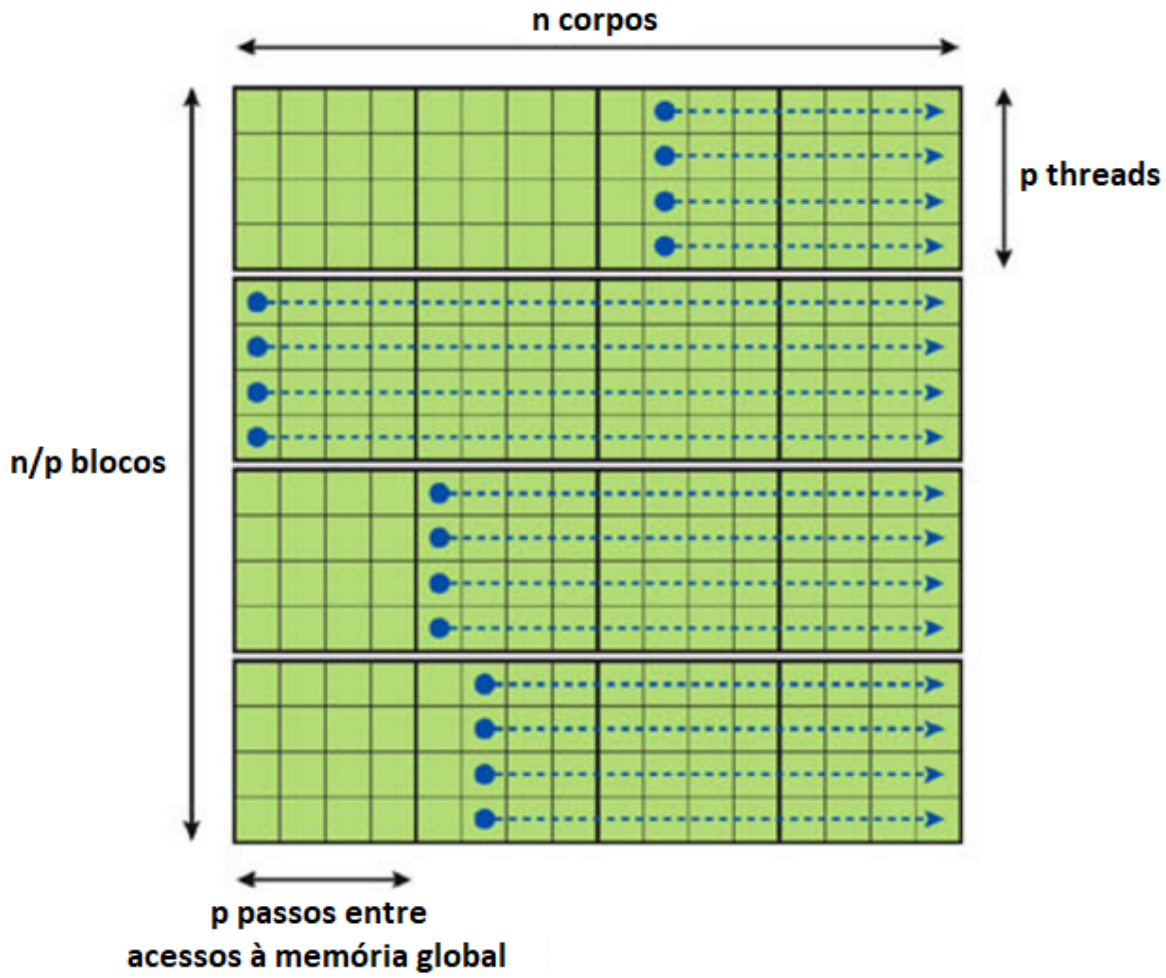

Figura 14 – Padrão de execução dos blocos e *threads* na simulação de n-corpos em CUDA (NYLAND *et al.*, 2007).

4.4 Implementações com API OpenGL

Para se adaptar ao *pipeline* gráfico da GPU e evitar sincronizações desnecessárias em qualquer *shader*, desenvolvemos uma implementação em dois passos para o Pseudocódigo 4.1: o estágio de cômputo da aceleração (linha 9) e o estágio de atualização da velocidade (linha 11) e da posição (linha 12) dos corpos. A Figura 15 mostra como esses dois passos são acoplados a um terceiro passo de renderização para a visualização dos resultados da simulação.

O primeiro passo é onde a simulação numérica realmente acontece. Nele interações, par a par, entre todos os corpos do *vertex buffer object* (*VBO*) de entrada são calculados

Pseudocódigo 4.2 Algoritmo para *worker thread* em CUDA

```
1:  a = 0
2:  mydata = fetch(blockIdx * p + threadIdx)
3:  for i = 0 to n/p do
4:      synchronize thread group
5:      shared[threadIdx] = fetch(i * p + threadIdx)
6:      synchronize thread group
7:      for j = 0 to p do
8:          a = a + interaction(mydata, shared[j].pos) * shared[j].m
9:      end for
10: end for
11: update_body(threadIdx, a)
```

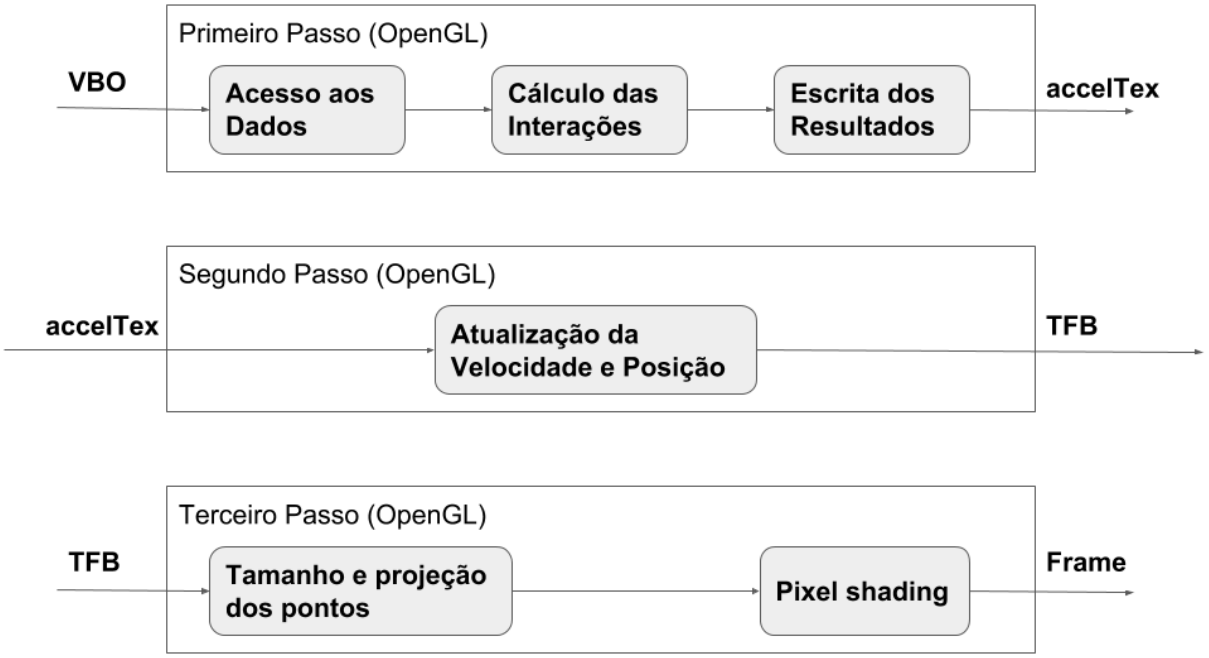

Figura 15 – Implementação em 3 passos baseada em OpenGL de uma simulação de n-corpos.

e os resultados são escritos numa memória de textura. Inspirados pelo trabalho de Griffin et al. (GRIFFIN *et al.*, 2012) nós propomos que o *hardware* de *blend* seja ativado para acumular todas as interações de cada corpo com todos os outros numa posição específica da textura, conforme descrito na Seção 3.4.1.

No segundo passo a velocidade e a posição de cada corpo no instante $t + \Delta t$ são atualizadas como mostra o Pseudocódigo 4.4. As chamadas de *draw* devem ser feitas novamente para todos os corpos a fim de inicializar essa segunda passada. As acelerações calculadas na passada anterior são transmitidas através de uma memória de textura atribuída à variável *accelTex* (linha 3). Como essa segunda passada envolve apenas o processamento dos corpos como vértices, nós propomos o uso de *transform feedback buffers*, TFB, para capturar as novas posições, *newPos* (linha 10), novas velocidades, *newVel* (linha 11), e resubmetê-las diretamente

para o terceiro passo, o estágio de renderização, sem nenhuma transferência entre CPU e GPU.

Pseudocódigo 4.3 Funções auxiliares

```
1:  idx2tc(index){
2:    pos = vec2(index%size.x, index/size.x);
3:    return vec2(pos.x/size.x, pos.y/size.y);
4:  }
5:
6:  tc2pos(texCoord){
7:    x = -1 + 2texCoord.x;
8:    y = -1 + 2texCoord.y;
9:    return vec4(x, y, 0, 1);
10: }
11:
12: interaction(a, b){
13:   r_ij = b.xyz - a.xyz;
14:   dist = dot(r_ij, r_ij) + softeningSq;
15:   invSqrtDist = inversesqrt(dist);
16:   return r_ij * invSqrtDist^3;
17: }
```

Pseudocódigo 4.4 *Vertex shader* na segunda passada

```
1:  getAccel(index){
2:    texelCoord = ivec2(index%size.x, index/size.x);
3:    return texelFetch(accelTex, texelCoord, 0).xyz;
4:  }
5:
6:  main(){
7:    a = getAccel(gl_VertexID);
8:    vel = (oldVel.xyz + a * dt) * d;
9:    pos = oldPos.xyz + vel * dt;
10:   newPos = vec4(pos, oldPos.w);
11:   newVel = vec4(vel, oldVel.w);
12: }
```

Para analisar o impacto dos diferentes recursos gráficos disponíveis em GPUs, e expostos na API OpenGL, numa simulação de n-corpos, nós implementamos o passo de *Cálculo das Interações* de três formas diferentes: com *geometry shader*, com *tessellation evaluation shader* e com primitivas instanciadas.

4.4.1 Utilizando *Geometry Shader*

O primeiro recurso que nós exploramos na implementação do passo de *Cálculo das Interações* foi a capacidade do *geometry shader* de acessar os dados de todos os vértices que compõem a primitiva sendo renderizada, como pode ser visto no Pseudocódigo 4.5. O

programa renderiza um conjunto de n^2 segmentos de reta de forma que cada par de corpos i e j é processado no *geometry shader*, onde as suas posições, $vOut[0].pos$ e $vOut[1].pos$, são lidas dos dados de entrada (linha 1) e a aceleração parcial de cada corpo devido à sua interação gravitacional é escrita nos dados das primitivas de saída (linha 2). O *shader* calcula a interação entre os dois corpos, representados pelos vértices do segmento de reta (linha 4) e emite dois vértices, representando os dois corpos, com a aceleração em decorrência da interação entre eles (linhas 7 e 10). Note que o elemento $vOut[i].pos.w$ corresponde à massa do corpo i.

Pseudocódigo 4.5 *Geometry shader* para interação entre pares de corpos

```
1:  layout(lines) in;
2:  layout(points, max_vertices = 2) out;
3:  main(){
4:    int = interaction(vOut[0].pos, vOut[1].pos);
5:    gl_Position = tc2pos(vOut[0].texCoord);
6:    gOut.accel = int * vOut[1].pos.w;
7:    EmitVertex();
8:    gl_Position = tc2pos(vOut[1].texCoord);
9:    gOut.accel = -(int * vOut[0].pos.w);
10:   EmitVertex();
11: }
```

4.4.2 Utilizando *Tessellation Evaluation Shader*

Com o uso do *geometry shader* nós conseguimos um alto paralelismo para a simulação de n-corpos, mas o grande número de interações que devem ser calculadas, com seu crescimento quadrático, faz com que isso não seja facilmente escalável mesmo nas GPUs mais modernas. Quando olhamos novamente a implementação em CUDA (Seção 4.3), observamos que sua característica marcante é a forma como a memória compartilhada é exposta para o programador, o que facilita a implementação de 1:m interações numa única *thread*. Nós nos perguntamos se é possível implementar uma forma equivalente de execução das *threads* usando *shaders*.

Nossa inspiração vem do trabalho apresentado por Nießner et al. que afirma que o *tessellation shader* otimiza o acesso aos dados dos vértices num *patch* através do uso intensivo de memória compartilhada (NIESSNER *et al.*, 2015). Sendo assim, nós agrupamos as n^2 interações em *patches* de 32 corpos mutualmente excludentes, esperando dessa forma maximizar o paralelismo em nível de *threads* sem exceder o número de *threads* sendo executadas.

Nossa proposta é renderizar os n corpos como $\frac{n}{32}$ *patches* quadrados com os níveis de tesselação exterior, *TessLevelOuter*, configurados para {3,7,3,7} e de tesselação interior, *TessLevelInner*, configurados para {7,3}. A Figura 16 mostra como essa configuração de níveis de tesselação resulta em 32 pontos de tesselação por *patch*. Fazendo com que cada ponto de tesselação esteja associado a um corpo, essa organização de *patches* se torna similar à organização

por blocos da implementação em CUDA descrita na Seção 4.3. Podemos então calcular a interação gravitacional por *patch* para cada corpo iterando sobre todos os 32 corpos no *patch*. Isso é feito no *tessellation evaluation shader*.

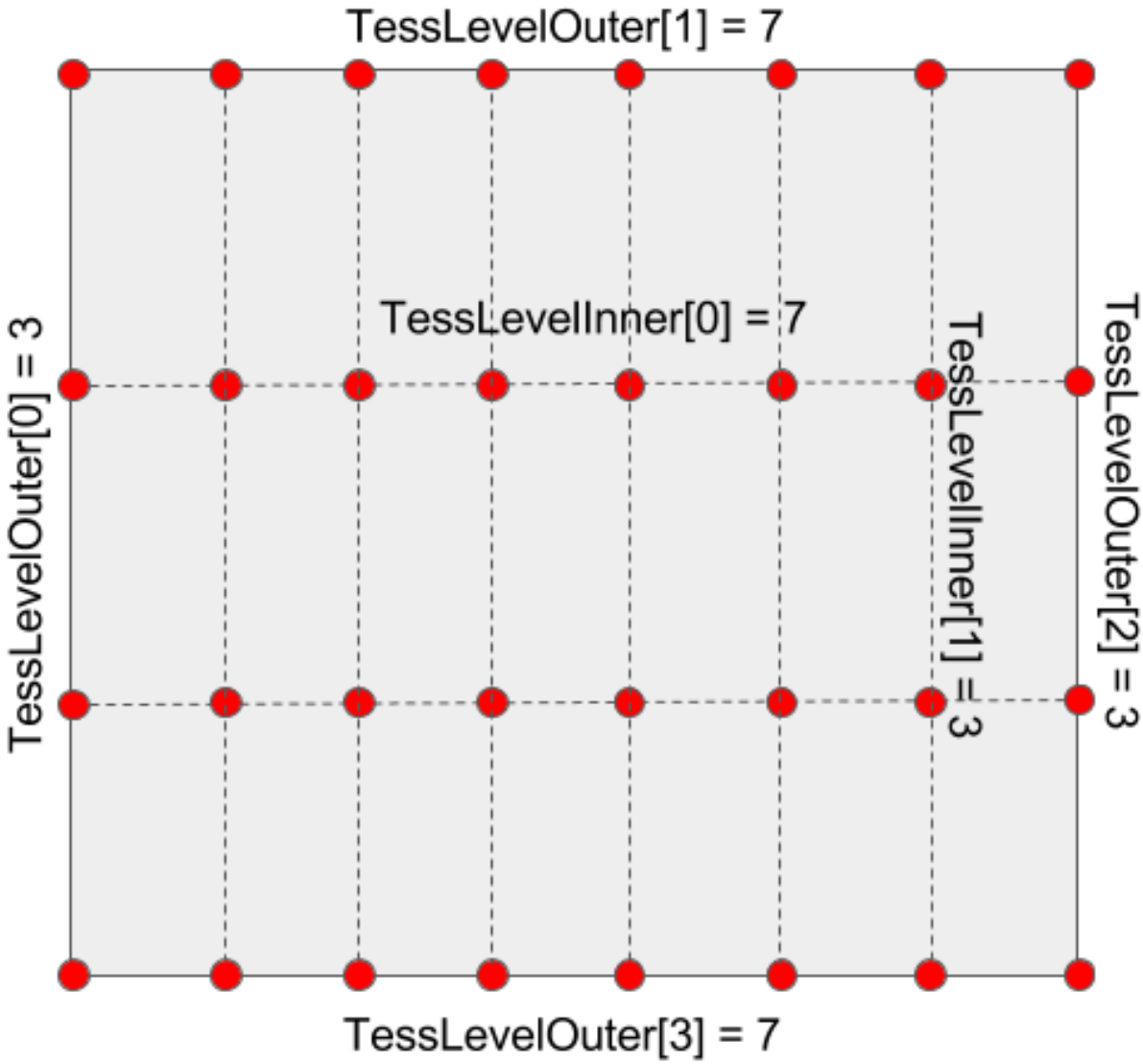

Figura 16 – Tesselação de um quadrado com 32 pontos.

O Pseudocódigo 4.6 detalha essa proposta. O laço nas linhas 5–8 implementa o acesso aos 32 corpos e a soma das suas contribuições, ponderadas por sua massa $bodies[i].pos.w$, para a aceleração a do atual ponto de tesselação. A localização desse ponto atual no *patch* é obtida usando sua posição de textura em *gl_TessCoord*. A linha 3 demonstra como é obtido o índice *myIdx* do ponto de tesselação atual no vetor *vOut* onde os dados dos 32 corpos estão organizados em linhas. Vale a pena notar que o padrão de acesso à memória proposto é similar ao padrão descrito no Pseudocódigo 4.2. A aceleração acumulada passa sem alteração pelo *fragment shader* (linha 9) e é escrita na posição correspondente da textura *accelTex*.

Pseudocódigo 4.6 *Tessellation shader* para interação entre pares de corpos

```
1:  layout(quads, equal_spacing, cw, point_mode) in;
2:  main(){
3:    myIdx = ⌈24 * gl_TessCoord.y + 7 * gl_TessCoord.x⌉;
4:    a = 0;
5:  for i = 0 to 32 do
6:      int = interaction(bodies[myIdx].pos, bodies[i].pos);
7:      a = a + int * bodies[i].pos.w;
8:  end for
9:    tOut.accel = a;
10:   gl_Position = tc2pos(bodies[myIdx].texCoord);
11: }
```

4.4.3 Utilizando *Tessellation Shader* com instâncias

Mesmo com o uso de memória compartilhada através de primitivas de tesselação nós observamos que o desempenho da implementação baseada em OpenGL ainda estava aquém do desempenho da versão otimizada em CUDA. Uma comparação mais cuidadosa dos algoritmos nos levou a detectar uma diferença sutil entre as implementações em CUDA e em OpenGL usando *tessellation shader*. Ao invés de um padrão de acesso à memória de $(\frac{n}{p} \times \frac{n}{p})(p \times p)$, como descrito na Seção 4.4.2, o padrão adotado na implementação em CUDA é $\frac{n}{p}(p \times (\frac{n}{p} \times p))$, como mostra a Figura 14. Apesar da quantidade de corpos processados ser a mesma, o número de acessos à memória varia largamente entre os dois métodos. Novamente nós nos perguntamos se é possível replicar o padrão do CUDA no *pipeline* de renderização da OpenGL.

Observando como o método de renderização instanciada renderiza múltiplas instâncias de um mesmo VBO em uma única chamada de *draw*, nós tivemos a idéia de aplicar essa forma de renderização para iterar sobre todos os *patches* descritos na Seção 4.4.2. O principal ponto da nossa proposta é iterar sobre todos os $\frac{n}{32}$ *patches* instanciados usando a renderização por instâncias de um único *patch*. Para cobrir todos os $\frac{n}{32}$ *patches* nós despachamos $\frac{n}{32}$ chamadas instanciadas de *draw*, como ilustrado na Figura 17. Com o método proposto de renderização instanciada, nosso padrão de acesso à memória se torna $\frac{n}{32}(32 \times (\frac{n}{32} \times 32))$, aproximando-se ainda mais do padrão da implementação em CUDA descrito na Figura 14.

Para computar as interações gravitacionais entre cada corpo *myIdx* do *patch* sendo renderizado e cada corpo *i* do *patch* instanciado no *tessellation evaluation shader* (linhas 4–8 descrito no Pseudocódigo 4.8), nós precisamos ter acesso aos dados de ambos os *patches*. Para isso nós propomos que esses dados sejam obtidos do VBO no *vertex shader* usando o índice do vértice corrente, *gl_VertexID*, e o índice do *patch* instanciado corrente, *gl_InstanceID* (linha 4 no Pseudocódigo 4.7). Dessa forma, cada vértice de um *patch* é responsável por obter os dados de um único corpo do *patch* instanciado e esses dados são escritos na variável $vOut.pos2$.

Note que, similar ao Pseudocódigo 4.6, a saída do *tessellation evaluation shader*

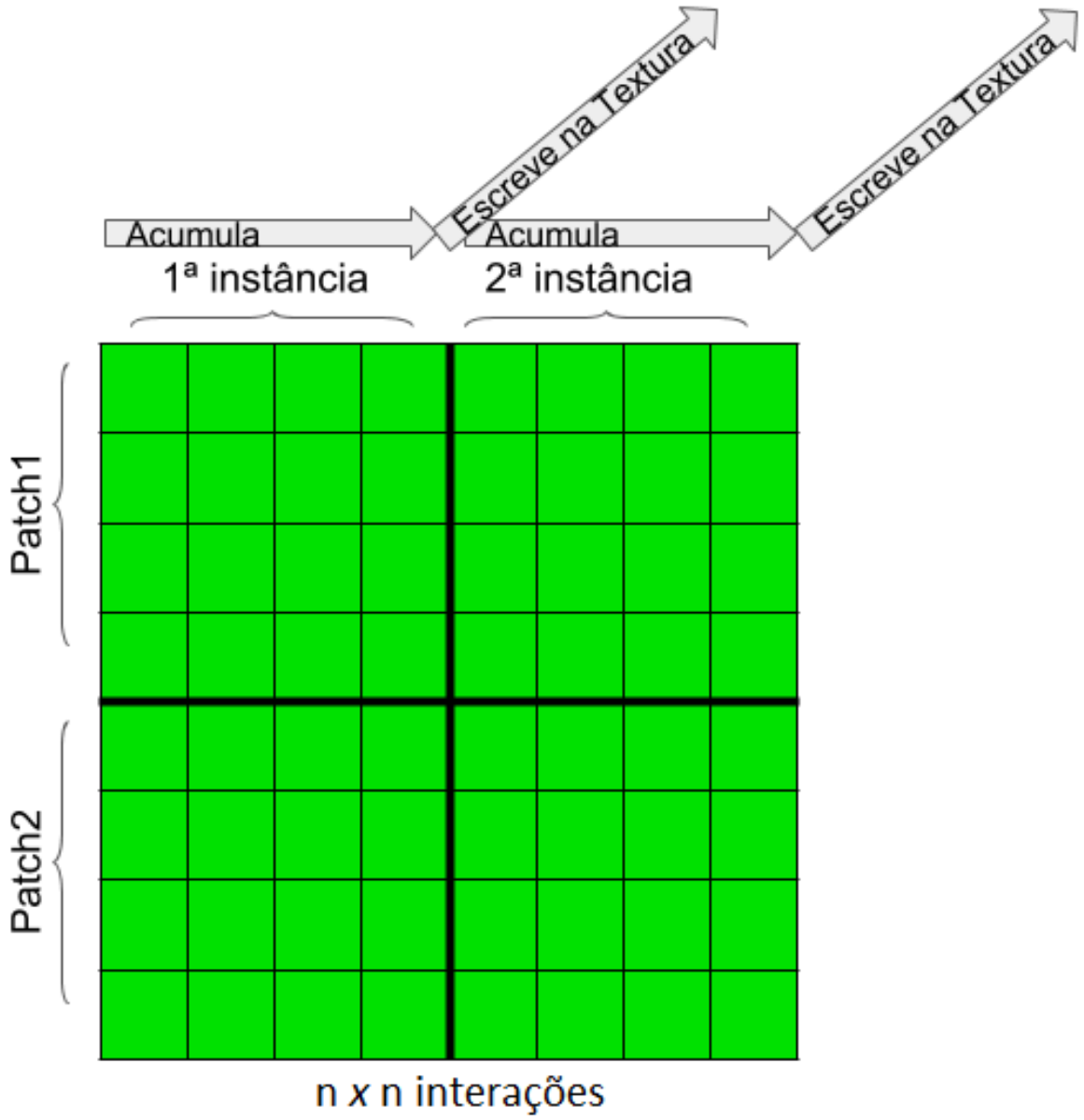

Figura 17 – Padrão de execução dos *patches* e *patches* instanciados.

é o somatório das acelerações do corpo *myIdx* resultante da interação com 32 corpos (linha 9 no Algoritmo 4.8), e, diferentemente do Algoritmo 4.6, esses 32 corpos pertencem ao *patch* instanciado e não ao *patch* sendo renderizado onde o ponto de tesselação corrente se encontra.

Pseudocódigo 4.7 *Vertex shader* para obtenção de dados dos *patches* instanciados

```
1: main(){
2:   vOut.pos = pos;
3:   extraIdx = 32 * gl_InstanceID + gl_VertexID%32;
4:   vOut.pos2 = positions[extraIdx];
5:   vOut.texCoord = idx2tc(gl_VertexID);
6: }
```

Pseudocódigo 4.8 *Tessellation evaluation shader* para interação entre pares de corpos

```
1:  layout(quads, equal_spacing, cw, point_mode) in;
2:  main(){
3:    myIdx = ⌈24 * gl_TessCoord.y + 7 * gl_TessCoord.x;⌉
4:    a = 0;
5:  for i = 0 to 32 do
6:      int = interaction(bodies[myIdx].pos, bodies[i].pos2);
7:      a = a + int * bodies[i].pos2.w
8:  end for
9:    tOut.accel = a;
10:   gl_Position = tc2pos(bodies[myIdx].texCoord);
11: }
```

Nome da Máquina	Processador	Tamanho da RAM	GPU	# de CUDA cores
Fermi	Intel core i5	8 GB	NVidia GeForce 540M 1 GB VRAM	96
Kepler	Intel core i7	16 GB	NVidia GeForce GTX Titan 6 GB VRAM	2688
Maxwell	Intel Xeon	16 GB	NVidia GeForce GTX Titan X 12 GB VRAM	3072
Pascal	Intel core i7	16 GB	NVidia GeForce GTX 1080 8 GB VRAM	2560

Tabela 2 – Especificação das máquinas usadas nos testes descritos na Seção 4.5

4.5 Resultados

Na Seção 4.4 nós propomos diferentes formas de gradualmente melhorar uma implementação da simulação de n-corpos em OpenGL com o propósito de deixar o padrão de acessos à memória o mais próximo possível do padrão apresentado pela implementação em CUDA descrita na Seção 4.3. Nessa seção nós avaliamos o quão efetiva é nossa proposta em tornar o desempenho de duas implementações, usando API OpenGL e API CUDA, o mais próximo possível.

Nós realizamos dois testes, um para avaliar a melhora do desempenho conforme novos recursos gráficos eram incorporados ao programa e outro para avaliar a escalabilidade da nossa proposta nas últimas quatro arquiteturas de GPU da NVidia, Fermi, Kepler, Maxwell e Pascal (Tabela 2 e Apêndice A). O código-exemplo de simulação de N-corpos, que faz parte do SDK CUDA da NVidia (NYLAND *et al.*, 2007), é executável tanto na GPU como na CPU e mostra a quantidade de quadros por segundo (FPS) como medida de desempenho, com uma taxa de atualização de um segundo. Um exemplo da tela de saída desse programa pode ser visto na Figura 18. O algoritmo implementado em CUDA é apresentado no Pseudocódigo 4.1. Nós apenas modificamos o código do estágio *Cálculo das Interações* para cada caso de implementação apresentado na Seção 4.4.

No primeiro teste nós executamos as quatro diferentes implementações da simulação de n-corpos (uma em CUDA e três em OpenGL) na máquina Kepler para avaliar a diferença em desempenho que cada circuito dedicado de OpenGL causa e quão próximo esse desempenho chega do apresentado pela implementação em CUDA. Olhando o gráfico de desempenho

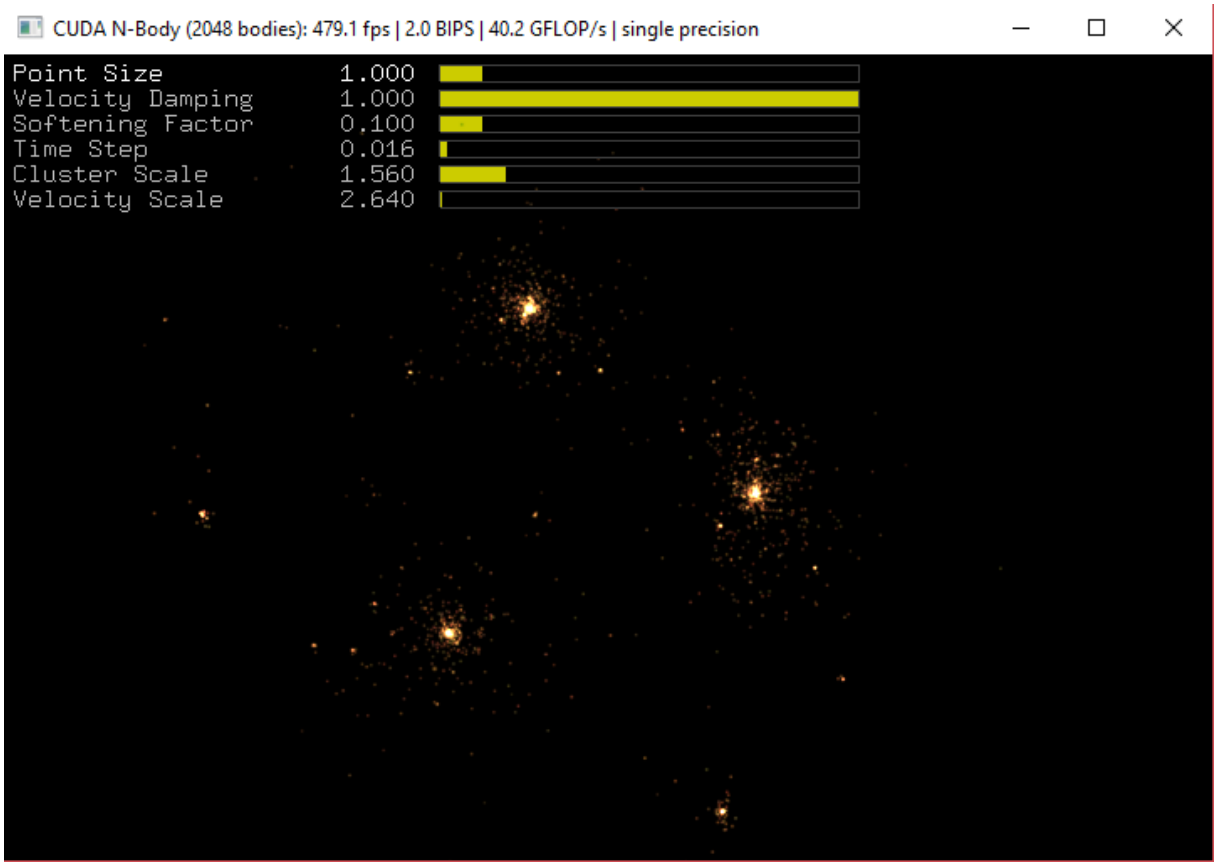

Figura 18 – Examplo da saída do programa de simulação N-body.

em FPS $\times$ número de corpos mostrado na Figura 19, podemos concluir que os circuitos de *geometry shader* e *tessellation shader* não escalam bem. O seu desempenho degrada rapidamente com um leve aumento no número de corpos, enquanto a implementação em CUDA e a em OpenGL que combina *tessellation* e renderização instanciada mantém um bom desempenho mesmo com o crescimento do número de corpos. Nós podemos observar também que a implementação com *tessellation* e renderização instanciada é a que mais se aproxima do desempenho da implementação em CUDA.

Num segundo teste nós comparamos, para as últimas quatro arquiteturas de GPU NVidia, o desempenho da implementação em OpenGL usando *tessellation shader* e renderização instanciada em relação à implementação em CUDA. Com isso nós conseguimos avaliar a influência da evolução da GPU no desempenho do *pipeline* gráfico para computação de propósito geral e também a escalabilidade das APIs OpenGL e CUDA em diferentes arquiteturas de GPU. Dado o gráfico de desempenho em FPS $\times$ número de corpos apresentado na Figura 20, é interessante notar que a implementação em OpenGL apresenta melhor desempenho que a implementação em CUDA para um menor número de corpos, mas ela degrada mais rapidamente que a versão em CUDA à medida que esse número cresce. Também vale ressaltar que, apesar de ter um desempenho menor, a GPU NVidia da arquitetura Maxwell parece oferecer uma melhor escalabilidade do que as outras.

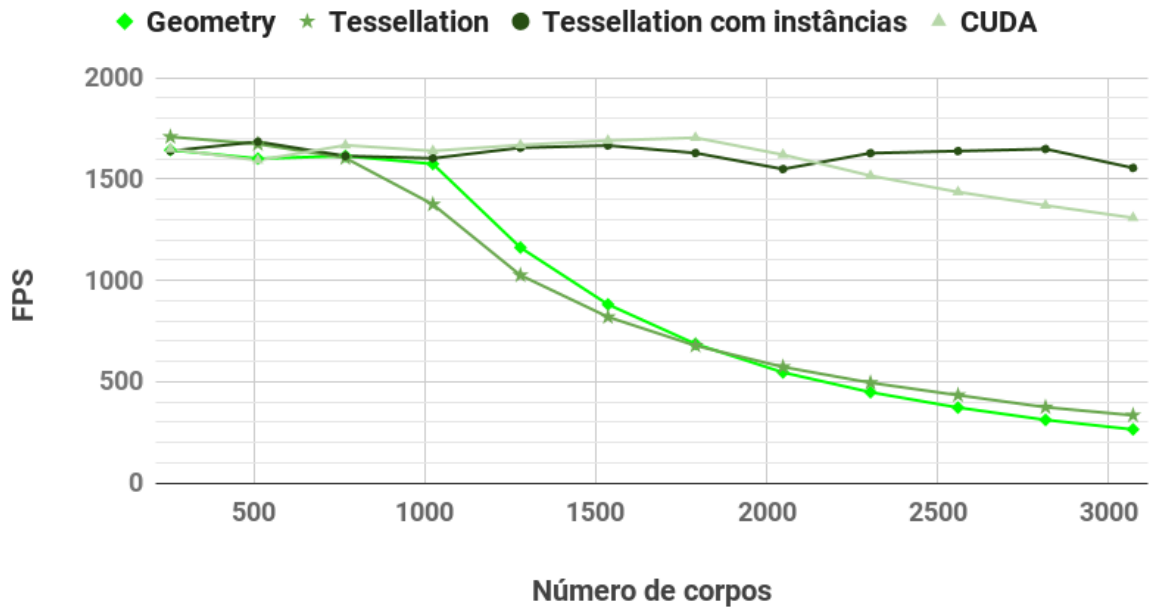

Figura 19 – Comparação de desempenho das 4 implementações na máquina Kepler.

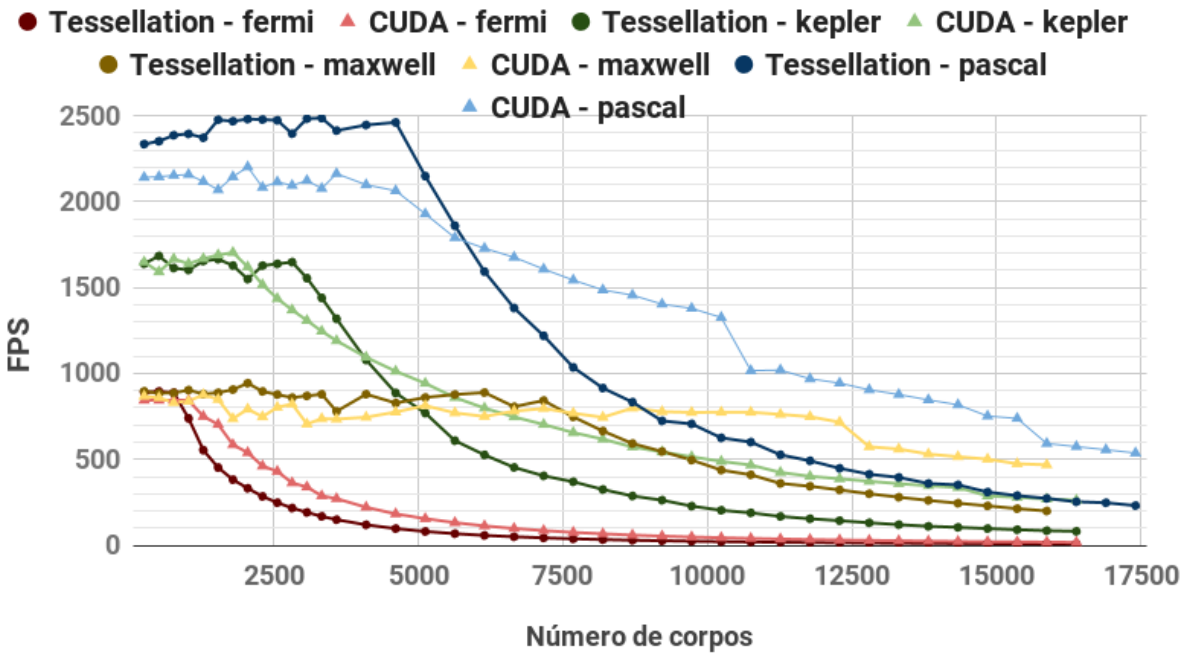

Figura 20 – Comparação do desempenho das implementações em OpenGL e CUDA em 4 diferentes arquiteturas de GPUs da NVidia (Fermi, Kepler, Maxwell e Pascal).

4.6 Discussões

O principal problema do uso de OpenGL em computação não-gráfica antes do advento da versão 4.0 era que algumas características de *hardware* importantes para processamento paralelo, tais como hierarquia de memória e sincronização de *threads* e blocos, não eram diretamente acessíveis através de conceitos gráficos em versões anteriores. Isso tornava a oti-

mização do fluxo de uma aplicação não-gráfica em uma tarefa trabalhosa.

Como vimos na Seção 2.4.2, vários novos conceitos gráficos foram introduzidos na OpenGL a partir da versão 3.0, tais como *transform feedback*, *tessellation shader* e renderização instanciada, com o intuito de melhorar o desempenho gráfico. *Transform feedback* evita a transmissão desnecessário de dados entre CPU e GPU, os *tessellation shaders* permitem mudar a forma como uma malha grosseira é refinada na GPU, e renderização instanciada provê uma forma eficiente de renderizar múltiplas instâncias de uma mesma malha numa única chamada de *draw*.

Baseado em (NIESSNER *et al.*, 2015) e uma série de experimentos, nós chegamos à conclusão que esses conceitos são implementados sobre um *hardware* otimizado. A partir de (NIESSNER *et al.*, 2015) nós inferimos que melhores desempenhos em tesselação são devidos ao pré-carregamento de todos os dados dos vértices de um *patch* na memória compartilhada de uma forma similar a todo o dado de um bloco de *threads* sendo carregado, como no Pseudocódigo 4.2. E, segundo nossos testes empíricos, levantamos a hipótese de que atributos de múltiplas instâncias são carregados em blocos da memória global, o que aumenta a localidade dos dados. Para testar essa hipótese nós implementamos diferentes versões do estágio de *Cálculo das Interações* com melhorias graduais nos recursos gráficos usadas, como detalhado na Seção 4.4.

Os resultados de desempenho na máquina Kepler apresentados na Figura 19 nos permitem afirmar que nossa hipótese está de acordo com os dados experimentais. As estratégias de otimização que apresentamos para a API OpenGL são suficientes para alcançar um desempenho que é comparável à implementação otimizada em CUDA. Mais intrigante é a rápida queda em desempenho para números maiores de corpos quando comparados com a versão em CUDA. Uma análise cuidadosa nos levou a identificar a única diferença entre eles: o número de corpos por grupo de processamento. Na versão em OpenGL nós temos 32 corpos por *patch* e na versão em CUDA temos 256 corpos por bloco. Dessa forma, é esperado que o segundo tenha um desempenho melhor que o primeiro dado o tamanho de bloco suportado.

Apesar da queda de desempenho reforçar nossa hipótese, nós ainda decidimos comparar o desempenho das duas implementações quando o número de corpos por grupo de processamento é o mesmo. Quando configuramos o número de corpos por bloco na versão em CUDA para 32, nós obtemos uma curva de desempenho bem próxima à obtida na versão em OpenGL, conforme podemos observar na Figura 21. Ou seja, enquanto OpenGL limitar o número de vértices em cada *patch* a 32, CUDA, com sua maior flexibilidade no tamanho dos blocos, deve apresentar um melhor desempenho.

Devido à limitação do tamanho de um *patch* imposta pela implementação OpenGL da GPU NVidia, a versão em OpenGL é menos escalável que a versão em CUDA. Sendo assim o gráfico na Figura 20 está de acordo com nossas expectativas. Outro ponto que vale ressaltar é que, apesar de apresentar um desempenho menor, a GPU Maxwell tem uma melhor escalabi-

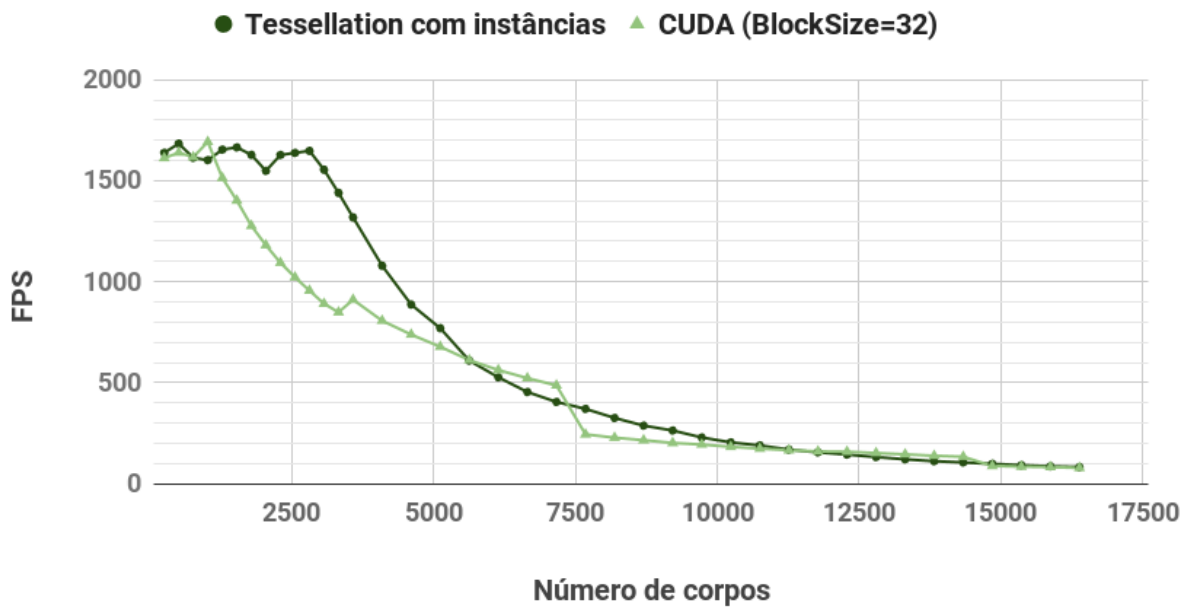

Figura 21 – Comparação da versão em *tessellation shader* com instâncias e da versão em CUDA com tamanho do bloco de 32 corpos.

lidade. Nós conjecturamos que esse comportamento se dê por ela apresentar um maior número de CUDA *cores* que as outras GPUs utilizadas (Tabela 2).

5 Conclusões

O avanço das placas gráficas como processadores massivamente paralelos de uso geral resultou em vários estudos sobre a portabilidade de algoritmos da CPU para a GPU. No começo esses algoritmos eram desenvolvidos usando os estágios programáveis em bibliotecas gráficas como o OpenGL. O aparecimento de CUDA, que permite um maior controle sobre os recursos das GPUs e provê uma interface mais direta de programação do que o reaproveitamento de funções gráficas, fez com que pesquisadores dessem maior ênfase para CUDA do que OpenGL, mesmo em aplicações com finalidade gráfica. Porém, diferentes estudos mostram que, para aplicações de renderização, é possível conseguir um desempenho melhor ao se implementar os algoritmos baseados somente em OpenGL ao invés de utilizar CUDA interoperando com OpenGL.

Almejando desenvolver aplicativos gráficos com padrão de acessos irregular sem interoperabilidade, esse trabalho fez uma análise da competitividade entre CUDA e OpenGL na solução dos problemas/aplicações que envolvem acessos aleatórios a uma topologia fixa armazenada na memória da GPU. Foram identificadas duas sub-classes de problemas baseadas na valência dos vértices das malhas: (1) valência limitada; e (2) valência ilimitada.

Para problemas de valência limitada foi escolhido para análise o problema de cálculo de tensores de curvatura. O trabalho de Griffin et al. foi utilizado como base e foram apresentadas três diferentes implementações, uma em CUDA utilizando memória compartilhada, uma tradução do algoritmo de Griffin et al. para GLSL e uma versão utilizando *tessellation shaders*. Os resultados obtidos corroboraram estudos anteriores mostrando que o uso de algoritmos adaptados para OpenGL ao invés de CUDA+OpenGL resulta em melhor desempenho. O acesso à vizinhança de primeira ordem de forma implícita utilizando *geometry shader*, ao invés do uso de memória compartilhada, e o uso de *blending* como ferramenta para somatórios, ao invés de operações atômicas, resultou em um melhor desempenho para a implementação em OpenGL quando comparado com CUDA+OpenGL, resultado que corrobora estudos anteriores, incluindo o de Griffin et al.

Infelizmente o desempenho do nosso algoritmo utilizando *tessellation shader* não validou nossa hipótese de que as otimizações de *hardware*/*driver* criadas para o mesmo são capazes de melhorar o desempenho a níveis próximos da implementação do Griffin et al.. Mesmo assim, nosso algoritmo é capaz de realizar o cálculo dos tensores de curvatura em malhas arbitrárias com mais de 1 milhão de polígonos em tempo real (mais de 120 quadros por segundo) com pouca adaptação do código originalmente pensado para a CPU. Os nossos resultados também mostram que o ganho de desempenho demonstrado pela mudança da arquitetura nas GPUs NVidia é significativamente maior para o nosso algoritmo, com os ganhos obtidos pelo mesmo

estando constantemente acima da média dos ganhos dos três algoritmos analisados. Em retrospecto, o problema de tensores de curvatura pode não ser um bom representante de problemas de acesso não-contíguo à vizinhança de primeira ordem com valência limitada pois, como mostraram Griffin et al., ele pode ser reduzido ao cômputo de parcelas independentes com um somatório não sincronizado ao final. Isso reforça ainda mais a ideia de que o custo de acessos à memória é muito maior do que o custo de processamento lógico-aritmético.

O problema de simulação de N-corpos foi utilizado como exemplo de problemas com valência ilimitada. Tomamos a implementação de Nyland em CUDA, disponível nos exemplos do SDK distribuído pela NVidia, como base de comparação. Foram apresentadas três novas implementações em OpenGL explorando novas aplicações dos recursos gráficos *geometry shader*, *tessellation shader* e renderização instanciada. Nós demonstramos que, devido ao pré-carregamento de dados para a memória compartilhada, realizado implicitamente pelo *tessellation shader*, a primitiva *patch* pode ser utilizada para maximizar acessos à memória de baixa latência. Nós também demonstramos experimentalmente que o uso de renderização instanciada pode ser aplicado para maximizar o uso da largura de banda da memória global devido aos dados de múltiplas instâncias serem acessados em blocos. Os resultados obtidos mostram que nossa implementação usando essas técnicas tem desempenho comparável ao de CUDA, tendo uma menor escalabilidade devido apenas às limitações de *software* impostas pela implementação da API OpenGL nos *drivers* da fabricante NVidia.

A Tabela 3 sintetiza o mapeamento do acesso aos recursos de *hardware* investigados em CUDA e OpenGL. Aplicando nossas contribuições (linhas 2 e 3) e a de Griffin et al. (linha 1) pode-se obter um melhor aproveitamento dos recursos de *hardware* expostos pelo *pipeline* gráfico da OpenGL.

Operação em *Hardware*	Acesso em CUDA	Acesso em OpenGL
Operações de Redução	Uso de memória compartilhada e operações atômicas	Uso de *blending* e renderização em textura
Agrupar a execução em blocos que acessam dados próximos para otimizar o uso da largura de banda da memória global.	Definição das dimensões de blocos e do *grid* na chamada de execução do *kernel*	Uso de renderização instanciada
Acesso à memória compartilhada de um SM	Declaração de variáveis no *kernel* com a palavra chave *__shared__*	Dados de um *patch* durante a execução de *tessellation shaders*

Tabela 3 – Mapeamento do acesso aos recursos de *hardware* em CUDA e OpenGL

Para trabalhos futuros nós gostaríamos de expandir o estudo aplicando as técnicas descritas nesse trabalho em outros problemas que se encaixam nas duas classes identificadas como, por exemplo, simulação de materiais granulados. Além disso, os resultados obtidos no Capítulo 3 demonstram que *tessellation shaders* ainda estão sendo otimizados em novas gerações de GPUs e seria interessante observar como os ganhos dessas otimizações afetam as implementações apresentadas. Por fim, as comparações aqui realizadas levam em conta somente

placas NVidia, devido ao uso de CUDA. É, no entanto, pertinente um estudo da aplicabilidade dos novos usos de *tessellation shader* e renderização instanciada em GPUs de outras fabricantes já que essas otimizações podem ser dependentes das implementações de cada fabricante.

O código fonte dos aplicativos utilizados nos experimentos deste trabalho, assim como publicações decorrentes do mesmo estão disponíveis no *web site* do projeto em <http://www.dca.fee.unicamp.br/projects/desmo/camillo/index.html>.

Referências

AARSETH, S. *Gravitational N-Body Simulations: Tools and Algorithms*. [S.l.]: Cambridge University Press, 2009. (Cambridge Monographs on Mathematical Physics).

ALUMBAUGH, T. J.; JIAO, X. Compact Array-Based Mesh Data Structures. In: *International Meshing Roundtable*. Springer Berlin Heidelberg, 2005. p. 485–503. Disponível em: <http://link.springer.com/chapter/10.1007\%2F3-540-29090-7_29>.

BAYRAKTAR, S. *Simulating cloth behavior by using mass-spring networks*. Tese (Doutorado) — bilKent university, 2002.

BUCK, I.; FOLEY, T.; HORN, D.; SUGERMAN, J.; FATAHALIAN, K.; HOUSTON, M.; HANRAHAN, P. Brook for gpus: Stream computing on graphics hardware. In: *ACM SIGGRAPH 2004 Papers*. New York, NY, USA: ACM, 2004. (SIGGRAPH '04), p. 777–786.

CAMILLO, M. S.; WU, S.-t. *Um Estudo do Mapeamento de Estruturas BRep em GPU*. [S.l.], 2013. 1–18 p.

COSTA, M. R.; WU, S.-t. *Deformação de Malhas de Arbitrária Topologia*. [S.l.], 2012. 1–18 p. Disponível em: <http://www.dca.fee.unicamp.br/projects/prosim/publications/qualifying/costa-2012-desmo.pdf>.

DYER, C.; IP, P. Softening in N-body Simulations of Collisionless Systems. *The Astrophysical Journal*, v. 409, p. 60–67, maio 1993.

FRATARCANGELI, M. Gpgpu cloth simulation using glsl, opencl, and cuda. In: LENGYEL, E. (Ed.). *Game Engine Gems 2*. [S.l.]: A K Peters, 2011. cap. 22, p. 365–378.

GORDON, B.; SOHONI, S.; CHANDLER, D. Data handling inefficiencies between cuda, 3d rendering, and system memory. In: *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC'10)*. Washington, DC, USA: IEEE Computer Society, 2010. (IISWC '10), p. 1–10. ISBN 978-1-4244-9297-8. Disponível em: <http://dx.doi.org/10.1109/IISWC.2010.5648828>.

GRIFFIN, W.; WANG, Y.; BERRIOS, D.; OLANO, M. Real-time GPU surface curvature estimation on deforming meshes and volumetric data sets. *IEEE transactions on visualization and computer graphics*, v. 18, n. 10, p. 1603–13, out. 2012. ISSN 1941-0506. Disponível em: <http://www.ncbi.nlm.nih.gov/pubmed/22508906>.

HJELMERVIK, J.; LEON, J.-C. GPU-Accelerated Shape Simplification for Mechanical-Based Applications. In: *IEEE International Conference on Shape Modeling and Applications 2007 (SMI '07)*. IEEE, 2007. p. 91–102. ISBN 0-7695-2815-5. Disponível em: <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=4273372>.

HUNZ, J. *The Possibilities of Compute Shaders: an Analysis*. 2013. Bachelor's Thesis. Disponível em: <https://kola.opus.hbz-nrw.de/files/786/JochenHunzBachelorThesis.pdf>.

KHRONOS GROUP. *OpenGL - Open Graphics Library*. [S.l.], 1994. Ver. 1.0.

KHRONOS GROUP. *OpenGL - Open Graphics Library*. [S.l.], 2014. Ver. 4.5.

LOBEIRAS, J.; AMOR, M.; DOALLO, R. Influence of memory access patterns to small-scale FFT performance. *The Journal of Supercomputing*, v. 64, n. 1, p. 120–131, jul. 2012. ISSN 0920-8542. Disponível em: <http://link.springer.com/10.1007/s11227-012-0807-5>.

MCCLANAHAN, C. *History and Evolution of GPU Architecture*. [S.l.], 2010. 1–7 p. Disponível em: <http://mcclanahoochie.com/blog/2011/03/the-history-and-evolution-of-gpu-hardware/>.

MEYER, M.; DESBRUN, M.; SCHRöDER, P.; BARR, A. H. *Discrete Differential-Geometry Operators for Triangulated 2-Manifolds*. 2002.

NIEMEYER, K. E.; SUNG, C.-J. Recent progress and challenges in exploiting graphics processors in computational fluid dynamics. *J. Supercomput.*, Kluwer Academic Publishers, Hingham, MA, USA, v. 67, n. 2, p. 528–564, fev. 2014. ISSN 0920-8542. Disponível em: <http://dx.doi.org/10.1007/s11227-013-1015-7>.

NIESSNER, M.; KEINERT, B.; FISHER, M.; STAMMINGER, M.; LOOP, C.; SCHÄFER, H. Real-time rendering techniques with hardware tessellation. *Computer Graphics Forum*, EG, 2015.

NVIDIA. *NVIDIA's Next Generation CUDA Compute Architecture: Fermi*. [S.l.], 2009. Disponível em: <http://www.nvidia.com/content/pdf/fermi_white_papers/nvidia_fermi_compute_architecture_whitepaper.pdf>.

NVIDIA. *Cuda C Programming Guide*. [S.l.], 2012. Disponível em: <http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf>.

NVIDIA. *NVIDIA's Next Generation CUDA Compute Architecture: Kepler GK110*. [S.l.], 2012. Disponível em: <https://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf>.

NVIDIA. *NVIDIA GeForce GTX 980*. [S.l.], 2014. Disponível em: <https://international.download.nvidia.com/geforce-com/international/pdfs/GeForce_GTX_980_Whitepaper_FINAL.PDF>.

NVIDIA. *NVIDIA GeForce GTX 1080*. [S.l.], 2016. Disponível em: <http://international.download.nvidia.com/geforce-com/international/pdfs/GeForce_GTX_1080_Whitepaper_FINAL.pdf>.

NVIDIA. *NVIDIA TESLA V100 GPU ARCHITECTURE*. [S.l.], 2017. Disponível em: <http://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf>.

NVIDIA. *Cuda C Programming Guide v9.1*. [S.l.], 2018. Disponível em: <http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf>.

NYLAND, L.; HARRIS, M.; PRINS, J. Fast n-body simulation with cuda. In: NGUYEN, H. (Ed.). *GPU Gems 3*. [S.l.]: Addison-Wesley Professional, 2007. cap. 31.

OWENS, J. D.; LUEBKE, D.; GOVINDARAJU, N.; HARRIS, M.; KRüGER, J.; LEFOHN, A. E.; PURCELL, T. J. A Survey of General-Purpose Computation on Graphics Hardware. *Computer Graphics Forum*, v. 26, n. 1, p. 80–113, mar. 2007. ISSN 0167-7055. Disponível em: <http://doi.wiley.com/10.1111/j.1467-8659.2007.01012.x>.

PRESS, W. H.; TEUKOLSKY, S. A.; VETTERLING, W. T.; FLANNERY, B. P. *Numerical Recipes in C (2Nd Ed.): The Art of Scientific Computing*. New York, NY, USA: Cambridge University Press, 1992. ISBN 0-521-43108-5.

RUSINKIEWICZ, S. Estimating curvatures and their derivatives on triangle meshes. In: *Symposium on 3D Data Processing, Visualization, and Transmission*. [S.l.: s.n.], 2004.

SANS, F.; CARMONA, R. Volume ray casting using different gpu based parallel apis. In: *XLII Latin American Computing Conference (CLEI)*. [S.l.: s.n.], 2016. p. 1–11.

SHIUE, L.-j.; JONES, I.; PETERS, J. A realtime GPU subdivision kernel. In: *ACM SIGGRAPH 2005 Papers on - SIGGRAPH '05*. New York, New York, USA: ACM Press, 2005. p. 1010–1015. Disponível em: <http://portal.acm.org/citation.cfm?doid=1186822.1073304>.

VASSILEV, T. I. Comparison of parallel algorithms for modelling mass-springs systems with several apis on modern gpus. In: *Proceedings of the 12th International Conference on Computer Systems and Technologies*. New York, NY, USA: ACM, 2011. (CompSysTech '11), p. 204–209.

WALTERS, J. P.; BALU, V.; CHAUDHARY, V.; KOFKE, D.; SCHULTZ, A. *Accelerating Molecular Dynamics Simulations with GPUs.* 2008. 44-49 p.

WU, B.; ZHAO, Z.; ZHANG, E. Z.; JIANG, Y.; SHEN, X. Complexity analysis and algorithm design for reorganizing data to minimize non-coalesced memory accesses on gpu. In: *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. New York, NY, USA: ACM, 2013. (PPoPP '13), p. 57–68. ISBN 978-1-4503-1922-5.

YOON, S.-E.; LINDSTROM, P. Mesh layouts for block-based caches. *IEEE transactions on visualization and computer graphics*, v. 12, n. 5, p. 1213–20, 2006. ISSN 1077-2626. Disponível em: <http://www.ncbi.nlm.nih.gov/pubmed/17080854>.

ZHANG, Y.; PENG, L.; LI, B.; PEIR, J.-K.; CHEN, J. Architecture comparisons between nvidia and ati gpus: Computation parallelism and data communications. In: *Proceedings of The 2011 IEEE International Symposium on Workload Characterization (IISWC)*. [S.l.: s.n.], 2011.

Apêndices

APÊNDICE A – Arquiteturas de GPUs CUDA

Nesse apêndice mostramos brevemente as diferentes famílias de GPUs NVidias utilizadas nos experimentos deste trabalho. Para cada família são apresentadas a arquitetura geral e do SM do *chip*, assim como as principais mudanças e inovações em relação à família anterior.

A.1 Fermi

A família de GPUs Fermi é a segunda geração de placas CUDA e foi onde a NVidia redesenhou a arquitetura para dar maior foco a GPGPU (NVIDIA, 2009).

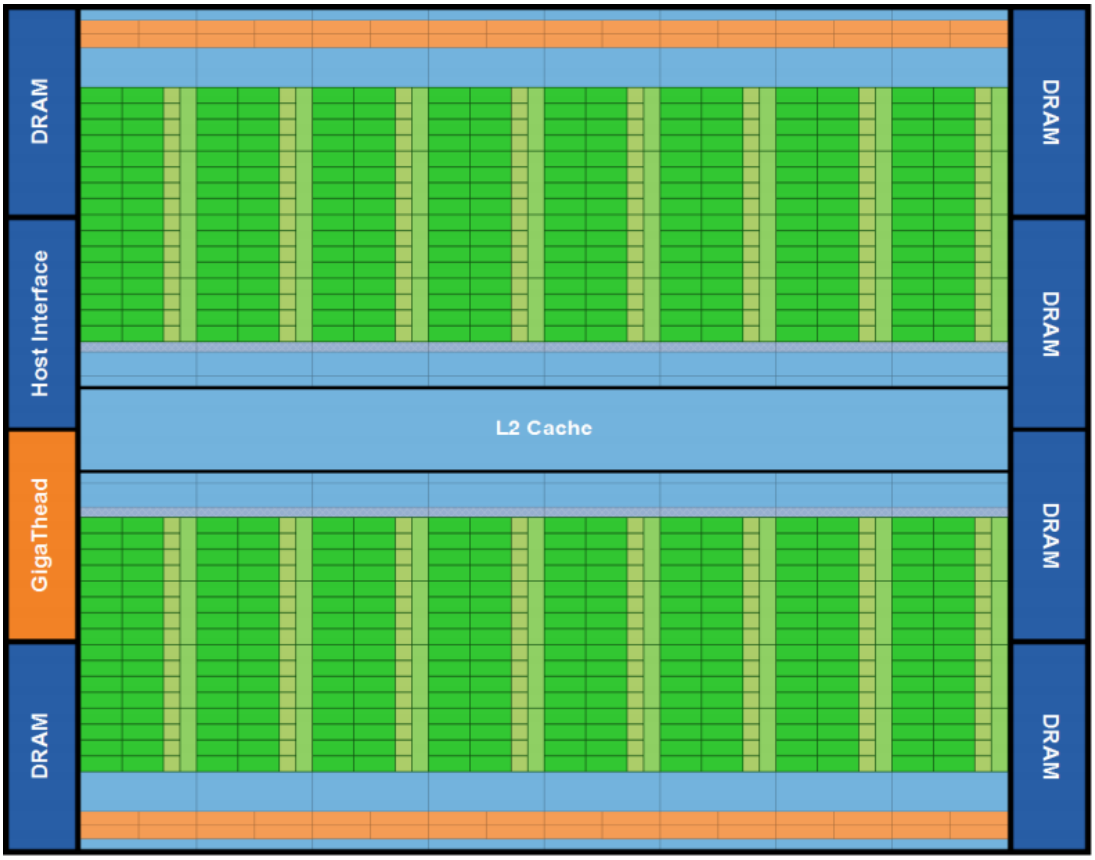

Figura 22 – Visão geral da arquitetura de um *chip* Fermi (NVIDIA, 2009)

As principais mudanças e inovações foram:

- maior capacidade de computação em ponto flutuante de precisão dupla (aproximadamente 8.5 vezes maior que a geração anterior);

- duplicação de SFUs (*Special Function Units*) por SM. Essas unidades são responsáveis por cálculos complexos como funções trigonométricas e raiz quadrada;
- introdução do *cache* L1, compartilhado de forma configurável com a memória compartilhada e aumento da memória disponível para 64KB;
- introdução do *cache* L2; e
- suporte a *kernels* concorrentes.

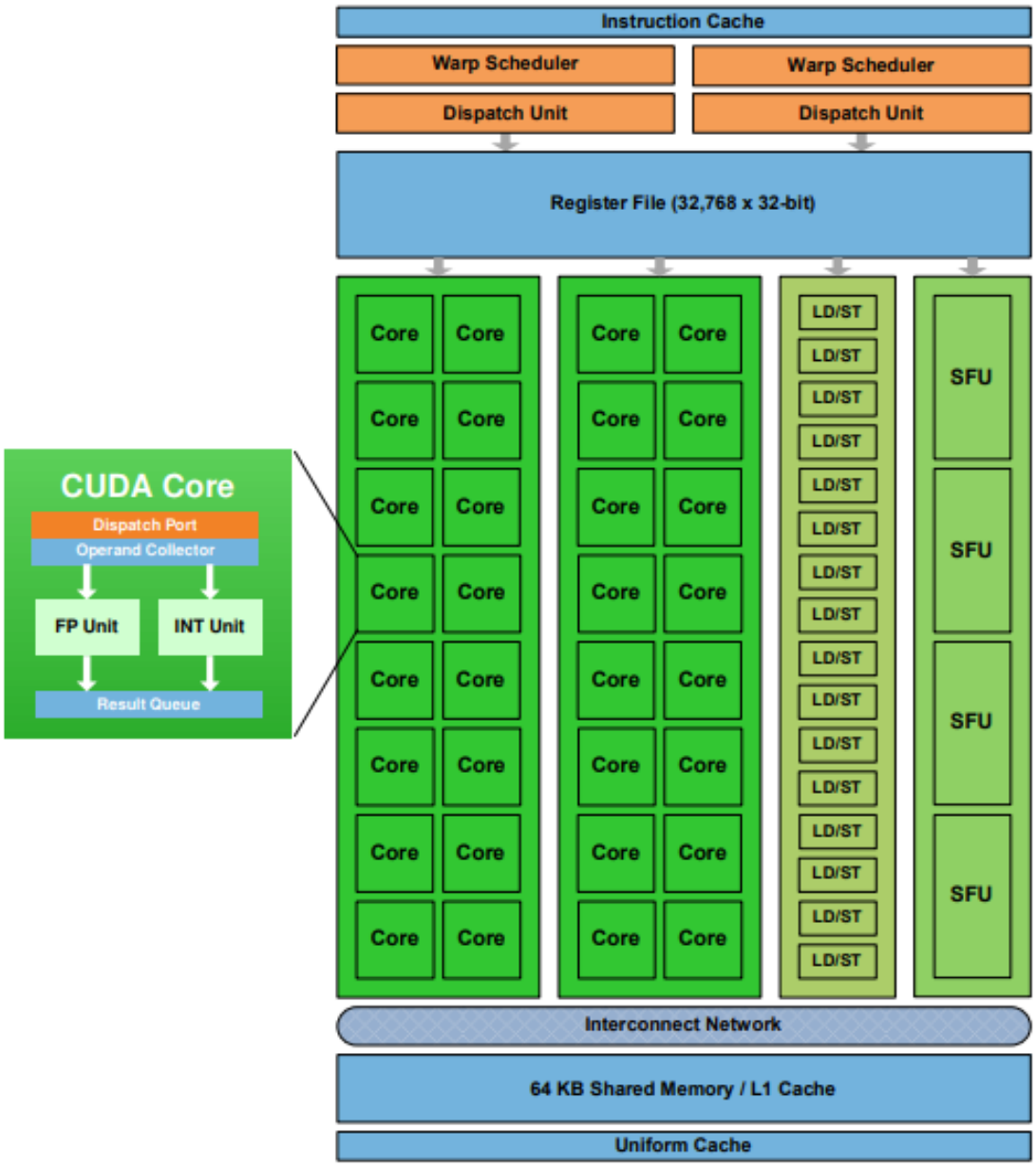

Figura 23 – Arquitetura de um SM num *chip* Fermi (NVIDIA, 2009)

A.2 Kepler

A geração seguinte das GPUs NVidia, chamada Kepler, expande os recursos de *hardware* voltados para GPGPU, aumentando o número de *threads* por SM, a quantidade de configurações de tamanho do *cache* L1 e memória compartilhada, entre outros.

Figura 24 – Visão geral da arquitetura de um *chip* Kepler (NVIDIA, 2012b)

As principais mudanças e inovações foram:

- adição de mais dois escalonadores de *warp*, totalizando quatro, por SM;
- nova configuração de tamanho para a divisão da memória interna do *chip* entre memória compartilhada e *cache* L1;
- aumento (4x) da quantidade de unidades de textura por SM;
- aumento do tamanho e da largura de banda da memória *cache* L2; e
- introdução do recurso de paralelismo dinâmico, onde a GPU tem controle sobre sincronismo e execução de novos *kernels* sem o uso da CPU.

Figura 25 – Arquitetura de um SM num *chip* Kepler (NVIDIA, 2012b)

A.3 Maxwell

Na sua família Maxwell de GPUs a fabricante NVidia volta a focar em aplicações gráficas e apresenta melhorias do *hardware* para tecelamento e iluminação global, por exemplo.

Figura 26 – Visão geral da arquitetura de um *chip* Maxwell (NVIDIA, 2014)

As principais mudanças e inovações foram:

- aumento da largura de banda da memória principal da GPU;
- aumento do tamanho da memória de *cache*;
- memória compartilhada passa a ser exclusiva, com tamanho de 96KB, enquanto a memória *cache* L1 passa a compartilhar o espaço do *cache* de textura;
- duplicação da quantidade de *Polymorph Engines* (responsáveis por tecelamento) e melhorias no mesmo permitem desempenho até três vezes melhor que a geração anterior; e
- aceleração de *hardware* para etapa de *voxelização* do algoritmo de iluminação global: *Viewport Multicast* e *Conservative Raster*.

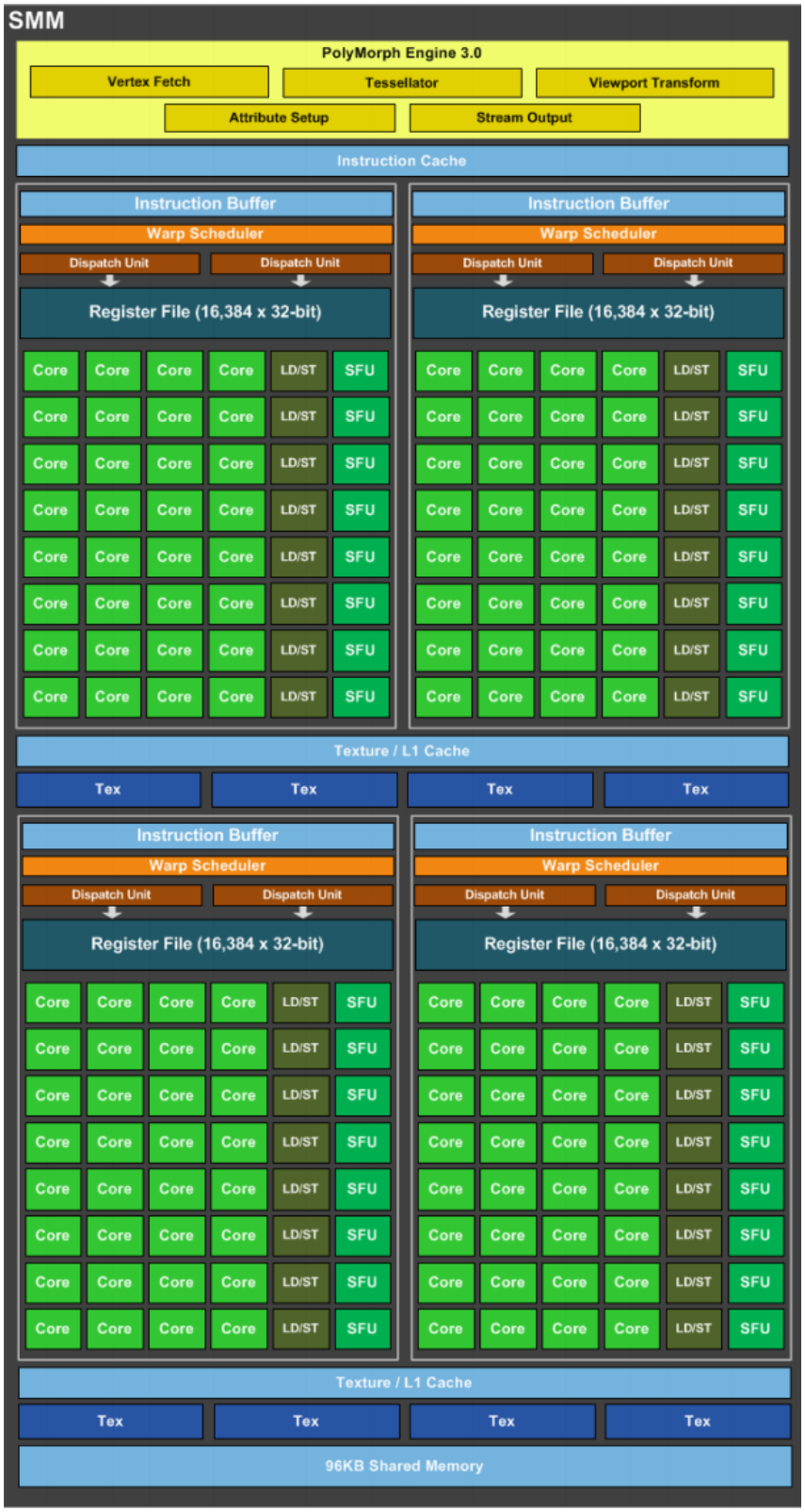

Figura 27 – Arquitetura de um SM em um *chip* Maxwell (NVIDIA, 2014)

A.4 Pascal

A última geração de GPUs utilizada neste estudo é a família Pascal. Essas placas apresentam inovações tanto para aplicações gráficas como para GPGPU.

Figura 28 – Visão geral da arquitetura de um *chip* Pascal (NVIDIA, 2016)

As principais mudanças e inovações foram:

- aumento da largura de banda da memória principal através do uso de GDDR5X;
- compressão de memória propiciando um melhor uso da largura de banda e melhor utilização das memórias de *cache*;
- melhorias no mecanismo das operações atômicas com o uso de *Unified Memory*;
- preempção a nível de *pixel* (gráfico) e instrução (GPGPU), permitindo um melhor balanceamento de carga em diferentes contextos na GPU; e
- multi-projeção simultânea, para aplicações de realidade virtual ou configurações de *display surround*.

Figura 29 – Arquitetura de um SM num *chip* Pascal (NVIDIA, 2016)

APÊNDICE B – Cálculo da Área de Voronoi

Neste trabalho, seguindo o algoritmo proposto por Rusinkiewicz (RUSINKIEWICZ, 2004), a área de Voronoi de um vértice com relação a uma face é utilizada para ponderar a contribuição do tensor de curvatura da face no tensor de curvatura do vértice. Essa área de Voronoi, conforme definida por Meyer et al. (MEYER *et al.*, 2002), é a porção da área da face mais próxima a um vértice. Em um triângulo não obtuso $V_0V_1V_2$ pode-se dividí-lo em três sub-triângulos isósceles conectando o seu circuncentro C com os vértices V_0, V_1 e V_2, como ilustra a Figura 30. Dividindo cada um destes sub-triângulos pelas respectivas bissetrizes relativas às suas bases, obtemos seis sub-triângulos. A área de Voronoi de um vértice, por exemplo do vértice V_0, é a soma das áreas dos dois sub-triângulos adjacentes a ele, $\triangle V_0CE$ e $\triangle V_0CF$. Para $\triangle V_0CF$, ela pode ser obtida por

$$A_{\triangle V_0CF} = \frac{1}{2}\frac{e_1}{2}\left(\frac{e_1}{2cos(a)}sen(a)\right). \tag{B.1}$$

Como valem as seguintes igualdades $\widehat{V_0} = a+b$, $\widehat{V_1} = b+c$, $\widehat{V_2} = a+c$ e $a+b+c = \frac{\pi}{2}$, obtemos com uso de identidades trigonométricas

$$\begin{aligned} sen(a) &= sen(\frac{\pi}{2} - \widehat{V_1}) = cos(\widehat{V_1}) \\ cos(a) &= cos(\frac{\pi}{2} - \widehat{V_1}) = sen(\widehat{V_1}). \end{aligned}$$

Substituindo-as na Eq. B.1 temos

$$A_{\triangle V_0CF} = \frac{1}{2}\frac{e_1}{2}\left(\frac{e_1}{2sen(\widehat{V_1})}cos(\widehat{V_1})\right) = \frac{1}{8}e_1^2 cot\widehat{V_1}.$$

Analogamente, obtemos $A_{\triangle V_0CE} = \frac{1}{8}e_2^2 cot\widehat{V_2}$. Somando as duas parcelas obtemos

$$A_0 = \frac{1}{8}(e_1^2 cot\widehat{V_1} + e_2^2 cot\widehat{V_2}). \tag{B.2}$$

Há uma outra forma de cálculo das áreas de Voronoi A_0, A_1 e A_2, respectivamente, de cada vértice V_0, V_1 e V_2, de um triângulo arbitrário somente em termos dos seus lados e_0, e_1 e e_2. Neste caso, as igualdades $\widehat{V_0} = a+b$, $\widehat{V_1} = b+c$, $\widehat{V_2} = a+c$ e $a+b+c = \frac{\pi}{2}$ são mantidas e fazendo uso das identidades trigonométricas temos

$$\begin{aligned} sen(a) &= sen(\frac{\pi}{2} - \widehat{V_1}) = cos(\widehat{V_1}) \\ sen(b) &= sen(\frac{\pi}{2} - \widehat{V_2}) = cos(\widehat{V_2}) \\ sen(c) &= sen(\frac{\pi}{2} - \widehat{V_0}) = cos(\widehat{V_0}). \end{aligned}$$

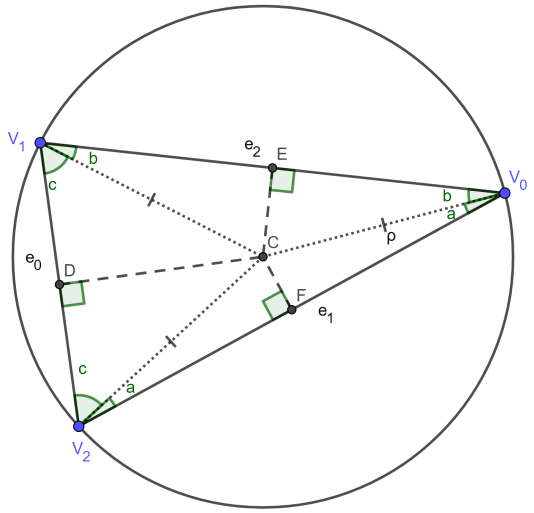

Figura 30 – Triângulo não-obtuso, seu circuncentro C e pontos médios das arestas D, E e F.

Podemos escrever as distâncias do circuncentro C em relação aos lados e_0, e_1 e e_2, ou seja, às alturas dos sub-triângulos, em termos de razões (com sinal) do raio do circuncírculo ρ

$$\begin{aligned} |FC| &= \rho sen(a) = \rho cos(\widehat{V_1}) \\ |DC| &= \rho sen(c) = \rho cos(\widehat{V_0}) \\ |EC| &= \rho sen(b) = \rho cos(\widehat{V_2}). \end{aligned}$$

E ao multiplicarmos estas alturas pelos respectivos lados, temos as áreas dos sub-triângulos

$$\begin{aligned} A_{\triangle V_0V_1C} &= \frac{1}{2}e_2(\rho cos(\widehat{V_2})) = (\frac{1}{2}\rho)(e_2 cos(\widehat{V_2})) \\ A_{\triangle V_0V_2C} &= \frac{1}{2}e_1(\rho cos(\widehat{V_1})) = (\frac{1}{2}\rho)(e_1 cos(\widehat{V_1})) \\ A_{\triangle V_1V_2C} &= \frac{1}{2}e_0(\rho cos(\widehat{V_0})) = (\frac{1}{2}\rho)(e_0 cos(\widehat{V_0})). \end{aligned} \tag{B.3}$$

Ou seja, $(\frac{A_{\triangle V_0V_1C}}{A_\triangle}, \frac{A_{\triangle V_0V_2C}}{A_\triangle}, \frac{A_{\triangle V_1V_2C}}{A_\triangle})$ são as coordenadas baricêntricas do circuncentro C. Usando a identidade

$$k^2 = i^2 + j^2 - 2ij\, cos(\widehat{ij}) \rightarrow cos(\widehat{ij}) = \frac{i^2 + j^2 - k^2}{2ij},$$

onde i, j e k são os comprimentos dos lados e $\widehat{ij}$ é o ângulo entre os lados i e j, e definindo a área total do triângulo como

$$A_\triangle = A_{\triangle V_0V_1C} + A_{\triangle V_0V_2C} + A_{\triangle V_1V_2C} = \frac{\rho}{2}(e_0 cos(\widehat{V_0}) + e_1 cos(\widehat{V_1}) + e_2 cos(\widehat{V_2})),$$

podemos escrever as coordenadas baricêntricas em função dos lados do triângulo como

$$\begin{aligned}
\frac{A_{\triangle V_0V_1C}}{A_\triangle} &= \frac{e_2^2(e_0^2+e_1^2-e_2^2)}{(e_2^2(e_0^2+e_1^2-e_2^2)+e_1^2(e_0^2+e_2^2-e_1^2)+e_0^2(e_1^2+e_2^2-e_0^2))},\\
\frac{A_{\triangle V_0V_2C}}{A_\triangle} &= \frac{e_1^2(e_0^2+e_2^2-e_1^2)}{(e_2^2(e_0^2+e_1^2-e_2^2)+e_1^2(e_0^2+e_2^2-e_1^2)+e_0^2(e_1^2+e_2^2-e_0^2))},\\
\frac{A_{\triangle V_1V_2C}}{A_\triangle} &= \frac{e_0^2(e_1^2+e_2^2-e_0^2)}{(e_2^2(e_0^2+e_1^2-e_2^2)+e_1^2(e_0^2+e_2^2-e_1^2)+e_0^2(e_1^2+e_2^2-e_0^2))}.
\end{aligned}$$

Com isso a área de Voronoi de um vértice em um triângulo não-obtuso pode ser calculada por (linhas 18–21 do Pseudocódigo 3.2)

$$A_0 = \frac{1}{2}A_\triangle\left(\frac{(e_2^2(e_0^2+e_1^2-e_2^2)+e_1^2(e_0^2+e_2^2-e_1^2))}{((e_0^2+e_1^2-e_2^2)+(e_0^2+e_2^2-e_1^2)+(e_1^2+e_2^2-e_0^2))}\right). \tag{B.4}$$

Quando uma das coordenadas baricêntricas é negativa, o circuncentro fica fora do triângulo como ilustra a Figura 31. Meyer et al. aproximam as áreas para $\frac{A_\triangle}{2}$ e $\frac{A_\triangle}{4}$ para o vértice de ângulo obtuso e os outros dois vértices, respectivamente. Rusinkiewicz redefine as áreas de Voronoi nesse caso para a área do polígono delimitado pelos lados adjacentes ao vértice e a as bissetrizes que cortam o triângulo. No exemplo da Figura 31 as áreas dos vértices V_0, V_1 e V_2 são delimitadas pelos polígonos V_0EG, V_1DHGE e V_2DH, respectivamente. A área dos triângulos retângulos V_0EG e V_2DH é facilmente obtida por

$$\begin{aligned}
A_{\triangle V_0EG} &= \frac{base \times altura}{2} = \frac{1}{2}\frac{e_2}{2}\frac{e_2\ sen(\widehat{V_0})}{2\ cos(\widehat{V_0})},\\
A_{\triangle V_2DH} &= \frac{base \times altura}{2} = \frac{1}{2}\frac{e_0}{2}\frac{e_0\ sen(\widehat{V_2})}{2\ cos(\widehat{V_2})}.
\end{aligned} \tag{B.5}$$

Sabendo que a área de um triângulo pode ser escrita em função de dois lados i, j e o ângulo entre eles $\widehat{ij}$ como $A_\triangle = \frac{1}{2}ijsen(\widehat{ij})$, e que $ijcos(\widehat{ij}) = \vec{i}\cdot\vec{j}$ a Equação (B.5) pode ser reescrita sem o uso de ângulos como

$$\begin{aligned}
A_0 = A_{\triangle V_0EG} &= \frac{e_2^2}{4}\left(\frac{1}{2}e_1e_2sen(\widehat{V_0})\right)\left(\frac{1}{e_1e_2cos(\widehat{V_0})}\right) = \frac{e_2^2}{4}A_\triangle\frac{1}{\vec{e_1}\cdot\vec{e_2}},\\
A_2 = A_{\triangle V_2DH} &= \frac{e_0^2}{4}\left(\frac{1}{2}e_0e_1sen(\widehat{V_2})\right)\left(\frac{1}{e_0e_1cos(\widehat{V_2})}\right) = \frac{e_0^2}{4}A_\triangle\frac{1}{\vec{e_0}\cdot\vec{e_1}}
\end{aligned} \tag{B.6}$$

Por fim a área de Voronoi do vértice de ângulo obtuso é facilmente obtida por $A_1 = A_\triangle - A_0 - A_2$ (linhas 8, 12 e 16 do Pseudocódigo 3.2)

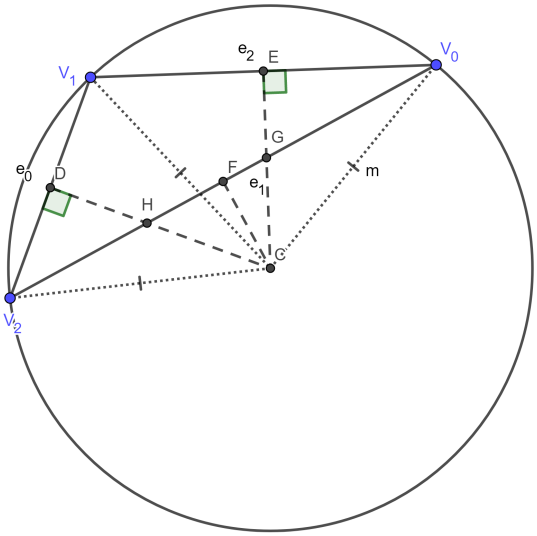

Figura 31 – Triângulo obtuso, seu circuncentro C e pontos médios das arestas D, E e F.

APÊNDICE C – Cálculo da Matriz de Rotação de Uma Base

No algoritmo de estimativa de tensores de curvatura de Rusinkiewicz (RUSINKIEWICZ, 2004), descrito na Seção 3.3, tensores de curvatura calculados em diferentes referenciais (bases), são somados de forma ponderada para formar um novo tensor. Para que essas grandezas possam ser somadas é preciso que elas sejam giradas para um mesmo referencial. Dados um referencial antigo $(u_{old}, V_{old}, norm_{old})$ e um referencial novo $(u_{new}, V_{new}, norm_{new})$ conforme mostra a Figura 32, Rusinkiewicz determinou as transformações necessárias para levar o referencial antigo ao novo, usando o fato de que somente as projeções dos vetores-base u_{old} e V_{old} sobre o plano formado pelos vetores $norm_{old}$ e $norm_{new}$, $proj(u_{old})$ e $proj(V_{old})$ em verde na Figura 32, são giradas por θ quando alinhamos o vetor $norm_{old}$ com o vetor $norm_{new}$.

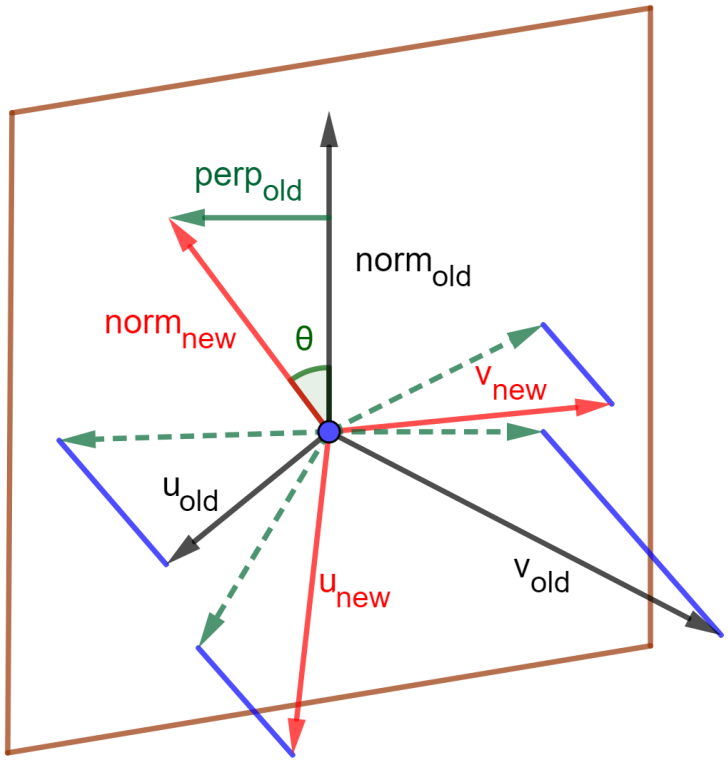

Figura 32 – Rotação da base antiga (preta) para a nova base (vermelha) de forma que as normais $norm_{old}$ e $norm_{new}$ se tornem colineares

Sendo $norm_{old}$ e $norm_{new}$ vetores unitários e $ndot = old_{norm} \cdot new_{norm} = cos(\theta)$ (linha 4 do Pseudocódigo 3.3), a projeção $proj(u_{old})$ pode ser obtida por:

$$proj(u_{old}) = u_{old} \cdot \frac{perp_{old}}{\sqrt{1 - ndot^2}},$$

que, girado pelo ângulo θ se transforma em

$$\begin{aligned} proj(u_{new}) &= (u_{old} \cdot \frac{perp_{old}}{\sqrt{1-ndot^2}}) cos(\theta) \frac{perp_{old}}{\sqrt{1-ndot^2}} - (u_{old} \cdot \frac{perp_{old}}{\sqrt{1-ndot^2}}) sen(\theta) norm_{old} \\ &= (u_{old} \cdot \frac{perp_{old}}{\sqrt{1-ndot^2}}) ndot \frac{perp_{old}}{\sqrt{1-ndot^2}} - (u_{old} \cdot \frac{perp_{old}}{\sqrt{1-ndot^2}}) \sqrt{1-ndot^2} norm_{old}. \end{aligned}$$

Sabendo que as projeções perpendiculares ao plano formado por $norm_{old}$ e $norm_{new}$ (em azul na Figura 32) não são alteradas por essa rotação, podemos obter o vetor u_{new} simplesmente subtraindo $proj(u_{old})$ e somando $proj(u_{new})$ (linhas 10–12 no Pseudocódigo 3.3)

$$\begin{aligned} u_{new} &= u_{old} - proj(u_{old}) + proj(u_{new}) \\ &= u_{old} - u_{old} \cdot \frac{perp_{old}}{\sqrt{1-ndot^2}} ndot + \\ &\quad + ((u_{old} \cdot \frac{perp_{old}}{\sqrt{1-ndot^2}}) ndot \frac{perp_{old}}{\sqrt{1-ndot^2}} - (u_{old} \cdot \frac{perp_{old}}{\sqrt{1-ndot^2}}) \sqrt{1-ndot^2} norm_{old}) \\ &= u_{old} - (u_{old} \cdot perp_{old}) \frac{1}{1+ndot} (norm_{new} + norm_{old}). \end{aligned}$$

Analogamente, para V_{new} temos

$$V_{new} = V_{old} - (V_{old} \cdot perp_{old}) \frac{1}{1+ndot} (norm_{new} + norm_{old}),$$

e $perp_{old}$ é facilmente calculado por (linha 9 no Pseudocódigo 3.3)

$$norm_{new} = ndot\ norm_{old} + perp_{old} \implies perp_{old} = norm_{new} - ndot\ norm_{old}.$$

A partir dos vetores-base dos dois referenciais, podemos derivar as variações dos vetores-base do novo referencial em relação aos vetores-base do referencial antigo pelos produtos escalares entre eles

$$\begin{aligned} \frac{\partial u_{new}}{\partial u_{old}} &= u_{new} \cdot u_{old} \\ \frac{\partial u_{new}}{\partial V_{old}} &= u_{new} \cdot V_{old} \\ \frac{\partial V_{new}}{\partial u_{old}} &= V_{new} \cdot u_{old} \\ \frac{\partial V_{new}}{\partial V_{old}} &= V_{new} \cdot V_{old}. \end{aligned}$$

Substituindo esses valores na matriz Jacobiana para transformação dos valores de um tensor na base antiga para a base nova

$$\begin{bmatrix} \tilde{b}_{11} & \tilde{b}_{12} \\ \tilde{b}_{21} & \tilde{b}_{22} \end{bmatrix} = \begin{bmatrix} \frac{\partial u}{\partial \tilde{u}} & \frac{\partial v}{\partial \tilde{u}} \\ \frac{\partial u}{\partial \tilde{v}} & \frac{\partial v}{\partial \tilde{v}} \end{bmatrix} \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix} \begin{bmatrix} \frac{\partial u}{\partial \tilde{u}} & \frac{\partial u}{\partial \tilde{v}} \\ \frac{\partial v}{\partial \tilde{u}} & \frac{\partial v}{\partial \tilde{v}} \end{bmatrix},$$

temos

$$
\begin{aligned}
new\_b_{11} &= old\_b_{11}(u_{new}\cdot u_{old})^2 + 2\ old\_b_{12}(u_{new}\cdot u_{old})(u_{new}\cdot V_{old}) + old\_b_{22}(u_{new}\cdot V_{old})^2 \\
new\_b_{12} &= old\_b_{11}(u_{new}\cdot u_{old})(V_{new}\cdot u_{old}) + \\
&\quad + old\_b_{12}((u_{new}\cdot u_{old})(V_{new}\cdot V_{old}) + (V_{new}\cdot u_{old})(u_{new}\cdot V_{old})) + \\
&\quad + old\_b_{22}(u_{new}\cdot V_{old})(V_{new}\cdot V_{old}) \\
new\_b_{11} &= old\_b_{11}(V_{new}\cdot u_{old})^2 + 2\ old\_b_{12}(V_{new}\cdot u_{old})(V_{new}\cdot V_{old}) + old\_b_{22}(V_{new}\cdot V_{old})^2.
\end{aligned}
$$