Point Based Graphics e Aplicações em Jogos

Luciano Silva
Laboratório de Processamento Gráfico e Mídias Digitais
Faculdade de Computação e Informática, Universidade Presbiteriana Mackenzie

Abstract

Point based Graphics embraces a set of processes, techniques and algorithms for the acquisition, representation, processing and storage of point clouds, aiming at applications in graphics processing. As multi and manycore processors have increased their storage and processing powers, this area has become very attractive for applications that require high performance graphics, eg, digital games. Within this context, this chapter introduces the basics of Point Based Graphics, highlighting the main applications in modeling, rendering and animation in GPU.

Resumo

Point based Graphics refere-se a um conjunto de processos, técnicas e algoritmos para aquisição, representação, processamento e armazenamento de nuvens de pontos, visando às aplicações em processamento gráfico. Com o aumento da capacidade de processamento e armazenamento de processadores multi e many-cores, esta área tornou-se bastante atrativa para aplicações gráficas que requerem alto desempenho como, por exemplo, jogos digitais. Dentro deste contexto, este capítulo apresenta as bases de Point Based Graphics, evidenciando as principais aplicações em modelagem, renderização e animação em GPU.

1.1. Introdução

A tecnologia de *Point Based Graphics*, com o aumento do poder de processamento das unidades de processamento gráfico (GPU), tem oferecido ao segmento de desenvolvimento de jogos digitais novas possibilidades para aumento de desempenho das aplicações. Neste contexto, ao invés de se trabalhar com estruturas de dados complexas, que envolvem, por exemplo, vértices, arestas e faces, utiliza-se somente uma nuvem de pontos para representar o objeto a ser processado.

A partir desta nuvem de pontos, com o poder de processamento gráfico e genérico das GPUs, procedimentos como modelagem, transformações, renderização e animação são efetuados diretamente nesta nuvem. Mesmo procedimentos considerados não-gráficos como, por exemplo, simulações de Física ou inferências em Inteligência Artificial podem ser efetuadas nas nuvens com o auxílio de linguagens como CUDA ou OpenCL.

Assim, dentro deste contexto, este texto tem como objetivo introduzir os conceitos fundamentais de nuvens de pontos para suporte a *Point Based Graphics* e processamento dentro de GPUs, visando às aplicações de desenvolvimento de jogos digitais.

O texto está organizado da seguinte forma:

- a Seção 1.2 traz o conceito de nuvens de pontos e algumas formas para sua representação
- a Seção 1.3 discute o processo de aquisição de nuvens de pontos através de *scanning* 3D
- a Seção 1.4 apresenta funcionalidades de processamento gráfico de nuvens de pontos através de *shaders*
- a Seção 1.5 apresenta detalhadamente as funcionalidades de processamento genérico de nuvens de pontos em GPU, com as arquiteturas CUDA e OpenCL. Como o processamento de nuvens muitas vezes não requer saída gráfica, deu-se uma atenção especial a este tópico.
- finalmente, a Seção 1.6 apresenta o fechamento do capítulo e, em seguida, são apresentadas algumas sugestões de referências bibliográficas.

O autor deseja que este texto possa disponibilizar um suporte simples e direto para todos aqueles que queiram iniciar trabalhos na área de *Point Based Graphics*, especialmente aplicados os aplicados em jogos.

1.2. Nuvens de Pontos e Fundamentos de Representação

A estrutura básica em *Point Based Graphics* é uma nuvem de pontos (*point cloud*). Essencialmente, uma nuvem de pontos é uma coleção de pontos com coordenadas tridimensionais e, normalmente, sem relações entre os pontos. A figura abaixo mostra dois exemplos de nuvens de pontos, onde se pode ver um cenário (à esquerda) e dois objetos (à direita):

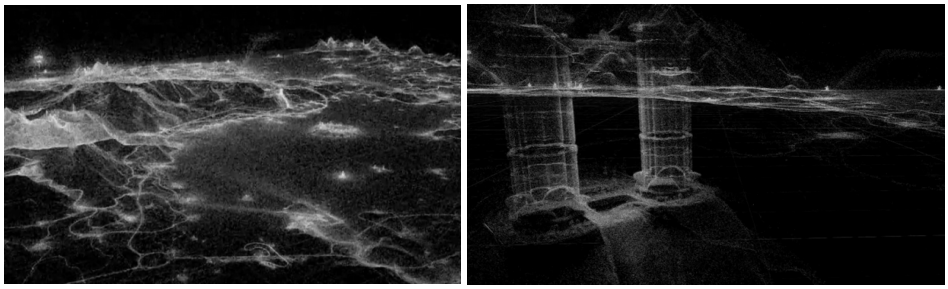
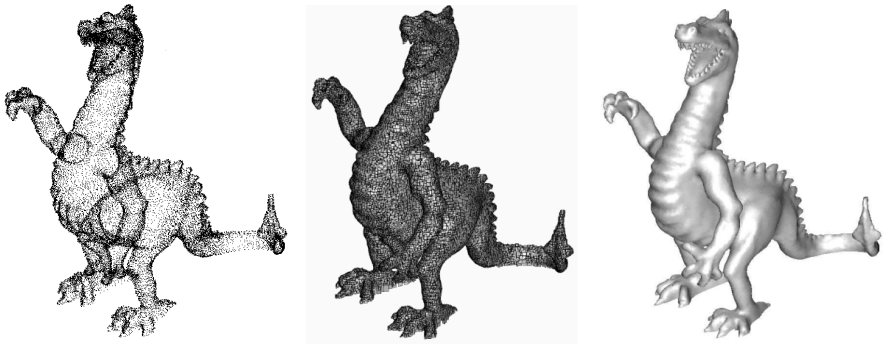


Figura 1: Exemplos de nuvens de pontos (cenário e objetos).
Fonte: *Jogo Just Cause 2*.

A partir de uma nuvem de pontos, pode-se construir uma visualização básica, através de acumulação de patches, ou se aproximar ou interpolar uma superfície pelos pontos. A figura abaixo mostra três níveis usuais de visualização de uma malha de pontos: a própria nuvem, uma acumulação de *patches* e uma superfície interpolada (já com renderização):



(a) Nuvem (b) *Patches* acumulados (c) Superfície

Figura 2: Níveis de visualização de uma nuvem de pontos.
Fonte: **(Linsen, 2001)**.

Uma das grandes vantagens de falta de relações entre os pontos reside no fato da velocidade de alteração das estruturas de representação das nuvens de pontos, uma vez que, normalmente, não há necessidade de atualização de arestas, faces ou mesmo objetos.

Formalmente, um **ponto** P em uma nuvem de ponto é uma t-upla formada, geralmente, pela sua posição (x,y,z) e alguma informação de cor (r,g,b):

$$\mathrm{p}_i = \{x_i, y_i, z_i, r_i, g_i, b_i, dist_i, ...\}.$$

Outras informações podem incluir, por exemplo, informações de normais, curvatura, tangentes, dentre outros. A partir do conceito de ponto, define-se uma **nuvem de pontos** como uma coleção indexada de pontos:

$$\mathcal{P} = \{\mathrm{p}_1, \mathrm{p}_2, \ldots, \mathrm{p}_i, \ldots, \mathrm{p}_n\}.$$

Para se representar eficientemente pontos e nuvens de pontos, existem diversas alternativas interessantes para o segmento de jogos digitais. O *framework* PCL (*Point Cloud Library*) **(PCL, 2012)** , por exemplo, é um conjunto de classes em C++ para representação tanto de nuvens de pontos 2D quanto 3D. A definição de uma nuvem de pontos em PCL toma, como base, uma estrutura para representar cada ponto e, a partir de um ponto, constrói-se a nuvem. Abaixo, tem-se um exemplo de representação de nuvem de ponto em PCL:

```
pcl::PointCloud<pcl::PointXYZ> cloud;
std::vector<pcl::PointXYZ> data = cloud.points;
```

A classe PointCloud disponibiliza uma série de métodos básicos para manipulação de pontos isolados ou conjuntos de pontos. Além desta classe, bastante eficiente para representação de nuvens de pontos, a PCL ainda disponibiliza duas outras formas básicas para representação de nuvens: *quadtrees* e *octrees*, conforme mostrado na figura abaixo:

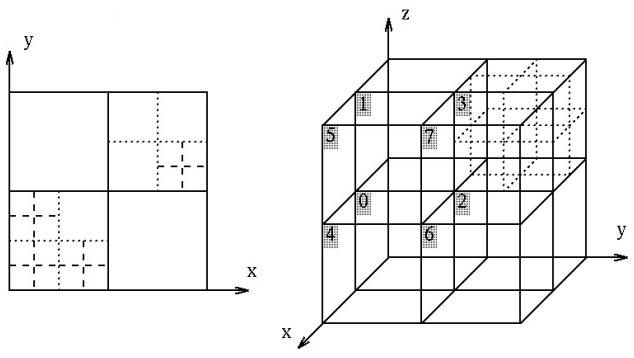


Figura 3: Quadtree (à esquerda) e Octree (à direita).

Normalmente, uma *quadtree* é utilizada para representação de nuvens de pontos bidimensionais, enquanto que *octrees* são utilizadas para nuvens de pontos tridimensionais.

1.3. Aquisição de Nuvens de Pontos

Existem diversas estratégias para aquisição de nuvens de pontos (Gross & Pfister, 2007). No contexto de jogos digitais, uma maneira bastante comum é via *scanners* tridimensionais, como mostrado na figura abaixo:

Figura 4: *Scanner* 3D manual, baseado em Visão Estéreo e projeção laser.

A projeção laser permite indicar qual segmento de reta está sendo escaneado. O sistema duplo de câmeras permite utilizar técnicas de reconstrução 3D, baseadas em Visão Estereoscópica. A coordenada do ponto que está sendo escaneado pode ser obtida através de uma intersecção de retas definidas pelas duas câmeras e os planos de projeção das imagens, conforme mostra a figura a seguir:

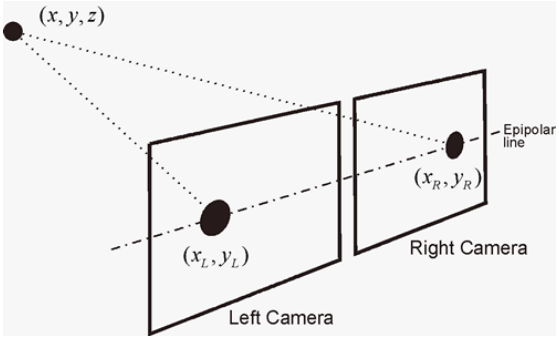


Figura 5: Esquema de obteção das coordenadas de um ponto baseado em Visão Estereoscópica.

Este esquema exige uma calibração de todo o sistema de aquisição como, por exemplo, distância entre as câmeras ou estimação das distâncias focais das duas câmeras. Uma vez calibrado o sistema, a granularidade dos pontos escaneados pode ser controlada

tanto no processo de aquição, quanto no processo de pós-processamento. A figura abaixo mostra duas nuvens de pontos, com as respectivas superfícies rescontruídas:

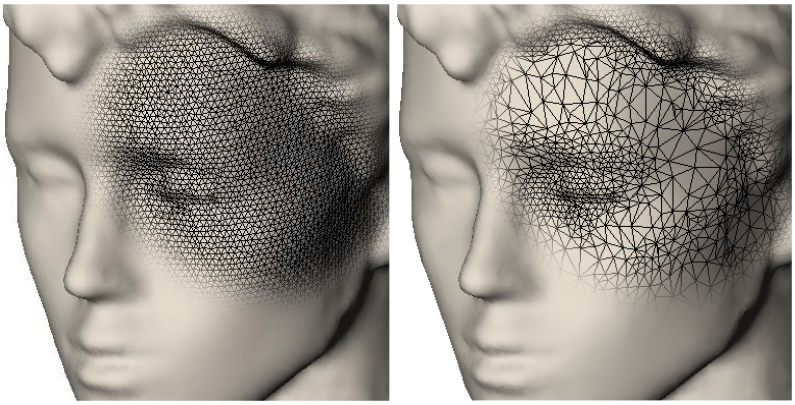


Figura 6: Resultado de um processo de *scanning* para um modelo de jogo.

Na imagem da esquerda, tem-se uma nuvem de pontos bastante regular, resultado comum de um processo de *scanning*. Na imagem da direita, os pontos foram processados e, regiões que não necessitam de muitos detalhes podem e devem ser simplificadas.

Os equipamentos para *scanning* 3D, mesmo para pequenos objetos, ainda tem um custo elevado. Uma alternativa bastante interessante atualmente é o uso do *gadget* de interação para jogo Kinect. Para reconhecer profundidade dos objetos, o Kinect projeta uma nuvem estruturada de pontos, que pode ser percebida e capturada por câmeras de infra-vermelho. A imagem abaixo mostra um exemplo de nuvem projetada de pontos pelo Kinect:

Figura 7: Nuvem de pontos projetada pelo Kinect.

O processo de obtenção da nuvem de pontos pelo Kinect pode ser feita através do Kinect SDK. A seguir, tem-se um exemplo de código que realiza esta tarefa:

```
using Microsoft.Kinect;
KinectSensor sensor;

sensor = KinectSensor.KinectSensors[0];

sensor.DepthStream.Enable(DepthImageFormat.Resolution320x240Fps30);
sensor.DepthFrameReady+=DepthFrameReady;
sensor.Start();

void DepthFrameReady(object sender, DepthImageFrameReadyEventArgs e)

{
 DepthImageFrame imageFrame =  e.OpenDepthImageFrame(); // obtém imagem para extrair
                                                        // os pontos
 short[] pixelData = new short[imageFrame.PixelDataLength];
 imageFrame.CopyPixelDataTo(pixelData);

}

int temp = 0; int i = 0;
for (int y = 0; y < 240; y +=s)
 for (int x = 0; x < 320; x +=s)
 {
  temp = ((ushort) pixelData[x+y*320])>>3;
  ((TranslateTransform3D) points[i].Transform).OffsetZ = temp; // nuvem de pontos
  i++;
 }
}
```

A partir deste processo de obtenção de imagens e extração de pontos, pode-se obter nuvens de pontos como mostrado a seguir, onde os pontos foram renderizados como triângulos:

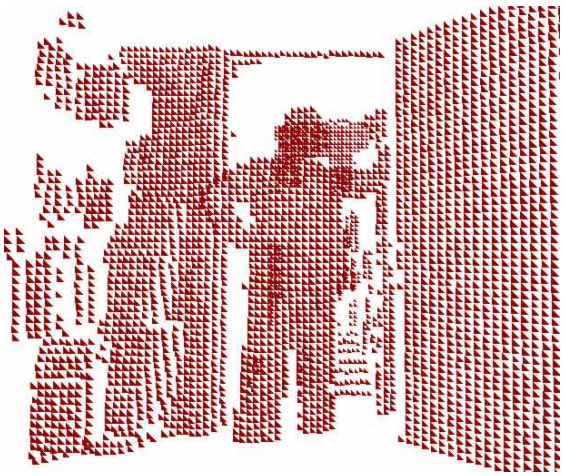


Figura 8: Nuvem de pontos extraída de imagens do Kinect.

Além dos pontos, o Kinect ainda permite obter animações baseadas em um esqueleto humano de referência.

Uma vez obtida a nuvem de pontos, as próximas seções abordarão como processá-las dentro de uma GPU com propósitos gráficos (via *shaders*) ou com propósitos gerais (via CUDA ou OpenCL).

1.4. Processamento Gráfico de Nuvens de Pontos com *Shaders*

Shaders são pequenos programas executados dentro de unidades gráficas de processamento (GPU) e são extremamente adaptados para processamento de nuvens de pontos devido a sua natureza.

Existem, essencialmente, três tipos de *shader*:

- *Vertex Shaders*
- *Pixel Shaders*
- *Geometry Shaders*:

Os *vertex shaders* recebem como entrada um vértice (ponto) e retornam um outro vértice, resultado de alguma transformação. No contexto de nuvens de pontos para jogos, este tipo de shader é muito utilizado para os mecanismos de modelagem (transformações) e animação. O código abaixo mostra o código de vertex shader utilizado na simulação de líquidos baseados em nuvens de pontos em OSGL (*OpenGL Shading Language*):

```
uniform float time;
uniform float xs, zs;
void main()
{
float s;
s = 1.0 + 0.1*sin(xs*time)*sin(zs*time);
gl_Vertex.y = s*gl_Vertex.y;
gl_Position =
   gl_ModelViewProjectionMatrix*gl_Vertex;
}
```

Outro exemplo bastante importante em animação de nuvens de pontos é conhecido comumente como sistemas de partículas, conforme mostra o exemplo do *vertex shader* abaixo, onde, além da posição, controla-se também parâmetros de ordem física:

```
uniform vec3 init_vel;
uniform float g, m, t;
void main()
{
     vec3 object_pos;
     object_pos.x = gl_Vertex.x + vel.x*t;
     object_pos.y = gl_Vertex.y + vel.y*t
       - g/(2.0*m)*t*t;
     object_pos.z = gl_Vertex.z + vel.z*t;
     gl_Position =
               gl_ModelViewProjectionMatrix*
          vec4(object_pos,1);
}
```

Os *pixel shaders* recebem como entrada um vértice (ponto) e retornam uma cor associada ao vértice. No contexto de nuvens de pontos de jogos, este tipo de shader é muito utilizado para os mecanismos de *renderização*. O trecho de código a seguir mostra parte do cálculo do Modelo de Phong para nuvens de pontos:

```
varying vec3 normale;
varying vec4 positione;
void main()
{
vec3 norm = normalize(normale);
vec3 lightv = normalize(gl_LightSource[0].position-positione.xyz);
vec3 viewv = normalize(positione);
vec3 halfv = normalize(lightv + viewv);
vec4 diffuse = max(0, dot(lightv, viewv))
  *gl_FrontMaterial.diffuse*gl_LightSource[0].diffuse;
vec4 ambient = gl_FrontMaterial.ambient*gl_LightSource[0].ambient;
```

Finalmente, existem os *geometry shaders*, que permitem, dentro do contexto de nuvens de pontos, a geração de novos pontos. Como são executados depois dos vertex shaders, possuem aplicação imediata nos processos de refinamento de nuvens de pontos.

1.5. Processamento Genérico de Nuvens de Pontos com CUDA e OpenCL

Atualmente, existem duas tecnologias (e linguagens) importantes para processamento genérico de nuvens de pontos em GPU: CUDA e OPENCL (Silva e Stringhini, 2012).

1.5.1 Linguagem CUDA C

A arquitetura CUDA (*Compute Unified Device Architecture*) (NVIDIA, 2011) unifica a interface de programação para as GPUs da NVIDIA, assim como define um modelo de programação paralela que pode ser usado de forma unificada em dezenas de dispositivos diferentes. A linguagem CUDA C possibilita que se inclua comandos direcionados às GPUs da NVIDIA em programas escritos em linguagem C/C++.

Apesar de ser uma interface unificada que possibilita a programação em diferentes placas gráficas, CUDA possui características intrínsecas ao hardware das placas NVIDIA. Assim, antes de apresentar o modelo de programação CUDA, uma breve descrição da arquitetura Fermi será apresentada a fim de justificar o modelo de programação CUDA e familiarizar o leitor com este tipo de dispositivo que tem sido referenciado como *acelerador*.

1.5.2 Arquitetura FERMI

As GPUs são compostas de centenas de núcleos (*cores*) simples que executam o mesmo código através de centenas a milhares de threads concorrentes. Esta abordagem se opõe ao modelo tradicional de processadores *multicore*, onde algumas unidades de núcleos completos e independentes são capazes de processar threads ou processos. Estes núcleos completos, as CPUs, possuem poderosas unidades de controle e de execução capazes de executar instruções paralelas e fora de ordem, além de contarem com uma poderosa hierarquia de cache. Já as GPUs contam com unidades de controle e de execução mais simples, onde uma unidade de despacho envia apenas uma instrução para um conjunto de núcleos que a executarão em ordem. O modelo de execução das GPUs é conhecido como SIMT (*Single Instruction Multiple Threads*), derivado do clássico termo SIMD (*Single Instruction Multiple Data*). A figura 9 apresenta as diferenças nas arquiteturas de CPU e GPU.

Figura 9: arquitetura de CPU e de GPU (fonte: NVIDIA, 2011).

A arquitetura Fermi da NVIDIA segue este princípio de dedicar uma maior quantidade de transístores às unidades de execução, ao invés de dedica-los às unidades de controle e cache. A figura 10 apresenta uma visão geral da arquitetura Fermi:

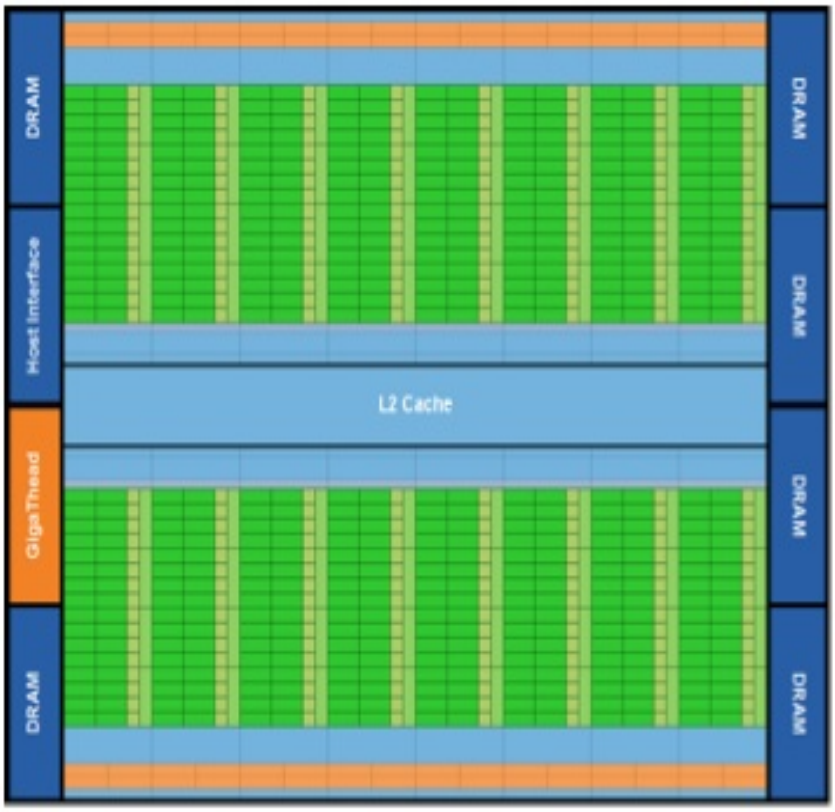


Figura 10: visão geral da arquitetura Fermi (fonte: NVIDIA, 2009).

A arquitetura conta com 16 SM (*streaming multiprocessors*), cada um composto por 32 núcleos de processamento (*cores*), resultando num total de 512 núcleos. É possível observar uma cache de segundo nível (L2) compartilhada por todos os SM. A cache de primeiro nível (L1) é compartilhada pelos 32 núcleos de cada SM. A figura 11, próxima página, mostra a hierarquia de cache da Fermi, juntamente com dois outros tipos de memória presentes na arquitetura. A memória compartilhada (*shared memory*) pode ser usada explicitamente pelo programador como uma memória de "rascunho" que pode acelerar o processamento de uma aplicação, dependendo do seu padrão de acesso aos dados. Esta memória é dividida fisicamente com a cache de primeiro nível com um total de 64KB, cujo tamanho é configurável: 16 KB – 48KB para cache e memória

compartilhada respectivamente ou ao contrário. Além dos dois níveis de cache e da memória compartilhada, a Fermi conta com uma memória global (DRAM) de até 6GB.

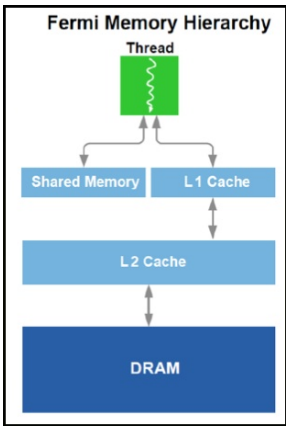


Figura 11: hierarquia de memória da FERMI (fonte: NVIDIA, 2009).

A figura 12 apresenta a arquitetura dos SM. Cada um é composto por quatro blocos de execução controlados por duas unidades escalonamento de *warps* (grupos de 32 *threads*).

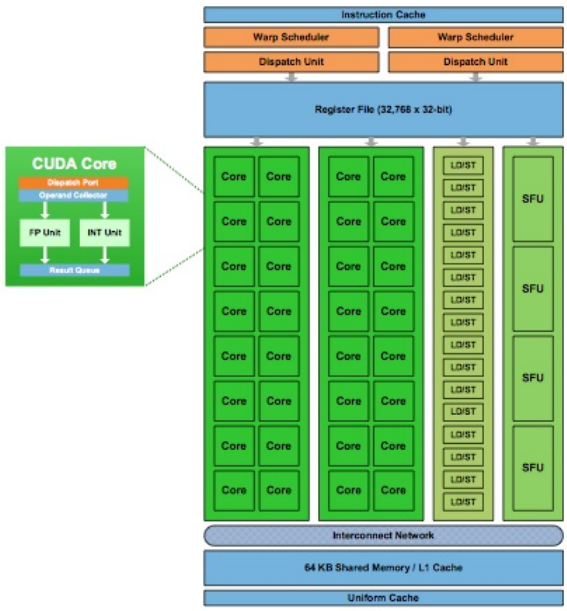


Figura 12: arquitetura de um SM (*Streaming Multiprocessor*) (fonte: NVIDIA, 2009).

Além disso, cada SM conta com uma memória cache L1/memória compartilhada de 64KB, já mencionada, e com 32KB de registradores, compartilhados entre todas as threads que executarão no SM.

1.5.3 Programação CUDA

O modelo de programação de CUDA C é composto de uma parte sequencial executada na CPU (*host*) e de uma parte paralela executada na GPU (*device*). O programador desenvolve uma função especial chamada *kernel* que será replicada em até milhares de threads durante a execução na GPU. As threads realizam as mesmas operações simultaneamente, porém atuam ao mesmo tempo sobre dados diferentes .

Em primeiro lugar, é importante observar a organização das threads em CUDA (figura 5). Elas são organizadas em blocos e, dentro destes blocos, podem estar dispostas em 1, 2 ou até 3 dimensões.Os blocos são organizados em grades de uma ou duas dimensões. Da mesma forma, cada thread também terá disponível a informação de a qual bloco dentro da grade ela pertence. Por exemplo, numa grade 1D, pode-se dizer que a primeira thread entre todas pertencerá ao bloco 0 e terá seu índice dentro do bloco como 0 (bloco 0, thread 0).

A figura 13 mostra uma representação clássica desta organização, apresentando uma grade bidimensional (2D), com blocos de threads também bidimensionais (2D) (NVIDIA, 2011). Estas dimensões, assim como a quantidade de threads e blocos em cada uma delas, são definidas pelo programador no momento em que ele inicia (lança) o *kernel*.

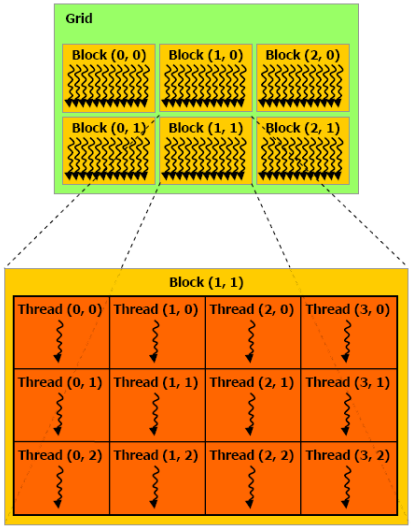


Figura 13: Organização de blocos e threads (fonte: NVIDIA, 2011).

Além disso, CUDA suporta uma série de tipos de memória que podem ser usadas pelos programadores para que possam acelerar a velocidade de execução de um *kernel*. A figura 14 mostra de forma esses tipos de memória num dispositivo CUDA. A memória global (*global memory*) pode ser escrita ou lida a partir do código que executa na CPU, chamado usualmente de *host*. Estas operações são realizadas utilizando-se funções da API (*Aplication Programming Interface*) de CUDA.

Internamente, a memória global pode ser acessada por qualquer thread em execução no dispositivo. Entretanto, a tecnologia usada no desenvolvimento de tal memória não possui taxa de velocidade de acesso que acompanhe a velocidade dos cálculos que pode ser obtida pela GPU, tornando-se um gargalo de desempenho. Por conta disso, a organização de memória conta com outros tipos de memória que podem ser usadas pelo programador para otimizar o desempenho de acesso à memória. São elas a memória local (*shared memory*), compartilhada apenas por threads num mesmo bloco, e os registradores (*registers*), que não são compartilhados entre as threads e são alocados previamente pelo compilador. Existe ainda uma memória somente de leitura, também compartilhada entre todas as threads de um grid, a memória constante (*constant memory*), que possui um tempo de acesso melhor que o da memória global.

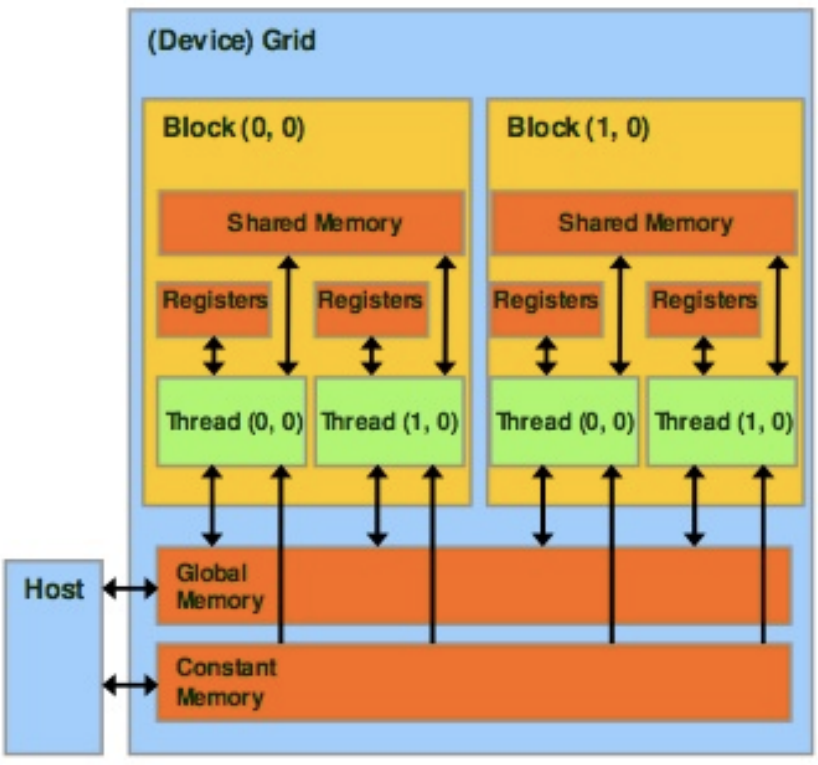


Figura 14: organização de memória em CUDA (fonte: NVIDIA, 2011).

Embora os registradores e a memória local possam ser extremamente efetivos na redução da quantidade de acessos à memória global, o programador deve ser cuidadoso para que não exceda a capacidade efetiva destas memórias considerando os limites de hardware da GPU. Cada dispositivo oferece uma quantidade limitada de memória CUDA, que pode limitar a quantidade de threads que pode executar simultaneamente nos multiprocessadores de um dispositivo.

Como os registradores e a memória local são compartilhados entre as threads, quanto maior for a quantidade destes recursos que cada thread necessitar, menor será a quantidade de threads que poderão executar num processador.

O esquema de escalonamento de threads depende de uma boa quantidade threads em execução para que se obtenha uma boa utilização do dispositivo. Todas as threads em um bloco são executadas em conjunto num SM, que por sua vez pode ter múltiplos blocos concorrentes em execução. Assim, a quantidade de blocos (ocupação) será limitada pela quantidade de recursos de hardware disponíveis no SM (como a quantidade de registradores e memória local, por exemplo).

O esquema de escalonamento é baseado em *warps* – cada bloco é divido em *warps* de 32 threads cada, ou seja, o número de *warps* de um bloco é igual ao número de threads no bloco dividido por 32. Visto que o escalonamento é realizado em grupo de threads, a organização em *warps* serve para que o processador não fique parado quando ocorrer algum bloqueio num grupo de threads – este será *desescalonado* e um outro grupo (*warp*) poderá ser imediatamente executado. Daí a importância de se ter uma boa quantidade de threads em execução em cada SM.

CUDA para a linguagem C consiste numa série de extensões de linguagem e de biblioteca de funções. O modelo de programação assume que o sistema é composto de um host (CPU) e de um dispositivo (device ou GPU).

A programação consiste em definir o código de uma ou mais funções que executarão no dispositivo (kernel) e de uma ou mais funções que executarão no host (a main(), por exemplo). Quando um kernel é invocado, centenas ou até milhares de threads são iniciadas no dispositivo, executando simultaneamente o código descrito no kernel. Os dados utilizados devem estar na memória do dispositivo e CUDA oferece funções para realizar esta transferência.

1.5.4 Exemplo: soma de nuvens de pontos em CUDA

O código a seguir apresenta um exemplo que código em CUDA que implementa a soma de vetores no dispositivo. O comando de invocação do kernel define a quantidade de threads e as dimensões do bloco e da grade.

```
01 #include <stdio.h>
02 #include <stdlib.h>
03 #include <cuda.h>
04
05 __global__ void VecAdd(float* A, float* B, float* C){
06       int i = blockDim.x * blockIdx.x + threadIdx.x;
07       if (i < N)
08          C[i] = A[i] + B[i];
09 }
10 int main(){
11       int N = 1024;
12       size_t size = N * sizeof(float);
13
14       // Aloca memória principal (host) para os vetores de entrada
15       float* h_A = malloc(size);
16       float* h_B = malloc(size);
17       //... (omitida a inicialização dos vetores com os valores a serem
18 somados)
19       // Aloca memória no dispositivo (GPU)
20       float *d_A, *d_B, *d_C;
21       cudaMalloc((void**)&d_A, size);
22       cudaMalloc((void**)&d_B, size);
23       cudaMalloc((void**)&d_C, size);
24
25       // Copia os vetores da memória do host para a do dispositivo
26       cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
27       cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);
28
29       // Invoca o kernel
30       int threadsPerBlock = 256;
31       int     blocksPerGrid     =     (N+threadsPerBlock     –
32 1)/threadsPerBlock;
33
34       dim3 dimGrid(blocksPerGrid, 1, 1); //define grade 1D
35       dim3 dimBlock(threadsPerBlock, 1, 1); //define bloco 1D
36       VecAdd<<<dimGrid, dimBlock>>>(d_A, d_B, d_C);
37
38       // Copia o resultado do dispositivo para a memória do host
39       // h_C contem o resultado na memória do host
40       cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);
41
42       // Libera a memória do dispositivo
43       cudaFree(d_A);
44       cudaFree(d_B);
45       cudaFree(d_C);
46 }
```

A partir deste código, é possível observar algumas das principais características da programação CUDA. São elas:

- uso da palavra-chave **__global__** que indica que a função é um kernel e que só poderá ser invocada a partir do código do host, criando uma grade de threads que executarão no dispositivo (linha 05);
- uso das variáveis pré-definidas **blockDim.x**, **blockIdx.x** e **threadIdx.x** que identificam o bloco e a thread dentro do bloco através de suas coordenadas (linha 06);
- uso da função **cudaMalloc()** que aloca memória no dispositivo (linhas 21 a 23);
- uso da função **cudaMemcpy()**, que copia os dados da memória do *host* para a memória do dispositivo (linhas 27 e 28) e vice-versa (linha 40);
- invocação do kernel e definição de suas dimensões (linha 36);
- uso da função **cudaFree()**, para liberar a memória do dispositivo (linhas 43 a 45).

1.5.5 Linguagem OpenCL

OpenCL (Munchi, 2011) possui uma filosofia ligeiramente diferente de CUDA. A ideia é que a linguagem e seu sistema de tempo de execução sirvam como uma camada de abstração ao hardware heterogêneo que é extremamente comum nos dias de hoje. Assim, um programa OpenCL tem o objetivo de aproveitar todos os dispositivos presentes na máquina, tais como processadores multicore, GPUs, DSPs (*Digital Signal Processors*), entre outros. Uma aplicação que executa em um hardware heterogêneo deve seguir os seguintes passos:

1. Descobrir os componentes que compõem o sistema heterogêneo.
2. Detectar as características do hardware heterogêneo tal que a aplicação possa se adaptar a elas.
3. Criar os blocos de instruções (*kernels*) que irão executar na plataforma heterogênea.
4. Iniciar e manipular objetos de memória.
5. Executar os *kernels* na ordem correta e nos dispositivos adequados presentes no sistema.
6. Coletar os resultados finais.

Estes passos podem ser realizados através de uma série de APIs do OpenCL juntamente com um ambiente de programação e execução dos *kernels*. Esta seção apresenta um resumo do modelo de abstração do OpenCL juntamente com um exemplo simples de código.

Em primeiro lugar, é importante conhecer o **modelo de plataforma** heterogênea do OpenCL. Ele é composto por um **host** e um ou mais dispositivos OpenCL (***devices***). Cada dispositivo possui uma ou mais unidades de computação (*compute units*), que por

sua vez são compostos por um conjunto de elementos de processamento (*processing elements*). A figura 15 apresenta esta organização.

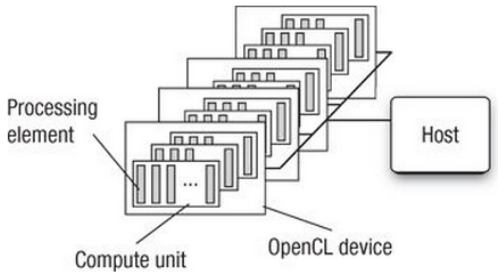


Figura 15: modelo de plataforma do OpenCL (fonte: Munchi, 2011)

O *host* é conectado a um ou mais dispositivos e é responsável por toda a parte de inicialização e envio dos *kernels* para a execução nos dispositivos heterogêneos. Os dispositivos normalmente são compostos por unidades de computação que agrupam uma determinada quantidade de elementos de processamento. Em relação a CUDA, as unidades de computação correspondem aos *Streaming Multiprocessors* da GPU (dispositivo) e os elementos de processamento correspondem aos núcleos (*cores*). Um dispositivo, portanto, pode ser uma CPU, GPU, DSP ou outro qualquer, dependendo da implementação do OpenCL.

O modelo de execução define que uma aplicação OpenCL é composta por um programa *host* e um conjunto de *kernels*. O programa host executa no host (normalmente uma CPU) e os *kernels* executam nos dispositivos disponíveis.

O programa *host*, ou simplesmente *host*, envia o comando de execução de um *kernel* para um dispositivo. Isto faz com que várias instâncias da função que implementa o *kernel* sejam executadas no dispositivo alvo. Em OpenCL estas instâncias são chamadas de *work-items* (itens de trabalho) e correspondem às *threads* de CUDA. Assim como em CUDA, cada *thread* ou *work-item* é identificado por suas coordenadas no espaço de índices (que aqui também pode ter até 3 dimensões) e correspondem ao seu *global ID*.

Os *work-items* são organizados, por sua vez, em *work-groups*. Estes, oferecem uma maneira de estabelecer granularidades diferentes aos grupos de itens de trabalho, o que normalmente facilita a divisão de trabalho e a sincronização. Os *work-groups* correspondem aos blocos de CUDA e podem ser situados num espaço de até três dimensões. Assim, os *work-items* possuem dois tipos de coordenadas: local (dentro do *work-group*) e global (dentro do conjunto completo de *work-items* em execução).

O *host* deve ainda definir um contexto (***context***) para a aplicação OpenCL. Um contexto define o ambiente de execução no qual os *kernels* são definidos e executam e é definido em termos dos seguintes recursos: dispositivos, conjunto de *kernels*, objetos de programa (códigos fonte e executável dos *kernels* que executam a aplicação) e objetos de memória (dados que serão utilizados pelos *kernels* durante o processamento). Assim,

um contexto é todo o conjunto de recursos que um *kernel* vai utilizar durante sua execução.

O contexto é definido em tempo de execução pelo *host* de acordo com os dispositivos disponíveis na máquina alvo. Para possibilitar uma escolha dinâmica do dispositivo onde os *kernels* vão executar o OpenCL compila os *kernels* dinamicamente, gerando os objetos de programa, portanto, em tempo de execução.

A interação entre o *host* e os dispositivos é realizada através de uma fila de comandos (***command-queue***). Os comandos são colocados nesta fila e aguardam seu momento de executar. A fila é criada pelo *host* e conectada a um dispositivo logo após a criação do contexto. Esta fila suporta três tipos de comandos: execução de *kernel*, transferência de dados (objetos de memória) e comandos de sincronização.

Os comandos colocados em uma fila executam de forma assíncrona com relação ao *host*. Comandos de sincronização podem ser utilizados caso uma ordem deva ser estabelecida. Os comandos na fila normalmente executam em ordem (***in-order execution***), porém algumas implementações de OpenCL podem oferecer o modo de execução fora de ordem (***out-of-order execution***), que prevê uma execução assíncrona dos comandos enfileirados.

O **modelo de memória** do OpenCL define dois tipos de objetos de memória: ***buffers*** (blocos contíguos de memória aos quais é possível mapear estruturas de dados) e **imagens**. Estas, podem ser manipuladas através de funções específicas presentes na API do OpenCL.

O modelo de memória define cinco diferentes regiões (figura 16, próxima página):

- ***Host memory***: visível apenas ao host.
- ***Global memory***: permite acesso para leitura e escrita a partir de todos os *work-items* em todos os *work-groups*.
- ***Constant memory***: é uma memória global que é inicializada pelo *host* e permite acesso somente de leitura aos *work-items*.
- ***Local memory***: é compartilhada apenas entre os *work-items* de um mesmo *work-group*.
- ***Private memory***: é privada a cada *work-item*.

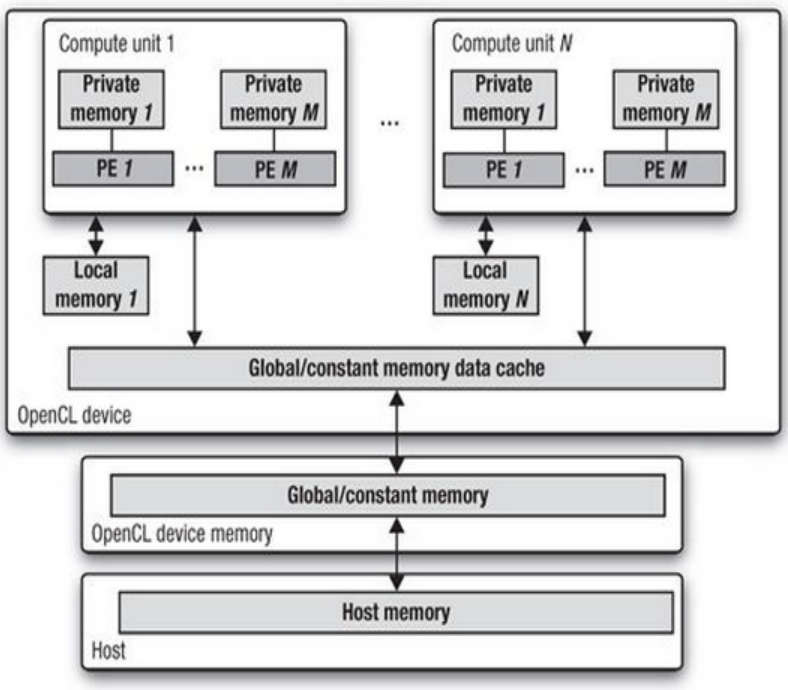


Figura 16: Regiões de memória de OpenCL (fonte: Munchi, 2011)

A interação entre o *host* e o modelo de memória pode ocorrer de duas maneiras: cópia explícita ou mapeamento de regiões de um objeto de memória. Na cópia explícita, comandos de transferência entre *host* e dispositivos são enfileirados na fila de comandos e podem ser executados de forma síncrona ou assíncrona. No método de mapeamento, os objetos de memória são mapeados na memória do *host*, que pode também realizar acessos a estes objetos. O comando de mapeamento também deve ser enfileirado na fila de comandos.

1.5.6 Exemplo: soma de nuvens de pontos em OpenCL

Os códigos a seguir apresentam um exemplo de soma de vetores em OpenCL. Este exemplo é baseado em um tutorial oferecido pelo OLCF (*Oak Ridge Leadership Computing Facility*), um dos maiores centros de processamento de alto desempenho do mundo (OLCF, 2012).

O primeiro código apresenta o código do *kernel*, que pode ficar num arquivo separado (.cl) ou pode ser formatado no próprio código como uma *string-C*. Este código será passado como argumento à função OpenCL que o compilará em tempo de execução.

```
01  __kernel void vecAdd(__global const float *a,
02                        __global const float *b,
03                        __global float *c,
04                        const unsigned int n)
05  {
06      int gid = get_global_id(0);
07      if (gid < n)
08            c[gid] = a[gid] + b[gid];
09  }
```

A partir deste código é possível observar algumas características de OpenCL:

- A definição de um kernel é feita através de uma função que utiliza o modificador **__kernel** (linha 01).
- O modificador **__global** indica que os parâmetros estão na memória global do dispositivo (linhas 01 a 03).
- A função **get_global_id()** retorna o identificador global da thread (*work item*) na dimensão 0 (linha 06).
- A verificação do identificador (linha 07) é comumente realizada neste tipo de computação, pois por motivos de desempenho é possível que threads a mais venham a ser lançadas. A verificação serve para que somente as threads "dentro do problema" executem o trabalho. Este tipo de verificação também é comum em CUDA.
- Na linha 08 a soma é realizada (*n* threads serão iniciadas e cada uma realizará uma soma).

O código a seguir apresenta a *main()* juntamente com funções auxiliares do OpenCL que devem ser executadas pelo *host*. Para reduzir o tamanho do código os testes de erro retornados pelas funções não foram realizados.

```
01  #include <stdio.h>
02  #include <stdlib.h>
03  #include <math.h>
04   #include <CL/opencl.h>
05   #include <iostream>
06   #include <fstream>
07   #include <sstream>
08
09
    int main( int argc, char* argv[] )
```

```
10  {
11     const char* filename; //Deve conter o nome do .cl
12
13     // Tamanho dos vetores
14      unsigned int n = 100000;
15
16      // Vetores de entrada no host
17      float *h_a;
18      float *h_b;
19      // Vetor de saída no host
20      float *h_c;
21
22      // Buffers de entrada no dispositivo
23      cl_mem d_a;
24      cl_mem d_b;
25      // Buffer de saída no dispositivo
26      cl_mem d_c;
27
28      cl_platform_id cpPlatform;        // OpenCL platform
29      cl_device_id device_id;           // device ID
30      cl_context context;               // context
31      cl_command_queue queue;           // command queue
32      cl_program program;               // program
33      cl_kernel kernel;                 // kernel
34
35      // Tamanho, em bytes, de cada vetor
36      size_t bytes = n*sizeof(float);
37
38      // Aloca memória para cada vetor no host
39      h_a = (float*)malloc(bytes);
40      h_b = (float*)malloc(bytes);
41      h_c = (float*)malloc(bytes);
42
43      // Inicializa vetores no host
44      int i;
45      for( i = 0; i < n; i++ )
46      {
47          h_a[i] = sinf(i)*sinf(i);
48          h_b[i] = cosf(i)*cosf(i);
49      }
50
51      size_t globalSize, localSize;
52      cl_int err;
53
54      // Número de work items em cada work group local
55      localSize = 64;
56
57      // Número total de work items
58      globalSize = ceil(n/(float)localSize)*localSize;
59
60      // Obtém a plataforma
61      err = clGetPlatformIDs(1, &cpPlatform, NULL);
62
63      // Obtém o ID do dispositivo (GPU)
64      err   =   clGetDeviceIDs(cpPlatform,   CL_DEVICE_TYPE_GPU,   1,
```

```
65  &device_id, NULL);
66
67      // Cria um contexto
68      context = clCreateContext(0, 1, &device_id, NULL, NULL, &err);
69
70      // Cria uma command queue
71      queue = clCreateCommandQueue(context, device_id, 0, &err);
72
73      //Lê o código fonte do kernel e transforma numa string-C
74       std::ifstream kernelFile(fileName, std::ios::in);
75       std::ostringstream oss;
76       oss << kernelFile.rdbuf();
77       std::string srcStdStr = oss.str();
78       const char *kernelSource = srcStdStr.c_str();
79      // Cria o programa
80      program = clCreateProgramWithSource(context, 1,
81                      (const char **) &kernelSource, NULL, &err);
82
83      // Compila o executável
84      clBuildProgram(program, 0, NULL, NULL, NULL, NULL);
85
86      // Cria o kernel
87      kernel = clCreateKernel(program, "vecAdd", &err);
88
89      // Cria os arrays de entrada e saída na memória do dispositivo
90     d_a =clCreateBuffer(context,CL_MEM_READ_ONLY,bytes,NULL,NULL);
91     d_b =clCreateBuffer(context,CL_MEM_READ_ONLY,bytes,NULL,NULL);
92     d_c =clCreateBuffer(context,CL_MEM_WRITE_ONLY,bytes,NULL,NULL);
93
94      // Escreve os dados no array de entrada do dispositivo
95      err = clEnqueueWriteBuffer(queue, d_a, CL_TRUE, 0,
96                                      bytes, h_a, 0, NULL, NULL);
97      err |= clEnqueueWriteBuffer(queue, d_b, CL_TRUE, 0,
98                                      bytes, h_b, 0, NULL, NULL);
99
100     // Envia argumentos ao kernel
102     err  = clSetKernelArg(kernel, 0, sizeof(cl_mem), &d_a);
102     err |= clSetKernelArg(kernel, 1, sizeof(cl_mem), &d_b);
103     err |= clSetKernelArg(kernel, 2, sizeof(cl_mem), &d_c);
104     err |= clSetKernelArg(kernel, 3, sizeof(unsigned int), &n);
105
106     // Executa o kernel em todo o intervalo de dados
107

    err = clEnqueueNDRangeKernel(queue, kernel, 1, NULL,
108                         &globalSize, &localSize,0, NULL, NULL);
109
110     // Espera que finalização do kernel antes de ler os resultados
111     clFinish(queue);
112
113     // Lê resultados a partir do dispositivo
114     clEnqueueReadBuffer(queue, d_c, CL_TRUE, 0,
115                                  bytes, h_c, 0, NULL, NULL );
116
117     //Soma o vetor c e imprime o resultado dividido por n, este
```

```
118 deve ser 1 com erro
119     float sum = 0;
120     for(i=0; i<n; i++)
121         sum += h_c[i];
122     printf("resultado final: %f\n", sum/n);
123 
124     // libera recursos do OpenCL
125     clReleaseMemObject(d_a);
126     clReleaseMemObject(d_b);
127     clReleaseMemObject(d_c);
128     clReleaseProgram(program);
129     clReleaseKernel(kernel);
130     clReleaseCommandQueue(queue);
131     clReleaseContext(context);
132 
133     //libera a memória do host
134     free(h_a);
135     free(h_b);
136     free(h_c);
137 
138     return 0;
139 }
140 
```

A seguir, destaca-se as principais características de OpenCL presentes no código:

- Na linha 55 é definido o *localSize* que é a quantidade de *work items* em cada *work group* – 64 neste caso. Isto equivale em CUDA a definir a quantidade de *threads* em cada grupo.

- A linha 59 define a quantidade total de *work items* lançados (*globalSize*). Num primeiro momento pensaríamos que este número deve ser igual ao tamanho do vetor (*n*). Porém, *globalSize* deve ser divisível por *localSize*, por isso o arredondamento realizado nesta linha.

- Entre as linhas 61 e 72 é realizado o *setup* do OpenCL: plataforma, dispositivo, contexto e fila de comandos (*command queue*).

- Entre as linhas 75 e 88 o *kernel* é lido e compilado.

- Entre as linhas 90 e 105 os dados são enviados ao *kernel* no dispositivo.

- Na linha 108 o *kernel* é enfileirado e por fim lançado no dispositivo.

- Na linha 112 o *host* espera a finalização do *kernel* (sincronização).

- Na linha 115 o resultado é lido da memória do dispositivo.

1.6. Comentários Finais

Conforme evidenciado no texto, um dos grandes motores propulsores da tecnologia atual de *Point Based Graphics* são GPUs, que disponibilizam, além de suporte a operações gráficas como modelagem, renderização e animação, suporte a operações como Física e Inteligência Artificial.

Esta área em jogos digitais, apesar de recente, tem oferecido diversas oportunidades de pesquisas tanto no desenvolvimento de novas técnicas e algoritmos para nuvens de pontos, como também de aplicação direta em *game engines*.

Espera-se, com o caráter introdutório deste texto, que as bases de nuvens de pontos tenham sido compreendidas, assim como as possibilidades de desenvolvimento tanto para o contexto gráfico como para contextos mais genéricos.

1.7. Referências

FARBER, R. **CUDA Application Design and Development**. New York: Morgan Kaufmann, 2011.

FARIAS, T.S.M.C., TEIXEIRA, J.M.N.X., LEITE, P.J.S., ALMEIDA, G.F., TEICHRIEB, V., KELNER, J. High Performance Computing: CUDA as a Supporting Technology for Next Generation Augmented Reality Applications. In: **RITA**, 16(1), 2009, pp. 71-96.

GASTER, B., HOWES, L., KAELI, D.R., MISTRY, P. **Heterogeneous Computing with OpenCL**. New York: Morgan Kaufmann, 2011.

GROSS, M., PFISTER, H. **Point-Based Graphics**. New York: Morgan Kaufmann, 2007.

KIRK, D.B., HWU, W.W. Programming Massively Parallel Processors: A Hands-on Approach. New York: Morgan Kaufmann, 2010.

LINSEN, L. **Point Cloud Representation**. Karlsruhe, Alemanha: Universität Karlsruhe, 2001.

MUNSHI, A., GASTER, B., MATTSON, T.G., FUNG, J., GISBURG, D. **OpenCL Programming Guide**. New York: Addison-Wesley Professional, 2011.

NVIDIA Corporation, **FERMI Whitepaper**, 2009.

NVIDIA Corporation, **NVIDIA CUDA C Programming Guide - 4.0**, 2011.

OLCF, **Oak Ridge Leadership Computing Facility Tutorial**, disponível em http://www.olcf.ornl.gov/training_articles/opencl-vector-addition/, acessado em abril de 2012.

OPENCV, **OpenCV GPU**, disponível em http://opencv.willowgarage.com/wiki/OpenCV_GPU, acessado em abril de 2012.

PCL. (2012). **Point Cloud Library**. Fonte: Point Cloud: http://pointclouds.org, Acesso em 01/08/2012.

SANDERS, J., KANDROT, E**. CUDA by Example: An Introduction to General-Purpose GPU Programming.** New York: Addison-Wesley, 2010**.**

SCARPINO, M. **OpenCL in Action: How to Accelerate Graphics and Computations**. New York: Manning Publications, 2011.

SILVEIRA, C.L.B., SILVEIRA, L.G.S. Programação Introdutória em OpenCL e Aplicações em Realidade Virtual e Aumentada. In: **Tendências e Técnicas em Realidade Virtual e Aumentada (Capítulo 3)**, Anais do SVR'2010, pp. 65-101.

STRINGHINI, D., SILVA, L. Programação em CUDA e OpenCL para Realidade Virtual e Aumentada. **In: Tendências e Técnicas em Realidade Virtual e Aumentada (Capítulo 1),** Anais do SVR'2012, pp. 1-35..

SINHA, S.N., FRAHM, J.M., POLLEFEYS, M., GENC, Y. **GPU-based Video Feature Tracking and Matching**. Relatório Técnico TR 06-012, Departamento de Ciência da Computação, Universidade da Carolina do Norte – Chapel Hill, 2006.

SIZINTESEV, M., KUTHIRUMMAL, S., SAMARASEKERA, S., KUMAR, R., SAWHNEY, H.S., CHAUDHRY, A. GPU Accelerated Real-time Stereo for Augmented Reality. In: **Proceedings of the 5th International Symposium 3D Data Processing, Visualization and Transmission (3DPVT)**, 2010.