Universidade Estadual de Campinas
Faculdade de Engenharia Elétrica e de Computação

Mário Santos Camillo

Um Estudo do Mapeamento de Estruturas BRep em GPU

Monografia apresentada à Faculdade de Engenharia Elétrica e de Computação como parte do Exame de Qualificação para a obtenção do título de Mestre em Engenharia Elétrica. Área de concentração: Engenharia de Computação.

Orientadora: Profa. Dr. Wu, Shin-Ting

Campinas
2013

Sumário

1 Introdução

O desenvolvimento de tecidos é de interesse para diversas aplicações diferentes, desde aquelas para entretenimento e propaganda até o altamente lucrativo mundo da moda. Apesar de várias técnicas diferentes para a simulação de tecidos em computador estarem sendo desenvolvidas e evoluindo desde o começo da década de setenta, simulações realistas de tecidos em tempo real ainda são um problema desafiador para ambas as comunidades de computação gráfica e engenharia têxtil.

Dentre os diferentes modelos desenvolvidos nos chama a atenção aqueles baseados em modelos consistentes com os conceitos da Física devido a sua eficiência e vasta aplicabilidade (como, por exemplo, simulação de tecidos comuns e pele humana). Em especial, damos continuidade aos estudos da teoria de superfície de Cosserat aplicada a tecidos [6, 5, 4] devido a duas características principais: o modelo representa a dinâmica do tecido levando em conta a correlação entre enrugamento e esticamento do tecido de forma que as rugas se formam naturalmente; e porque uma simples analogia pode ser feita entre quantidades estáticas(forças de contato) e geométricas (medidas de enrugamento e estimanto). Com o intuito de alcançar um desempenho de tempo real (mais de 24 frames por segundo) acreditamos que a implementação de um modelo fundamentado nessa teoria utilizando o *hardware* das placas gráficas atuais seja uma alternativa viável.

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 progrmáveis e altamente paralelos [16]. Com o advento do *pipeline* programável em GPUs, diversos trabalhos de adaptação de algoritmos para GPGPU (*General Purpose GPU*) passaram a ser desenvolvidos [19]. A maioria dos trabalhos desenvolvidos trata de cômputo de dados maciços armazenados em matrizes de grandes dimensões. Nesses casos uma das técnicas de otimização possível é modificar a estrutura de dados para tirar proveito da alta largura de banda no acesso à memória global das GPUs. Dentro das pesquisas desenvolvidas algumas focam em estruturas de dados genéricas na GPU [14] enquanto outras criam novos mapeamentos de malhas 3D específicos para resolver os problemas propostos [22, 10].

A modelagem da teoria de superfície de Cosserat para malhas de topologia arbitrária proposta por Costa e Wu [4], em específico, apresenta um algoritmo altamente paralelizável mas também é bastante complexo e extremamente dependente de acessos aleatórios à memória para travessia de vizinhanças na malha 3D, o que torna a sua adaptação para GPGPU desafiadora. A proposta desse trabalho de mestrado é realizar um estudo do mapeamento de estruturas *BRep* em GPU e seu impacto no desempenho, tendo como motivação (e aplicação de avaliação) desenvolver uma versão massivamente paralela (em GPGPU) desse algoritmo que tenha taxa iterativa em tempo real.

Essa monografia está organizada da seguinte forma: a seção 1.1 traz um pequeno resumo de trabalhos relacionados; a seção 2 faz uma revisão da arquitetura das GPGPUs atuais enquanto a seção 3 apresenta, de forma resumida, a teoria de Cosserat para simulação de tecidos. Por fim, a seção 4 mostra os experimentos realizados e resultados preliminares obtidos, com a seção 5 fazendo uma breve conclusão e apresentando os planos futuros para a continuação do trabalho proposto.

1.1 Trabalhos Relacionados

As estruturas de dados tradicionalmente utilizadas para armazenar as informações de uma malha 3D e sua topologia (*Boundary Representation*), tal como a estrutura *half-edge* não são apropriadas para a arquitetura da GPU e, devido à sua complexidade podem apresentar desempenho insatisfatório. Diversos trabalhos tentam atacar esses problemas através de mapeamentos dos dados da malha para

GPU [10, 22] e criação de novas estruturas ou organização desses dados na memória [1, 25].

Partindo de uma estrutura *half-edge* em CPU, Hjelmervik e Léo [10] descrevem uma estrutura de dados para realizar a simplificaçao de malhas utilizando *shaders* na GPU. Essa estrutura baseia-se em duas texturas 2D que contém as posições de todos os vértices candidatos à remoção e a posição de todos seus vizinhos *1-ring*. Essas texturas são preenchidas de forma que os vértices fiquem agrupados nas linhas por valência (número de vizinhos). Isso é feito para diminuir a necessidade de condicionais e a diferença de tamanho de loop no *shader* sendo executado. Essa é uma técnica simples para melhorar o desempenho na GPU e nós a utilizamos nos primeiros experimentos de prova de conceito (ver seção 4). Mas ela tem uma limitação com relação ao consumo de memória devido a alta taxa de redundância de dados.

Já Shiue et al. [22] mapeiam os dados de uma malha de forma particionada com o intuito de executar subdivisões de Catmull-Clark de forma massivamente paralela na GPU. Essas subdivisões (chamadas de *fragment-mesh*) são obtidas através da malha original e de uma primeira subdivisão feita na CPU. A partir desses dados as partições são feitas de um vértice central e de sua vizinhança *2-ring*, organizados numa textura 1D de acordo com uma numeração em espiral, partindo do centro em direção às bordas. Essa numeração especial garante a localidade dos dados e proporciona a obtenção das vizinhanças de uma forma trivial. A idéia de uma estrutura de dados que mantenha a localidade alterando-se a numeração dos vértices e fazendo particionamentos é interessante e merece maior investigação para analisar sua aplicabilidade no nosso problema.

Alumbaugh e Jiao [1], por sua vez, definem uma estrutura de dados baseada em vetores com o intuito de oferecer uma fexibilidade e facilidade de uso parecidos com a *half-edge* mas com um gasto menor de memória e maior facilidade de compartilhamento dos dados assim como seu particionamento para uso em ambientes paralelos. Comparado com a estrutura *half-edge*, a estrutura proposta poderia ser portada mais facilmente para a GPU, mas ainda é preciso verificar o desempenho oferecido pela mesma devido aos acessos aleatórios à memória global durante a travessia da malha.

Enquanto isso, Yoon e Lindstrom [25] atacaram o problema de desempenho de algoritmos geométricos e gráficos, como extração de iso-superfícies e renderização dependente de vista, pelo lado da organização dos dados na memória. Baseados no fato de que em alguns casos o tempo de execução de um programa pode ser tanto função de seu padrão e eficiência de acesso ao *cache*, como do seu número de operações [7], Yoon e Lindstrom definem métricas de estimativa de erros de *cache* (*cache miss*) de uma determinada organização dos dados de uma malha na memória. Com isso eles caem no problema de minimizar essa métrica, o que é feito através de algoritmos de particionamento de dados da biblioteca METIS[12]. Essa simples reorganização dos dados baseada no conhecimento do *cache* do sistema pode gerar ganhos de mais de 100x (na CPU) na extração de iso-superfícies, quando comparados com dados organizados apenas pela ordenação dos vértices no eixo X. Ainda no assunto de organização de dados, Lobeiras et al. [15] apresentam um estudo de como diferentes formas de acessar e estruturar uma matriz de dados na GPU podem influenciar no desempenho de uma transformada de Fourier. Apesar do trabalho de Yoon et al. ser voltado para memórias *cache* em CPU a sua teoria pode ser adaptada para a forma de acesso à memória global baseada em blocos (ver seção 2.2) como uma forma de diminuir o número de requisições feitos à essa memória.

Há ainda esforços, como a *GLIFT* [14], de criar bibliotecas com estruturas de dados de uso geral (tal como a biblioteca *STL* para *C++*) que sejam portadas e otimizadas para a GPU e facilitem o desenvolvimento de programas GPGPU. Bibliotecas como essa poderia ser utilizada para fazer uma tradução da estrutura de dados de uma malha 3D (tal como uma *half-edge*) para a GPU.

2 Arquitetura da GPU

A arquitetura geral das GPUs segue o padrão definido por von Neuman, sendo dividida em processadores e memórias ligadas através de um barramento de comunicação. Porém, devido à sua origem como processadores gráficos, a arquitetura das GPUs tem certas características específicas. Nessa seção apresentamos as especificidades da arquitetura das GPUs atuais. Focamos nos *chips* fabricados pela empresa NVidia, com sua arquitetura CUDA, mas os pontos levantados aqui podem ser generalizados para chips de outras fabricantes que suportem GPGPU.

2.1 Arquitetura dos Processadores

As GPUs, como o próprio nome diz, começaram como *chips* específicos para fazer 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 *Kepler* (GK104), que implementam a versão (*compute capability*) 3.0 da arquitetura CUDA [18]. Uma visão geral de um processador desse tipo pode ser visto na Fig. 1

As GPUs são compostas por centenas de de processadores mais simples numa arquitetura chamada de massivamente paralela. Esses processadores são chamados de *Stream Multiprocessors* (SM) e cada um deles pode rodar várias centenas de *threads*. Uma coleçao 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, onde cada um desses *warps* é executado de forma SIMD (single instruction multiple data) 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 [18]. A Fig. 2 mostra a relação entre *threads* e SMs de forma clara e sucinta, enquanto a Fig. 3 mostra como um exemplo de como um escalonador de *warps* funciona.

Essas características dos processadores de uma GPU precisam ser conhecidas por um programador para que o *kernel* desenvolvido possa ser otimizado e aproveitar 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 arquiteturada 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 para cada thread (registradores). Um exemplo dessa hierarquia pode ser visto na Fig. 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.

A primeira característica diferenciada da memória da GPU é o largura de banda do barramento de acesso à memória global. Esse tamanho varia entre 128 e 386 bits, dependendo da versão da placa de vídeo. O que isso significa, na prática, é que a memória global da GPU é cara (alta latência) para acessos, mas pode ser usada para acessar um maior número de dados a cada requisição.

Outra especificidade relacionada com essa capacidade de acessar um maior número de dados da memória global por requisição é a capacidade que os SMs têm de organizar os acessos feitos em uma instrução sendo executada por um *warp* para que eles sejam feitos no menor número de requisições possíveis. Esses acessos são agrupados em acessos contíguos de 32, 64 ou 128 bytes da melhor forma possível [17]. A Fig. 5 mostra como esse agrupamento de acessos é realizado.

As GPUs também tem mais dois tipos de memória além das que já foram 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 [17]. Esses dois tipos de memória também são mostrados na Fig. 4

2.3 Estratégias de Otimização

Conforme mostrado anteriormente, a GPU tem diversas características únicas quando comparada à já conhecida arquitetura de uma CPU. Essas especificidades precisam ser conhecidas pelo programador que estiver desenvolvendo os *kernels* que irão ser executados para que seja possível extrair o melhor desempenho possível da placa gráfica. Nas próximas seções discorremos sobre algumas estratégias de otimização advindas desse conhecimento e que devem ser levadas em consideração durante o desenvolvimento da solução do nosso problema.

2.3.1 Simplificação dos Kernels

A primeira estratégia que podemos extrair vem da característica SIMD 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* rebarramentocados 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.

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 [18]. Essas

limitações podem fazer com que um *kernel* mal implementado tenha uma ocupação pobre da GPU. Ou seja, cada SM estará utilizando apenas um pequena porcentagem da sua capacidade ótima de *threads* simultâneas. Isso pode causar um grande impacto na desempenho de algoritmos para GPGPU [15]. 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 fechados no código. 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 [17] para o cálculo dos parâmetros para conseguir uma ocupação ótima para cada *kernel*.

2.3.3 Minimizar Requisições 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 desse recurso, maximizando a quantidade de dados obtidos em cada acesso. Lobreiras et al. [15] mostram qual o impacto que acessos não agrupados podem ter na 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 um certo pedaço dos dados da memória global no início da execução e 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 fazer uma análise dos ganhos e perdas ao se utilizar essa estratégia.

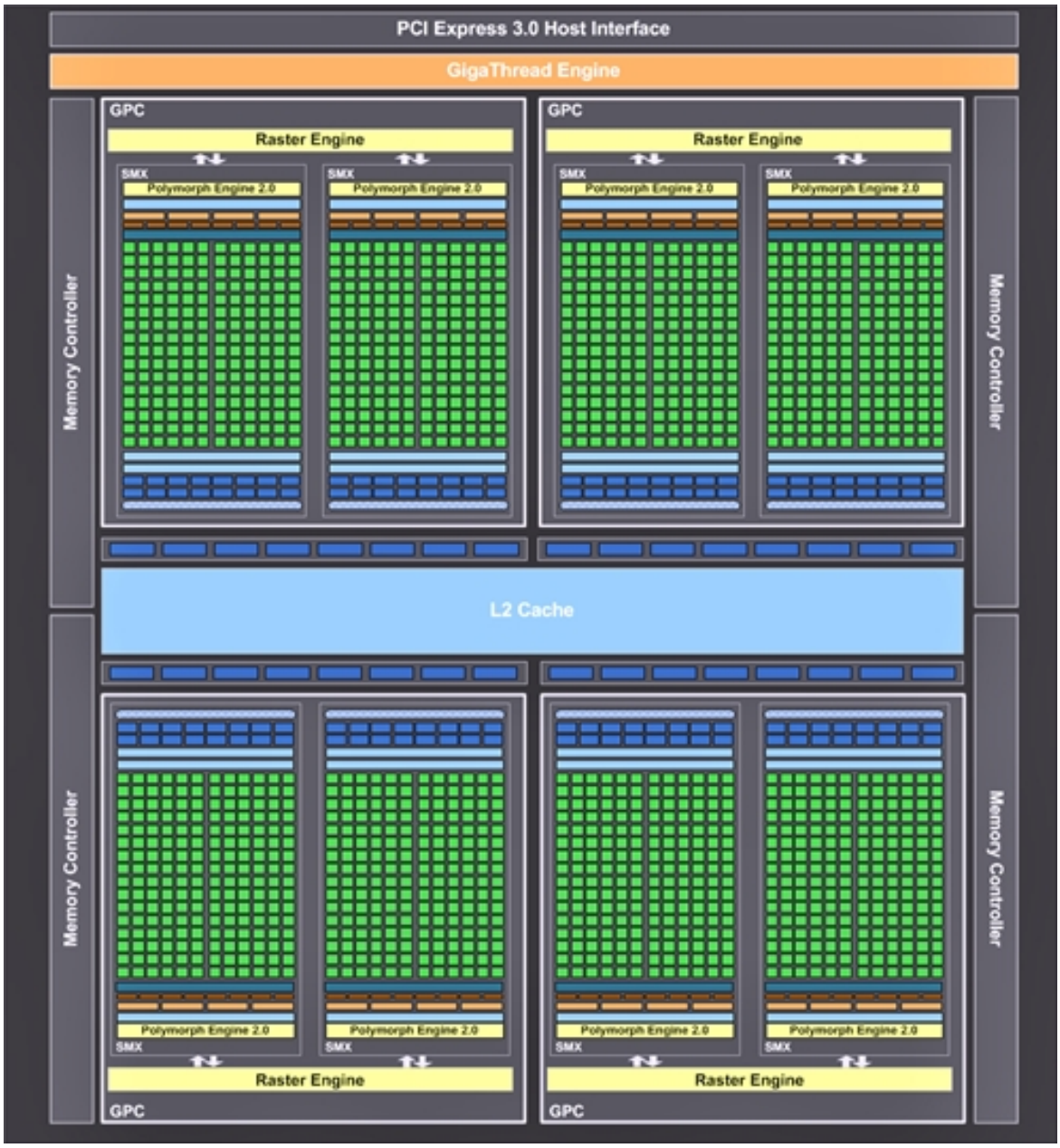

Figura 1: Visao geral da arquitetura de um *chip Kepler* GK104

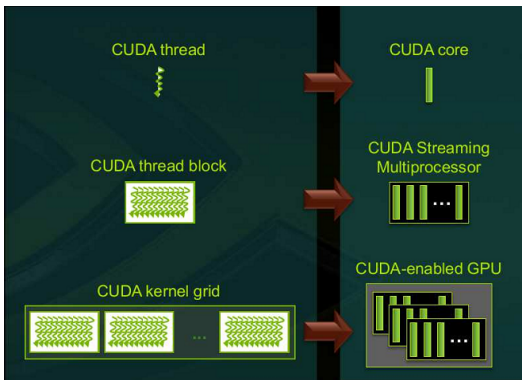

Figura 2: Mapeamento entre *threads* e SMs

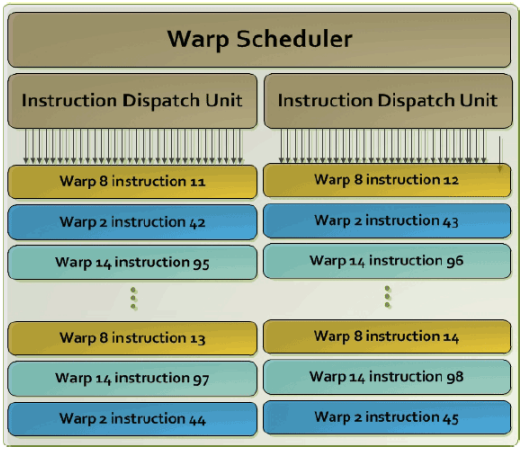

Figura 3: Exemplo de escalonamento de *warps* em um SM

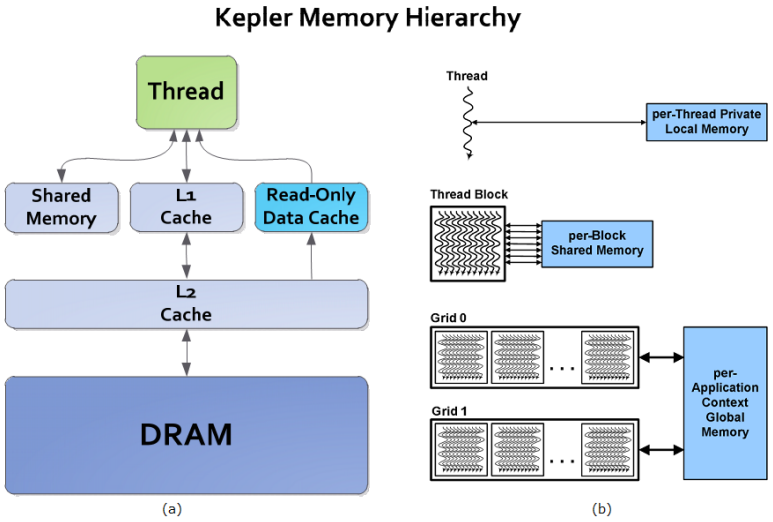

Figura 4: Memorias de uma GPU CUDA. (a) mostra a hierarquia entre elas enquanto (b) mostra o nivel de acesso

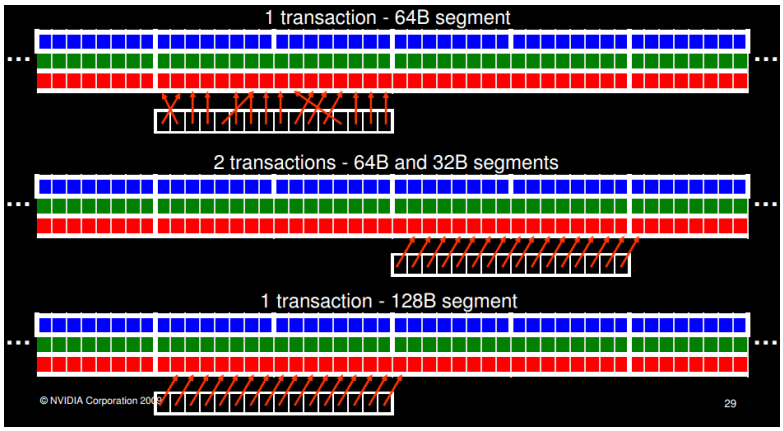

Figura 5: Exemplo de acesso à memória global agrupado no menor número de segmentos possível

3 Simulação de Tecidos

Existem diferentes formas de se simular tecidos de forma discreta no computador [5]: modelo geométrico, modelo massa-mola e modelos baseados na física. O modelo geométrico consiste em representar o tecido como uma superfície parametrizada em curvas. Com isso o tecido pode ser simulado através de alterações nos parâmetros das curvas que caracterizam a sua superfície. Já o modelo massa-mola baseia-se no paradigma de partículas, no qual um tecido é representado por um conjunto de pontos com massa que são interligados por molas e amortecedores para simular as características do material [20, 3, 11]. Por último, o modelo físico usa uma técnica baseada em mecânica contínua, na qual um tecido é considerado como um meio contínuo onde a teoria de placas finas não lineares é aplicada para a análise do comportamento de compressão, cisalhamento e flexão do material [23, 9, 2]. Um ponto importante a ser frisado é que, depois de diferenciações finitas no espaço e tempo, tanto os modelos baseados em partículas como em mecânica contínua tem formulações diferenciais ordinarias bastante parecidas. Eles se diferenciam, essencialmente, nas *equações constituintes*, que são reponsáveis pelas forças internas devido à deformação do tecido.

Nessa seção nós apresentamos um modelo baseado na teoria de Cosserat [5, 4, 6], que é considerado um modelo geometricamente exato para placas finas [8]. Esse modelo é usado como uma aplicação para a demonstração e validação das técnicas de otimização de layout de malhas 3D para GPU sendo desenvolvidas.

3.1 Teoria de Cosserat

A teoria de Cosserat por si só merece várias páginas e capítulos para ser explicada e alguns anos para ser devidamente entendida. O que iremos apresentar a seguir é apenas um pequeno resumo das principais equações e modelos necessários para entender os cálculos e resultados obtidos na seção 4. Mais especificamente, apresentaremos o modelo proposto por Green et al. [8], baseado na teoria de superfícies de Cosserat da Mecânica do Contínuo. Ele utiliza propriedades de geometria diferencial, como tensores métricos e de curvatura, para considerar alterações de área e curvatura na estimação das forças internas

3.1.1 Notação de Einstein

A notação de Einstein permite escrever equações complexas, especialmente envolvendo somatório de termos, de maneira compacta [13]. Ela é extensivamente utilizada no desenvolvimento das superfícies de Cosserat.

Nessa notação, letras romanas representam os valores 1,2 e 3 e as gregas apenas 1 e 2, numa combinação de diversos fatores. Por exemplo, uma equação $A = x_i y_\alpha$, é uma forma compacta de expressar

$$\begin{aligned} A = x_i y_\alpha &= \sum_{i=1}^{3}\sum_{\alpha=1}^{2} x_i y_\alpha \\ &= x_1 y_1 + x_2 y_1 + x_3 y_1 \\ &= x_1 y_2 + x_2 y_2 + x_3 y_2 \,. \end{aligned} \tag{1}$$

3.1.2 Tensores métrico e de curvatura

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

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

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

$$d\mathbf{r} = \frac{\partial \mathbf{r}}{\partial x^1} dx^1 + \frac{\partial \mathbf{r}}{\partial x^2} dx^2, \tag{3}$$

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

$$I(\alpha(t)) \;=\; d\mathbf{r} \cdot d\mathbf{r} = a_{\alpha\beta} dx^\alpha dx^\beta = \begin{bmatrix} dx^1 & dx^2 \end{bmatrix} \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \begin{bmatrix} dx^1 \\ dx^2 \end{bmatrix} \tag{4}$$

onde

$$a_{\alpha\beta}(\mathbf{r}(a)) = \frac{\partial \mathbf{r}}{\partial x^\alpha} \cdot \frac{\partial \mathbf{r}}{\partial x^\beta}. \tag{5}$$

A Eq. (4) é a *primeira forma fundamental* da superfície $\mathcal{S}$, enquanto $a_{\alpha\beta}$ são chamados de componentes de *tensor métrico*. Os coeficientes $a^{\alpha\beta}$ podem ser obtidos através da matriz inversa da Eq. (4), sendo

$$a^{11} = \frac{a_{22}}{a}, \quad a^{12} = a^{21} = -\frac{a_{12}}{a}, \quad a^{22} = \frac{a_{11}}{a},$$

onde

$$a = a_{11}a_{22} - a_{12}a_{21}.$$

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

$$II(\alpha(t)) \;=\; b_{\alpha\beta} dx^\alpha dx^\beta = \begin{bmatrix} dx^1 & dx^2 \end{bmatrix} \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix} \begin{bmatrix} dx^1 \\ dx^2 \end{bmatrix} \tag{6}$$

onde

$$b_{\alpha\beta} = \mathbf{n} \cdot \frac{\partial^2 \mathbf{r}}{\partial x^\alpha \partial x^\beta} \tag{7}$$

com

$$\mathbf{n} = \left(\frac{\partial \mathbf{r}}{\partial x^1} \times \frac{\partial \mathbf{r}}{\partial x^2}\right) \Big/ \left\| \frac{\partial \mathbf{r}}{\partial x^1} \times \frac{\partial \mathbf{r}}{\partial x^2} \right\|. \tag{8}$$

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

De forma simplificada, ao consideramos um ponto P pertencente a superfície $\mathcal{S}$, $a_{\alpha\beta}$ está relacionado as propriedades comprimento e área enquanto $b_{\alpha\beta}$ descreve o quão curva é a superfície.

Em condições onde $\frac{\partial r}{\partial x^1}$ e $\frac{\partial r}{\partial x^2}$ são ortonormais, o tensor de curvatura fica igual 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, na base de referência escolhida, as direções principais no ponto. Com isso, podemos escrever as seguintes igualdades

$$\begin{aligned}\frac{\partial \mathbf{n}}{\partial x^1} &= b_{11}\frac{\partial \mathbf{r}}{\partial x^1} + b_{12}\frac{\partial \mathbf{r}}{\partial x^2} \\ \frac{\partial \mathbf{n}}{\partial x^2} &= b_{21}\frac{\partial \mathbf{r}}{\partial x^1} + b_{22}\frac{\partial \mathbf{r}}{\partial x^2}.\end{aligned} \tag{9}$$

3.1.3 Modelo de Cosserat

Seja $\mathcal{S}(t) = \mathbf{r}(x^1, x^2, t)$ uma superfície elástica deformável no instante t, e sejam curva-x^1 e curva-x^2 as curvas coordenadas sobre $\mathcal{S}$ e x^3 a normal para $\mathcal{S}(t)$. x^i são identificados como *coordenadas variáveis* (*convected coordinates*) porque qualquer ponto em $\mathcal{S}$ possui as mesmas coordenadas curvilíneas tanto no estado de referência como no deformado. As derivadas ao longo dessas curvas são representados por a_i. Estas quantidades são linearmente independentes e compõem os *vetores-base*, onde $\frac{\partial \mathbf{r}}{\partial x^1}$ e $\frac{\partial \mathbf{r}}{\partial x^2}$ estão sobre o plano tangente normal a $\mathbf{n} = \mathbf{a}_3$. Associado a cada ponto de $\mathcal{S}(t)$, temos ainda um vetor $\mathbf{d}$ com direção saindo da superfície, mas não necessariamente ao longo de sua normal.

Destes elementos, podemos computar os componentes $a_{\alpha\beta}$ do tensor métrico e os componentes $b_{\alpha\beta}$ de seu tensor de curvatura para cada ponto de $\mathcal{S}(t)$. Particularmente para $t = t_0$, a superfície não deformada é referida por $\mathcal{S}(t_0) = \mathbf{R}(x^1, x^2) = \mathbf{r}(x^1, x^2, t_0)$, com os tensores métricos e de curvatura em cada ponto denotados por $A_{\alpha\beta} = a_{\alpha\beta}(t_0)$ e $B_{\alpha\beta} = b_{\alpha\beta}(t_0)$ respectivamente, e o vetor diretor como $\mathbf{D} = \mathbf{d}(t_0)$.

Os vetores diretores de uma superfície de Cosserat não estão necessariamente na direção da normal associada ao mesmo ponto, nem tem obrigatoriamente norma unitária. No caso onde os vetores coincidem com os vetores normais para qualquer t, ou seja $\mathbf{d}(t) = \mathbf{a}_3(t)$, a superfície passa a ser denominada como *superfície de Cosserat com vetor diretor inextensível e paralelo à normal*. De acordo com Green et al., nesses casos temos relações explícitas entre as *variáveis cinéticas*, *tensão de esticamento* $\varepsilon_{\alpha\beta}$ e tensão de curvatura $\kappa_{\beta i}$, e as propriedades geométricas, tensor métrico $a_{\alpha\beta}$ e tensor de curvatura $b_{\alpha\beta}$ [8]:

$$\varepsilon_{\alpha\beta} = \frac{1}{2}(a_{\alpha\beta} - A_{\alpha\beta}) \qquad \kappa_{\beta\alpha} = -(b_{\beta\alpha} - B_{\beta\alpha}). \tag{10}$$

Melo e Wu mostram em [24] que pequenas modificações na formulação para energia interna $\mathcal{A}(\mathbf{r}, t)$ proposta por Green et al. leva a uma expressão onde variáveis possuem um interpretação geométrica mais significativa.

$$\begin{aligned}\mu\mathcal{A}(\mathbf{r}, t) &= \Phi^{\alpha\beta\gamma\delta}\varepsilon_{\alpha\beta}(t)\varepsilon_{\gamma\delta}(t) + \Psi^{\alpha\beta\gamma\delta}\kappa_{\alpha\beta}(t)\kappa_{\gamma\delta}(t) + \\ &\quad \Theta^{\alpha\beta\gamma\delta}\varepsilon_{\alpha\beta}(t)\kappa_{\gamma\delta}(t),\end{aligned} \tag{11}$$

onde μ é a densidade de massa por unidade de área de $\mathcal{S}$ no instante t, e

$$\begin{aligned}\Phi^{\alpha\beta\gamma\delta} &= \zeta[A^{\alpha\beta}A^{\gamma\delta} + (A^{\alpha\gamma}A^{\beta\delta} + A^{\alpha\delta}A^{\beta\gamma})] \\ \Psi^{\alpha\beta\gamma\delta} &= \xi[A^{\alpha\beta}A^{\gamma\delta} + A^{\alpha\gamma}A^{\beta\delta} + A^{\alpha\delta}A^{\beta\gamma}] \\ \Theta^{\alpha\beta\gamma\delta} &= \phi[A^{\alpha\beta}A^{\gamma\delta} + (A^{\alpha\gamma}A^{\beta\delta} + A^{\alpha\delta}A^{\beta\gamma})].\end{aligned} \tag{12}$$

Os termos $\Phi^{\alpha\beta\gamma\delta}$ e $\Psi^{\alpha\beta\gamma\delta}$ afetam predominantemente os comportamentos de esticamento e curvamento, respectivamente. Logo, denominamos ζ e ξ *coeficientes elásticos* e os parâmetros $\Phi^{\alpha\beta\gamma\delta}$ e $\Psi^{\alpha\beta\gamma\delta}$ *propriedades do material*. O termo $\Theta^{\alpha\beta\gamma\delta}$ por sua vez, afeta de maneira mais significativa o número

de ondulações (comportamento fora-do-plano) na superfície sendo deformada, sendo chamado de *fator de enrugamento*.

3.2 Teoria de Cosserat Aplicada a Simulacao de Tecidos

De acordo com a teoria de superfícies de Cosserat a *rigidez do tecido* em cada amostra $\mathbf{r}(t)$ em um instante de tempo t pode ser expressada como uma função da variação da energia interna $\mathcal{A}(\mathbf{r}, t)$ com relação às medidas de esticamento ε e de curvatura κ:

$$K(\mathbf{r}, t)\mathbf{r}(t) = \mu \frac{\delta \mathcal{A}(\mathbf{r}, t)}{\delta \mathbf{r}(t)} = \frac{\partial \mathcal{A}(\mathbf{r}, t)}{\partial \varepsilon(t)} + \frac{\partial \mathcal{A}(\mathbf{r}, t)}{\partial \kappa(t)}. \tag{13}$$

Usando a Eq. (13) na equação parcial diferencial de equilíbrio [23], nos temos:

$$\mu \frac{\partial^2 \mathbf{r}}{\partial t^2} + \varrho \frac{\partial \mathbf{r}}{\partial t} + \mathbf{K}(\mathbf{r}, t)\mathbf{r}(t) = \mu \mathbf{F}(\mathbf{r}, t) \ , \tag{14}$$

onde μ é a densidade de massa (por unidade de área), ϱ é o coeficiente de amortecimento das forças que agem contrárias às fricções internas do tecido e às fricções externas com fluidos, e $\mathbf{F}$ corresponde à contribuição total de forças externas por unidade de massa em $\mathbf{r}$.

O diagrama de fluxo de dados de uma simulação iterativa pode ser visto na Fig. 6. Começando com uma superfície amostrada, para cada amostra são computados, no tempo t, os componentes dos tensores de superfície $N^{i\alpha}$ e $M^{\gamma\alpha}$ através das derivadas dos tensores métrico ($a_{\alpha\beta}$) e de curvatura ($b_{\alpha\beta}$). Com isso são calculadas as forças internas conforme a Eq. (13), que são equivalentes às derivadas covariantes $\mathbf{N}^1_{|1}$ e $\mathbf{N}^2_{|2}$. Substituindo-as na Eq. (14), obtemos a aceleração $\dot{\mathbf{v}}(t)$ e a velocidade $\mathbf{v}(t + \Delta t)$ do ponto amostrado. Isso permite o cálculo da posição da amostra no próximo instante de tempo $t + \Delta t$. Aplicando sucessivamente esses passos temos uma série de malhas evoluindo com o tempo, obedecendo restrições físicas e geométricas, o que cria uma ilusão da animação do tecido.

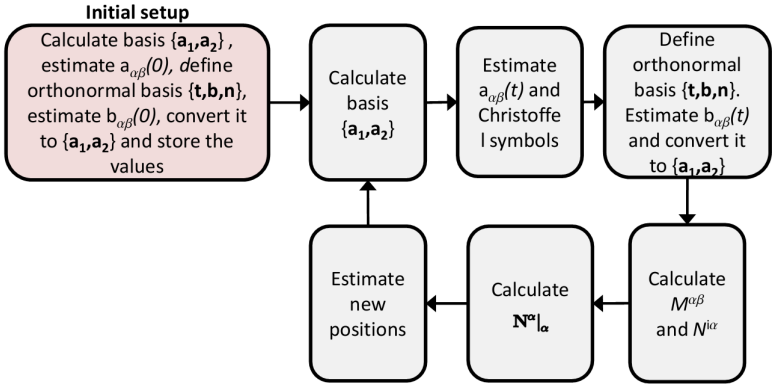

Figura 6: Fluxo de dados de uma simulação.

4 Experimentos e Resultados Preliminares

Durante o período que antecedeu a redação dessa monografia foram realizados alguns experimentos para averiguar, primeiramente, a existência de um problema a ser resolvido, e a viabilidade do trabalho sendo proposto. Como um primeiro passo foi realizado um estudo do desempenho do algoritmo desenvolvido por Costa e Wu [4], cuja implementação é apresentada de forma resumida na seção 4.1.

Depois foram desenvolvidos dois programas, derivados do algoritmo de Costa e Wu, em CUDA. Esses programas são descritos nas seções 4.2 e 4.3. Por fim, todos os algoritmos foram executados e dados dos seus desempenhos foram recolhidos. Esses resultados e sua análise podem ser vistos na seção 4.4

4.1 Implementação Costa e Wu

Em [4] Costa e Wu derivam um algoritmo baseado nos trabalhos de Rusinkiewicz [21] para o cálculo dos tensores métrico e de curvatura em uma malha de topologia arbitrária. Os passos para esse cálculo são descritos à seguir.

4.1.1 Determinação da base $\mathbf{a}_i$

Para criar uma base de vetores $\mathbf{a}_i$ que satisfaçam as propriedades de *coordenadas móveis* (*convected coordinates*), são escolhidos em passo de inicialização, para cada vértice v, vértices vizinhos de referência, e a aresta relacionada é usada para determinar $\mathbf{a}_\alpha$. $\mathbf{a}_3$ é determinado utilizando-se métodos tradicionais de cálculo de normal de superfícies. Finalmente os vetores $\mathbf{a}_\alpha$ são girados para que fiquem pertencentes ao plano perpendicular a $\mathbf{a}_3$.

Com $\mathbf{a}_i$ estimado, é possível calcular os componentes $a_{\alpha\beta}$ do tensor métrico conforme a Eq. (5).

4.1.2 Estimativa de tensores métrico e de curvatura

Para cada vértice v da malha, é escolhida arbitrariamente uma das arestas adjacentes como v_0v para definir um referencial local (Fig. 7).

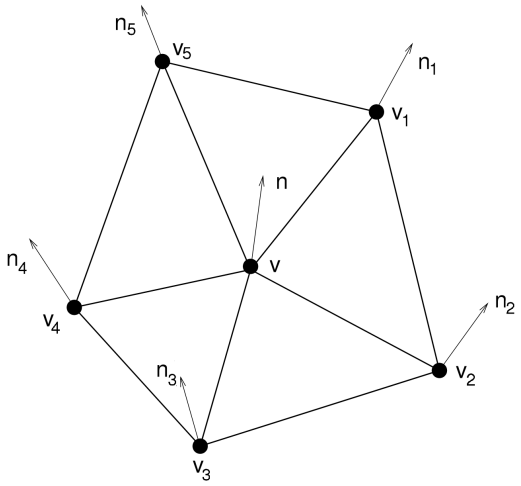

Figura 7: Vizinhança *1-ring* de um vértice v

$$\begin{aligned} \mathbf{t} &= \frac{v_0v}{|v_0v|}. \\ \mathbf{b} &= \mathbf{n} \times \mathbf{t} \\ \mathbf{n} &= \mathbf{n}. \end{aligned}$$

Todas as arestas adjacentes a v, $\mathbf{e}_i$, são representadas como combinações lineares dos vetores-base desse referencial local$\{\mathbf{t}, \mathbf{b}, \mathbf{n}\}$.

$$\begin{aligned} \mathbf{e}_i &= \mathbf{t}t_i + \mathbf{b}b_i \\ t_i &= \mathbf{e}_i \cdot \mathbf{t} \\ b_i &= \mathbf{e}_i \cdot \mathbf{b}\,. \end{aligned}$$

Utilizando Eq. Eq. (9), o seguinte sistema de equações é montado:

$$\begin{aligned} \triangle\mathbf{n}_{i,t}t_i &= b_{11}t_i^2 + b_{12}t_ib_i \\ \triangle\mathbf{n}_{i,b}b_i &= b_{12}b_it_i + b_{22}b_i^2 \\ \triangle\mathbf{n}_{i,t}b_i &= b_{11}t_ib_i + b_{12}b_i^2 \\ \triangle\mathbf{n}_{i,b}t_i &= b_{12}t_i^2 + b_{22}t_ib_i \end{aligned}$$

A solução desse sistema nos dá os elementos do tensor de curvatura em v com respeito a $\{\mathbf{t}, \mathbf{b}, \mathbf{n}\}$.

Para produzir boas estimativas independente das distribuição das arestas, o sistema de equações montado de fato é resultado da soma das equações envolvendo as mesmas incógnitas de todas as arestas adjacentes.

4.2 Implementação ingênua na GPU

A primeira implementação feita para a GPU foi apenas uma traduçao do algoritmo de Costa e Wu para a linguagem CUDA. Porém, devido à arquitetura da memória dos *chips* gráficos (ver seção 2.2) também foi necessário fazer uma tradução da estrutura de dados sendo utilizada. No algoritmo original é utilizada uma estrutura *half-edge* que utiliza bastante ponteiros e memória dinâmica. Para os algoritmos em GPU foi criada uma simples estrutura de lista de adjacências, com dois vetores de *float4* (estrutura com quatro *floats*) para a posição e normal dos vértices, um vetor de *float4* para as normais da face e mais dois vetores auxiliares para a adjacência dos vértices e das faces. Um exemplo do acesso não agrupado causado por esse tipo de estrutura pode ser visto na Fig. 8.

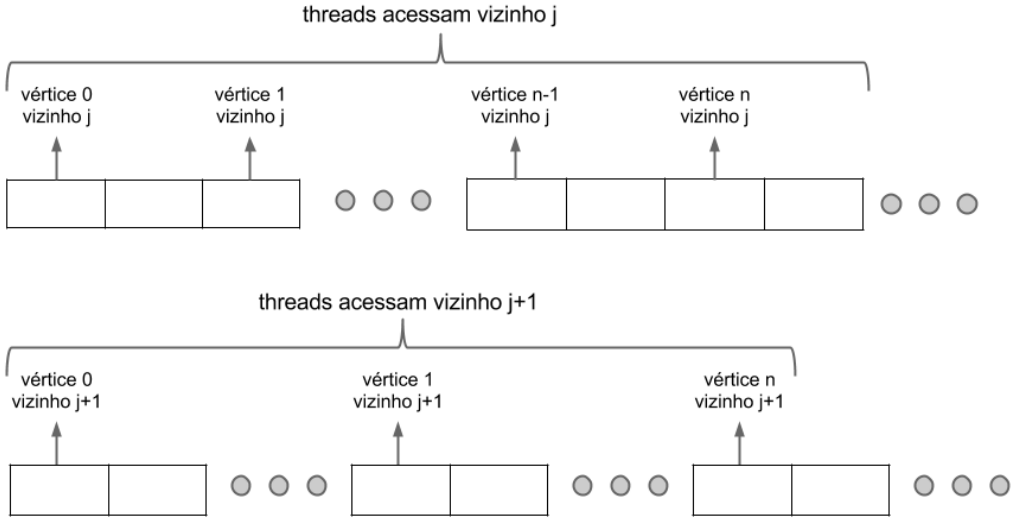

Figura 8: Exemplo da organização dos dados nos vetores aglutinados (*coalesced*)

4.3 Implementação com memória agrupada na GPU

Uma segunda implementação para GPU foi feita com o intuito de demonstrar o impacto de performance que uma estrutura de dados pode ter no caso de um algoritmo como o de cálculo dos tensores métrico e de curvatura. Esse algoritmo é altamente paralelizável pois os mesmos cálculos podem ser executados para cada vértice de forma independente entre eles. Porém, por precisar percorrer a vizinhança *1-ring* de cada vértice, são realizados muitos acessos aleatórios à memória global o que, em escalas maiores, pode comprometer o desempenho de todo o algoritmo. Para contornar esse problema, foram criados alguns vetores de dados auxiliares para tornar o acesso à memória global mais aglutinado (*coalesced*), conforme a estratégia de otimização descrita na seção 2.3. Dessa forma o hardware da GPU pode cobrir mais requisições por acesso.

Foram gerados então três novos vetores de dados (*float3*) com tamanho igual a valência máxima de um vértice mais um: um vetor para as posições dos vértices, um para as normais e outro para as normais das faces. Cada um desses vetores foi então preenchido com os dados redundantes do próprio vetor e todos os vetores/faces na sua vizinhança *1-ring*. A forma de preenchimento desses vetores também é feita para otimizar os acessos à memória conforme pode ser visto na Fig. 9. Por fim, o algoritmo ingênuo para GPU foi alterado para que esses vetores auxiliares fosse utilizados durante os cálculos dos tensores métrico e de curvatura, ao invés dos vetores originais.

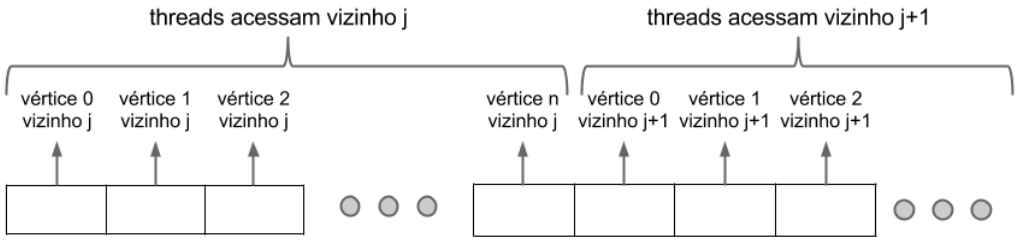

Figura 9: Exemplo da organização dos dados nos vetores aglutinados (*coalesced*)

4.4 Resultados e Análises

Para termos de comparação foram executados os três algoritmos descritos anteriormente. Além disso também foi executada uma versão para a CPU idêntica à versão ingênua para GPU (*CPU+vetor* nas tabelas de resultado). Foram realizadas 10 execuções de cada algoritmo e calculados os tempos médios para cada passo de execução com malhas de tamanho 10x10 (Tabela 1) e 100x100 (Tabela 2)vértices. Cada linha nas Tabelas 1,2mostra os tempos de execução obtidos com cada algoritmo para cada passo, sendo que cada cálculo é feito para a malha no estado inicial (t_0, plano) e no estado final (t, cilindro). Também são mostrados os tempos de cópia dos dados das malhas da CPU para a GPU.

Como podemos ver pela coluna *CPU+half-edge*, o cálculo dos tensores métrico e de curvatura foi o passo que demorou mais tempo para ser executado. Esse é o passo com maior número de acessos à memória de forma não agrupada pois precisa acessar os vértices adjacentes conforme descrito anteriormente.

Os tempos obtidos para a execução algoritmo ingênuo na GPU podem ser vistos na coluna *GPU ingênuo*. Essa implementação manteve os acessos não agrupados à memória quando percorrendo a vizinhança *1-ring* do vértice. Na coluna *CPU+vetor* temos os tempos para a execução de exatamente o mesmo algoritmo na CPU. Podemos ver um ganho de 200% no desempenho da GPU sobre a CPU.

Passos/Algoritmos	CPU+half-edge	CPU+vetor	GPU ingênuo	GPU agrupado
Criação da estrutura de dados	6599.74	9757.39	9803.48	10038
Transmissão CPU->GPU	0	0	2176.17	3860.15
Estimativa das normais	291.745	23.3377	256.493	295.939
Definição das arestas base	145.783	4.23923	102.99	116.823
Estimativa dos tensores métrico e de curvatura	1413.21	253.994	1108.17	212.004
Tempo total gasto com os cálculos	1850.738	281.571	1467.65	624.766
Tempo total gasto (incluindo estrutura de dados)	8450.478	10039	13447.3	14522.9

Tabela 1: Tabela dos tempos de execução, em μs, para uma malha 10x10 vértices

Passos/Algoritmos	CPU+half-edge	CPU+vetor	GPU ingênuo	GPU agrupado
Criação da estrutura de dados	749487	5.01362e+006	5.06538e+006	5.0822e+006
Transmissão CPU->GPU	0	0	4654.8	10967.5
Estimativa das normais	78905.5	2349.35	1255.47	1436.19
Definição das arestas base	36997.4	339.001	139.357	136.189
Estimativa dos tensores métrico e de curvatura	1.09207e+006	25936.2	12600.2	2112.85
Tempo total gasto com os cálculos	1.20797e+006	28624.6	13995.1	3685.23
Tempo total gasto (incluindo estrutura de dados)	1.95746e+006	5.04224e+006	5.08403e+006	5.09686e+006

Tabela 2: Tabela dos tempos de execução, em μs, para uma malha 100x100 vértices

Também vemos um ganho do algoritmo da CPU com vetores sobre o algoritmo usando *half-edge*. Isso se dá porque percorrer a vizinhança de um nó numa estrutura de dados com vários ponteiros e indireções é bem mais custoso que um simples acesso a um vetor.

Por último, o tempo de execução para a versão com o acesso aglutinado à memória global pode ser visto na coluna *GPU agrupado*. Como era esperado, o ganho obtido com essa simples alteração na estrutura de dados e como esses foram distribuídos na memória global foi bastante perceptível (600%).

Logicamente uma solução como essa serve apenas como prova de conceito, não sendo aplicável em uma situação real pois, conforme pode ser visto na Tabela 3, a quantidade de memória utilizada, em comparação com a implementação original, é 232% maior. Dessa forma, faz-se necessária uma investigação de estruturas de dados que podem ser utilizadas na GPU para se solucionar ou, ao menos, contornar o problema de desempenho gerado pelo acesso aleatório à memória global da GPU quando percorrendo a vizinhança *1-ring* dos vértices de uma malha 3D.

Por último precisamos analisar os tempos necessários para a criação das estruturas de dados em cada versão do algoritmo. Como podemos ver na linha *Criação da estrutura de dados* nas Tabelas 1,2, esse tempo é bastante significativo e cresce exponencialmente com o aumento do número de vértices da malha. Porém, no caso da simulação de tecido pela teoria de Cosserat, esse passo precisaria ser executado apenas uma vez em um pré-processamento da malha, ou mesmo carregar a malha diretamente na estrutura de dados final ao invés de linearizar a partir de uma estrutura *half-edge* como é feito atualmente. Isso mitigaria os impactos de desempenho causados por esse passo dos algoritmos.

Algoritmo	Memória utilizada malha 10x10	Memória utilizada malha 100x100
GPU ingênuo	0.0549927	4.80383
GPU agrupado	0.13023	11.1468

Tabela 3: Memória utilizada por cada algoritmo, em mega bytes, para uma malha 100x100 vértices.

5 Conclusões e Perspectivas

Os resultados preliminares obtidos demonstram que, apesar de extremamente paralelizável, o algoritmo proposto por Costa e Wu [4] tem um gargalo de desempenho devido aos acessos aleatórios à memória global por causa da constante necessidade de travessia dos vizinhos *1-ring* de um nó. Através de um simples experimento também demonstramos, conforme esperado, que uma alteração na organização e estruturação dos dados da malha pode gerar ganhos expressivos de desempenho, o que pode proporcionar simulações com taxas iterativas de tempo real para malhas com milhares de vértices.

Apesar dos resultados obtidos serem significativos, eles foram recolhidos como uma prova de conceito para validar a hipótese de que, para alcançar uma simulação de tecidos em tempo real para malhas grandes é necessário um estudo de como mapear uma estrutura *BRep* para GPU de forma otimizada. Para termos uma simulação de tecidos em GPGPU precisamos ainda finalizar a implementação de todos os passos necessários para a simulação de tecidos usando a teoria de superfícies de Cosserat. E para alcançarmos o objetivo de realizar essa simulação em tempo interativo para malhas de grandes dimensões nós pretendemos:

- Aplicar técnicas de otimização para melhorar a ocupação da GPU e tamanho dos *kernels*;
- Implementar diferentes estruturas de dados que mapeiem *BRep* em GPU (incluindo as apresentadas na seção 1.1 de trabalhos relacionados);
- Colher dados de desempenho (tempo de execução, memória utilizada, requisições à memória) para cada uma das estruturas implementadas;
- Fazer uma análise comparativa dos resultados obtidos para escolher a melhor representação de *BRep* em GPU para a teoria de superfícies de Cosserat.

Com esses planos pretendemos alcançar, ao final dos trabalhos, o objetivo de estudar o mapeamento de estruturas *BRep* em GPU e seu impacto na versão massivamente paralela do algoritmo de Costa e Wu.

Referências

[1] Tyler J Alumbaugh and Xiangmin Jiao. Compact Array-Based Mesh Data Structures. In *International Meshing Roundtable*, pages 485–503. Springer Berlin Heidelberg, 2005.

[2] Miklós Bergou, Max Wardetzky, David Harmon, Denis Zorin, and Eitan Grinspun. {A} {Q}uadratic {B}ending {M}odel for {I}nextensible {S}urfaces. In *Symposium on Geometry Processing*, pages 227–230, June 2006.

[3] R Bridson, S Marino, and R Fedkiw. Simulation of Clothing with Folds and Wrinkles. In *Eurographics/SIGGRAPH Symposium on Computer Animation*, 2003.

[4] Matias Rodrigues Costa and Shin-ting Wu. Deformação de Malhas de Arbitrária Topologia. Technical Report 1, University of Campinas, Faculty of Electrical and Computer Engineering, 2012.

[5] Vanio Fragoso de Melo. *Modeling and Controle of Drape and Wrinkles in Deformable Surfaces*. PhD thesis, FEEC/State University of Campinas, 2004.

[6] Leandro de Pinho Monteiro. Numerical solutions to a cloth model based on the Cosserat surface. Master's thesis, School of Electrical and Computer Engineering, State University of Campinas, Campinas, 2008.

[7] Matteo Frigo, Charles E. Leiserson, Harald Prokop, and Sridhar Ramachandran. Cache-oblivious algorithms (extended abstract). In *In Proc. 40th Annual Symposium on Foundations of Computer Science*, pages 285–397. IEEE Computer Society Press, 1999.

[8] A E Green, P M Naghdi, and W L Wainwright. A General Theory of a Cosserat Surface. *Archive for Rational Mechanics and Analysis*, 20:287–308, 1965.

[9] Eitan Grinspun, Anil Hirani, Mathieu Desbrun, and Peter Schröder. Discrete Shells. *Eurographics/SIGGRAPH Symposium on Computer Animation*, pages 62–67, 2003.

[10] Jon Hjelmervik and Jean-Claude Leon. GPU-Accelerated Shape Simplification for Mechanical-Based Applications. In *IEEE International Conference on Shape Modeling and Applications 2007 (SMI '07)*, pages 91–102. IEEE, June 2007.

[11] Feng Ji, Ruqin Li, and Yiping Qiu. Simulate the Dynamic Draping Behavior of Woven and Knitted Fabrics. *Journal of Industrial Textiles*, 35(3):201–215, 2006.

[12] G. Karypis, V. Kumar, and Vipin Kumar. Multilevel k-way partitioning scheme for irregular graphs. *Journal of Parallel and Distributed Computing*, 48:96–129, 1998.

[13] L P Kuptsov. Einstein rule. *Hazewinkel, Michiel, Encyclopaedia of Mathematics*, 2001.

[14] Aaron E Lefohn, Joe Kniss, Robert Strzodka, Shubhabrata Sengupta, and John D Owens. Glift : Generic , Efficient , Random-Access GPU Data Structures. *ACM Transactions on Graphics*, 25(1):1–37, 2006.

[15] J. Lobeiras, M. Amor, and R. Doallo. Influence of memory access patterns to small-scale FFT performance. *The Journal of Supercomputing*, 64(1):120–131, July 2012.

[16] Chris Mcclanahan. History and Evolution of GPU Architecture. Technical report, Georgia Tech - College of Computing, 2010.

[17] NVidia. Cuda C Programming Guide. Technical Report October, NVidia Corporation, 2012.

[18] NVidia. NVidia's Next Generation CUDA Compute Architecture: Kepler GK110. Technical report, NVidia Corporation, 2012.

[19] John D. Owens, David Luebke, Naga Govindaraju, Mark Harris, Jens Krüger, Aaron E. Lefohn, and Timothy J. Purcell. A Survey of General-Purpose Computation on Graphics Hardware. *Computer Graphics Forum*, 26(1):80–113, March 2007.

[20] Xavier Provot. Deformation Constraints in a Mass-Spring Model to Describe Rigid Cloth Behavior. In *Proceedings of Graphics Interface*, pages 147–154, 1995.

[21] S. Rusinkiewicz. Estimating curvatures and their derivatives on triangle meshes. In *Proceedings. 2nd International Symposium on 3D Data Processing, Visualization and Transmission*, pages 486–493. IEEE, 2004.

[22] Le-jeng Shiue, Ian Jones, and Jörg Peters. A realtime GPU subdivision kernel. In *ACM SIGGRAPH 2005 Papers on - SIGGRAPH '05*, pages 1010–1015, New York, New York, USA, 2005. ACM Press.

[23] Demetri Terzopoulos, John C Platt, Alan H Barr, and Kurt Fleischer. Elastically Deformable Models. *Comp. Graph.*, 21(4):205–214, 1987.

[24] Shin-Ting Wu and Vanio Fragoso de Melo. A Deformable Surface Model on the Basis of the Theory of a Cosserat Surface. In *Proceedings of the Computer Graphics and Image Processing, XVII Brazilian Symposium on (SIBGRAPI'04)*, pages 274–281, Curitiba, Brazil, 2004. IEEE Computer Society.

[25] Sung-Eui Yoon and Peter Lindstrom. Mesh layouts for block-based caches. *IEEE transactions on visualization and computer graphics*, 12(5):1213–20, 2006.