UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
CURSO DE CIÊNCIA DA COMPUTAÇÃO

VINÍCIUS BREDA

Detecção Automática de Planos em Nuvens de Pontos Não-Estruturadas Usando Paralelismo Dinâmico em CUDA

Monografia apresentada como requisito parcial para a obtenção do grau de Bacharel em Ciência da Computação

Orientador: Prof. Dr. Manuel Menezes de Oliveira Neto

Porto Alegre
2015

RESUMO

A detecção de regiões planares em nuvem de pontos é um problema recorrente em visão computacional com as mais diversas aplicações práticas, como a calibração automática de câmeras, o reconhecimento de estruturas em cenas 3D, engenharia reversa e realidade aumentada.

O algoritmo de detecção de regiões planares em nuvem de pontos não-organizadas estudado neste trabalho (LIMBERGER; OLIVEIRA, 2015) é uma forma eficiente de resolver este problema em complexidade $O(NlogN)$. O método consiste em três etapas: (1) Subdivisão da nuvem de pontos em regiões aproximadamente coplanares, utilizando uma octree; (2) Esquema de votação para cada nodo definido nesta octree, acumulando os votos em um acumulador esférico; e (3) Detecção de picos no acumulador esférico, definindo as regiões planares de maior importância.

Este trabalho explora o paralelismo de GPUs para tentar melhorar o desempenho do algoritmo definido utilizando a arquitetura CUDA, criada pela NVIDIA, com foco no uso de sua função de Paralelismo Dinâmico. Foram propostas duas abordagens para geração da octree, uma puramente em GPU utilizando o Paralelismo Dinâmico e outra híbrida intercalando processamento entre CPU e GPU, e uma abordagem para a etapa de votação aproveitando o paralelismo nas dimensões do acumulador esférico.

Nos resultados das abordagens de octree foram observadas as dificuldades da utilização de paralelismo em GPU quando existe uma necessidade alta de sincronismo e a indispensabilidade de estruturas de dados melhores adaptadas a arquitetura CUDA. Já na abordagem de votação foi observado ganho de desempenho para nuvens de até um milhão de pontos.

Palavras-chave: Detecção de Regiões Planares em Nuvem de Pontos Não-Organizadas. CUDA. Paralelismo Dinâmico.

Automatic Plane Detection in Unstructured Point Clouds Using Dynamic Parallelism in CUDA

ABSTRACT

Detection of planar regions in point clouds is a recurring problem in computer vision with various practical applications like automatic cameras calibration, structures recognition in 3D scenes, reverse engineering and augmented reality.

The algorithm for detection of planar regions in unorganized point clouds studied in this work (LIMBERGER; OLIVEIRA, 2015) presents a very efficient way of solving this problem in complexity of $O(NlogN)$. The method consists in three stages: (1) A point cloud subdivision in approximately planar regions, using an octree; (2) A voting scheme for every node defined in the octree, accumulating the votes in a spherical accumulator; and (3) A peak detection in the spherical accumulator, finding the most important planar regions.

This work explores the GPU parallelism with the goal to improve the algorithm performance using the CUDA architecture, created by NVIDIA, with focus in its feature of Dynamic Parallelism. Two approaches for the octree generation are proposed, one exclusively on the GPU using the CUDA's Dynamic Parallelism and a second hybrid approach, interleaving between CPU and GPU. Also an approach for the voting procedure is presented, taking advantage of the dimensions of the spherical accumulator.

In the results of both the octree approaches is observed the difficulties in the use of parallelism in GPU when there is a high need for synchronism and the necessity of structures better adapted to the CUDA architecture. However, in the voting approach is observed a performance gain for clouds up to a million points.

Keywords: Detection of planar regions in unorganized point clouds, CUDA, Dynamic Parallelism.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

GPGPU	General-Purpose Computing on Graphics Processing Units
SDRAM	Synchronous Dynamic Random Access Memory
CUDA	Compute Unified Device Architecture
CPU	Central Processing Unit
GPU	Graphics Processing Unit
RAM	Random Access Memory
STL	Standard Template Library

SUMÁRIO

1 INTRODUÇÃO

Detecção de regiões planares em nuvem de pontos é um problema de visão computacional que possui diversas aplicações práticas, como a calibração automática de câmeras, o reconhecimento de estruturas em cenas 3D, engenharia reversa, e realidade aumentada. É portanto essencial solucionar este problema da maneira mais otimizada possível, explorando todas as possibilidades de computação que o hardware atual oferece.

O algoritmo estudado neste trabalho (LIMBERGER; OLIVEIRA, 2015) é uma forma muito eficiente de detectar regiões planares em nuvens de pontos utilizando a transformada de Hough (HOUGH, 1962), com complexidade $O(nlogn)$. Ele generaliza para três dimensões um esquema de votação criado para detecção de linhas em tempo real (FERNANDES; OLIVEIRA, 2008).

O método consiste basicamente na execução de três passos: (1) Subdivisão da nuvem de pontos em regiões aproximadamente coplanares, utilizando uma Octree; (2) Utilização de um esquema de votação eficiente para cada nodo aproximadamente coplanar desta Octree, acumulando os votos em um acumulador esférico; e (3) Detecção de picos no acumulador esférico de votação, definindo as regiões planares de maior importância.

Diversos aspectos do algoritmo estudado demonstram potencial de processamento em paralelo, como por exemplo a detecção de estruturas aproximadamente coplanares em nuvens de pontos massivas, e o processo de votação. O uso do paralelismo das GPUs pode, potencialmente, levar a um melhor desempenho do que o processamento serial da nuvem de pontos na CPU. Como forma de facilitar o desenvolvimento de aplicações que explorem o uso desse paralelismo, a NVIDIA criou o CUDA (NVIDIA, 2015a), um modelo de arquitetura e programação para GPUs. Em sua arquitetura Kepler (NVIDIA, 2012) foi introduzido um novo conceito em CUDA chamado de Paralelismo Dinâmico, que criou novas possibilidades para a programação em GPU permitindo a chamada de kernels recursivos em CUDA.

1.1 Objetivos

O objetivo deste trabalho é explorar o paralelismo de GPUs para tentar melhorar o desempenho do algoritmo definido por Limberger e Oliveira (LIMBERGER; OLIVEIRA, 2015). Para isso é utilizada a arquitetura CUDA com foco no uso de sua função de *Paralelismo Dinâmico*.

1.2 Estrutura do Trabalho

Este trabalho é organizado do seguinte modo: o Capítulo 2 detalha do algoritmo de detecção de regiões planares em nuvens de pontos não-organizadas de Limberger e Oliveira (LIMBERGER; OLIVEIRA, 2015), discutindo suas estruturas de dados e métodos utilizados. O Capítulo 3 descreve o modelo de paralelismo em CUDA, mostrando sua arquitetura de memória e características de programação. O Capítulo 4 apresenta os detalhes do algoritmo paralelo criado, explicando as abordagens realizadas, bem como suas vantagens e desvantagens. O Capítulo 5 apresenta os resultados obtidos com a implementação realizada. Por fim, o Capítulo 6 apresenta as conclusões do trabalho e discute possibilidades de trabalhos futuros.

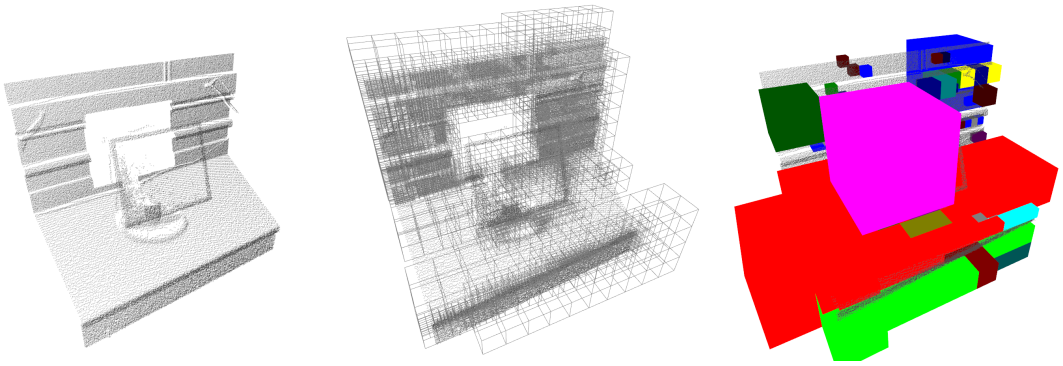
2 ALGORITMO DE DETECÇÃO DE REGIÕES PLANARES

Este capítulo apresenta uma breve descrição da técnica de detecção de estruturas aproximadamente co-planares em nuvens de pontos não-estruturadas (LIMBERGER; OLIVEIRA, 2015) que constitui o foco deste trabalho. O algoritmo detecta planos utilizando a transformada de Hough (HOUGH, 1962) de forma eficiente e determinística, com complexidade $O(N \log N)$ no número de pontos da nuvem. Serão discutidos os três passos do algoritmo. O primeiro passo consiste na subdivisão do espaço em amostras aproximadamente coplanares utilizando uma Octree. O segundo passo refere-se ao processo de votação em um acumulador esférico. Já o terceiro passo realiza a detecção de picos no acumulador de votos, determinando a localização e orientação das regiões co-planares.

2.1 Geração da Octree

O primeiro passo do algoritmo subdivide a nuvem de pontos utilizando uma octree, na qual os nodos são refinados de modo a manter somente amostras aproximadamente coplanares. Com isso, pode-se acelerar significativamente o processo de votação da transformada de Hough. Um exemplo desta subdivisão pode ser observado na Figura 2.1.

Figura 2.1 – Subdivisão da nuvem de pontos utilizando uma octree. Nuvem de pontos de entrada (esquerda). Estágio intermediário de subdivisão da octree (centro). Nodos da octree contendo elementos aproximadamente co-planares resultantes (direita).



A geração da octree inicia com um nodo raiz que contém toda a nuvem de pontos e é recursivamente subdividido em oito nodos filhos. Em cada nodo que deve ser subdividido, o algoritmo verifica se os pontos existentes naquele nodo são aproximadamente coplanares. Em caso afirmativo, o nodo não é mais subdividido. Isso permite menor quantidade de amostras a serem consideradas pelo passo de votação e melhor desempenho na execução do algoritmo.

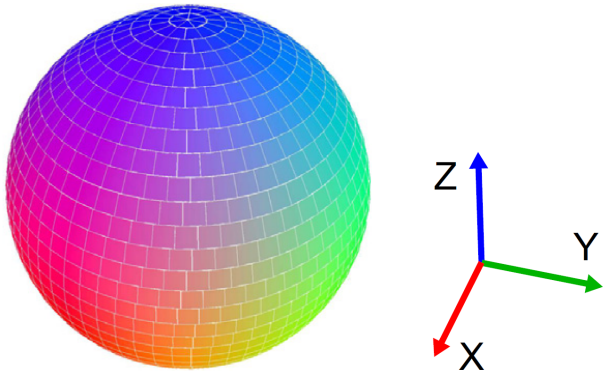
O procedimento para verificar se os pontos de um nodo são aproximadamente coplanares consiste em calcular a matriz de covariância dos pontos da nuvem e, a partir desta, seus autovetores e autovalores. Os autovalores representam as variâncias das amostras em cada direção (X, Y e Z), e são utilizados para testar a coplanaridade dos pontos, verificando se os clusters de pontos satisfazem um valor de espessura máxima e isotropia mínima.

No procedimento de divisão da octree, os pontos referentes ao nodo pai são distribuídos para seus nodos filhos através da verificação de sua posição espacial, calculando também os seus centroides. Após a inicialização dos nodos filhos, o nodo pai pode chamar a função de subdivisão recursivamente para cada um deles.

2.2 Votação no Acumulador Esférico

Para cada nodo da octree, sua correspondente matriz de covariância $\Sigma_{(X,Y,Z)}$ expressa em coordenadas Cartesianas (X,Y,Z) é convertida para coordenadas esféricas (θ, ϕ, ρ), onde $\theta \in [0°, 360°)$, $\phi \in [0°, 180°]$ e $\rho \in \mathbb{R}_{>0}$. Esta conversão de sistema de coordenadas é obtida como $\Sigma_{(\theta,\phi,\rho)} = J\Sigma_{(X,Y,Z)}J^T$, onde J é a matriz Jacobiana da transformação. Com isso é possível distribuir os votos em um acumulador esférico (Figura 2.2) utilizando um kernel Gaussiano trivariado expresso em coordenadas esféricas. Para assegurar 95.4% de certeza de que existe um plano na posição e orientação estimadas, votos são depositados no acumulador em células até dois desvios padrões do centro do kernel Gaussiano.

Figura 2.2 – Representação do acumulador esférico para uma dada coordenada ρ onde as cores representam as direções das normais.

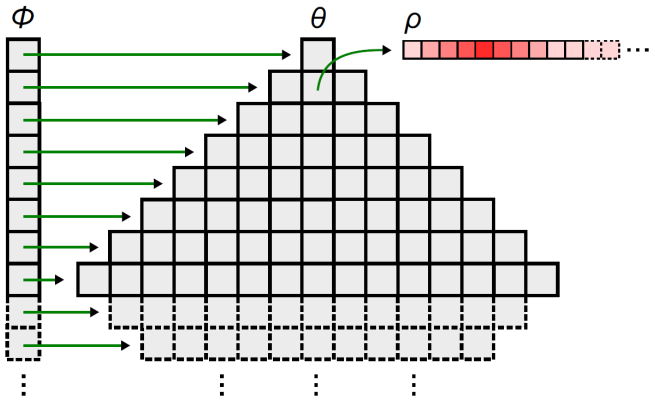


Fonte: (LIMBERGER; OLIVEIRA, 2015)

O acumulador esférico utiliza discretizações das coordenadas (θ, ϕ, ρ) para criar uma estrutura de dados onde os votos serão acumulados em células referente as suas direções. Sim-

plesmente armazenar um array de três dimensões completo para representar este acumulador é ineficiente, visto que existe uma menor quantidade de célula nas regiões dos polos do acumulador do que no centro. Para compensar este fato, o algoritmo utiliza uma representação definida por Borrman e outros (BORRMANN et al., 2011), que define uma estrutura cujas dimensões variam com o ângulo ϕ do acumulador (Figura 2.3).

Figura 2.3 – Estrutura de dados do acumulador esférico indexado por (ϕ, θ, ρ)



Fonte: (LIMBERGER; OLIVEIRA, 2015)

2.3 Detecção de Picos

Durante a votação, o algoritmo armazena em um vetor auxiliar as posições das células do acumulador que receberam votos, gerando uma lista de possíveis picos de votação. Com isso, o último passo do algoritmo consiste em utilizar este vetor auxiliar para detectar os picos armazenados no acumulador esférico. Um filtro passa-baixas é aplicado às células do acumulador representadas no vetor auxiliar, suavizando os picos detectados e evitando a detecção de picos espúrios. Este processo é análogo ao utilizado no algoritmo de detecção de linhas de Fernandes e Oliveira (FERNANDES; OLIVEIRA, 2008). Após a filtragem, o vetor auxiliar é classificado em ordem decrescente do número de votos. Este vetor é então percorrido e, cada célula ainda não visitada, é registrada como pico, marcando-se a célula e suas vizinhas como visitadas. Durante o percorrimento do vetor, caso uma célula vizinha da atual (no acumulador) já tenho sido visitada, a célula atual é marcada como visitada. Este procedimento garante que somente os picos que verdadeiramente representem planos sejam detectados.

2.4 Resumo

Este capítulo apresentou os detalhes do algoritmo de Limberger e Oliveira (LIMBERGER; OLIVEIRA, 2015) para detecção de estruturas aproximadamente planares em nuvens de pontos não estruturadas. Tal algoritmo é a base do trabalho desenvolvido nesta monografia.

3 PARALELISMO EM GPU

Este capítulo discute o funcionamento da arquitetura de CUDA, modelo de programação de propósito geral em GPU (GPGPU) da NVIDIA. São cobertos os aspectos de programação e estrutura de memória, bem como suas vantagens para implementação do algoritmo estudado. Também realiza-se uma discussão sobre a funcionalidade de Paralelismo Dinâmico introduzida na arquitetura Kepler.

3.1 Definição de CUDA

Mesmo com os comprovados benefícios da GPGPU para solução de problemas e aceleração de algoritmos, a programação em GPUs antes do surgimento de CUDA era excessivamente complicada. Os dados precisavam ser representados como primitivas geométricas e objetos de textura, diminuindo assim a produtividade dos programadores e dificultando a sua adoção pela indústria.

Como forma de facilitar o uso destes recursos computacionais, a NVIDIA em 2006 introduziu CUDA, um modelo de arquitetura e programação utilizado em suas GPUs. Este modelo foi criado de modo com que o programador possa rodar seu código aproveitando ao máximo a vazão dos multiprocessadores disponíveis na GPU. Para que o programador possa usufruir deste modelo, também foi definido uma plataforma com diretivas na linguagem C, criando um ambiente para programação em mais alto nível. Apesar da linguagem C ter sido definida como padrão pelos criadores da arquitetura, diversas linguagens foram estendidas para utilização do sistema, como C++, Fortran, e Python, por exemplo.

3.2 Arquitetura CUDA

Na arquitetura CUDA, um **kernel** é definido como uma função padrão em C, usando o declarador **__global__**, onde seu código será executado N vezes em paralelo por N threads diferentes. O número de threads e blocos de threads que será executado por um kernel deve ser definido no momento da chamada da função (NVIDIA, 2015a). A Figura 3.1 mostra um exemplo definição e chamada de um kernel de multiplicação de um array por uma constante.

As threads que executam um kernel em CUDA são definidas por índices em até 3 dimensões, e são organizadas em blocos de threads. Os blocos por sua vez também são indexados

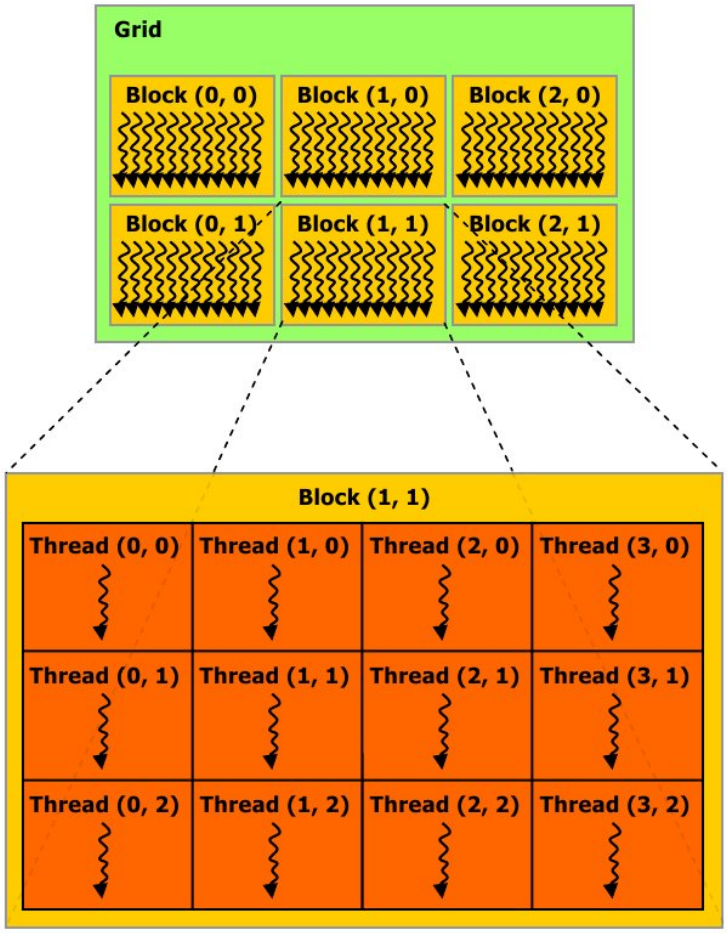
Figura 3.1 – Exemplo de definição e chamada de kernel de multiplicação de array por constante em CUDA.

```
// Definição de kernel de multiplicação de array por constante
__global__ void kernelMult(float* A, float B)
{
    int i = threadIdx.x + blockIdx.x * blockDim.x; // Define indice baseado em indice da thread e do seu bloco.
    A[i] = A[i] * B; // Multiplica valores por constante em paralelo.
}

int main()
{
    const int S = 10240; //Tamanho do array.
    float h_a[S], h_out[S]; //Definição dos arrays de entrada e saída na CPU.
    float *d_a; //Definição de ponteiro de array na GPU.
    float val = 5; //Valor que multiplicará o array.

    //Define valores randomicos no array inicial.
    for (int i = 0; i < S; ++i) h_a[i] = (float)rand()/RAND_MAX;

    cudaMalloc(&d_a, sizeof(float)*S); //Aloca array na GPU.
    cudaMemcpy(d_a, h_a, sizeof(float)*S, cudaMemcpyHostToDevice); //Copia da memória da CPU para memória global da GPU.

    // Chamada de Kernel com 10 blocos e 1024 threads.
    kernelMult <<< 10, 1024 >>>(d_a, val);

    cudaMemcpy(h_out, d_a, sizeof(float)*S, cudaMemcpyDeviceToHost); //Copia da memória da GPU para a CPU.

    cudaFree(d_a); //Libera memória alocada na GPU.
}
```

no mesmo sistema de indices em até 3 dimensões e são definidos como um grid, onde cada bloco de threads é executado em um único multiprocessador, dividindo portanto os recursos de memória compartilhada disponíveis. Devido a limitação de recursos disponíveis em cada multiprocessador da GPU, o número máximo de threads por bloco é definido como 1024 (NVIDIA, 2015a). A indexação de threads e blocos permite com que o kernel possa identificar qual thread está executando em determinado momento e realizar suas funções com base nisso. Na Figura 3.2 pode-se observar graficamente como as threads são organizadas.

Numa visão de nível arquitetônico existe a definição de *warp*, que representa grupos de 32 threads paralelas que são gerenciadas em conjunto por um multiprocessador. As threads de um mesmo *warp* executam em conjunto as mesmas instruções em paralelo, e por isso, a arquitetura alcança maior eficiência se as threads de um mesmo *warp* seguem um mesmo fluxo de execução.

Figura 3.2 – Hierarquia de threads em CUDA (indexadas em duas dimensões).



Fonte: CUDA C Programming Guide - NVIDIA

3.2.1 Organização de Memória

A hierarquia de memória em CUDA define diversas camadas em que os dados podem ser armazenados e acessados por uma thread. Cada thread possui uma **memória local** privada armazenada em SDRAM, mas com utilização da cache L1 tem-se o acesso rápido em casos de usos repetitivos de dados.

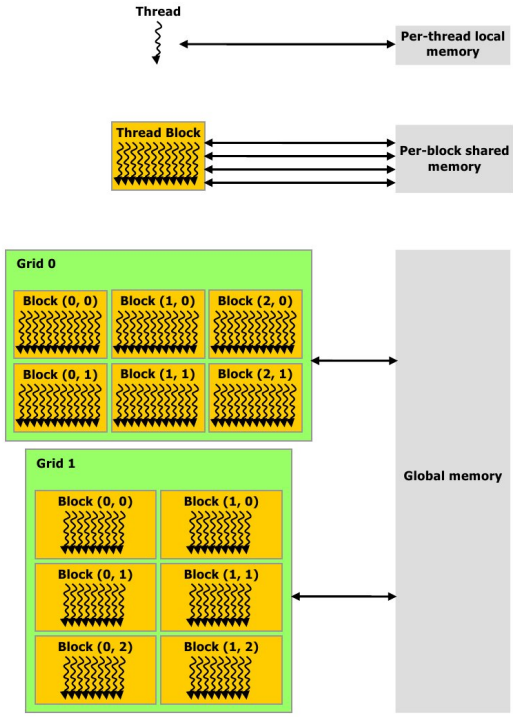
Os blocos de threads por sua vez também possuem uma memória privada chamada de **memória compartilhada**, pois pode ser acessada por qualquer thread do bloco. A memória compartilhada é pequena (máximo de 64kB) porém extremamente rápida e portanto essencial para algoritmos que precisem de comunicação e sincronismo entre threads de um block, como algoritmos que utilizem redução, por exemplo.

No nível mais alto da hierarquia de memória fica a **memória global** onde estão locali-

zados dados que podem ser acessados por qualquer thread executando um kernel, independente de seu bloco. É na memória global em que é realizada a troca de dados entre a CPU e a GPU (NVIDIA, 2015a).

A Figura 3.3 define como são organizados os componentes principais da hierarquia de memória em CUDA.

Figura 3.3 – Hierarquia de memória em CUDA.



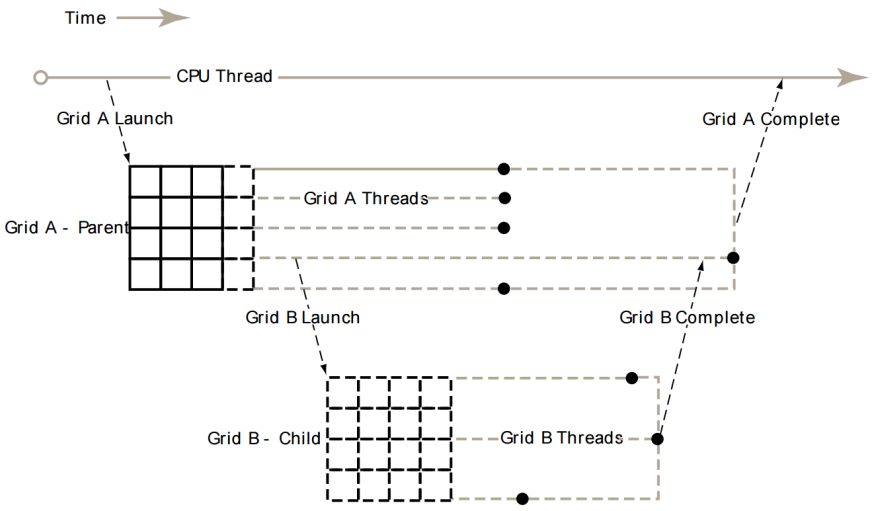
Fonte: CUDA C Programming Guide - NVIDIA

3.3 Paralelismo Dinâmico

Até a arquitetura Kepler, CUDA possuía a limitação de que só era possível lançar um kernel diretamente da CPU, impossibilitando encadeamento entre novos kernels. O conceito de **paralelismo dinâmico** (NVIDIA, 2012) foi introduzido com a arquitetura Kepler e permite a GPU gerar novos kernels sem necessidade da intervenção da CPU. No paralelismo dinâmico, a chamada de kernels internamente na GPU é similar a realizada na CPU, tornando a programação mais simples e suportando uma gama maior de algoritmos possíveis de serem implementados. Ao criar um kernel filho, o kernel pai automaticamente espera pela sua execução (Figura 3.4), criando também uma possibilidade de sincronismo.

A principal vantagem de um algoritmo recursivo utilizar este novo tipo de abordagem é a possibilidade de adaptar melhor a GPU aos dados sem intervenção da CPU e definir melhor seu nível de paralelismo, evitando trabalho desnecessário.

Figura 3.4 – Representação de uma chamada de kernel com Paralelismo Dinâmico. A CPU cria um kernel pai, e uma de suas threads cria um kernel filho. No final da execução, o kernel filho retorna ao kernel pai, que por sua vez retorna a CPU.



Fonte: CUDA C Programming Guide - NVIDIA

3.4 Biblioteca Thrust

Thrust é uma biblioteca de templates baseada na STL do C++ e permite utilizar funções de mais alto nível em uma implementação em CUDA. A alocação de memória é facilitada através de primitivas de vetores similares aos da STL, simplificando a troca de dados entre CPU e GPU. Além disso, a biblioteca Thrust disponibiliza algoritmos padrões em paralelo, como *sort* e *reduce*, permitindo uma implementação mais rápida de algoritmos mais complexos (NVIDIA, 2015c). Neste trabalho a biblioteca Thrust foi utilizada para facilitar a alocação de vetores na GPU e para a realização dos ordenamentos necessários nas abordagens de octree criadas.

3.5 Resumo

Este capítulo apresentou os prinicipais conceitos de CUDA necessários ao entendimento do trabalho desenvolvido nesta monografia.

4 PARALELIZAÇÃO DO ALGORITMO DE DETECÇÃO DE PLANOS

O algoritmo descrito no Capítulo 2 possui várias etapas que demonstram potencial de paralelismo a ser explorado, como subdivisão do espaço utilizando uma Octree e a votação no acumulador esférico. Este capítulo descreve a abordagem utilizada para paralelizar cada uma das etapas do algoritmo em CUDA, buscando analisar vantagens e desvantagens encontradas na implementação.

4.1 Geração de Octree para Subdivisão do Espaço em Regiões Planares

A geração da octree separando aglomerações de pontos coplanares foi identificado como o principal gargalo para a execução do algoritmo original, principalmente em nuvens de pontos massivas (LIMBERGER; OLIVEIRA, 2015). Por esse motivo, este processo foi considerado como ponto principal para ser otimizado via paralelismo em GPU. Para esta monografia, foram realizadas duas implementações do processo de geração de octree: uma buscando explorar o paralelismo completo em GPU utilizando paralelismo dinâmico, e outra procurando misturar as vantagens de uma computação híbrida entre CPU e GPU. Ambas as implementações possuem similaridades no sistema de alocação de memória, mas possuem grandes diferenças na questão de sincronismo e comunicação de dados.

4.1.1 Alocação de Memória

Os algoritmos para geração da octree recebem como entrada um vetor com as coordenadas Cartesianas (x, y, z) dos pontos da nuvem não-estruturada. São então realizadas operações simples de alocação de memória global na GPU (vetor P de pontos), seguida de cópia dos valores da CPU para GPU. A octree resultante é representada utilizando um vetor N de estruturas de nodos de tamanho fixo e pré-alocado. A quantidade de nodos de uma octree é definida pela expressão $(8^d - 1)/7$, onde d é a profundidade máxima da octree. Cada elemento deste vetor define os valores relevantes para um nodo da octree: número de pontos, bounding box, centroide, matriz de covariância, autovalores, autovetores, além de um valor booleano indicando a coplanaridade ou não das amostras contidas no nodo.

Para criar uma relação entre pontos e nodos da octree, utiliza-se um vetor T de tuplas (p, n), onde p é o índice do ponto no vetor de pontos P, e n é o índice do nodo no vetor de nodos

N. O vetor T permite a identificação de todos os pontos contidos em um dado nodo da octree a partir de um índice para um ponto inicial e o do número de pontos contidos no nodo. Porém, este modelo de estrutura de dados necessita que sejam realizados ordenamentos em trechos do vetor de modo em que os pontos de cada nodo estejam sempre linearmente distribuídos.

Um problema enfrentado na utilização de CUDA para esta etapa foi a necessidade de pré-alocação destes vetores. Em CPU, tais vetores podem ser alocados dinamicamente sem maiores problemas, por causa da serialidade de acesso a memória, que permite operações de *push* e *pop* em vetores. Em CUDA, este tipo de operação é contraproducente e deve ser evitado ao máximo.

4.1.2 Geração da Octree Utilizando Paralelismo Dinâmico

Dada a natureza recursiva do processo de criação de uma octree, este possui características que podem ser exploradas com a utilização de paralelismo dinâmico. O pacote de exemplos de CUDA na arquitetura Kepler (NVIDIA, 2015b) apresenta uma implementação de quadtree simples utilizando este novo recurso. Este exemplo foi utilizado como base teórica para esta solução, visto a semelhança entre os algoritmos.

Nesta abordagem criamos um único kernel de CUDA para computar a octree que será chamado recursivamente. Ele recebe como entrada um valor de profundidade d, um índice de nodo n, ponteiros para os vetores globais pré-alocados, um nível s de início para verificação de coplanaridade, e parâmetros auxiliares, como espessura máxima de um plano, isotropia mínima, e tamanho do vetor de pontos P, por exemplo. As threads e blocos foram utilizadas para subdividir os pontos contidos no nodo atual n, onde cada thread pode ser responsável por processar uma ou mais coordenadas de pontos, dependendo da quantidade de dados.

4.1.2.1 Cálculo da Matriz de Covariância dos Pontos de um Nodo

O refinamento de um nodo da octree depende do resultado do teste de coplanaridade das pontos nele contido. Este resultado, por sua vez, depende dos autovalores da matriz de covariância definida pelas coordenadas Cartesianas destes pontos no espaço. Assim, o primeiro passo é calcular a matriz de covariância do nodo atual n. Para cada ponto processado p_i, do nodo n, é calculada em paralelo a sua contribuição $Cov_{p_i}(n)$ para a matriz de covariância $Cov(n)$ do nodo:

$$Cov_{p_i}(n) = \begin{bmatrix} (p_{ix} - c_{nx})^2 & (p_{ix} - c_{nx})(p_{iy} - c_{ny}) & (p_{ix} - c_{nx})(p_{iz} - c_{nz}) \\ & (p_{iy} - c_{ny})^2 & (p_{iy} - c_{ny})(p_{iz} - c_{nz}) \\ & & (p_{iz} - c_{nz})^2 \end{bmatrix},$$

onde c_n é o centroide de n, e $p_{i.x}$, $p_{i.y}$, $p_{i.z}$, $c_{i.x}$, $c_{i.y}$ e $c_{i.z}$ são, respectivamente, as coordenadas x, y, e z de p_i e de c_n.

Neste momento é necessário utilizar uma barreira para sincronizar a computação das várias matrizes $Cov_{p_i}(n)$ relativas a cada um dos pontos p_i do nodo n. Estas são então somadas e a matriz resultante é processada serialmente dividindo-se cada um de seus elementos pelo número de pontos contidos no nodo n, obtendo-se assim a matriz $Cov(n)$. Neste ponto são calculados e armazenados os autovalores e autovetores de $Cov(n)$.

Através dos autovalores computados é possível verificar a "espessura" e grau de isotropia da nuvem de pontos no nodo n e decidir se estes são, pelo menos, aproximadamente coplanares. Em caso afirmativo, não há mais necessidade de subdividir este nodo da octree, e o kernel termina sua computação.

4.1.2.2 Geração dos Nodos Filhos

O segundo passo trata da criação dos nodos filhos de n, calculando os seus centroides e sua subdivisão de pontos. O processo inicia verificando, em paralelo, para cada ponto p_i a qual octante definido pelos nodos filhos ele pertence. Esta computação é feita de maneira eficiente, calculando-se a diferença entre as coordenadas de p_i e as do centro do bounding box de n, e utilizando o sinal das coordenadas geradas para criar um número de 3 bits, representando a indexação dos nodos filhos. Neste momento, utiliza-se uma barreira para sincronizar os blocos de threads de modo a que os valores sejam computados corretamente antes que se possa verificar a necessidade de subdivisão dos nodos filhos.

Para cada um dos nodos filhos m é realizado um ordenamento paralelo no vetor auxiliar da relação entre índices e pontos, para manter os pontos de um mesmo nodo sequencialmente distribuídos. Além disso, é realizado o cálculo do centroide de m através da média dos seus pontos, somados anteriormente. Por fim, cada um dos nodos filhos de n realiza uma chamada recursiva de kernel, utilizando o recurso de paralelismo dinâmico, e modificando os valores de profundidade e nodo atual.

4.1.2.3 Sincronismo Entre Blocos

Em dois pontos do algoritmo descrito foi citada a necessidade de sincronismo entre as threads para evitar acesso a dados ainda não computados. O modelo de programação CUDA fornece algumas primitivas de sincronismo como a função *__syncthreads()*. Porém, estas primitivas fornecidas somente permitem sincronismo interno a um bloco de threads e no algoritmo existe uma necessidade de conter muitos blocos, devido a limitação da arquitetura de 1024 threads por bloco.

Para realizar o sincronismo, foi utilizado um algoritmo para criação de barreiras (XIAO; FENG, 2010) entre blocos de threads de uma maneira eficiente utilizando as primitivas fornecidas e variáveis globais auxiliares. Este tipo de sincronismo, porém, possui a desvantagem de só poder ser realizado se o número de blocos executados em um kernel for menor ou igual ao numero de multiprocessadores disponível na GPU. Esta restrição limita o número de threads que podem ser executadas e consequentemente a quantidade de memória compartilhada disponível.

4.1.3 Geração da Octree Utilizando uma Abordagem Híbrida

Devido aos problemas encontrados na criação da octree utilizando paralelismo dinâmico, uma abordagem alternativa foi proposta e implementada. Esta abordagem, aqui chamada de *híbrida*, busca explorar um melhor uso conjunto dos recursos de CPU e GPU. Ela busca utilizar a computação em GPU somente nos momentos de alto processamento paralelo, deixando partes essencialmente seriais para serem executadas com a baixa latência da CPU.

Os dois pontos de alto processamento concorrente no algoritmo anterior são as somas das matrizes de covariâncias $Cov_{p_i}(n)$ do nodo atual n, e a separação dos pontos em seus nodos filhos. Para realizar estas tarefas na abordagem híbrida foram criados dois kernels que utilizam reduções para minimizar os conflitos de acesso a memória na GPU.

4.1.3.1 Redução em Memória Compartilhada

A redução em memória compartilhada consiste em realizar operações cumulativas em endereços de memória compartilhada distintos e depois, utilizando a indexação de threads, combinar os elementos dois a dois, até que eles sejam reduzidos a um valor final. Para realizar o cálculo da matriz de covariância e centroides dos nodos filhos foram criados dois kernels utilizando um algoritmo de redução otimizada com memória compartilhada (HARRIS et al., 2007), que busca diminuir ao máximo os conflitos de leitura e escrita no algoritmo padrão de redução

utilizando diversas otimizações de acesso a memória e organização de threads.

4.1.3.2 Cálculo da Matriz de Covariância e Teste de Coplanaridade

Para o cálculo da matriz de covariância foi utilizado um kernel de redução que utiliza um vetor de matrizes de covariância em memória compartilhada com tamanho dado pelo número de threads por bloco. Isto permite que as threads consigam acessar separadamente os valores de memória compartilhada, evitando conflitos. Cada thread é responsável por um ou mais pontos da nuvem de entrada que são utilizados em cada uma das matrizes armazenadas em memória compartilhada. Cada bloco de threads então é reduzido até ser obtido uma matriz de covariância por bloco, que serão acumuladas em memória global. As operações seriais de cálculo de autovalores, autovetores, e teste para verificar se o nodo é aproximadamente coplanar são realizadas na CPU.

4.1.3.3 Cálculo dos Centroides

O cálculo de centroides e número de pontos dos filhos de um nodo da octree também ocorre utilizando o algoritmo de redução descrito anteriormente. Neste caso, é preciso armazenar em memória compartilhada os acumuladores de centroides e número de pontos para cada um dos oito filhos do nodo processado. Esta necessidade de ter oito acumuladores diminui o número de threads por bloco, pois a memória compartilhada possui um limite atual de 64 kB por bloco.

De modo similar ao cálculo das matrizes de covariância, cada thread processa um ou mais pontos e decide em qual dos oito endereços de nodos filhos em memória compartilhada deve ser acumulado o valor para o centroide e somado o número de pontos. Os blocos são reduzidos acumulando os valores de cada nodo filho e somente um vetor com oito valores de centroide e oito contagens. Uma thread de cada bloco então soma estes valores em memória global e o kernel finaliza sua execução deixando os valores disponíveis para a CPU.

4.1.3.4 Subdivisão dos Nodos Filhos

Após os cálculos de centroides é realizado um ordenamento no trecho do vetor T (que associa pontos da nuvem a nodos da octree) referentes ao nodo atual. Utiliza-se a biblioteca Thrust (NVIDIA, 2015c) para realizar este ordenamento de forma otimizada. Por fim é realizada uma chamada recursiva para cada um dos oito nodos filhos.

4.2 Votação

A etapa de votação recebe como entrada um vetor com os parâmetros dos kernels Gaussianas trivariados obtidos a partir do processamento dos nodos da octree contendo conjuntos de pontos (aproximadamente) coplanares. Cada elemento deste vetor armazena a normal, a binormal e a matriz de covariância em coordenadas esféricas associadas a um nodo. Cada kernel deposita seus votos no acumulador. Esta etapa de votação é realizada em paralelo utilizando processamento em GPU (Seção 4.2.2).

4.2.1 Alocação de Memória e Estrutura de Dados

A representação dos elipsoides que serão processados é feita alocando na GPU um vetor com estruturas de dados que armazenam a matriz de covariância em coordenadas esféricas $\Sigma_\theta\phi\rho$, o índice do nodo na Octree n, a direção inicial de votação $(\theta_s, \phi_s, \rho_s)$ definida pelo menor autovetor de n e um grau de representatividade r. O tamanho deste vetor é definido pela quantidade de nodos marcados como colineares na geração da Octree.

O acumulador esférico em CPU é representado por uma matriz de três dimensões de estruturas de células, onde a dimensão de θ é alocada dinamicamente dependendo do índice ϕ. Realizar alocações dinâmicas e em múltiplas dimensões pode prejudicar o desempenho em CUDA, e portanto é pouco recomendado. Para representar o acumulador esférico em CUDA, realiza-se a linearização da matriz 3D em um vetor 1D. O tamanho do vetor do acumulador é calculado por $tam = \phi_{max}\theta_{max}\rho_{max}$, onde ϕ_{max},θ_{max} e ρ_{max} são os tamanhos máximos de cada dimensão na matriz 3D.

Cada célula do acumulador deve armazenar os nodos que votaram naquela direção, criando a necessidade de outro vetor dinâmico. A solução neste caso foi a alocação fixa de um vetor com número de elipsoides, que armazena o valor do voto do elipsoide. Consequentemente, este vetor também é utilizado para dizer se um certo elipsoide votou ou não em determinada célula do acumulador.

Por fim, um vetor auxiliar é criado para armazenar o primeiro voto em cada uma das células do acumulador, que posteriormente será utilizado para detecção de picos. Este vetor também é linearizado como o acumulador e possui armazenado os valores de θ, ϕ, ρ e valor do voto realizado.

4.2.2 Kernel de Votação

O kernel CUDA de votação irá contabilizar os votos no acumulador para cada Gaussiana trivariada em paralelo. Um kernel Gaussiano será avaliado por um certo número de blocos de threads, onde cada thread contabilizará os votos em uma direção (ϕ, θ), variando somente o raio ρ. Cada bloco de threads então será responsável por contabilizar votos para uma Gaussiana em uma parte do acumulador esférico. A direção (ϕ,θ) a ser votada por uma thread é definida através do índice da thread em relação ao kernel Gaussiano de seu bloco. Para cada valor de raio ρ, a thread calcula um valor de voto utilizando a distribuição Gaussiana trivariada, e caso este voto ultrapasse um valor mínimo ele é contabilizado na célula. O índice da célula que deve contabilizar o voto é definido por $i_c = \rho + \rho_{dim}(\theta + \theta_{dim} \times \phi)$, onde ρ_{dim} e θ_{dim} são as dimensões de ρ e θ, e (ϕ, θ, ρ) são os índices calculados pela thread atual. Este procedimento pode ser melhor observado em pseudocódigo no Algoritmo 1.

Para computar o voto (Algoritmo 2), a thread verifica se a célula já foi votada pela Gaussiana. Em caso negativo, a thread armazena atomicamente o seu voto no vetor de Gaussianas da célula e realiza uma soma atômica dos votos na célula do acumulador. Neste momento também é verificado se a célula já recebeu algum voto, e caso não tenha recebido o a thread armazena no vetor auxiliar de possíveis picos a posição e o valor do voto, marcando a célula como já votada. No caso da Gaussiana já ter votado nesta célula, é armazenado o maior valor entre o voto atual e o antigo, atualizando também o acumulador de votos da célula.

Algoritmo 1 Votação

```
1:  função VOTACAO(vetorKernels, vetorCelulas, vetorPicos)
2:      //Define os valores de Kernel Gaussiano, φ e θ com base nos índices de bloco e thread.
3:      kernel = defineKernelGaussiano(vetorKernels, indice_bloco);
4:      φ = definePhi(indice_thread);
5:      θ = defineTheta(indice_thread, phi);
6:
7:      //Realiza votação para os índices ρ
8:      para ρ ← kernel.ρ_min até kernel.ρ_max faça
9:          //Computa voto através da Gaussiana.
10:         voto = kernel.valorGaussiana(phi, theta, rho);
11:         se voto > kernel.votoMinimo então
12:             //Contabiliza voto na célula.
13:             ContabilizaVoto(voto, φ, θ, ρ, kernel, vetorCelulas, vetorPicos);
14:         fim se
15:     fim para
16: fim função
```

O número de blocos do kernel de votação pode ser definido pela equação $N_G\lceil(\frac{\theta_{dim}\phi_{dim}}{\omega})\rceil$,

Algoritmo 2 Contabiliza um voto em determinada célula (ϕ, θ, ρ) do acumulador

```
1: função CONTABILIZAVOTO(voto, φ, θ, ρ, kernelGaussiano, vetorCelulas, vetorPicos)
2:     i_c = ρ + ρ_dim(θ + θ_dim × φ); //Lineariza posição da célula do acumulador.
3:     celula = vetorCelulas[i_c]; //Célula atual.
4:
5:     //Verifica se kernel não votou na célula.
6:     se celula.kernels[kernelGaussiano] == 0 então
7:         atômico celula.kernels[kernelGaussiano] = voto; //Armazena voto do kernel.
8:         celula.acumulador += voto; //Acumula voto na célula.
9:         se celula não votou então
10:            atômico celula.votou = true; //Define que célula já votou.
11:            vetorPicos[i_c] = (θ, φ, ρ, voto); //Armazena no vetor de picos.
12:        fim se
13:    senão //Se nodo já votou.
14:        voto_antigo = celula.kernels[kernelGaussiano]; //Voto atual.
15:        se voto > celula.kernels[kernelGaussiano] então //Se novo voto maior que atual.
16:            atômico celula.kernels[kernelGaussiano] = voto; //Armazena voto do kernel.
17:            celula.acumulador += voto - voto_antigo; //Troca o voto no acumulador.
18:        fim se
19:    fim se
20: fim função
```

onde N_G é o número de Gaussianas que devem ser processados, θ_{dim} e ϕ_{dim} são as dimensões do acumulador nas direções θ e ϕ, e ω é a quantidade de threads por bloco, limitada por $\omega < 1024$.

4.3 Detecção de Picos

A paralelização eficiente do processo de detecção de picos se mostrou uma tarefa não trivial para ser realizada em tempo para este trabalho, devido a dependência de dados existente entre as células do vetor de possíveis picos. Por esse motivo, esta etapa foi mantida serialmente CPU. Uma investigação sobre formas eficientes de paralelizar esta etapa é deixada como sugestão para trabalhos futuros. O objetivo principal da detecção de picos é realizar uma ordenação do vetor de células que receberam votos e selecionar os principais picos ao mesmo tempo que seus vizinhos são identificados e descartados. Uma possível estratégia de paralelismo neste caso seria alocar threads para verificar todos os possíveis picos em paralelo, marcando seus picos vizinhos como inválidos posteriormente.

4.4 Resumo

Este capítulo descreveu as abordagens de implementação utilizadas para tentar paralelizar o processo de detecção de conjuntos de pontos aproximadamente coplanares. Foram discutidas duas estratégias para construção de octrees: uma explorando paralelismo dinâmico, e outra híbrida que divide o processamento entre a GPU e a CPU. Também foram detalhados procedimentos paralelos para obtenção de matrizes de covariâncias associados aos nodos da octree, bem como para votação no acumulador esférico.

5 RESULTADOS

Este capítulo apresenta os resultados numéricos obtidos com as implementações dos algoritmo descritos nesta monografia.

5.1 Ambiente de Testes

Os experimentos foram realizados em ambiente Windows com um processador Intel i7 2600 3.4GHz com 16GB de RAM para o código serial e uma GPU NVidia GTX-970 para execução dos kernels em CUDA. O código foi implementado utilizando as linguagens C++ e CUDA C. Para geração da octree em CPU foi utilizado paralelismo na subdivisão dos nodos utilizando OpenMP. Os datasets de testes são os mesmos utilizados no artigo de Limberger e Oliveira (LIMBERGER; OLIVEIRA, 2015) de modo a permitir uma comparação de desempenho entre a versão serial e as versões paralelas do algoritmo. Para captura dos tempos em CUDA foram utilizadas as primitivas de *cudaEvents*, que permitem capturar os tempos de uso da GPU com maior precisão do que o uso de contadores em CPU. Para todos os casos de teste foram realizadas três execuções e obtidos os valores médios encontrados. Os resultados foram arrendondados para até três casas decimais e variabilidade dos tempos obtidos nas execuções demonstrou ser muito baixa e insuficiente para alteração dos resultados médios.

5.2 Resultados da Geração da Octree

Os tempos de execução das três versões de geração da octree podem ser observados na Tabela 5.1. Estes dados demonstram que o desempenho da versão em CPU foi superior aos desempenhos da versão utilizando paralelismo dinâmico em GPU (GPU PD) e também da versão híbrida CPU-GPU (GPU H). Percebe-se ainda que a abordagem baseada em paralelismo dinâmico não conseguiu executar para os datasets maiores (*Box* e *Bremen*). Por outro lado, a vantagem relativa da implementação em CPU com relação a implementação híbrida se reduz com o tamanho do dataset. O gráfico apresentado na Figura 5.1 compara os desempenhos das implementações com paralelismo dinâmico e híbrida. Entre estas duas abordagens, fica evidenciada a superioridade relativa da abordagem híbrida. A Figura 5.2 ilustra uma octree gerada em GPU para segmentar diversos conjuntos de pontos coplanares distribuídos em algumas regiões do espaço e contendo orientações variadas.

Uma das razões do desempenho inferior das duas abordagens que utilizam GPU quando comparadas com a abordagem em CPU é a impossibilidade de alocação de memória dinamicamente. Esta limitação impõe a necessidade de ordenamento sucessivo de trechos do vetor auxiliar T. Especificamente no caso da abordagem com paralelismo dinâmico, existe a dificuldade de sincronizar blocos de threads via hardware, tornando necessária a criação de barreiras utilizando operações atômicas que diminuem significativamente o desempenho do algoritmo.

Estes resultados demonstram que nem sempre problemas que utilizem uma abordagem recursiva e dinâmica produzirão resultados eficientes na GPU. Um meio de corrigir estes problemas em trabalhos futuros seria construir a octree nível a nível, eliminando o uso de recursão.

Tabela 5.1 – Tempo (em segundos) de geração da octree para nuvens de pontos de vários tamanhos e configurações. CPU refere-se à versão em CPU usando OpenMP. GPU PD refere-se ao uso de paralelismo dinâmico e GPU H refere-se à abordagem híbrida.

Nuvem de Pontos	*N° de Pontos*	*CPU*	*GPU PD*	*GPU H*
Computer	68.852	0,005	1,54	1,87
Room	112.586	0,009	2,65	1,46
Utrecht	160.256	0,024	4,47	1,61
Museum	179.744	0,013	6,68	0,91
Box	964.806	0,054	-	1,40
Bremen	20.332.246	2,050	-	4,94

Figura 5.1 – Comparação de tempos (em segundos) para geração de octree utilizando paralelismo dinâmico (GPU PD) e usando a versão híbrida (GPU H)

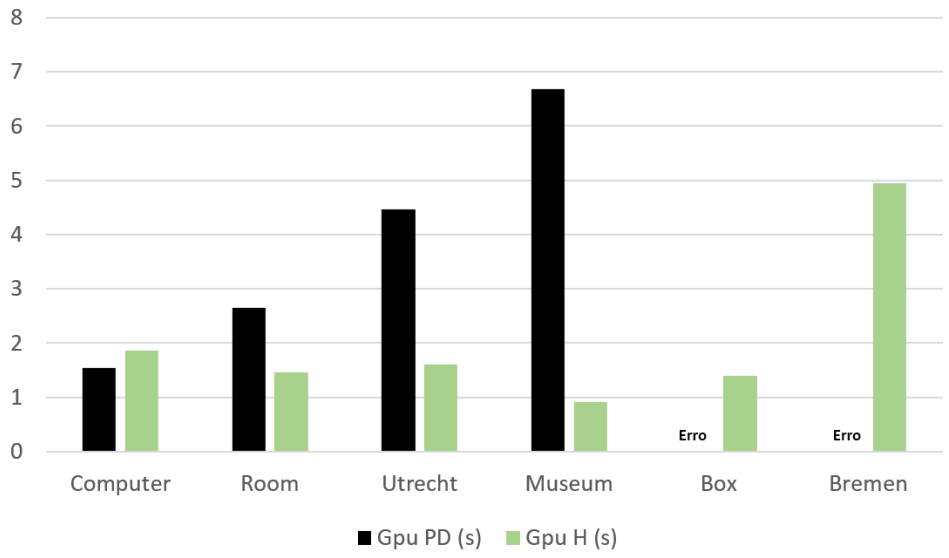
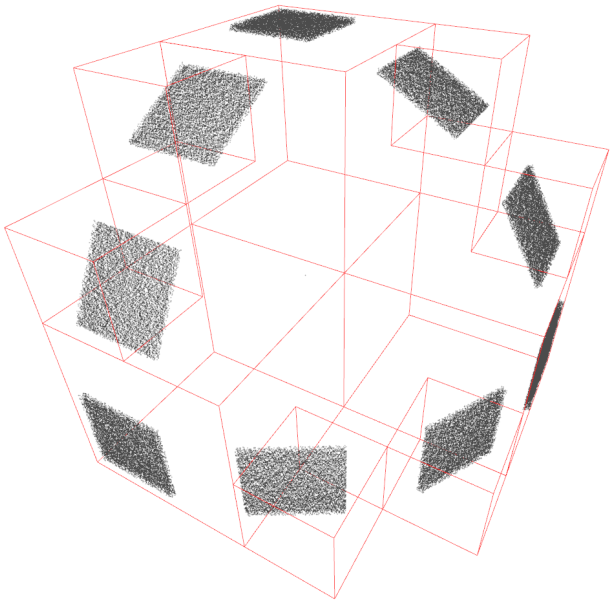


Figura 5.2 – Exemplo de resultado da geração da octree.



5.3 Resultados da Votação

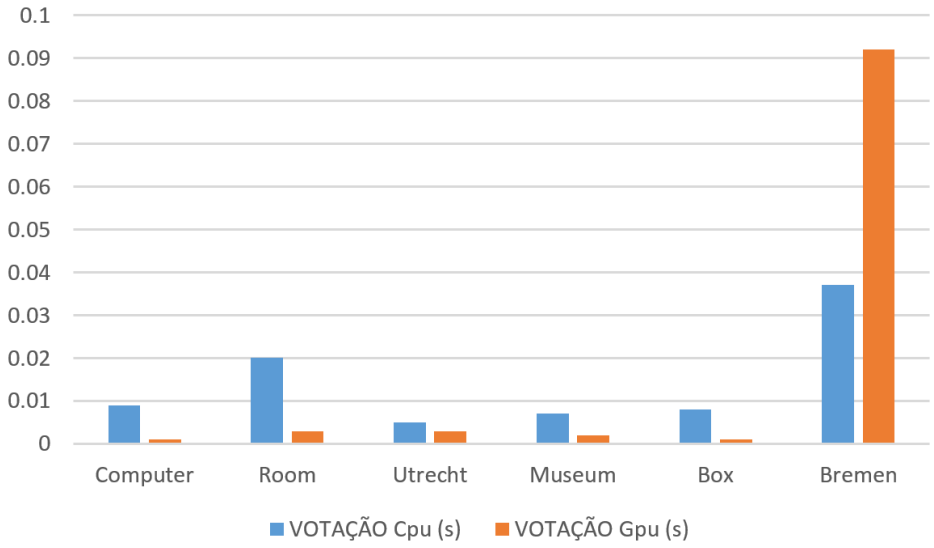
A Tabela 5.2 sumariza os resultados de desempenho obtidos com o processo de votação, os quais são apresentados graficamente na Figura 5.3. Neste caso, o processo de paralelização em GPU obteve um ganho de performance de até 9 vezes com relação ao procedimento executado em CPU para datasets com até 1 milhão de pontos. Para o dataset *Bremen*, com 20 milhões de pontos, entretanto, a implementação em GPU apresentou significativa perda de desempenho, atingindo apenas 40% do desempenho da implementação em CPU. As causas desta perda de desempenho podem ser explicadas pelo uso de operações atômicas na computação de cada voto, que podem ocasionar uma maior quantidade de colisões de acesso a memória no caso da utilização de muitos kernels Gaussianos para a votação. A Figura 5.4 mostra uma visualização do acumulador, mostrando os votos (em azul) distribuídos nas várias células.

Tabela 5.2 – Resultados para o kernel de votação

Nuvem de Pontos	*N° de Pontos*	*CPU (s)*	*GPU (s)*	*Speedup*
Computer	68.852	0,009	0,001	9
Room	112.586	0,020	0,003	6,7
Utrecht	160.256	0,005	0,003	1,7
Museum	179.744	0,007	0,002	3,5
Box	964.806	0,008	0,001	8
Bremen	20.332.246	0,037	0,092	0,4*

*Para o caso do dataset Bremen foi observado uma perda de desempenho (*slowdown*) de 2,42 vezes na GPU.

Figura 5.3 – Gráfico comparativo de tempos em segundos da votação em CPU e GPU.



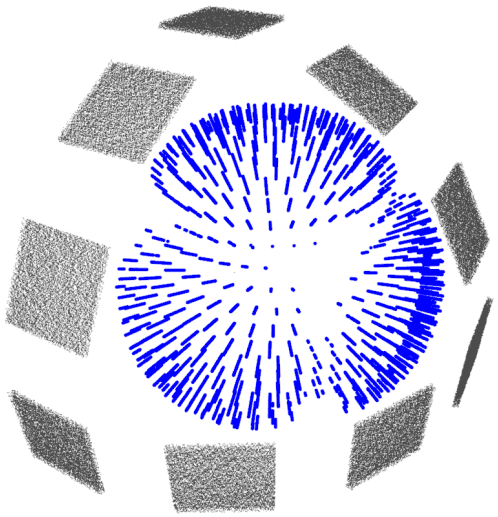
5.4 Discussão

A experiência deste trabalho demonstrou a complexidade do processo de mapear uma implementação eficiente em CPU para GPU. Também demonstrou a importância do algoritmo a ser paralelizado se ajustar à arquitetura de CUDA. Deve ser destacado que a realização deste trabalho propiciou o primeiro contato do estudante com CUDA, bem como a primeira experiência prática de tentar paralelizar um algoritmo. Este foi um grande aprendizado que possivelmente nos permitirá obter no futuro implementações mais eficientes que a atualmente disponível para CPU.

5.5 Resumo

Este capítulo apresentou os resultados obtidos com as implementações realizadas. Enquanto o processo de criação da octree apresentou desempenho inferior àquele alcançado com a implementação em CPU, foram observados ganhos de desempenho com a implementação da etapa de votação para datasets com até 1 milhão de pontos.

Figura 5.4 – Demonstração da votação em um acumulador esférico.



6 CONCLUSÕES E TRABALHOS FUTUROS

Neste trabalho foram implementadas duas versões paralelas para GPU de um algoritmo para detecção de regiões planares em nuvens de pontos não-estruturadas. Tais implementações foram feitas em CUDA, e uma delas explorou a funcionalidade de paralelismo dinâmico disponível em algumas GPUs modernas.

Uma das motivações para a realização deste trabalho foi o aprendizado de CUDA e a tentativa de verificar seu potencial como plataforma para programação paralela simples e de mais alto nível. O que foi observado é que nem todo algoritmo pode ser facilmente paralelizado com CUDA, havendo uma necessidade de projetar cuidadosamente as estruturas de dados que serão utilizadas na GPU, bem como a dinâmica de acesso a memória.

O maior problema encontrado ocorreu na geração da octree utilizando recursão e paralelismo dinâmico. Neste caso, a necessidade de sincronismo global demonstrou ser um gargalo. O sistema de votação, por sua vez, apresentou ganhos de desempenho para datasets com pelo menos 1 milhão de pontos.

O trabalho foi uma experiência extremamente enriquecedora. Possibilitou um primeiro contato com CUDA, e um aprendizado sobre os mais diversos aspectos de um assunto extremamente importante atualmente que é a programação paralela em GPUs. Foi possível observar suas vantagens e limitações.

Como trabalhos futuros, o algoritmo deverá ser reimplementado com estruturas de dados mais otimizadas para a computação em GPU, evitando a tendência de reproduzir estruturas similares e compatíveis com as utilizadas no algoritmo em CPU. Um aspecto importante deverá ser a utilização de outros métodos de indexação de octrees para aumentar a coalescência de memória. Outro aspecto ainda a ser estudado é uma forma efetiva para detecção picos paralelamente após a votação.

REFERÊNCIAS

BORRMANN, D. et al. The 3d hough transform for plane detection in point clouds: A review and a new accumulator design. **3D Research**, Springer, v. 2, n. 2, p. 1–13, 2011.

FERNANDES, L. A.; OLIVEIRA, M. M. Real-time line detection through an improved hough transform voting scheme. **Pattern Recognition**, Elsevier, v. 41, n. 1, p. 299–314, 2008.

HARRIS, M. et al. Optimizing parallel reduction in cuda. **NVIDIA Developer Technology**, Nvidia Corporation Santa Clara, CA, USA, v. 2, n. 4, 2007.

HOUGH, P. V. **Method and means for recognizing complex patterns**. [S.l.], 1962.

LIMBERGER, F. A.; OLIVEIRA, M. M. Real-time detection of planar regions in unorganized point clouds. **Pattern Recognition**, Elsevier, 2015.

NVIDIA. **Kepler GK110 Architecture Whitepaper**. 2012. Available from Internet: <http://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf>.

NVIDIA. **CUDA C Programming Guide Version 7.0**. 2015. Available from Internet: <http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf>.

NVIDIA. **CUDA SAMPLES**. 2015. Available from Internet: <http://docs.nvidia.com/cuda/pdf/CUDA_Samples.pdf>.

NVIDIA. **Thrust Quick Start Guide**. 2015. Available from Internet: <http://docs.nvidia.com/cuda/pdf/Thrust_Quick_Start_Guide.pdf>.

XIAO, S.; FENG, W.-c. Inter-block gpu communication via fast barrier synchronization. In: IEEE. **Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium on**. [S.l.], 2010. p. 1–12.