UNIVERSIDADE FEDERAL FLUMINENSE
INSTITUTO DE COMPUTAÇÃO
CURSO DE GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

LEONARDO LOURES QUIRINO DA SILVA

# AUTÔMATOS CELULARES EM GPU

Simulação de dinâmica de partículas baseado em autômatos celulares desenvolvida para placas gráficas.

Niterói
2010

LEONARDO LOURES QUIRINO DA SILVA

# AUTÔMATOS CELULARES EM GPU

Simulação de dinâmica de partículas baseado em autômatos celulares desenvolvida para placas gráficas.

Monografia apresentada ao Curso de Graduação em Ciência da Computação da Universidade Federal Fluminense, como requisito parcial para obtenção do Grau de Bacharel em Ciência da Computação.

Orientadora: Profª. Dr. REGINA CÉLIA P. LEAL TOLEDO

Co-Orientador: Msc. MARCELO ZAMITH

Niterói
2010

LEONARDO LOURES QUIRINO DA SILVA

# AUTÔMATOS CELULARES EM GPU

Simulação de dinâmica de partículas baseado em autômatos celulares desenvolvida para placas gráficas.

Monografia apresentada ao Curso de Graduação em Ciência da Computação da Universidade Federal Fluminense, como requisito parcial para obtenção do Grau de Bacharel em Ciência da Computação.

Aprovada em 08 de Dezembro de 2010.

BANCA EXAMINADORA

_____

Profª. Dr. REGINA CÉLIA P. LEAL
Orientadora
UFF

_____

Msc. MARCELO ZAMITH
Co-Orientador
UFF

_____

Prof. Dr. ESTEBAN WALTER GONZALEZ CLUA
UFF

_____

Prof. Dr. OTTON TEIXEIRA DA SILVEIRA FILHO
UFF

Niterói
2010

AGRADECIMENTO

A Deus, por tudo que nos proporciona,

Aos pais, por seu amor. carinho e compreensão,

Aos amigos, pelo apoio,

Aos professores, pelo conhecimento e dedicação,

A todos que, direta ou indiretamente contribuíram, para a realização deste trabalho.

*"É tudo muito pequeno — coisas microscópicas ou menores ainda, partículas elementares — ou muito grande, como astros e estrelas. São mundos completamente invisíveis para nós, mas que são revelados pela ciência."*

Marcelo Gleiser

RESUMO

O uso de *hardware* de processamento gráfico para computação de propósito geral tem sido área de pesquisa por muitos anos (GPGPU – *general-purpose computing on graphics processing units*). Com a crescente liberdade de programação das Unidades de Processamento Gráfico (GPUs), estes *hardware* são capazes de executar mais do que apenas as tarefas gráficas para as quais foram desenvolvidos, podendo ser utilizados como coprocessadores em uma grande variedade de aplicações. Este trabalho segue esta linha de desenvolvimento, estudando os conceitos e estrutura das GPUs para desenvolver uma abordagem paralela de um modelo de dinâmica de partículas baseado em autômatos celulares.

Palavras-chave: Computação Paralela, Autômatos Celulares, Vizinhança Margolus.

ABSTRACT

The use of Graphics Processing Hardware for general purpose computing has been an area of research for many years (GPGPU – general-purpose computing on graphics processing units). With the growing programmability of the Graphics Processing Units (GPU), these hardware are capable of executing more than just the graphic tasks they were designed for, enabling them to be used like co-processors in many applications. This work follows this line of development, studying the concepts and structure of the GPUs to develop an parallel approach to a cellular automata particle dynamics model.

keywords: Parallel Computing, Cellular Automata, Margolus Neighborhood.

## SUMÁRIO

# LISTA DE ACRÔNIMOS

| | |
|---|---|
| CPU | Central Processing Unit |
| GPU | Graphic Processing Unit |
| SIMT | Single Instruction Multiple Threads |
| SIMD | Single Instruction Multiple Data |
| SPA | Stream Processors Array |
| TPC | Texture/Processor Cluster |
| DRAM | Dynamic Random Access Memory |
| ROP | Raster Operation Processor |
| MAD | Multiply/Add unity |
| SFU | Special Function Unity |
| SMC | Streaming Multiprocessor Controller |
| CTA | Cooperative Thread Array |
| API | Application Programming Interface |
| MAD | Multiply/Add |

LISTA DE ILUSTRAÇÕES

LISTA DE TABELAS

# 1 INTRODUÇÃO

No início da década de 1990, jogos eletrônicos com gráficos 3D em tempo real estavam se tornando cada vez mais populares em computadores e consoles, e o que era anteriormente visualizado pela CPU (C*entral Processing Unit*) ganhou um *hardware* dedicado, conhecido por GPU (*Graphic Processing Unit*). Com a finalidade de se criar *softwares* altamente interativos e imagens cada vez mais realistas, os jogos eletrônicos impulsionaram a criação de *hardwares* cada vez mais poderosos para visualização. Com isso, no final dos anos 90, o *pipeline* de visualização 3D fixo já havia sido totalmente implementado dentro do *hardware de placas gráficas*. Nestes *hardwares* havia dois conjuntos de processadores de fluxo: o primeiro conjunto para processamento de vértices e um outro conjunto para processamento de *pixels*.

Com a velocidade da evolução dos jogos eletrônicos, no ano 2001 surgiram os primeiros *hardwares* gráficos programáveis, permitindo aos desenvolvedores de jogos eletrônicos maior liberdade através de pequenos programas escritos em linguagem *shader*. A linguagem shader é a linguagem padrão do mercado específica para o tratamento de dados dentro das GPUs que possuem hardware programável. Neste trabalho serão usados freqüentemente termos cuja origem vem do uso das funções originais das GPUs para aplicações gráficas, mas que são necessárias para o discorrer sobre estes hardwares. Assim serão citados texturas, *shaders*, etc. por herança.

A linguagem *shader* permite definir um modelo específico de processamento de vértices (*vertex shader*), carregado nos processadores de vértices e de *pixel* (*pixel shader*), carregado nos processadores de *pixel*. O primeiro trata de questões como: transformações, geometria, texturas e demais operações aplicadas sobre os vértices. O segundo trata todas as operações aplicadas sobre o *pixel*: *Bump mapping*, *shadow mapping*, HDR (*High Dinamic Range, uma técnica para simular a abertura da córnea e sua exposição à luz ambiente*).

A indústria do entretenimento adota três linguagens *shaders:* CG – C *for Graphics* da NVIDIA compatível tanto para OpenGL quanto para DirectX e disponível para Windows, Linux e MAC OS, GLSL – OpenGL *Shader Language* integrada a própria API do OpenGL a partir da versão 1.5 e HLSL – *High Level Shader Language* incorporada ao SDK do DirectX a partir da versão 8.

O grande poder de processamento massivamente paralelo, sendo cada vez menos específico, deste tipo de *hardware* vem incentivando trabalhos sobre o seu uso para processamento genérico. Como a programação de GPU era feita apenas por linguagem *shader*, os primeiros trabalhos nesta área exigiam o uso de uma das APIs (*Application Programming Interface*) gráficas (OpenGL e DirectX). Esta abordagem exigia que os dados fossem expressos na forma de fluxo, isto é, os dados eram modelados como vértices ou como *pixels*, bem como era exigido uma sobrecarga de instruções da API gráfica utilizada. Com o surgimento da arquitetura unificada, onde havia apenas um conjunto de processadores de fluxo menos especializado, que trabalhava tanto com processamento de vértices como de *pixels*, os fabricantes passaram a desenvolver bibliotecas baseadas na linguagem C que funcionam como uma extensão da mesma: A NVIDIA com a API CUDA (*Compute Unified Device Architecture*) [01], AMD com a API CAL (*Compute Abstract Layer*) [05] e OpenCL [06]. Estas tecnologias são abstrações de alto nível que provêm a utilização da GPU para processamento genérico.

A GPU é composta por unidades processadores de fluxo. Atualmente uma GPU pode conter centenas destas unidades, o que as torna processadores de múltiplos núcleos com suporte a inúmeras tarefas de computação, chamadas *threads*. Com isso as GPUs possuem um alto grau de paralelismo, com uma banda de memória e desempenho em operações aritméticas superiores aos das CPUs.

Com sua crescente popularidade e capacidade de programação, muitos programas de diferentes áreas comerciais também começaram a se aproveitar de suas capacidades. Esses programas vão da área multimídia, como tocadores e codificadores de vídeo e/ou som e manipulação de imagens, à até mesmo programas financeiros. Também se aproveitando dessa capacidade, muitos jogos já usam a GPU além da visualização gráfica, processando simulações físicas [17] e algoritmos de análise financeira [18].

Este projeto foi desenvolvido visando o estudo da utilização das GPUs no processamento de autômatos celulares. Para isso, no capítulo 2 iremos abordar a arquitetura das GPUs Tesla, desenvolvida pela Nvidia e que são programáveis através da linguagem CUDA, uma extensão de C. No capítulo 3 faremos uma pequena apresentação sobre os

conceitos básicos de Autômatos Celulares e exemplos de autômatos comumente utilizados. No capítulo 4 será apresentado o projeto em si, uma abordagem de paralelização de um sistema de autômato celular em GPU, as regras e o modelo utilizado no projeto. Por fim são apresentados os testes de execução do algoritmo.

# 2 UNIDADES DE PROCESSAMENTO GRÁFICO

## 2.1 INTRODUÇÃO

As primeiras GPUs continham um *pipeline* de função fixa de iluminação e transformação de vértices que trabalhava com pontos flutuantes de 32 *bits* e um *pipeline* de função fixa de processamento de *pixel* que trabalhava com inteiros. A partir de 2001 surgiram as primeiras GPUs com processadores de vértices programáveis que executavam pequenos trechos de código chamados de *shaders* de vértices, e um *pipeline* de processamento de *pixel* configurável. Em 2005, um console de jogos eletrônicos, o Xbox360, trouxe a primeira GPU unificada, permitindo o processamento de vértice e *pixel* em um mesmo processador.

Processadores de vértice operam nos vértices de primitivas como pontos, linhas e triângulos. As operações básicas incluem transformação de suas coordenadas para coordenadas no espaço da tela. Estas coordenadas são repassadas para as unidades de ajuste e rasterização, onde são ajustados os parâmetros de iluminação e textura, usados pelo processador de *pixel* para o preenchimento das primitivas.

Processadores de vértice e *pixel* evoluíram em velocidades diferentes: processadores de vértice foram desenvolvidos para operações matemáticas de alta precisão com maior tempo de processamento, enquanto processadores de *pixel* foram otimizados para filtragem de texturas com baixa precisão matemática e um tempo de processamento menor. Processadores de vértice tradicionalmente executavam um tipo de processamento mais complexo. Devido a isto, tornaram-se programáveis primeiro. Os dois tipos de processadores, de vértices e *pixels*, convergiram devido a esta necessidade de generalização de sua programação. Além disso, essa crescente generalização também aumentou a complexidade de sua arquitetura, a área do chip e o custo no desenvolvimento de dois chips separados.

Pelo fato das GPUs tipicamente processarem mais *pixels* do que vértices, processadores de *pixel* eram em geral mais numerosos do que processadores de vértice, numa

razão de três para um. No entanto, a carga de trabalho dificilmente é balanceada desta forma, levando a ineficiência. Isto pode ocorrer quando há triângulos grandes, subtilizando os processadores de vértice, enquanto os processadores de *pixel* são usados ao máximo, e com triângulos pequenos o contrário pode vir a acontecer. A adição de processamento de primitivas geométricas, como pontos, retas e etc., ainda mais complexas, tratando de primitivas inteiras como um objeto único, torna a escolha de uma razão fixa entre processadores de vértice e *pixel* ainda mais difícil. Estes fatores influenciaram a decisão de se utilizar uma arquitetura unificada. Isto levaria à possibilidade de se balancear a variação de prioridade nas cargas entre processamento de *pixel* e vértice.

A generalização dos processadores gráficos abriu a porta para uma nova capacidade de processamento paralelo baseado em GPUs. Os arquitetos de *hardware* desenvolveram os recursos gráficos da arquitetura de forma coordenada com o desenvolvimento da nova API DirectX 10, e em paralelo desenvolveram recursos de computação de propósito geral de forma coordenada com o desenvolvimento do compilador, ferramentas e linguagem de programação paralela CUDA C.

Este projeto se baseia nos conceitos da arquitetura unificada da Nvidia chamada Tesla, que foi utilizada em suas GPUs GeForce série 8000, 9000 e 200. Foi lançada uma nova arquitetura, chamada Fermi, juntamente com uma nova versão da API DirectX, mas esta não será abordada neste projeto pois não faz parte do escopo, nem do hardware utilizado no desenvolvimento.

## 2.2 CONCEITOS BÁSICOS

A cada nova geração de GPUs, a discrepância de capacidade de processamento de ponto flutuante aumenta entre CPU e GPU. Isso ocorre, pois as GPUs foram desenvolvidas especificamente para tarefas de computação intensiva e altamente paralelas, justamente o que a visualização em tempo real necessita. Para alimentar uma grande quantidade de dados em paralelo, as GPUs também dispõem de uma largura de banda de memória muito maior em comparação com as CPUs. A figura 1 apresenta uma comparação básica entre o desempenho em operações de ponto flutuante de CPUs e GPUs.

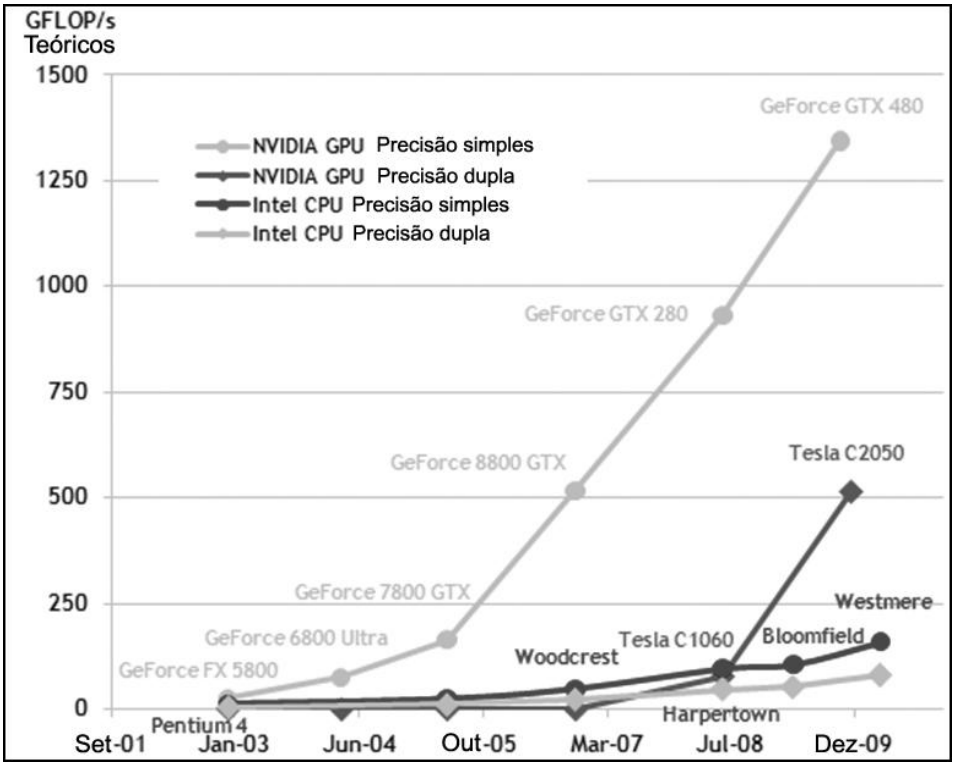


Figura 1: Operações de ponto flutuante para GPUs e CPUs. [1]

A arquitetura Tesla e outras arquiteturas de GPU atuais são baseadas em um vetor de processadores escaláveis, que possuem apenas uma unidade de multiplicação e adição,, e por isso, têm em seus desenhos, mais transistores voltados para o processamento dos dados do que para memória *cache* e controle de fluxo, como nas CPUs. As GPUs são adaptadas de forma específica para lidar com problemas que podem ser expressos como computação de dados em paralelo. O mesmo programa é executado em diversos elementos em paralelo com grande intensidade de operações aritméticas, ou seja, com uma grande razão entre operações aritméticas e operações de memória, de forma que a latência no acesso à memória pode ser contornada com operações de cálculo em vez de memória *cache*. Já que o mesmo código será executado para diversos elementos em paralelo não há tanta necessidade para controle de fluxo individual, diminuindo a complexidade da arquitetura e salvando espaço no projeto de hardware. Na figura 2 pode-se visualizar um esquema básico das arquiteturas de CPUs e GPUs, tornando clara a diferença entre as duas.

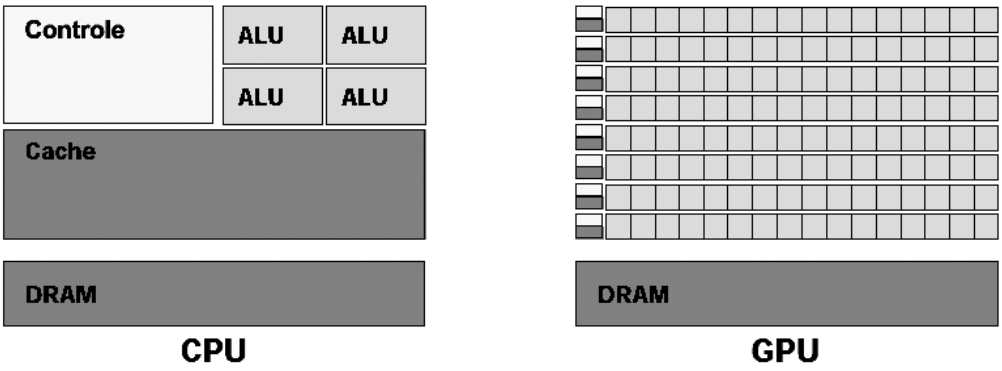


Figura 2: Diferença de arquitetura entre GPU e CPU. [1]

O processamento de dados em paralelo funciona mapeando os elementos de dados para diversas *threads*, ou tarefas de computação, que executam em paralelo. Muitas aplicações que processam grandes conjuntos de dados podem ser implementados no modelo de dados em paralelo para acelerar suas execuções. Na visualização 3D, grandes conjuntos de *pixels* e vértices são mapeados para diversas *threads* que executam em paralelo. De forma similar, o processamento de aplicações de imagem e mídia, como processamento de imagens e codificação e decodificação de vídeo podem mapear blocos de imagem e *pixels* para *threads* em paralelo. Muitas outras aplicações fora da área de processamento e visualização de imagens são acelerados quando desenvolvidos com o modelo de dados em paralelo em mente, desde processamento de sinais ou simulações físicas até finanças ou biologia computacional.

### 2.2.1 Arquitetura

O processador da GPU é construído em volta de um vetor escalável de multiprocessadores de fluxo (conhecidos como *Streaming Multiprocessors* ou *SMs*). Quando um programa na CPU chama uma função de GPU, chamada *kernel*, são criadas e enumeradas *threads* que são executadas cada uma com seu conjunto de dados, e que são distribuídas para os multiprocessadores que se encontram disponíveis.

Para gerenciar centenas de milhares de *threads* executando, o multiprocessador emprega uma arquitetura chamada SIMT *(Single Instruction Multiple Threads*). A arquitetura SIMT remete à SIMD (*Single Instruction Multiple Data*) [19]. As duas arquiteturas trabalham de formas similares, executando cálculos em diversos dados em paralelo, mas a arquitetura SIMT possui como diferença uma abstração ao paralelismo SIMD original. A organização SIMD expõe a largura das instruções SIMD ao software enquanto a SIMT gerencia implicitamente o nível de paralelismo através de *threads*. As *threads* SIMT são criadas via *hardware*, de forma transparente ao desenvolvedor, e executam instruções escalares, de forma

a abstrair o tratamento do paralelismo dos dados, fazendo com que cada parte do conjunto de dados seja endereçado como um dado da *thread* a executar, dessa forma é alcançada uma escalabilidade automática na execução dos programas, pois de acordo com o número de processadores de fluxo disponíveis na GPU uma função será executada mais rapidamente, já que serão criadas mais *threads* em paralelo.

O multiprocessador mapeia cada *thread* para um núcleo escalar do processador, e cada *thread* executa independentemente com seus próprios endereços de instrução e registradores. A unidade SIMT do multiprocessador cria, gerencia, escalona e executa *threads* em grupos de trinta e duas *threads* paralelas, chamadas *Warps*. *Threads* individuais que compõem o mesmo *warp* iniciam no mesmo endereço de programa. Caso as *threads* encontrem um desvio condicional onde cada uma siga por um desvio diferente, estas *threads* são livres para divergir por estes desvios e executar independentemente. Isso é feito, serializando a execução das *threads* por cada desvio encontrado.

A figura 3 mostra um diagrama de bloco de uma GPU GeForce 8800, com 128 núcleos de processadores escalares (SP), organizados em 16 multiprocessadores de fluxo (SMs) em oito unidades de processamento independentes, chamadas de *Cluster* de processadores/texturas (TPC). Os TPCs são agrupados como um vetor de processadores de fluxo (SPA), e são responsáveis por todos os cálculos programáveis da GPU. O sistema de memória consiste em chips DRAM externos ao processador da GPU e processadores de operações de rasterização (ROPs), que realizam, diretamente na memória, operações de colorização e de ordenação de profundidade no *buffer* de quadros. Uma rede de intercomunicação transporta valores de cor e profundidade de *pixel*/fragmentos para os ROPs. Essa rede também conecta requisições de leitura da memória de texturas do SPA à DRAM e dados de textura da DRAM à memória *cache* de nível 2.

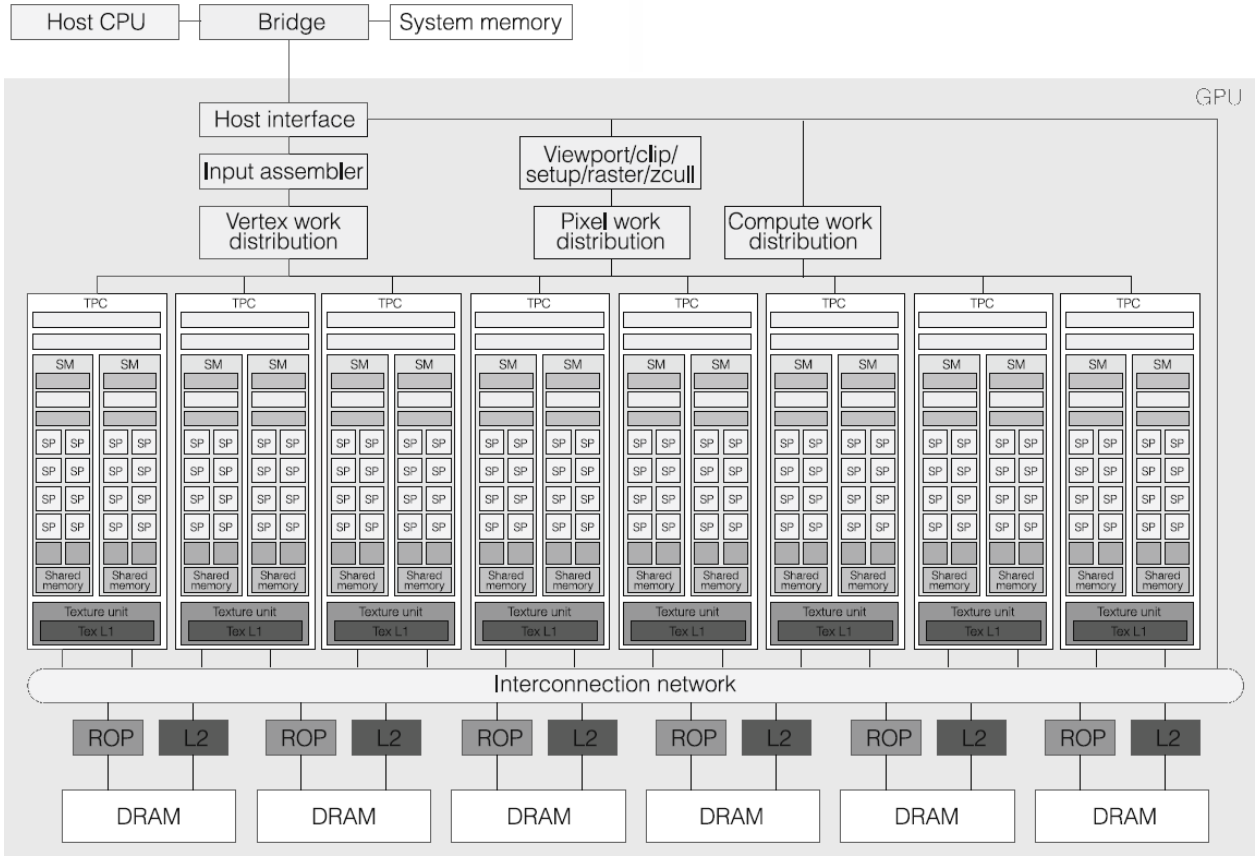


Figura 3: Arquitetura de GPU Tesla. [2]

Os blocos superiores na figura 3 são a interface de comunicação com a CPU do sistema. Através deles passam os dados de tarefas para o SPA. A interface de hospedeiro (*Host Interface*) é a ponte de comunicação com a CPU do sistema, e responde a comandos desta, busca dados da memória principal, verifica a consistência de comandos e realiza a troca de contexto.

O montador de entradas (*Input Assembler*) coleta vértices vindos do fluxo de entrada. O bloco de distribuição de trabalho de vértice distribui pacotes de trabalho de vértices para as várias TPCs no SPA. Os TPCs executam *shaders* de vértice, e (se habilitado) *shaders* de geometria. O resultado é escrito em *buffers* dentro do próprio chip e passado para o bloco de rasterização, tornando-se fragmentos de *pixel*. A unidade de distribuição de trabalho de *pixel* (*pixel work distribution*) distribui fragmentos de *pixel* para os TPCs apropriados para processamento, e então são enviados através da rede de intercomunicação para processamento de cor e profundidade nas unidades ROP. O bloco de distribuição de trabalho de computação envia vetores de *threads* para os TPCs. O SPA aceita e processa múltiplos fluxos simultaneamente. Múltiplos domínios de *clocks* para as unidades da GPU, processadores, DRAM e outras unidades permitem otimização de desempenho e gasto de energia.

2.2.1.1 Multiprocessadores de Fluxo

O SPA executa *threads* de *shaders* gráficos e programas de computação, e provem controle e gerenciamento de *threads*. Na figura 4 é apresentado o esquema de um TPC e seus dois SPA.

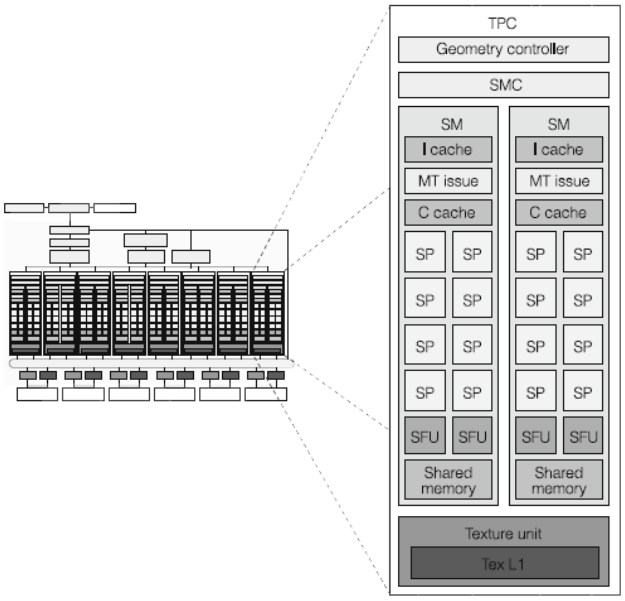


Figura 4: TPC e seus multiprocessadores de fluxo.[2]

Um multiprocessador consiste de:

- Oito núcleos de processadores escalares (SP);
- Duas unidades especiais para funções transcendentais, funções que não satisfazem uma equação polinomial, como Seno e Cosseno;
- Uma unidade de busca e entrega de instruções com suporte à múltiplas *threads*;
- Uma *cache* de instruções;
- Uma *cache* de constantes de somente leitura;
- Uma memória compartilhada de 16 *Kbyte*;
- Uma *cache* de texturas de somente leitura que é compartilhada por todos os SPs e acelera a leitura do espaço de memória de texturas, uma região de somente leitura na memória DRAM. [2]

O multiprocessador cria, gerencia e executa *threads* concorrentes via *hardware* com nenhum *overhead* de escalonamento, também é capaz de realizar sincronização de *threads* através de barreiras, intrinsecamente em apenas uma única instrução.

Cada núcleo SP contém uma unidade escalar de multiplicação/adição (MAD), sendo assim, cada SM possui oito unidades MAD. O SM usa suas duas unidades de funções especiais (SFU) para calcular funções transcendentais (funções trigonométricas, por exemplo)

e interpolação de atributos – a interpolação de atributos de vértice definindo uma primitiva em atributos de *pixel*. Cada SFU também possui quatro multiplicadores de ponto flutuante. O SM usa a Unidade de Textura do TPC como uma terceira unidade de execução e usa o Controlador de SM (SMC) e as unidades ROP para realizar acessos à memória.

Cada SM gerencia grupos de até 24 *warps*, chegando ao total de 768 *threads*. A cada tempo de instrução, a unidade de instruções escalona um *warp* que esteja pronto e busca e propaga de forma síncrona a próxima instrução para as *threads* ativas desse *warp*. *Threads* individuais podem estar inativas devido à divergência em desvios condicionais.

Um processador SIMT alcança seu desempenho máximo quando todas as 32 *threads* de suas *warps* seguem o mesmo fluxo de execução. Se uma *thread* diverge devido à um desvio condicional, a *warp* irá executar cada fluxo de cada desvio de forma seqüencial, desabilitando as *threads* que não estão executando no fluxo atual, até que todas as *threads* convirjam. Divergência entre *threads* só ocorre dentro de um *warp*, *warps* diferentes executam de forma independente mesmo que estejam executando o mesmo código, devido à isso, a arquitetura Tesla é muito mais eficiente e flexível em lidar com desvios do que GPUs de gerações passadas, pois suas 32 *threads* de cada *warp* são muito mais estreitas do que as antigas unidades SIMD das GPUs passadas.

## 2.3 MODELO DE PROGRAMAÇÃO CUDA

Para mapear de forma eficiente um problema de computação muito grande para uma arquitetura de processamento paralelo, o programador decompõe o problema em problemas menores, que podem ser resolvidos em paralelo.

CUDA é uma extensão das linguagens de programação C e C++. O compilador CUDA apenas procura e compila código específico CUDA, e deixa todo o resto para o compilador C padrão trabalhar e montar o programa final. Através desta extensão, o programador pode criar funções especiais, chamadas *kernel*, que são executadas na GPU. A GPU trabalha como um coprocessador, pois é necessário que o programa inicie pela CPU para que então seja possível chamar um *kernel*.

Para se executar um *kernel* CUDA, é necessário que a memória global da GPU seja previamente inicializada, alocando espaço em memória e iniciando variáveis com os valores necessários. A chamada do *kernel* leva então o contexto da função à GPU, que inicia e se configura para executar o *kernel*.

Esta configuração do *kernel* na GPU é uma abstração que é exposta ao programador como um pequeno conjunto de extensões à linguagem. Dessa forma o programador é guiado a dividir o problema em subproblemas que podem ser resolvidos por blocos de *threads*, e cada subproblema, em partes menores, que podem ser resolvidas cooperativamente em paralelo por *threads* dentro de um bloco.

Esta decomposição de problemas permite também uma escalabilidade automática dos programas CUDA. Cada bloco de *threads* pode ser designado para qualquer um dos núcleos de processador disponíveis em qualquer ordem, concorrentemente ou sequencialmente, fazendo com que um programa CUDA sempre tire proveito da quantidade total de núcleos de processador na GPU. A figura 5 exemplifica como se dá a distribuição de trabalho dentro da GPU para que essa escalabilidade seja alcançada.

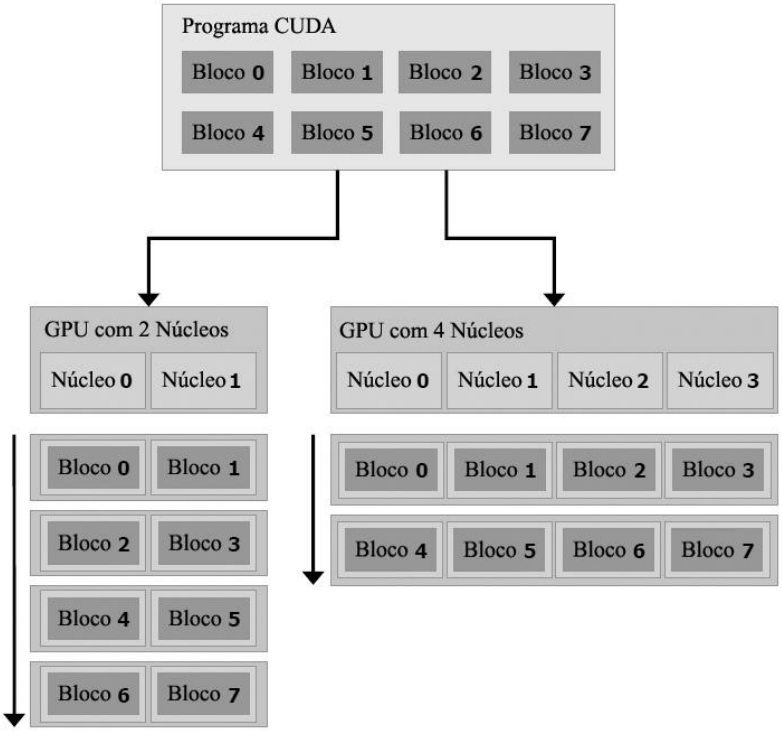


Figura 5: Blocos de *Threads* e escalabilidade de programas CUDA. [1]

A seguir serão introduzidos os principais conceitos do modelo de programação CUDA.

### 2.3.1 Hospedeiro e Dispositivo

É padrão da literatura sobre a arquitetura em questão, a utilização dos termos ‘hospedeiro’ (*host*) e ‘dispositivo’ (*device*) para se referir à CPU e à GPU. O modelo de

programação CUDA assume que as *threads* CUDA são executados em um dispositivo separado do resto do conjunto do computador, operando como um co-processador do sistema.

Essa separação ocorre, pois o modelo *CUDA* assume que tanto o hospedeiro quanto o dispositivo mantém suas próprias *DRAM* [1], as quais são referenciadas como 'memória do hospedeiro' e 'memória do dispositivo'. Portanto, um programa gerencia os espaços de memória global, de constantes e de texturas através de chamadas CUDA. Isso inclui chamadas de alocação e desalocação de memória, assim como transferências de dados entre as memórias de hospedeiro e de dispositivo.

### 2.3.2 Funções *Kernel*

CUDA extende a linguagem C. Isso permite aos programadores definirem *kernels*, funções escritras em linguagem C específicas para GPUs que quando chamadas, são executadas uma vez por cada N *threads* em paralelo. Um *kernel* é definido usando o qualificador de função **__global__** e o número de *threads* para cada chamada é especificado envolvendo os valores na nova sintaxe <<<...>>> utilizada ao se realizar a chamada à uma função CUDA a partir do programa hospedeiro. Também é possível definir funções internas à GPU, utilizando o qualificador de função **__device__**. Apesar disso, este é um recurso apenas para manter o código mais legível, pois estas funções são do tipo *inline*, o que significa que o compilador copia todo seu conteúdo para onde é realizada a chamada. Isso ocorre pois GPUs não possuem pilha de execução.

#### 2.3.2.1 Hierarquia de *threads*

Diferente do modelo de programação de gráficos, que executam *threads* de *shaders* independentes, o modelo de computação paralela requer que as *threads* se comuniquem, sincronizem-se, compartilhem dados e cooperem para chegar a um resultado. Para facilitar o gerenciamento de um grande número de *threads* que cooperam entre si, o modelo Tesla introduz o conceito de vetor de *threads* cooperativas (CTA), chamado de bloco de *threads* na terminologia CUDA.

*Threads* de um mesmo bloco podem cooperar entre si compartilhando dados através da memória compartilhada e sincronizando suas execuções para coordenar acessos à memória. Para que essa cooperação seja efetuada eficientemente, é esperado que *threads* de um mesmo bloco executem em um mesmo multiprocessador e, por isso, o número de *threads* por bloco é limitado pelos recursos de memória de um SM. Nas *GPUs* de arquitetura Tesla, um bloco de *threads* pode conter até 512 *threads* [1].

Para se criar domínios vetoriais, ou matrizes de duas ou três dimensões, é criada para cada *thread* uma identificação única, acessível de dentro do *kernel* através da variável implícita *threadIdx*, que é um vetor com 3 componentes. Desta forma as *threads* podem ser identificadas através de índices de até 3 dimensões, formando blocos de *threads* de até 3 dimensões. Isso fornece um meio natural de realizar a computação através dos domínios.

O índice de uma *thread* e seu identificador se relacionam diretamente. Para um bloco de uma dimensão, os valores são os mesmos, para um bloco bidimensional de tamanho (Dx, Dy), o ID da *thread* com um índice (x,y) é (x + yDx), de forma semelhante, para um bloco de três dimensões de tamanho (Dx, Dy, Dz) o ID será (x + yDx + zDxDy).

Como exemplo, o código a seguir soma as componentes de duas matrizes A e B de tamanho N x N e armazena o resultado na Matriz C.

O índice de uma *thread* e seu ID se relacionam diretamente. Para um bloco de uma dimensão, os valores são os mesmos, para um bloco bidimensional de tamanho (Dx, Dy), o ID da *thread* com um índice (x,y) é (x + yDx), de forma semelhante, para um bloco de 3 dimensões de tamanho (Dx, Dy, Dz) o ID será (x + yDx + zDxDy). A figura 6 exemplifica o agrupamento de blocos e *threads*.

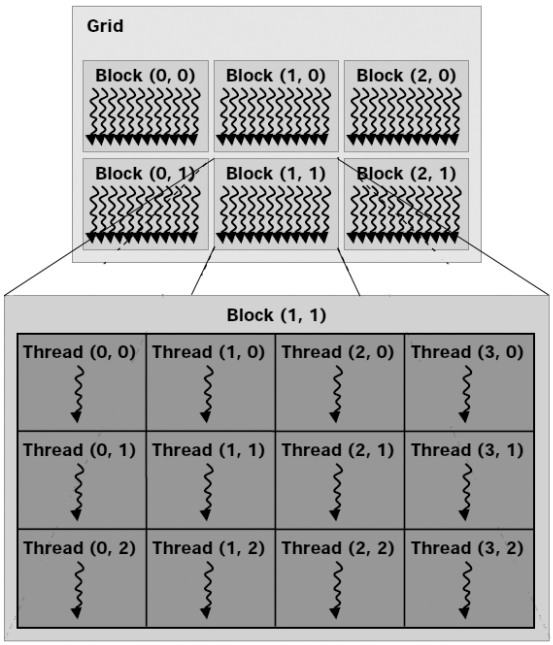


Figura 6: Hierarquia de *Threads*.[1]

Mesmo havendo o limite de apenas 512 *threads* por bloco, um *kernel* pode ser executado por múltiplos blocos de mesmo tamanho, assim o número de *threads* executado é igual ao número de *threads* por bloco vezes o número de blocos. Esses múltiplos blocos são

organizados em grades de uma ou duas dimensões. A dimensão dessas grades de blocos é especificada pelo primeiro parâmetro dentro da nova sintaxe <<<...>>>, onde são passados parâmetros de configuração do *kernel* a ser executado na chamada. Cada bloco dentro da grade pode ser identificado por um índice de mesma dimensão que a grade, acessível de dentro do *kernel* através da variável implícita *blockIdx*. A dimensão do bloco de *threads* é acessível de dentro do *kernel* através da variável implícita *blockDim.*

2.3.2.2 Hierarquia de Memória

Há vários níveis de paralelismo no modelo de computação na GPU:

- *Threads*: Executam em elementos selecionados pelo seu identificador.
- Blocos de *threads*: Executa em blocos de elementos, gerando blocos de resultados selecionados pelo sua identificador.
- Grade: Executa em diversos blocos de elementos, e grades seqüenciais executam passos seqüencialmente dependentes.

Níveis mais altos de paralelismo incluem mais GPUs por CPU e *clusters* de com mais GPUs por nó.

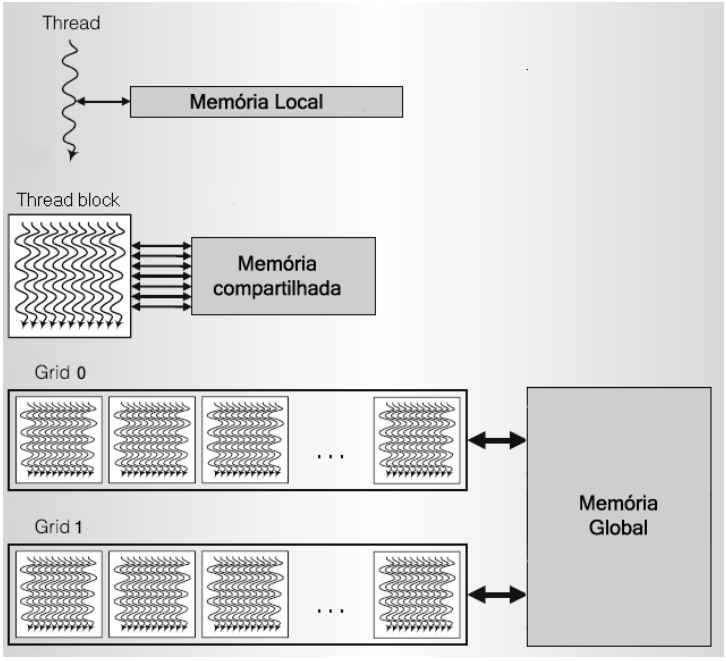


Figura 7: Hierarquia de memória.[1]

De forma semelhante, as *threads* podem acessar dados de diversos espaços de memória, como ilustrado pela figura 7. Cada *thread* possui uma memória local, individual e

privada. Cada bloco de *threads* possui uma memória visível a todas as suas *threads* e que tem o mesmo tempo de vida que o bloco. Por fim, todas as *threads* tem acesso à mesma memória global.

Há mais dois espaços de memória adicionais de somente leitura, acessível por todas as *threads,* as memórias de constantes e de textura. As memórias global, de constante e de textura são otimizadas para diferentes usos. Essas memórias também são persistentes entre *kernels* de uma mesma aplicação, podendo ser usadas para compartilhamento de dados entre *kernels*.

A comunicação de *threads* dentro de um bloco usa uma instrução de sincronização em barreira para esperar que operações de escrita que estejam sendo feitas na memória compartilhada ou global sejam terminadas antes que uma operação de leitura seja iniciada. O sistema de escrita/leitura de memória usa uma ordem de memória relaxada [1]que preserva a ordem das operações de escrita e leitura em um mesmo endereço, de uma mesma *thread* e do ponto de vista da coordenação de *threads* de um bloco usando a instrução de sincronização em barreira.

O desempenho do uso da largura de banda de cada espaço de memória depende significantemente do padrão de acesso à memória. A memória local e a global não possuem cache e são, assim, acessadas diretamente. Estes acessos são custosos em questão de tempo computacional, e por isso necessitam de certos cuidados. Para aperfeiçoar a utilização da largura de banda da memória e reduzir o *overhead* de múltiplos acessos, as instruções de escrita e leitura devem ser agrupadas entre *threads* de um mesmo *warp*. Os endereços de acesso devem pertencer a um mesmo bloco de memória e satisfazer um critério de alinhamento. No caso da memória local, todos os acessos são agrupados, já que esse espaço de memória é individual para cada *thread*.

A GPU é capaz de ler palavras de 4, 8 ou 16 bytes da memória global e escrever nos registradores em apenas uma instrução, tornando o acesso mais eficiente quando os acessos simultâneos de *threads* em um meio *warp* podem ser agrupados em uma única instrução de transação de memória de 32, 64, ou 128 bytes.

[1] O processador não necessita de resposta para operações de escrita, podendo continuar executando outras instruções mesmo que a operação de escrita não tenha sido finalizada na memória.

# 3 AUTÔMATOS CELULARES

## 3.1 INTRODUÇÃO

Autômatos celulares são sistemas dinâmicos discretos cujo comportamento é especificado em termos de uma relação local. Um autômato celular pode ser pensado como um universo estilizado, o espaço é representado por uma grade homogênea, com cada célula desta grade podendo assumir um número finito de estados. O tempo também é discretizado em intervalos de tempo e o funcionamento deste sistema é regido por um conjunto finito de regras, chamadas de regras de transição A cada passo de tempo, todas as células computam seu novo estado através das regras de transição dependendo apenas de seu estado atual e do estado de suas células vizinhas. Assim, autômatos celulares são um modelo matemático para sistemas naturais e complexos, compostos por um grande número de componentes idênticos com interações locais limitadas. Foram concebidos originalmente por Von Neumann como estruturas formais para modelar máquinas auto-reprodutoras

Formalmente, um autômato pode ser definido por ($Z^d$, Q, X, $f$), onde:

- Z é um conjunto de inteiros;
- Q é um conjunto finito de estados;
- X é o formato da vizinhança;
- $f$ é uma função de transição de estados; e,
- $Z^d$ é um espaço celular de *d* dimensões, no qual cada elemento é uma célula.

O comportamento de um autômato celular é determinado pela função de transição de estados que fixa o próximo estado de cada célula baseado no estado atual de seus vizinhos. O mapeamento do conjunto de células para um conjunto de estados é responsável pela configuração do espaço celular. A função $f$ é definida como um produto cartesiano de dimensão $n$ do conjunto de estados: $f:Q^n \rightarrow Q$. Dessa forma, sendo z o estado atual de uma

célula, por $c^t(z)$, seu próximo estado, $c^{t+1}(z)$, pode ser obtido através da função de transição, considerando o estado de suas células vizinhas $z_1$, $z_2$,...,$z_n$ da seguinte forma: $c^{t+1}(z) = f(c^t(z+z_1), c^t(z+z_2), ..., c^t(z+z_n))$.

## 3.2 ESTRUTURA

A seguir serão esclarecidos alguns dos conceitos básicos de autômatos celulares

### 3.2.1 Geometria

Através da definição formal, vê-se que um autômato celular deve possuir um espaço celular de *d* dimensões, no qual cada elemento é uma célula. Ou seja, é necessário que o autômato celular possua um espaço de geometria regular, onde se mantenha a mesma configuração por todo o espaço.

Essa geometria pode ser classificada quanto a dois aspectos: a dimensão e o formato da célula.

#### 3.2.1.1 Dimensão espacial

Um autômato celular pode ser construído com uma, duas, três dimensões. Em um autômato unidimensional, as células são distribuídas seqüencialmente como em um vetor. Em um autômato bidimensional são distribuídas como um plano, ou uma matriz de duas dimensões, o mesmo ocorre para autômatos de três ou mais dimensões. A figura 8 exibe exemplos visuais básicos da dimensão de autômatos celulares.

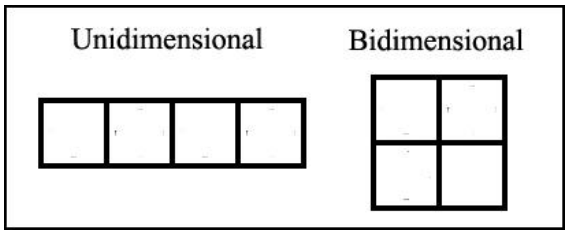


Figura 8: Dimensões de um autômato.

#### 3.2.1.2 Formato da célula

Em relação à distribuição de clélulas, estas podem ter vários formatos (triangular, quadrangular, hexagonal) contanto que essa distribuição seja homogênea em todo o espaço do autômato. Na figura 9 pode-se observar dois exemplos de formatos de célula.

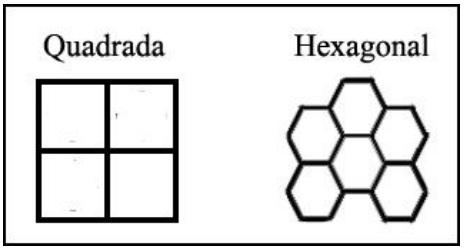


Figura 9: Exemplos de formato de célula.

### 3.2.2 Conjunto de Estados

Todo autômato celular deve ter um conjunto finito de estados possíveis para suas células. Este conjunto de estados pode ser qualquer tipo de conjunto, como cores (vermelho, azul e amarelo), números naturais (0, 1, 2 e 3), etc., sendo que cada célula é limitada a encontrar-se em somente um estado do conjunto a cada passo. Deve-se chamar atenção ao fato de que não necessariamente haverá pelo menos uma célula em determinado estado. É possível que em um determinado passo, um estado não ocorra em nenhuma célula, vindo a ocorrer em outros instantes da execução.

### 3.2.3 Regras de Transição

O conjunto de regras de transição especifica a função de transição de estados do autômato celular e é uma das partes mais importantes da especificação de um autômato, pois é ela que define a evolução do sistema de acordo com a passagem do tempo. Essas regras devem refletir a dinâmica do problema que se deseja modelar.

As regras são especificadas de forma a utilizar os dados disponíveis à célula atual, seu estado e o estado das células vizinhas, para definir seu estado no próximo intervalo de tempo. Dessa forma, conclui-se que há uma dependência muito forte entre as regras necessárias para se alcançar a evolução necessária, e a vizinhança necessária para implementar as regras desejadas.

#### 3.2.3.1 Vizinhança

O estado de uma célula em um intervalo de tempo é dado por uma função de transição através do seu estado anterior e o estado de seus vizinhos. O tipo de vizinhança depende da dimensão do autômato, do formato de sua célula e mais importante, das regras de transição utilizadas no sistema. Há diversos tipos de vizinhanças que podem ser implementadas em um autômato celular, e para cada um desses tipos, pode-se variar o tamanho da vizinhança. Dessa forma são alteradas a quantidade e a disposição dos vizinhos de uma célula.

A figura 10 apresenta para um autômato celular bidimensional, dois tipos de vizinhanças com tamanhos diferentes com r =1 e r=2, sendo *r* o tamanho do raio. As células alvo são representadas pela cor preta, e seus vizinhos pela cor cinza escuro.

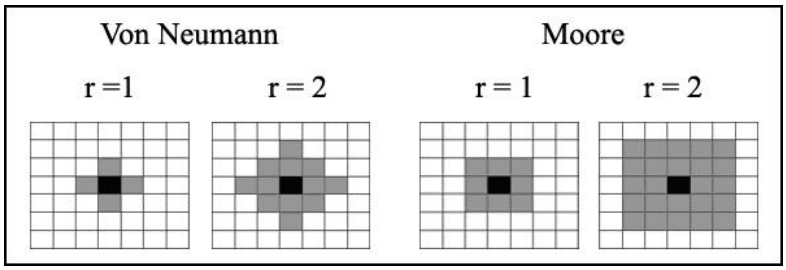


**Figura 10: Exemplos de vizinhança.**

### 3.2.4 Inicialização

Uma parte importante da execução de um autômato celular é a condição inicial das células de seu espaço. Dessa forma é especificado o estado inicial de cada célula no tempo inicial. A partir de diferentes inicializações, são obtidas evoluções diferentes para um determinado sistema.

### 3.2.5 Condições de Contorno

Há vários tipos de condições de contorno que podem ser usadas. A seguir são apresentados três exemplos de condições comumente usadas:

- Contorno periódico: seu funcionamento é como uma lista circular. Ao chegar ao final, o próximo passo leva à primeira célula, o inverso acontece ao se dar um passo para trás estando na primeira célula. A figura 11 apresenta um exemplo desta condição de contorno para o caso unidimensional. Este conceito é trivialmente estendido para casos com mais dimensões.

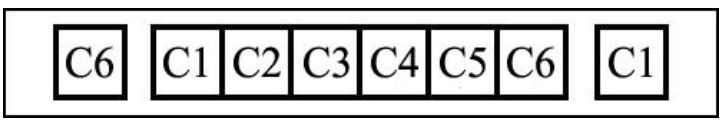


Figura 11: Contorno Periódico.

- Contorno reflexivo: seu funcionamento é como um espelho, fazendo com que as células do contorno enxerguem a si próprias ao acessar uma célula não existente. A figura 12 apresenta um exemplo desta condição para um caso unidimensional. Este conceito é trivialmente estendido para casos com mais dimensões.

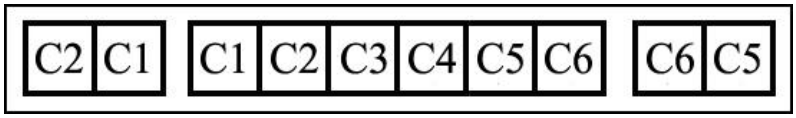


Figura 12: Contorno reflexivo.

- Contorno com valores fixos: É o tipo mais simples, pois é estabelecido um estado padrão para quaisquer células que estejam fora do espaço. A figura 13 apresenta um exemplo desta condição para um caso unidimensional. Este conceito é trivialmente estendido para casos com mais dimensões.

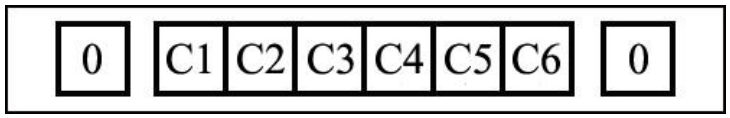


Figura 13: Contorno fixo.

## 3.3 REVERSIBILIDADE

### 3.3.1 Definição

Para se obter a reversibilidade, é necessário que o autômato celular utilizado seja um sistema determinístico. Para cada estado possível de todo o autômato celular, a função de transição retorna uma e somente uma possibilidade de estado futuro. Ainda assim não é trivial a tarefa de construir uma regra que execute um autômato de trás para frente, ou seja, dada uma regra qualquer, construir uma nova regra que ira forçar o sistema a retroceder seus passos no tempo. Para que isso seja possível, é necessário que a função original seja bijetora, tendo para cada estado possível do autômato celular, por definição, só um e somente um predecessor, assim como um e somente um sucessor.

### 3.3.2 Particionamento

Em um autômato celular convencional, as funções de transição podem ser vistas como portas lógicas, uma por célula. Cada porta lógica tem várias entradas (a célula atual e sua vizinhança) e somente uma saída – correspondendo ao estado futuro da célula atual. De forma geral, a função computada por uma porta lógica deste tipo não é reversível [10], pois somente uma fração da informação disponível nas entradas aparece na saída, ou seja, parte da informação é perdida.

Uma técnica utilizada para facilitar a criação de funções reversíveis é o particionamento de domínio. Esta técnica consiste em particionar o espaço em finitos grupos uniformes e disjuntos de células, denominados blocos. Dessa forma, a função de transição usa todas as células desses blocos como entrada, retornando os estados para todas as células do dado bloco. É importante ressaltar que os blocos não se sobrepõem uns aos outros e que não há troca de informação alguma entre blocos adjacentes. Assim pode-se estabelecer uma relação entre funções geradas por esta técnica com portas lógicas que possuem tantas linhas de entrada quanto de saída, onde nenhuma das linhas de entrada é compartilhada com outros portões lógicos. Nessa situação não há perda de informação, facilitando a criação de funções reversíveis. Uma função gerada utilizando esta técnica será reversível se e somente se ela estabelecer uma correspondência um para um entre o estado atual e o novo estado.

## 3.4 EXEMPLOS

Como apresentado anteriormente, ao se construir um autômato celular, um dos pontos a ser definido é a sua dimensão. A seguir serão apresentados dois exemplos, apresentando primeiramente um autômato celular de uma dimensão e seu funcionamento, posteriormente será apresentado um exemplo de um autômato de duas dimensões.

### 3.4.1 Autômato Celular Unidimensional

Basicamente um autômato celular de uma dimensão é uma linha ou vetor de células. Estas células devem se encontrar em um dos estados possíveis e com o passar do tempo, devem ser capazes de mudar de um estado para outro.

O conjunto de estados do autômato apresentado a seguir é bem restrito: duas cores, preto e branco. A regra de transição se baseará no estado atual da célula alvo, e de seus vizinhos diretos, portanto a vizinhança será composta de duas células, a célula à esquerda e a célula à direita da célula alvo.

Para este exemplo, serão usadas regras simples:

- Se todas as três células são brancas, então o novo estado da célula será branco;
- Se todas as três células são pretas, então o novo estado da célula será branco;
- Em qualquer outra situação, o novo estado da célula será preto.

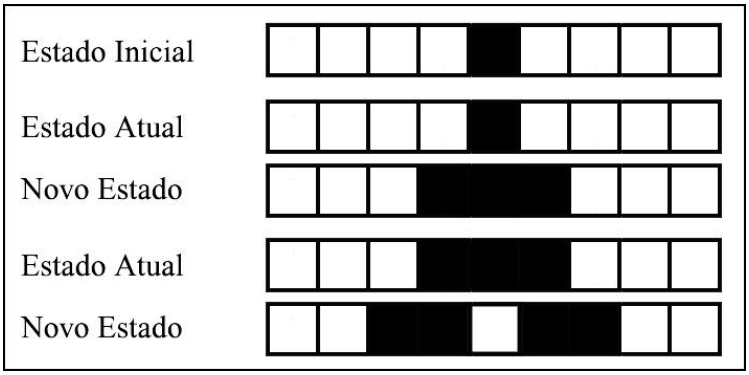


Figura 14: Evolução de um autômato unidimensional.

A figura 14 exibe o resultado de alguns passos de tempos, a partir de uma dada condição inicial. Nesse caso são usadas duas linhas, sendo uma a linha que contem os estados atuais das células, e a outra linha, a linha de saída com os estados futuros das células do autômato.

Para finalizar a especificação deste autômato, deve-se especificar as condições de contorno, uma condição simples a ser considerada seria o limite periódico, fazendo com que as células das extremidades assumam a célula da extremidade oposta como vizinha. Dessa forma, o autômato funcionaria como um círculo de células. Apesar da simplicidade de sua descrição, autômatos celulares unidimensionais são capazes de produzir padrões bastante complexos a partir de diferentes regras de transição.

Para autômatos desse tipo, com dois estados possíveis para as células, e uma vizinhança de três células: central, esquerda e direita; há uma combinação de $2^3=8$ padrões possíveis para uma vizinhança, sendo assim, há $2^8=256$ regras possíveis. A seguir, na figura 15, é apresentado um exemplo desse tipo de autômato, com uma regra diferente, chamada de regra 30, pois se considerarmos 0 como branco e 1 como preto, os resultados dos diferentes estados possíveis, colocados lado a lado formam o número 30 em binário.

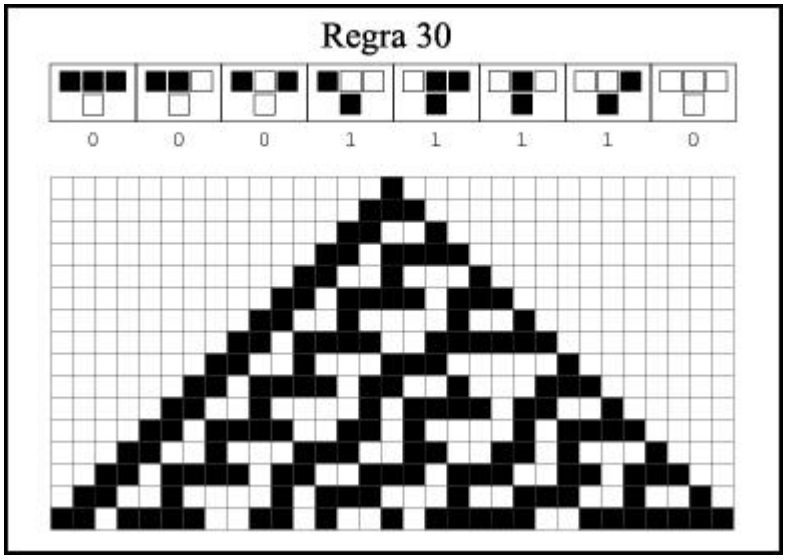


Figura 15: Exemplo de autômato unidimensional. [9]

### 3.4.2 Autômato Celular Bidimensional

Diferente dos autômatos celulares unidimensionais, que são dispostos como linhas de células, os autômatos celulares bidimensionais são dispostos como uma grade (ou matriz) de células, formando um painel bidimensional.

O funcionamento deste tipo de autômato é bastante similar de um autômato celular unidimensional. Dado um estado inicial, condições de contorno, um conjunto de regras de transição a serem seguidas e o tipo de vizinhança, o autômato evoluirá com o decorrer do tempo. A maior diferença entre autômatos celulares unidimensionais e bidimensionais é o maior número de possibilidades de se construir diferentes vizinhanças para o segundo. Com uma dimensão a mais há uma maior liberdade para se criar vizinhanças de diferentes formas e tamanhos. A figura 16 apresenta alguns tipos mais básicos de vizinhança.

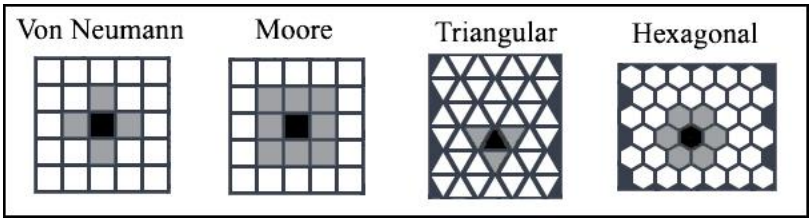


Figura 16: Exemplos de vizinhanças.

Um exemplo simples, e famoso, desse tipo de autômato é o Jogo da Vida, criado por John Conway. Este é um autômato bidimensional de células quadradas e com vizinhança de Moore. Neste autômato, as células podem viver, morrer ou se multiplicar. Dependendo da

condição inicial, vários padrões podem se formar. Cada célula tem 2 estados, vivo ou morto, e interage com seus oito vizinhos a cada passo de tempo.

As regras são:

- Células vivas com dois ou três vizinhos estarão viva no próximo passo.
- Células vivas com menos de dois vizinhos vivos, morre, devido à baixa população.
- Células vivas com mais de três vizinhos vivos, morrerá devido a superpopulação.
- Células mortas com exatamente três vizinhos se tornam vivas no próximo passo, como se houvesse reprodução entre as células vivas.

A partir de uma condição inicial e das regras apresentadas, o sistema evolui criando padrões interessante, sendo o mais notável o padrão conhecido como Planador. O Planador é um padrão periódico que se move através do espaço do autômato celular. A figura 17 exibe planadores que se formam na parte superior esquerda do domínio e se movem na direção da diagonal inferior direita.

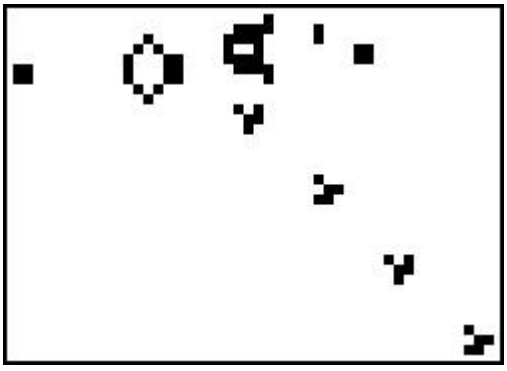

Figura 17: Jogo da Vida com uma configuração inicial geradora de Planadores. [21]

## 3.5 A VIZINHANÇA MARGOLUS

A vizinhança Margolus é uma vizinhança muito utilizada em autômatos celulares no tratamento de sistemas físicos, mais especificamente no tratamento de interação entre partículas.Esta vizinhança permite de maneira fácil a criação de regras para autômatos celulares com características como reversibilidade e conservação de partículas dentro do sistema. Isso ocorre pois essa vizinhança implementa o conceito já apresentado de particionamento de domínio. As células são divididas em blocos onde são aplicadas regras que recebem o conteúdo de todo o bloco e retornam uma atualização para todas as células

daquele bloco. A figura 18 apresenta um autômato celular particionado e um conjunto de regras para este autômato.

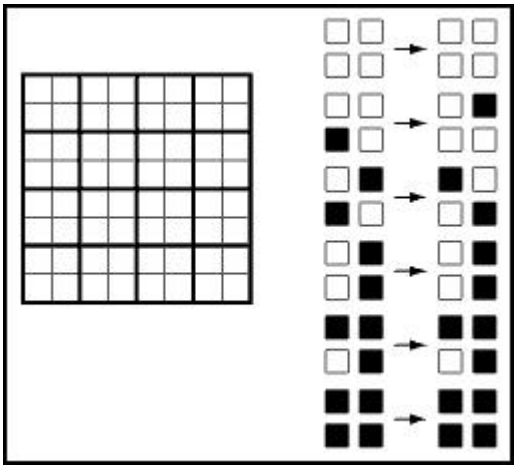

Figura 18: Particionamento de domínio e suas regras de transição. [10]

O problema desse tipo de vizinhança, é que se usarmos o mesmo particionamento em todos os passos, o autômato celular se torna uma coleção de subdomínios totalmente independentes entre si, o que faria com que a informação não se propagasse por todo o sistema. Para evitar que isto aconteça, a partição é alterada entre os passos do sistema, criando uma nova partição que sobreponha à partição de um passo anterior, e que será sobreposta pela partição criada no passo seguinte. Assim, alternamos entre duas partições similares, mas com um deslocamento entre elas, como se os próprios blocos se locomovessem pelo domínio entre passos pares e ímpares.

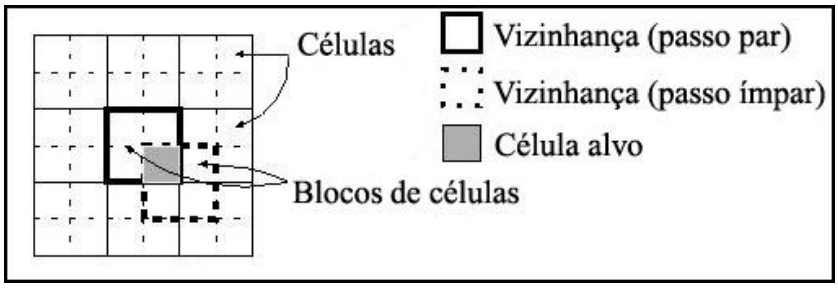


Figura 19:Vizinhança Margolus com blocos de células alternados. [11]

Na figura 19 as linhas sólidas representam as fronteiras entre os blocos de células pares, e as linhas pontilhadas as fronteiras dos blocos ímpares. Assim as células são posicionadas alternadamente em blocos pares e ímpares, gerando uma sobreposição entre os dois tipos de blocos permitindo a propagação da informação de um bloco para outro.

Em geral, as diferentes partições usadas passo a passo em um autômato celular, devem ser finitas em número e reusadas de forma cíclica, de forma a manter a uniformidade do tempo e do espaço [10]. Autômatos celulares são sistemas cujas leis são periódicas no tempo e no espaço. Em um autômato celular tradicional, um ciclo no tempo corresponde a um passo da regra e um ciclo no espaço corresponde a uma célula. Isso não ocorre com autômatos celulares com particionamento, pois suas regras têm um período mais longo tanto no tempo quanto no espaço. Especificamente, para a vizinhança Margolus são necessários 2 passos para completar um ciclo de regras no tempo além de serem necessárias 2 células (em um bloco 2x2) para fechar um ciclo no espaço

### 3.5.1 Um modelo simples de gases em movimento

Exemplificando os conceitos apresentados sobre a vizinhança Margolus, será apresentado um modelo simplificado de um gás, que consiste em um sistema simples de movimentação de partículas com velocidade uniforme. Neste exemplo a mecânica de movimento das partículas é trivial: uma partícula que está em um dos cantos de um bloco, se moverá na diagonal, ocupando o canto oposto do bloco no qual ela esta contida. Essa regra é conhecida como SWAP-ON-DIAG, onde as partículas sempre ocupam a diagonal oposta do bloco, sem preocupação com as partículas vizinhas, assim essa regra cria um sistema onde as partículas passam umas pelas outras e simplesmente continuam seus percursos. A figura 20 apresenta a tabela de casos para a regra SWAP-ON-DIAG.

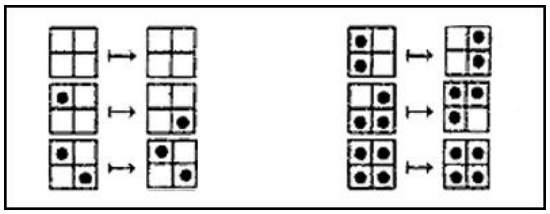

Figura 20: Regra SWAP-ON-DIAG. [10]

Com um mecanismo simples para a movimentação uniforme das partículas, devem ser tratadas as colisões que nesse caso serão tão simples quanto a movimentação. No mesmo modelo simplificado devemos tratar as partículas como tendo massa e, dessa forma, energia cinética. Já que todas as partículas são idênticas e viajam na mesma velocidade, todas elas têm a mesma energia. Dessa forma, conservação de energia é equivalente à conservação de partículas. Para conservar as partículas dentro de um dado bloco, usamos a regra SWAP-ON-

DIAG apresentada anteriormente. A quantidade de movimento também é conservada já que as partículas se movem uniformemente.

Para se introduzir colisões que mantenham a conservação de quantidade de movimento e energia, ao se analisar a tabela da regra SWAP-ON-DIAG, verifica-se que há somente uma possibilidade de modificação [10]. A terceira entrada da tabela, onde duas partículas se encontram frente a frente pode ser modificada para que no lugar onde as partículas se cruzariam livremente, elas colidam e desviem para a diagonal à sua esquerda. Essa nova regra é chamada de HPP-GAS. A figura 21 apresenta a tabela de casos para a regra HPP-GAS.

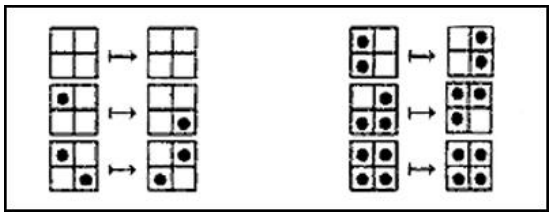

Figura 21: Regra HPP-GAS [10]

Apesar do fato das partículas estarem restritas a um conjunto limitado de posições e se moverem a uma velocidade uniforme em quatro direções possíveis, é válido notar que o comportamento macroscópico gerado pela regra *HPP-GAS* é, em muitos aspectos, idêntico ao comportamento físico dos gases [10].

# 4 SISTEMA DE DINÂMICA DE PARTÍCULAS EM GPUS

## 4.1 INTRODUÇÃO

Este trabalho tem por objetivo o estudo, implementação e a paralelização de um sistema de dinâmica de partículas em placas gráficas. Este sistema se baseia em um modelo bidimensional de autômatos celulares e trata a movimentação das partículas e as colisões entre elas.

A partir dos conceitos estabelecidos por [11] e [12], é apresentado um modelo de autômato celular desenvolvido para simular a dinâmica da movimentação de grupos de partículas e as interações entre eles. As partículas possuem atributos próprios, como massa, velocidade dentre outros, necessários para tornar possível a simulação da movimentação e da colisão.

Este sistema também permite a simulação de outras situações, como colisões entre um grupo de partículas e um obstáculo imóvel.

Inicialmente serão apresentados os conceitos básicos utilizados na construção do sistema de autômatos celulares, vizinhança, regras de transição e as equações físicas envolvidas, estados das células, etc. Em seguida será abordado a proposta de paralelização dada por [10] e as modificações e a estrutura criada para implementar o sistema em placas gráficas.

## 4.2 DINÂMICA DE PARTÍCULAS

Sem a influência de forças externas, uma partícula isolada se moveria com velocidade uniforme por todo o espaço. Num sistema de partículas, as únicas forças externas implementadas são aplicadas por outras partículas ou por obstáculos fixos durante uma colisão. Se nenhuma das duas condições ocorrer, a partícula permanecerá em seu trajeto até sair do domínio.

### 4.2.1 Tratamento físico das colisões

Como descrito em [14], a relação entre a velocidade relativa dos dois corpos depois do choque e a velocidade relativa dos corpos antes do choque é denominado coeficiente de restituição, cujo símbolo é a letra **e,** e é dado pela fórmula:

$$\vec{v}'_a - \vec{v}'_b = e(\vec{v}_a - \vec{v}_b) \qquad (1)$$

Onde $\vec{v}_a$ e $\vec{v}_b$ correspondem às velocidades iniciais dos corpos e $\vec{v}'_a - \vec{v}'_b$ as velocidades finais após a colisão. O coeficiente de restituição $e$ tem domínio limitado, $0 \leq e \leq 1$, onde $e$ = 1 corresponde a uma colisão elástica, onde há total conservação da energia cinética, e $e$ = 0 corresponde a uma colisão inelástica, onde há perda de energia cinética, mas conserva-se a energia mecânica do sistema.

Durante uma colisão entre dois corpos, a quantidade de movimento é conservada. Sendo assim, a quantidade de movimento total do sistema após a colisão é igual à quantidade de movimento total do sistema antes da colisão, conforme descreve a fórmula:

$$\vec{\rho}'_t = \vec{\rho}_t \rightarrow m_a \vec{v}'_a - m_b \vec{v}'_b = m_a \vec{v}_a - m_b \vec{v}_b \qquad (2)$$

onde $\vec{\rho}'_t$ corresponde à quantidade de movimento total depois da colisão; $\vec{\rho}_t$ corresponde à quantidade de movimento total antes da colisão; $m_a$ e $m_b$ as massas das partículas a e b envolvidas; $\vec{v}'_a$ e $\vec{v}'_b$ as velocidades das partículas a e b depois da colisão; e $\vec{v}_a$ e $\vec{v}_b$ as velocidades das partículas antes da colisão.

A partir da fórmula da colisão entre dois corpos (1) e da fórmula da conservação da quantidade de movimento (2), obtém-se:

$$\begin{cases} \vec{V}'_a = \vec{V}_0 + e\dfrac{m_b}{(m_a + m_b)}(\vec{V}_b - \vec{V}_a) \\ \vec{V}'_b = \vec{V}_0 + e\dfrac{m_a}{(m_a + m_b)}(\vec{V}_b - \vec{V}_a) \end{cases} \qquad (3)$$

$$\vec{V}_0 = \frac{m_a\vec{V}_a + m_b\vec{V}_b}{m_a + m_b} \qquad (4)$$

onde

Como as leis físicas utilizadas são determinísticas, isso faz com que o autômato celular possua um comportamento determinístico. Para dar uma impressão de explosão às colisões foi

utilizada uma técnica descrita em [11] e [12] que consiste em considerar um ângulo aleatório θ gerado entre 0 e 2π que determina a direção e sentido seguido pelas partículas após a colisão. Dessa forma, com várias partículas colidindo ao mesmo tem-se a simulação de uma explosão. Neste caso, é mantido o módulo do incremento da velocidade no segundo termo da fórmula (3), ou seja: $|\Delta \vec{v}| = |\vec{v}_a - \vec{v}_b|$.

Assim, a fórmula (3) pode ser reescrita de forma a considerar o seno e cosseno desse ângulo θ, da seguinte forma:

$$\begin{cases} \vec{V}'_a = \vec{V}_0 + e\dfrac{m_b}{(m_a + m_b)}|\vec{V}_b - \vec{V}_a|\,\vec{R} \\ \vec{V}'_b = \vec{V}_0 + e\dfrac{m_a}{(m_a + m_b)}|\vec{V}_b - \vec{V}_a|\,\vec{R} \end{cases} \quad (5)$$

onde

$$\vec{R} = (\cos\theta, \operatorname{sen}\theta) \quad (6)$$

### 4.2.2 Modelagem por autômatos celulares

Para criar o modelo de autômato celular, foi considerado um espaço bidimensional para as partículas. Este espaço é dividido em células separadas por passos discretos, e partículas ou obstáculos ocuparão estas células.

#### 4.2.2.1 Estados de uma célula

Para simular o movimento de partículas em um espaço, além de colisões com outras partículas e obstáculos, é necessária a definição de 3 estados possíveis para cada célula. Os estados são mapeados como números que representam os estados da célula em questão. Se o estado de uma célula for igual a zero, significa que a célula está vazia, se o estado for igual a um, a célula possui uma partícula, e se seu estado for igual a dois a célula possui um obstáculo imóvel e intransponível. Quando o estado é igual a 1, são válidos os atributos da célula:

- m: massa da partícula que ocupa a célula.
- v: vetor velocidade da partícula que ocupa a célula.

Quando o estado da célula for igual a 2, ou seja, um obstáculo ocupa a célula, há também um atributo a ser considerado:

- n: o vetor normal que define a superfície do obstáculo.

4.2.2.2 Vizinhança

Tanto a vizinhança de Von Neumann e a vizinhança de Moore são muito comuns na teoria de autômatos celulares. Padrões de propagação já foram criados nestes tipos de vizinhança, sendo o Glyder o exemplo mais óbvio.

Apesar disso, estas vizinhanças não são as mais recomendadas para este tipo de modelagem. Neste modelo, cada partícula é representada pelo estado da célula que a contém, não pelo padrão dos estados das células de toda a vizinhança. Também não é possível para uma célula alterar diretamente o estado de uma célula vizinha. Ela só obtém a informação de seu estado. Assim torna-se difícil criar uma função de transição de estados para propagação que preserve o número total de partículas. Para isso é necessário trabalhar com grupos de células que compartilhem de uma vizinhança em comum.

A vizinhança Margolus apresentada na Seção 3.5 apresenta exatamente este conceito, dividindo o domínio em blocos de tamanho 2x2 dentro dos quais ocorre a movimentação das partículas e as colisões entre as mesmas. Cada bloco é isolado do resto do domínio, não sendo permitido às partículas se deslocarem para fora de seus respectivos blocos.

Estes blocos de deslocam no domínio, no sentido diagonal inferior direita, a cada passo de tempo. Isso faz com que hajam dois tipos de subdivisões no domínio, nos passos pares, e outra nos passos ímpares. Na figura 22 pode-se observar o particionamento de um domínio em um intervalo de tempo ímpar

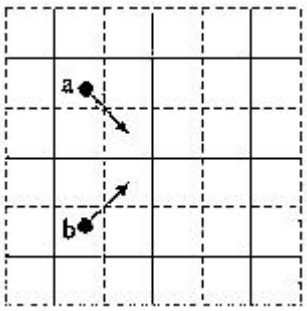


Figura 22: Particionamento de domínio em um intervalo de tempo ímpar. [11]

Como dito na Seção 3.5, esse particionamento alternado do domínio é importante para garantir o deslocamento das células, devido à independência de cada bloco em relação ao sistema. Esta limitação garante a conservação do número de partículas no bloco, fazendo com que as partículas não possam sair de seu limite. Caso não houvesse essa troca de particionamento, a cada passo, as partículas só poderiam se locomover entre as 4 células que compõem o bloco na qual estão contidas, para sempre.

Neste problema o domínio é infinito e é reproduzido apenas parte desde domínio. Para isto, é utilizada uma condição de contorno fixa e igual a zero, que significa que a partícula abandonou o subdomínio visualizado.

#### 4.2.2.3 Deslocamento de partículas

O movimento das partículas é discreto. Uma partícula se move em um intervalo de tempo do sistema para uma célula bem definida e a velocidade da partícula define quantos intervalos serão gastos para que a partícula possa enfim se mover. Dessa forma é simulada a velocidade da partícula e sua relação com partículas com velocidades diferentes. Uma partícula com velocidade maior esperará menos passos para se mover, enquanto uma partícula com velocidade menor terá de esperar mais passos.

Para isso, adota-se o seguinte procedimento:

1. Estabelece-se a relação entre a menor e a maior velocidade;
2. Calcula-se o módulo da maior velocidade;
3. Normalizam-se os vetores velocidade entre 0 e 1 a partir do módulo da maior velocidade;
4. Calcula-se o recíproco do valor absoluto das velocidades, obtendo-se assim um valor que funciona como um contador para cada partícula:

$$\text{Contador} = \frac{1}{|\vec{v}|}$$

Ou seja, como $|\vec{v}| \leq 1 \rightarrow$ contador $\geq 1$; e quanto maior o módulo da velocidade, menor o contador.

A cada intervalo de tempo decrementam-se os contadores de uma unidade. Quando esse contador chega a zero, a partícula é deslocada para a célula apontada pelo seu vetor velocidade.

#### 4.2.2.4 Colisões e posicionamento

Como as colisões ocorrem dentro dos blocos, elas se tornam mais fáceis de tratar. Em [11] esse tratamento é mais simples. A colisão é testada com a escolha aleatória de duas células dentro do bloco. Caso haja partículas nas células escolhidas, ocorrerá uma colisão. Dessa forma o sistema determina onde pode haver uma colisão, e não onde efetivamente ocorre uma. Assim, a cada passo, ocorre no máximo uma colisão em um mesmo bloco, podendo em certos casos não ocorrer nenhuma.

Em [12] o tratamento é um pouco mais complexo, dividindo este tratamento em dois casos separados. Caso haja duas partículas em um mesmo bloco, a colisão é dada como certa e é calculada a partir de seus atributos. Caso haja mais de duas partículas em um bloco, o tratamento se assemelha ao de [11], mas o número de colisões a serem testadas aumenta de acordo com o número de partículas no bloco. Dessa forma, se houver três partículas em um mesmo bloco, são escolhidas aleatoriamente 2 células. Caso hajam partículas nestas células, uma colisão ocorrerá. Em seguida, uma dessas células será testada com uma terceira célula também escolhida aleatoriamente e a célula restante da primeira dupla será testada com a única célula não escolhida. De forma análoga ocorre o tratamento de um bloco com 4 partículas, onde são tratadas duas colisões dentro do bloco, selecionando aleatoriamente duas duplas de células, uma para cada colisão. O modelo de tratamento de colisões utilizado neste projeto segue o modelo usado em [12].

Após o tratamento das colisões e o cálculo das novas velocidades de todas as partículas envolvidas, deve-se realizar o deslocamento destas de acordo com seus vetores velocidade. Para isso devem ser respeitadas duas regras de posicionamento, de acordo com a definição da vizinhança Margolus:

1. Nenhuma partícula de um bloco pode mover-se para fora do mesmo. E,
2. Cada célula do bloco deve possuir apenas uma partícula.

A escolha de uma nova célula para uma partícula é feita através de um estudo de seu vetor velocidade. Assim que ela esteja pronta para se mover, só há 3 possibilidades de movimento dentro de um bloco. Para decidir qual célula deve ser escolhida, verifica-se o ângulo formado pelo vetor velocidade com a horizontal. Depois de obtido o ângulo do vetor velocidade são usadas as seguintes regras para movimentar a partícula:

- Se $0 \leq \theta \leq \frac{\pi}{6}$, então a partícula é deslocada na horizontal;
- Se $\frac{\pi}{6} < \theta \leq \frac{\pi}{3}$, a partícula será deslocada na diagonal; e,
- Se $\frac{\pi}{3} < \theta \leq \frac{\pi}{2}$, a partícula será deslocada na vertical.

Quando uma partícula aponta para fora do bloco na qual ela está contida, é necessário definir qual a melhor posição para deslocar esta partícula sem infringir as regras de posicionamento da vizinhança Margolus. O número de possibilidades de preenchimento de um bloco é 4! = 24 possibilidades em cada bloco [11]. Para encontrar a melhor posição de

cada partícula dentro do bloco seria necessário resolver um problema de otimização combinatória. Como isso teria um custo alto, em [11] foi sugerido utilizar uma função mais direta, onde é definida a posição possível que seja mais próxima da desejada pela partícula. Isso é feito calculando-se a "distância" entre a célula desejada e as células livres, calculando-se a soma do quadrado da diferenças entre as coordenadas da célula desejada e das células que estão livres de restrições.

Em [12] foi sugerida a implementação de prioridades de movimento, onde as partículas com maior quantidade de movimento ($\vec{\rho} = m\vec{v}$) têm o direito de se deslocar primeiro dentro do bloco, empurrando as partículas com menos prioridade que estejam ocupando as células alvo das partículas com maior prioridade. Em casos onde uma dessas partículas queira se mover para fora do bloco é utilizada a função direta de [11] explicada anteriormente. Este foi o modelo de movimentação utilizado neste projeto.

#### 4.2.2.5 Modelagem de obstáculos

O modelo desenvolvido também é capaz de tratar obstáculos e a interação entre estes e as partículas do domínio. Os obstáculos tratados neste modelo são imóveis e não são deslocados durante toda a execução da aplicação.

Para calcular a nova velocidade $\vec{v}$', adquirida pela partícula após uma colisão com um obstáculo, utiliza-se a seguinte fórmula:

$$\vec{v}' = \vec{v} - (1+e)(|\vec{n}| \cdot \vec{v})\vec{n} \qquad (7)$$

onde *e* corresponde ao coeficiente de restituição entre a partícula e o obstáculo. Esta fórmula, além de considerar a velocidade da partícula antes da colisão, também leva em consideração o vetor normal $\vec{n}$ do obstáculo, que define sua superfície de contato. Por isso, a velocidade da partícula terá a direção definida explicitamente pela relação de sua velocidade anterior $\vec{v}$ e o vetor normal $\vec{n}$ do obstáculo, não sendo necessário um ângulo aleatório θ.

Em situações onde há obstáculos em um bloco, foi definido em [12] que obstáculos seriam tratados como as partículas de mais prioridade dentro de um bloco de células, portanto seriam as primeiras a serem posicionadas, podendo assim garantir que nenhuma partícula tome sua célula. Caso uma partícula tente se deslocar para uma célula ocupada por um obstáculo, esta célula é tratada como uma posição inválida, da mesma forma que uma posição fora do bloco.

## 4.3 ALGORITMO

O algoritmo utiliza dois domínios que se alternam a cada intervalo de tempo como um estado inicial e um estado final de cada passo. Basicamente o funcionamento do sistema pode ser descrito através do seguinte algoritmo:

- Alocar espaço em memória para a malha principal e a malha auxiliar;
- Iniciar a malha de acordo com os dados do sistema, criando grupos de partículas, obstáculos e configurando seus atributos;
- Atribuir o contador de movimento às partículas do sistema;
- Percorrer a malha, exibindo na tela as partículas e obstáculos, distinguindo-os em cor de acordo com o estado da célula corrente;
- Percorrer a malha decrementando o contador de cada partícula do domínio em uma unidade e criando cópias dos obstáculos na malha auxiliar, se houver obstáculos são criadas cópias no domínio do próximo passo;
- Mapear os blocos de células de acordo com o passo atual;
- Percorrer os blocos verificando o número de partículas e obstáculos no bloco corrente:
    - Havendo duas ou mais células preenchidas, inicia-se o tratamento de colisões de acordo com o número de células preenchidas;
    - Caso haja duas células preenchidas no bloco:
        - Se existirem duas partículas, elas irão colidir uma com a outra de acordo com seus vetores velocidade;
        - Se existirem uma partícula e um obstáculo eles irão colidir de acordo com seu vetor velocidade e seu vetor normal, respectivamente;
    - Caso haja três células preenchidas no bloco:
        - São escolhidas duas células aleatoriamente e caso estejam preenchidas, é tratada a colisão de acordo com o conteúdo das células;
        - A seguir é escolhida uma terceira célula aleatoriamente, e caso ela esteja preenchida, é tratada a colisão com uma das duas células anteriores, de acordo com seus conteúdos.
        - O tratamento de cada colisão é o mesmo modelo descrito para o caso de duas partículas;
    - Caso haja quatro células preenchidas no bloco:

        - São escolhidas duas células aleatoriamente e é tratada a colisão de acordo com o conteúdo das células;
        - Logo após são escolhidas outras duas células aleatoriamente para participar da segunda colisão;
        - O tratamento de cada colisão é o mesmo modelo descrito para o caso de duas partículas;
    - Atualizar o vetor velocidade e o contador de cada partícula que colidiu;
    - Iniciar o tratamento de deslocamento de partículas;
        - Definir prioridades dentro do bloco corrente;
        - Na ordem decrescente da quantidade de movimento de cada partícula definir a célula alvo do deslocamento;
        - Se for uma célula disponível, o deslocamento ocorre normalmente com uma cópia para o domínio do próximo passo;
        - Se for uma célula inválida (fora do bloco ou já preenchida) é realizada uma busca pela célula disponível mais próxima da desejada;
        - Deslocar a partícula para a célula alvo no bloco alocado no domínio do próximo passo;
- Alternar as malhas do passo atual e próximo passo;
- Incrementar o contador de passos;
- Repetir todo o processo a partir do quarto passo;

## 4.4 PARALELIZAÇÃO

Até então, todos os conceitos apresentados visavam a execução seqüencial deste sistema. A partir dessa seção serão apresentadas as modificações necessárias para o funcionamento do sistema em um ambiente multiprocessado.

### 4.4.1 Autômatos celulares em paralelo

A abordagem mais básica para se paralelizar o comportamento de um autômato celular seria tratar cada célula como um elemento paralelo independente. De fato, muitos modelos podem funcionar dessa maneira, sendo preferenciais autômatos onde as células sejam independentes dentro de suas vizinhanças, como o exemplo de autômato unidimensional apresentado na seção 3.4.1.

Apesar de existirem exemplos de regras de blocos que podem ser transformadas em um conjunto de regras de células, como a regra HPP-GAS apresentada na seção 3.5.1, possibilitando o tratamento paralelo de cada célula, este não é o caso do modelo utilizado. Neste trabalho é utilizada a vizinhança Margolus, onde são tratados blocos independentes, no lugar de células independentes, assim temos que tratar a paralelização de regras de bloco que atuam em várias células ao mesmo tempo.

O modelo paralelo criado utiliza informações de todas as células envolvidas para tratar as colisões, se existentes, e definir o próximo estado. Para isso são utilizados dois domínios que se intercalam a cada passo. A partir do passo atual, é desenhado o estado inicial do próximo passo no domínio que se encontra vazio. Ao final do passo corrente estes domínios se alternam de forma que o domínio do próximo passo será o domínio do passo atual. O modelo utilizado neste trabalho também é dependente da aleatoriedade para tratar as colisões, tanto na seleção de células ao colidir, quanto no ângulo da colisão.

Para se decidir as posições das partículas após as colisões, é necessário tratá-las em ordem de prioridade. Isso criaria a necessidade de passos de sincronização entre as *threads*, para garantir a consistência imposta pelo modelo matemático, trocando informações sobre qual *thread* endereça a melhor célula que a partícula pode ocupar. Esse procedimento seria um desperdício e diminuiria ou até mesmo acabaria com a vantagem da paralelização nas GPUs. Em [11] é sugerido tratar a paralelização com Elementos de Processamento (EPs) disjuntos que contém dois blocos sobrepostos. O número de EPs necessário para cobrir todo um domínio seria de um quarto do número de células do domínio. Estes elementos de processamento agrupariam um par de blocos, sendo um bloco do passo par e outro do passo ímpar. A figura 23 apresenta a esquematização dos EPs em um autômato celular.

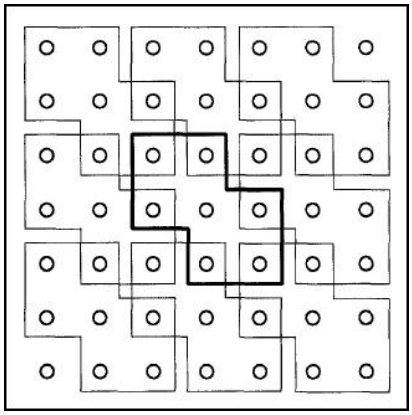

Figura 23: Organização dos elementos de processamento. [11]

Com esta configuração, cada EP possui uma rede de comunicação com 6 vizinhos, com os quais seria necessária a troca de informações de suas células sobrepostas a cada passo. A figura 24 exibe a rede de sincronização necessária para manter a consistência do sistema.

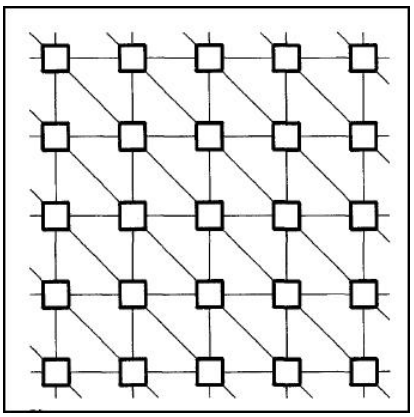

Figura 24: Rede de comunicação dos elementos de processamento. [11]

Esta configuração não retrata a realidade do *hardware* das GPUs, onde os EPs não são disjuntos. Sendo assim, neste projeto não há a necessidade de sincronização entre EPs. Com isso podemos criar uma estrutura de malha única que é dividida logicamente a cada passo, reorganizando os EPs para controlar um bloco completo do domínio no passo corrente. Dessa forma, os EPs sempre trabalham somente com os dados locais de cada bloco da vizinhança Margolus. Com isso, a sincronização se torna implícita, pois todos os elementos de processamento estariam trabalhando em uma malha em comum, mas em subdivisões independentes. Com essa configuração o número de EPs necessários é metade do número de células de um domínio, mas com a alternância entre EPs, temos que somente metade do total de EPs vai estar ativa a cada passo.

### 4.4.2 O sistema em GPUs

A estrutura apresentada anteriormente é facilmente configurável dentro do esquema da API CUDA. Para o sistema, criamos um *kernel* CUDA no espaço da GPU e através de parâmetros na chamada à função, estabelecemos parâmetros de configuração das *threads* e blocos de *threads* para a execução do *kernel*.

A abordagem natural seria estabelecer uma relação um para um entre as *threads* da GPU e as células do domínio. Como dito anteriormente isso causaria um gargalo com uma grande necessidade de sincronização entre as *threads*. Dessa forma, o *kernel* é configurado para que cada *thread* corresponda a um bloco da vizinhança Margolus em um dado passo. A

cada passo o número de *threads* é um quarto do número de células do domínio. Na figura 25 é possível ver essa organização de threads em blocos de células.

Com esta configuração as *threads* acessam a malha do domínio e tratam seus dados sem nenhuma sobreposição de células, não sendo necessário gastar instruções com sincronização.

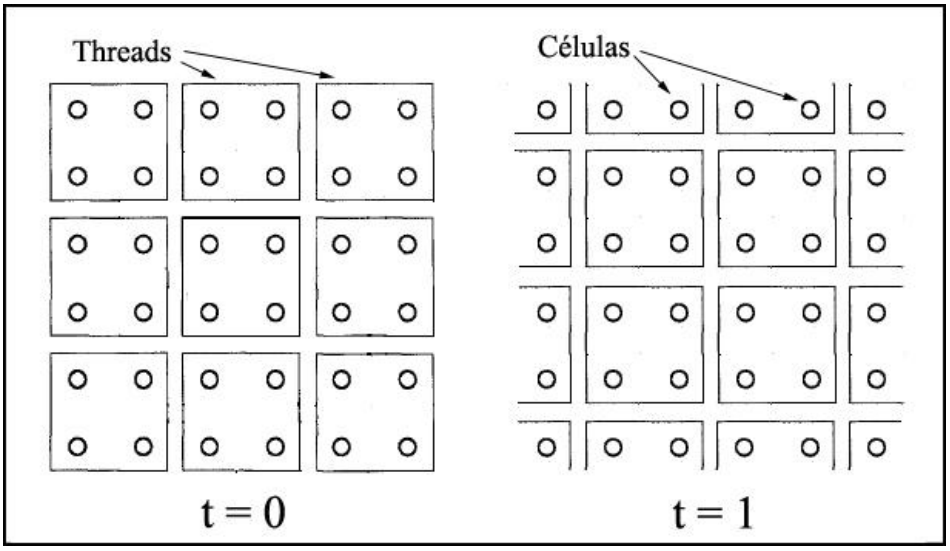


Figura 25: Disposição das threads na e as células do domínio.

4.4.2.1 Descrição do algoritmo em GPUs

O funcionamento do sistema nas GPUs não é de todo diferente da implementação sequencial. Apesar disso será apresentado o algoritmo de GPU para explicitar detalhes desta implementação em particular.

Primeiramente é feita toda a inicialização do sistema. São utilizados dois vetores que são alocados em memória como uma matriz através de uma função disponível na API CUDA chamada *cudaMallocPitch*. Nesse caso, os vetores são alocados como matrizes quadradas. Cada posição destes vetores contém uma célula, esta célula é uma estrutura que armazena todos os atributos necessários ao sistema, como o estado da célula, vetor velocidade, massa da partícula, contador, etc. Estes dois vetores funcionarão como o domínio atual e o domínio do passo seguinte.

Para visualizar o funcionamento do sistema, é utilizado o recurso de interoperabilidade entre a API CUDA e o DirectX. É criada uma janela para o sistema que é então passada ao DirectX para que este trate a atualização da mesma. Para isso é necessário criar diretamente na GPU uma malha poligonal que ira preencher esta janela e receber os dados de cor que serão exibidos na tela. O laço de repetição principal que atualiza a janela de visualização é

também o laço onde são executados os passos do sistema. Assim o sistema só terminar quando a janela executa o método de saída.

Nenhuma transferência de dados entre a CPU e a GPU é feita explicitamente através de chamadas de funções. Antes do primeiro passo, os vetores que já estão alocados na GPU são inicializados diretamente na GPU através de um *kernel* específico, criando partículas e/ou obstáculos, e atribuindo valores aos seus atributos.

O laço de repetição realiza a atualização da janela, e o passo de execução do autômato, que é dividido em três etapas como apresentado anteriormente. Para atualizar o vídeo, é necessário mapear um *buffer* de vértices do DirectX em uma estrutura do CUDA, assim é executado um *kernel* que atualiza os dados de cor destes vértices de acordo com os dados das células do vetor autômato.

Após todas as inicializações, se dará início ao tratamento do autômato no sistema. A cada iteração do laço principal é executado um passo do sistema que é dividido em três etapas diferentes. Cada etapa representa um *kernel* independente.

A primeira etapa é percorrer todo o domínio decrementando os contadores de todas as partículas e satisfazendo as condições de contorno, zerando o estado de todas as células na borda do domínio. Nesta etapa também é realizado uma cópia dos obstáculos para o domínio do passo seguinte. Este *kernel* é configurado de forma a gerar uma *thread* para cada posição do vetor. Em um vetor de tamanho 512x512, foram utilizados 32x32 blocos de 16x16 *threads* cada.

A segunda etapa é percorrer o domínio tratando as possíveis colisões. Cada *thread* verifica o número de células em seu bloco de células e, caso hajam mais de uma célula, é realizado o tratamento das colisões de acordo com o algoritmo apresentado anteriormente. É criada um vetor auxiliar de quatro posições fixas para tratas a aleatoriedade da seleção de células. Preenchendo este vetor de acordo com o número aleatório usado para obter o índice das células. Este *kernel* é configurado de acordo com a vizinhança Margolus. Em um vetor de tamanho 512x512, foram utilizados 16x16 blocos de 16x16 *threads* cada. Cada *thread* deste *kernel* executará em cima de um bloco de 2x2 posições do vetor. O mapeamento das *threads* é feito passando um parâmetro identificador de passo e alterando o cálculo do endereço da primeira posição do bloco a ser trabalhado.

A terceira etapa trata o deslocamento das partículas dentro de seus respectivos blocos. Cada *thread* verifica o número de células em seu bloco de células. Caso haja apenas uma célula, o tratamento do deslocamento é trivial. Caso hajam mais de uma partícula, é utilizado um vetor auxiliar de 4 posições que armazena as posições das partículas e é preenchido na

ordem decrescente de prioridade das partículas. Este vetor é então percorrido e as partículas contidas nos endereços armazenados são movidas para as células do domínio do passo seguinte. Este domínio encontra-se vazio, a não sei pelos obstáculos já copiados. Nesse caso as partículas de maior prioridade encontrarão mais células vazias a poderão tomar as células desejadas, enquanto as de menor prioridade serão alocadas nas células mais próximas o possível das desejadas. A configuração deste *kernel* é análoga à do *kernel* anterior.

Ao final destas três etapas, todas as partículas e obstáculos encontram-se atualizadas no domínio do passo seguinte. Os ponteiros destes domínios são então alternados para que o passo anterior possa ser apagado, fazendo com que todas as suas células fiquem vazias para receber o resultado do próximo passo.

A partir deste ponto, termina o laço principal, retornando ao primeiro passo. Caso seja necessário, o código do sistema estará disponível no Apêndice deste trabalho.

#### 4.4.2.2 Limitações

O sistema em sua forma atual depende da geração de números pseudoaleatórios em diversos passos, como na geração de ângulos aleatórios para a colisão. Isso é um entrave no desenvolvimento em CUDA, pois não há nenhuma função deste tipo já implementada em alguma biblioteca.

Em [13] foi desenvolvida uma versão CUDA da função chamada rand48. A função rand48 usa aritmética congruencial de 48bit para gerar valores pseudoaleatórios uniformemente distribuídos. Para inicializar a função, é necessário passar um número semente a ser usado na inicialização.

A implementação desta função no sistema traz alguns problemas. A execução de código CUDA não traz nenhuma implementação de pilha, não há chamadas a funções propriamente ditas. Todas as funções CUDA são funções *inline*. O compilador, ao gerar o código, simplesmente copia todo o trecho da função diretamente onde havia chamadas a ela. Com isso temos uma função CUDA, um *kernel*, que acaba por ser muito grande, e há uma limitação tanto de tamanho do *kernel*, quanto aos recursos necessários para este ser executado, como número de registradores em cada multiprocessador.

Para que o sistema pudesse ser executado, foi necessário reescrever o algoritmo de forma a separar etapas diferentes em mais de um *kernel*. Onde antes havia um passo único, onde se percorria toda a malha, decrementando contadores, tratando colisões e deslocamentos de partículas, ficou clara a necessidade de dividir em mais passos seqüenciais.

A função de transição fica composta de três etapas realizadas por três *kernels* diferentes:

- Uma etapa de controle da malha, onde é cumprida a condição de contorno, decrementados os contadores, e realizado o deslocamento de obstáculos;
- Uma etapa de tratamento de colisões; e
- Uma etapa para o deslocamento das partículas.

Note que as duas primeiras etapas apenas trabalham com os atributos das partículas. A posição da partícula no próximo passo é determinada somente na última etapa e depende fortemente dos passos anteriores.

# 5 SIMULAÇÕES

Serão apresentadas algumas simulações realizadas no projeto a partir do modelo apresentado no capítulo 4. Para isso, foram criadas partículas com os mesmos atributos que se localizavam próximas umas às outras, configurando um grupo de partículas. Cada uma das simulações a seguir foi realizada com 1941 partículas por grupo. Para este projeto foram desenvolvidas duas versões do sistema, uma seqüencial e outra em paralelo.

## 5.1 COLISÃO ENTRE GRUPOS DE PARTÍCULAS

O objetivo principal desde projeto é simular a dinâmica do choque entre grupos de partículas. Neste tipo de colisão, pode haver vários tipos de configurações iniciais. Algumas dessas situações são ilustradas a seguir.

### 5.1.1 Colisão frontal com velocidades opostas

Para gerar este tipo de colisão, foi usada a seguinte configuração:

- Coeficiente elástico = 1;
- Grupo à esquerda:
  - Massa = 10;
  - Velocidade = (10,0);
- Grupo à direita:
  - Massa = 10;
  - Velocidade = (-10,0);

Ao se encontrarem, as partículas vão colidindo e se reordenando dentro dos blocos, se espalhando em todas as direções devido ao ângulo aleatório inserido na equação. A figura 26 mostra três instantes de tempo desta simulação:

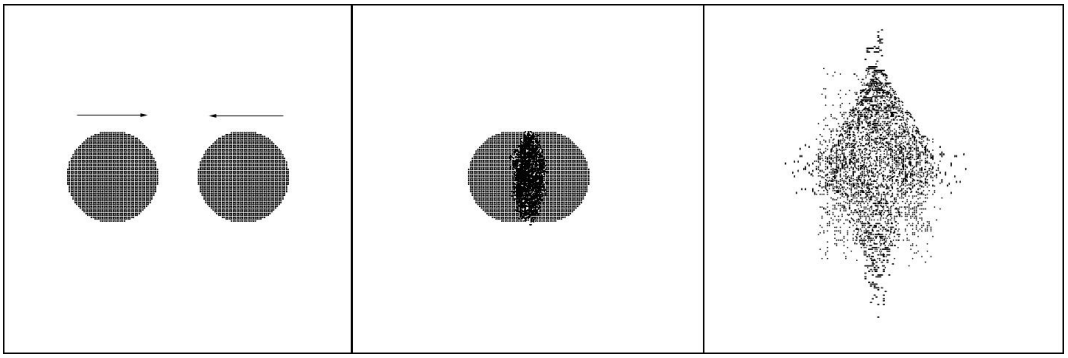

Figura 26: Colisão frontal entre grupos de partículas .1

Ainda utilizando a mesma configuração, alterando apenas a massa do grupo à direita para 2, resulta em uma colisão apresentada pela figura 27 em três instantes de tempo desta simulação:

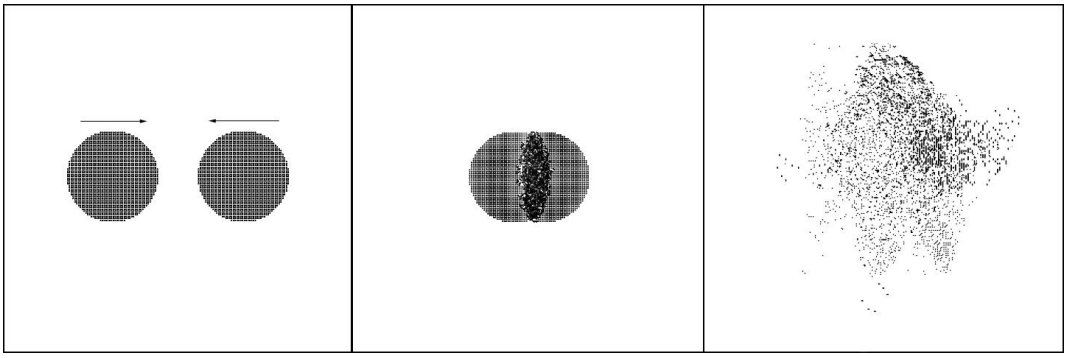

Figura 27: Colisão frontal entre grupos de partículas 2.

### 5.1.2 Colisão diagonal

Para a exibição de uma colisão com grupos se deslocando na diagonal, foi utilizada a seguinte configuração:

- Coeficiente elástico = 1;
- Grupo à esquerda:
  - Massa = 10;
  - Velocidade = (10, -10);
- Grupo à direita:
  - Massa = 10;
  - Velocidade = (-10, -10);

A figura 28 mostra três instantes desta simulação.

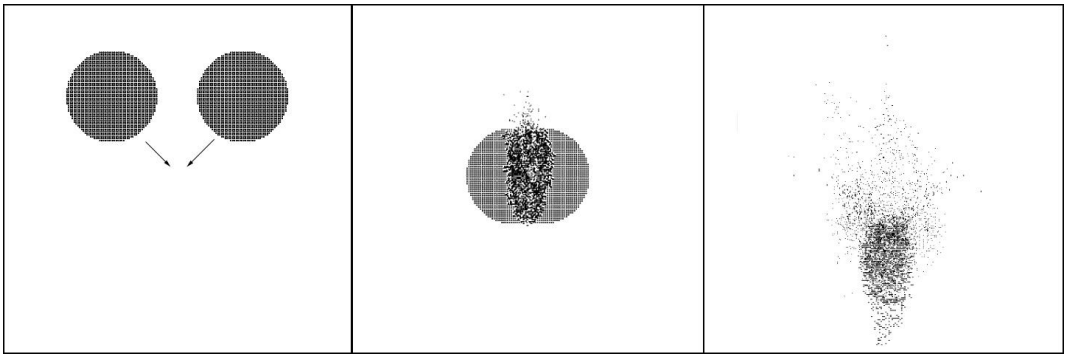

Figura 28: Colisão diagonal entre grupo de partículas 1

Ainda utilizando a mesma configuração, alterando apenas a massa do grupo à direita para 2, resulta em uma colisão apresentada pela figura 29 em três instantes de tempo desta simulação:

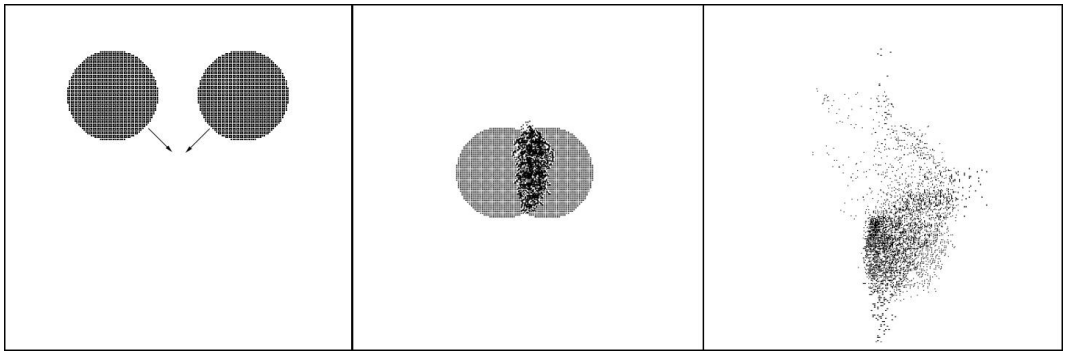

Figura 29: Colisão diagonal entre grupos de partículas.

Nesta simulação, em virtude da diferença de massas, as partículas mais pesadas empurraram as partículas mais leves, fazendo com que o grupo da esquerda, mais leve, fosse deformado mais rapidamente.

## 5.2 COLISÃO DE UM GRUPO DE PARTÍCULAS E OBSTÁCULOS

As simulações a seguir apresentam simulações onde um grupo de partículas colide com um obstáculo fixo. Neste tipo de colisão, o sentido do vetor velocidade após a colisão é definido pelo vetor normal da superfície do obstáculo.

### 5.2.1 Colisão de um grupo de partículas com uma parede

Neste tipo de colisão, um grupo de partículas é arremessado contra uma parede, colidindo com um obstáculo fixo de formato reto e liso. Para esta colisão, foram utilizadas as seguintes configurações:

- Coeficiente elástico entre partículas= 1;
- Coeficiente elástico entre partícula e obstáculo = 1;
- Grupo de partículas:
    - Massa = 10;
    - Velocidade = (10, 0);

A figura 30 apresenta três instantes da simulação obtida.

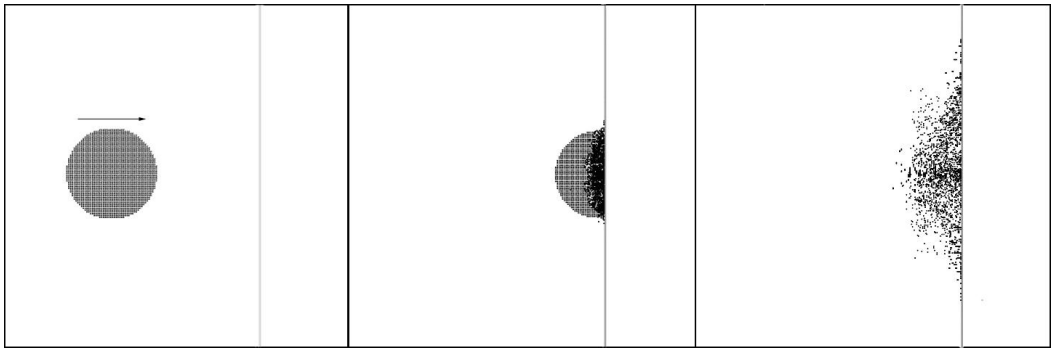

Figura 30: Colisão entre grupo de partículas e uma parede 1.

Com a mesma configuração da simulação, foram modificados apenas alterado apenas os valores dos coeficientes de restituição. A figura 31 apresenta três instantes de tempo da simulação:

- Coeficiente elástico entre partículas= 1;
- Coeficiente elástico entre partícula e obstáculo = 0;

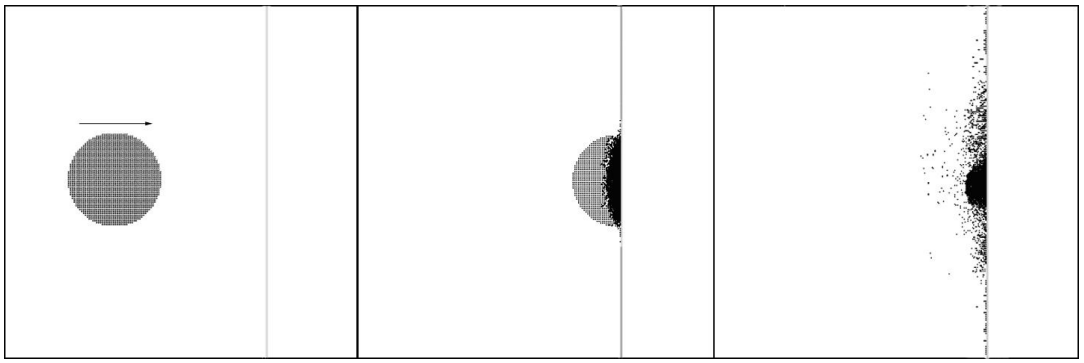

Figura 31: Colisão entre grupo de partículas e uma parede 2.

### 5.2.2 Colisão de um grupo de partículas contra duas paredes perpendiculares

Por último, é criada a situação onde há paredes perpendiculares no domínio, fazendo o grupo de partículas colidir de duas formas diferentes com este obstáculo.

A primeira simulação é feita colidindo o grupo de partículas de forma perpendicular a uma das paredes. Para isso foram utilizados os seguintes dados:

- Coeficiente elástico entre partículas= 1;
- Coeficiente elástico entre partícula e obstáculo = 1;
- Grupo de partículas:
    - Massa = 4;
    - Velocidade = (10,0);

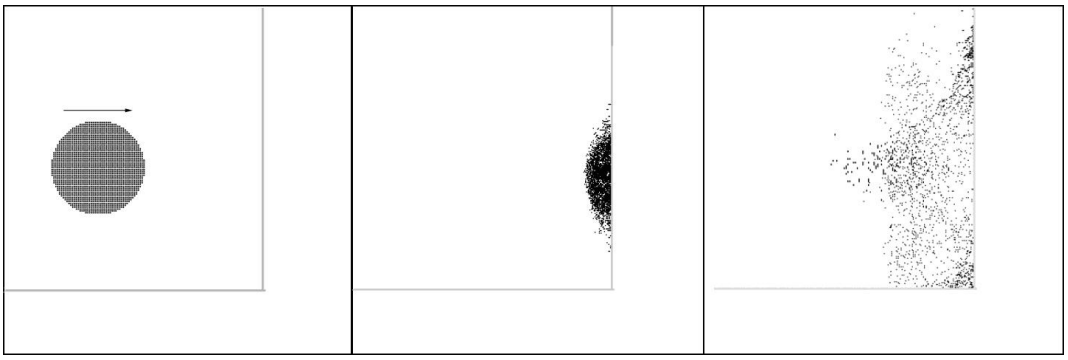

Figura 32: Colisão entre grupo de partículas e paredes perpendiculares 1.

Em seguida, alterando a velocidade do grupo de partículas, para que a colisão ocorra na diagonal, com o grupo acertando a junção entre as paredes.

- Grupo de partículas:
    - Massa = 10;
    - Velocidade = (10,0);

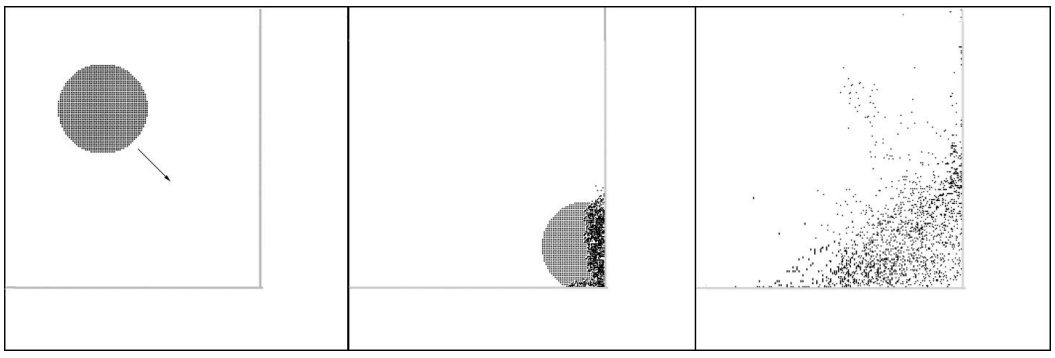

Figura 33: Colisão entre grupo de partículas e paredes perpendiculares 2.

## 5.3 UMA BREVE ANÁLISE DE DESEMPENHO

O desenvolvimento deste sistema não teve o desempenho geral como preocupação principal, mas este projeto estaria incompleto sem ao menos uma análise do funcionamento geral do sistema, por mais superficial que fosse.

Estes testes foram executados em uma Geforce 8600gt, que possui 4 multiprocessadores com 8 núcleos cada, é uma GPU de baixo custo e baixo desempenho. O modelo Geforce 8800, por exemplo, possui 16 multiprocessadores, também com 8 núcleos cada.

Foram coletados alguns dados da execução da versão seqüencial (1 *thread*) e da paralela para fins de comparação. Basicamente foi utilizada uma função de cronometragem de tempo para verificar o tempo necessário para ambas as versões completarem 1 passo simples, lembrando que na vizinhança Margolus, 1 passo de tempo completo é igual a um passo par e mais um passo ímpar. O tempo cronometrado refere-se apenas ao passo simples, não considerando a visualização, ou seja, é cronometrado apenas o tempo referente à execução do autômato. Os dados de tempo coletados para um domínio de 512x512 células com 2 grupos de 1941 partículas cada são apresentados a seguir:

| | CPU | GPU |
|---|---|---|
| Passo simples sem colisões | ~4.5 ms | ~5,7 ms |
| Passo simples com colisões – Min. | ~4,7 ms | ~16,3ms |
| Passo simples com colisões – Max. | ~6.8ms | ~1071,1ms |

Tabela 1: Coleta de tempo da execução 1.

A seguir são apresentados os tempos para o mesmo teste executado em um domínio de 1024x1024, com grupos de 7825 partículas cada:

| | CPU | GPU |
|---|---|---|
| Passo simples sem colisões | ~24.5. ms | ~23,1 ms |
| Passo simples com colisões – Min. | ~18,3 ms | ~19,1ms |
| Passo simples com colisões – Max. | ~39.2ms | ~1696,3ms |

Tabela 2: Coleta de tempo da execução 2.

Apesar dos tempos de execução terem sido coletados de uma forma simples, a diferença entre os tempos de ambas as versões fica clara. Ainda assim é necessário cuidado ao analisar estes números, pois a execução de código *CUDA* envolve muito *overhead* de função e sincronização de memória (mesmo implícita) entre chamadas de *kernels*, e neste caso, o sistema faz uma chamada a três *kernels* em sequencia  para cada passo simples.

O mais preocupante é a disparidade entre os tempos dos passos onde ocorreram colisões. Fica claro que o tratamento de colisões é o ponto fraco deste sistema. Para fins de experimentação, foi removida do sistema a função geradora de números pseudo-aleatórios, e no lugar de tratar a colisão de células aleatórias, as colisões foram tratadas seqüencialmente dentro de cada bloco. Dessa forma, ao ocorrerem as colisões, ocorreram apenas leves alterações no tempo de execução em relação ao passo simples sem colisão.

Após mais testes foi verificado que a função de movimentação de células, que trabalha com prioridade e percorre todo o bloco para encontrar a melhor célula disponível impacta igualmente o desempenho. O tempo coletado, se manteve próximo de 50% para o kernel de colisões e 50% para o kernel de deslocamento de partículas. Isso ocorre, pois além de ser uma função em que se percorre o bloco de células diversas vezes, também trata partículas de acordo com a prioridade destas, fazendo com que as *threads* acabem não percorrendo os mesmos caminhos, obrigando a GPU a serializar muitas *threads.*

# 6 CONCLUSÃO

Este projeto foi concebido para ser uma experiência na paralelização de sistemas de autômatos celulares com vizinhança de Margolus. Estes sistemas usam uma malha de células e um conjunto de regras de transição para definir o comportamento destas células, podendo gerar padrões complexos e simular fenômenos naturais. Autômatos celulares são sistemas com uma paralelização implícita, pois cada parte menor do sistema trabalha de forma independente do resto, e com isso são uma escolha óbvia para se experimentar com o poder de paralelização disponível nas placas gráficas atuais. Assim foi escolhido o desenvolvimento de um sistema que pudesse simular o comportamento de partículas, sua movimentação e interação com outras partículas e obstáculos fixos.

Este trabalho, mais do que representação visual ou desempenho, preocupou-se com o estudo e a abordagem necessários para realizar a implementação de um sistema de autômatos celulares em placas gráficas. Dessa forma foi criado uma primeira versão deste tipo de problema, onde é utilizado um modelo de autômatos celulares para desenvolver um sistema de dinâmica de partículas. O sistema desenvolvido é capaz de lidar com modelos de colisões entre partículas e entre partículas e obstáculos fixos. Este trabalho poderá servir de referência para que sejam criadas novas implementações e melhorias para este tipo de problema.

Devem-se ressaltar problemas encontrados no desenvolvimento deste trabalho. Um dos grandes prejudicadores foi o fato de não haver nenhuma biblioteca acessível de funções para a geração de números aleatórios congruenciais. Algumas versões já foram implementadas, mas se mostraram complexas e grandes demais para serem anexadas ao código do sistema, pois funções CUDA são do tipo *inline* e há uma limitação no tamanho total do código a ser executado. A versão utilizada demonstrou-se ineficaz, causando uma diferença muito grande no tempo de computação, em favor da versão seqüencial. É possível que somente a criação de uma biblioteca acessível para geração de números pseudo-aleatórios para placas gráficas seja assunto mais que suficiente para se tornar alvo de estudos mais

aprofundados, o que beneficiaria o desenvolvimento de outros sistemas que poderiam tirar vantagem de bibliotecas deste tipo.

Seria necessário testar outras funções geradoras de números pseudoaleatórios e verificar se estas teriam um menor impacto no sistema. Além disso, é necessário desenvolver um novo método de movimentação de partículas, que possa ser realizado por passos bem definidos, garantindo que todas as *threads* sejam executadas em paralelo.

Trabalhos futuros podem tentar melhorar o algoritmo, se preocupando com os parâmetros de desempenho do CUDA, para melhorar os tempos de execução, ou ser mais diretos e alterar o modelo utilizado para não sofrer com os mesmos problemas, além de melhorar a apresentação visual do sistema e introduzir o conceito de grupos de partículas fixas umas às outras, pois este projeto trabalha apenas com grupos de partículas soltas, não há nenhuma força de resistência que mantenha o grupo unido. Este tipo de sistema também pode ser expandido para tratar a dinâmica de outros tipos de partículas, como fluídos ou gases. E a própria modelagem pode ser expandida para representar o sistema em um domínio tridimensional.

# 7 REFERÊNCIAS BIBLIOGRÁFICAS


1. NVIDIA. *Nvidia. CUDA Zone*. Documentação. Disponível em: <http://www.nvidia.com/object/cuda_develop.html>. Acesso em: 5 de Ago 2009.

2. LINDHOLM, Erik. NICKOLLS, John. OBERMAN, Stuart. MONTRYM, John. Nvidia Tesla: A Unified Graphics and Computing Architecture. *IEEE Micro*, V. 28, n. 2, 2008. Disponível em <http://www.computer.org/portal/web/csdl/doi/10.1109/MM.2008.31>. Acesso em 5 de Ago 2009.

3. OWENS, John. LUEBKE, David. GOVINDARAJU, Naga. HARRIS, Mark. KRÜGER, Jens. LEFOHN, Aaron. PURCELL, Timothy. A Survey of General-Purpose Computation on Graphics Hardware. *Computer Graphics Forum*, V. 26, n. 1, 2007. Disponível em <http://onlinelibrary.wiley.com/doi/10.1111/j.1467-8659.2007.01012.x/full>. Acesso em 5 de Ago 2009.

4. CROW, Thomas Scott. *Evolution of the Graphical Processing Unit*. RENO, 2004. 59 F. Tese (Mestrado em Ciência da Computação) – Universidade de Nevada, Reno. 2004.

5. *Amd stream computing*. Disponível em: <http://ati.amd.com/technology/streamcomputing/firestream-sdk-whitepaper.pdf>. Acesso em 20 de Fev 2010.

6. Khronos Group. *Opencl - the open standard for parallel programming of heterogeneous systems*. Disponível em: <http://www.khronos.org/opencl/>. Acesso em 20 de Fev 2010.

7. SMITH, Michael. GOODCHILD, Michael. LONGLEY, Paul. Cellular Automata. Disponível em: <http://www.spatialanalysisonline.com/output/html/CellularautomataCA.html>. Acesso em 20 de Fev 2010.

8. WOLFRAM, Stephen. Cellular Automata (1983). Disponível em: <http://www.stephenwolfram.com/publications/articles/general/83-cellular/>. Acesso em 21 de Fev 2010.

9. ____**.**Cellular Automaton Supercomputing (1988). Disponível em: <http://www.stephenwolfram.com/publications/articles/ca/88-cellular/1/text.html>. Acesso em 21 de Fev 2010.

10. TOFFOLI, Tommaso. MARGOLUS, Norman. *Cellular Automata Machines*: A new Environment for Modeling. Cambridge, Massachusetts: The MIT Press, 1987. 255p.

11. TAKAI, Yoshiaki. ECCHU, Katsuyuki. TAKAI, Nami. A cellular automaton model of particle motions and its applications. *The Visual Computer*, Berlin: Nadia Magnenat-Thalmann, V. 11, 1995, n. 5, p. 240-252, Maio, 1995.

12. ARAÚJO, Diego Oliveira. JÚNIOR, Gilberto Paiva de Medeiros. *Collisions*: Um sistema para simular o movuimento de partículas através de um modelo de autômato celular. Niterói, 2005. 68 f. Trabalho de Conclusão de Curso (Graduação em Ciência da Computação) – Curso de Ciência da Computação – Instituto de Computação, Universidade Federal Fluminense, Niterói, 2005.

13. MEEL, J.A. Van. ARNOLD, A. FRENKEL, D. ZWART, S.F. Portegies. BELLEMAN, R.G.. Harvesting graphics power for MD simulations. *Molecular Simulation*, Amsterdam: N. Quirke.V. 34, 2008, n. p. 259 – 266, Maio, 2008.

14. DOS SANTOS, José Nazareno. DA SILVA, Romero Tavares. *Educação mediada por Computador*. Curso de Física, Universidade Federal da Paraíba. Disponível em <http://www.fisica.ufpb.br/prolicen/produtos.html> Acesso em 20 de Out. 2010

15. WEISSTEIN, Eric W. Von Neumann Neighborhood. *MathWorld:* A Wolfram Web Resource. <http://mathworld.wolfram.com/vonNeumannNeighborhood.html>. Acesso em 3 de Out. 2010.

16. ADAMATZKY, Andrew. *Collision-Based Computing*, Ed. Springer, 2002. Disponível em <HTTP://csdl2.computer.org/comp/mags/cs/2005/01/c1017.pdf>. Acesso em 20 de out. 2010.

17. KOMATITSCH, Dimitri. ERLEBACHER, Gordon. GÖDDEKE, Dominik. MICHÉA, David. *High-order finite-element seismic wave propagation modeling with MPI on a large GPU cluster*. *Journal of Computational Physics,* Notre dame: G. Tryggvason, V. 229, 2010, p. 7692–7714, junho 2010

18. PREIS, Tobias. VIRNAU, Peter. PAUL, Wolfgang. SCHNEIDER, Johannes J. *Accelerated fluctuation analysis by graphic cards and complex pattern formation in financial markets*. *New Journal of Physics*, Göttingen: Eberhard Bodenschatz, V. 11, 2009, n. 5, Setembro 2009. – disponível em <http://www.tobiaspreis.de/publications/pvps_njp_2009.pdf>. Acesso em 20 de Out. 2010.

19. STOKES, Jon. *SIMD architectures.* Ars Technica. <http://arstechnica.com/old/content/2000/03/simd.ars> Acesso em 20 de out. 2010.

20. WEISSTEIN, Eric W. Von Neumann Neighborhood. *MathWorld:* A Wolfram Web Resource. <http://mathworld.wolfram.com/CellularAutomaton.html>. Acesso em 3 de Out. 2010.

21. MARTIN, Edwin. *John Conway's Game of Life*. Bitstorm.org. <http://www.bitstorm.org/gameoflife/> Acesso em 20 de out. 2010.

22. ZAMITH, Marcelo. CLUA, Esteban. PAGLIOSA, Paula., CONCI, Aura. VALENTE, Luís. FEIJO, Bruno. LEAL-TOLEDO, Regina. MONTENEGRO, Anselmo. *The GPU Used as a Math Co-Processor in Real Time Applications. Computers in Entertainment.* In: Computers in Entertainment. , v.6, p.1 – 19, 2008.

23. ZAMITH, Marcelo, LEAL-TOLEDO, Regina Célia Paula. KISCHINHEVSKY, Mauricio, CLUA, Esteban. BRANDÃO, Diego, MONTENEGRO, Anselmo, LIMA, Edgar. *A probabilistic cellular automata model for highway traffic simulation. Procedia Computer Science*. , v.1, p.337 – 345, 2010.

# 8 APÊNDICE

## 8.1 CÓDIGO FONTE DO PROJETO

O código fonte do sistema está disposto em 4 arquivos diferentes. O código principal está disposto no arquivo cudacap.cu. A API CUDA dispõe de arquivos do tipo .cu para o desenvolvimento de suas funções, mesmo assim, código C pode ser escrito normalmente nestes arquivos. Outros arquivos como cudacap_globals.h e cudacap_globals.cpp apenas definem variáveis usadas no sistema e vis_kernel.cu define as funções usadas para atualizar o buffer da saída de vídeo.

```
// cudacap.cu
// CUDACAP
// CUDA CELLULAR AUTOMATA PARTICLES
// IMPLEMENTAÇÃO DE UM SISTEMA DE DINÂMICA DE PARTÍCULAS
// COM UMA VIZINHANÇA DE MARGOLUS EM GPUS CUDA
////////////////////////////////////////////////////////////////////////////
LEONARDO LOURES QUIRINO DA SILVA
////////////////////////////////////////////////////////////////////////////

#include <windows.h>
#include <d3dx10.h>
#include <d3dx9.h>
#include <cuda_d3d9_interop.h>
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include <cutil.h>
#include <d3dx9.h>
#include <cutil_inline.h>

#include "cudacap_globals.h"
#include "vis_kernel.cu"

#define PI 3.14159265358979f
```

```
/// magic constants for rand48
const unsigned long a = 0x5DEECE66DLL;
const unsigned long c = 0xB;

/*//////////////////////////////////////////////////////////////////////
<CUDA Utilities Functions>
//////////////////////////////////////////////////////////////////////*/

// Simple utility function to check for CUDA runtime errors
void checkCUDAError(const char *msg) {
   cudaError_t err = cudaGetLastError();
   if( cudaSuccess != err) {
      fprintf(stderr, "Cuda error: %s: %s.\n", msg, cudaGetErrorString( err) );
      exit(-1);
   }
}

void Cleanup() {
            if( g_pVB       != NULL ) g_pVB->Release();
            if( g_pIB       != NULL ) g_pIB->Release();
   if( g_pd3dDevice != NULL ) g_pd3dDevice->Release();
   if( g_pD3D      != NULL ) g_pD3D->Release();

            if( myVertices   != NULL ) free(myVertices);
            if( myIndices    != NULL ) free(myIndices);

            CUDA_SAFE_CALL(cudaFree(devicePointer));
   CUDA_SAFE_CALL(cudaFree(visualPointer));
}
/*//////////////////////////////////////////////////////////////////////
</CUDA Utilities Functions>
//////////////////////////////////////////////////////////////////////*/

/*//////////////////////////////////////////////////////////////////////
<Random Number Generator Utilities Functions>

"Harvesting graphics power for MD simulations" by J.A. van Meel, A.
Arnold, D. Frenkel, S. F. Portegies Zwart and R. G. Belleman, Molecular Simulation,
Volume 34, Pages 259 - 266 (2008).
//////////////////////////////////////////////////////////////////////*/

// setup RNG execution grid
void initRNG(int seed, int size) {
            int nThreads = pow((float)size, 2);
            uint2* seeds = new uint2[ nThreads ];
            if (res == 0) {
                        CUDA_SAFE_CALL(cudaMalloc( (void**) &res, sizeof(int)*nThreads));
            }
            CUDA_SAFE_CALL( cudaMalloc( (void**) &state, sizeof(uint2)*nThreads ) );

            // calculate strided iteration constants
```

```
        unsigned long long A, C;
        A = 1LL; C = 0LL;
        for (unsigned int i = 0; i < nThreads; ++i) {
                C += A*c;
                A *= a;
        }
        A0 = A & 0xFFFFFFLL;
        A1 = (A >> 24) & 0xFFFFFFLL;
        C0 = C & 0xFFFFFFLL;
        C1 = (C >> 24) & 0xFFFFFFLL;

        // prepare first nThreads random numbers from seed
        unsigned long long x = (((unsigned long long)seed) << 16) | 0x330E;
        for (unsigned int i = 0; i < nThreads; ++i) {
                x = a*x + c;
                seeds[i].x = x & 0xFFFFFFLL;
                seeds[i].y = (x >> 24) & 0xFFFFFFLL;
        }
        CUDA_SAFE_CALL(cudaMemcpy(state, seeds, sizeof(uint2)*nThreads,
        cudaMemcpyHostToDevice));

        delete[] seeds;
}

__device__ uint2 RNG_rand48_iterate_single(uint2 Xn, uint2 A, uint2 C) {
  // results and Xn are 2x 24bit to handle overflows optimally, i.e.
  // in one operation.
  // the multiplication commands however give the low and hi 32 bit,
  // which have to be converted as follows:
  // 48bit in bytes = ABCD EF (space marks 32bit boundary)
  // R0          = ABC
  // R1          =    D EF

  unsigned int R0, R1;
  // low 24-bit multiplication
  const unsigned int lo00 = __umul24(Xn.x, A.x);
  const unsigned int hi00 = __umulhi(Xn.x, A.x);

  // 24bit distribution of 32bit multiplication results
  R0 = (lo00 & 0xFFFFFF);
  R1 = (lo00 >> 24) | (hi00 << 8);
  R0 += C.x; R1 += C.y;

  // transfer overflows
  R1 += (R0 >> 24);
  R0 &= 0xFFFFFF;

  // cross-terms, low/hi 24-bit multiplication
  R1 += __umul24(Xn.y, A.x);
  R1 += __umul24(Xn.x, A.y);
```

```
  R1 &= 0xFFFFFF;

  return make_uint2(R0, R1);
}

__device__ int RNG_rand48_get_int(uint2 *state, int *res, int num_blocks, uint2 A, uint2 C, int id) {
  const int nThreads = blockDim.x*gridDim.x;

  // load the current state of the RNG into a register
  int   nOutIdx = threadIdx.x + blockIdx.x*blockDim.x;
  uint2 lstate = state[nOutIdx];
  int i;
  for (i = 0; i < num_blocks; ++i) {
    // get upper 31 (!) bits of the 2x 24bits
    res[nOutIdx] = ( lstate.x >> 17 ) | ( lstate.y << 7);
    nOutIdx += nThreads;
    // this actually iterates the RNG
    lstate = RNG_rand48_iterate_single(lstate, A, C);
  }

  nOutIdx = threadIdx.x + blockIdx.x*blockDim.x;
  state[nOutIdx] = lstate;

  return res[id];
}


/*////////////////////////////////////////////////////////////////////
</Random Number Generator Utilities Functions>
////////////////////////////////////////////////////////////////////*/

/*////////////////////////////////////////////////////////////////////
<Cellular Automaton Functions>
////////////////////////////////////////////////////////////////////*/

__global__ void initCellularSpace(CELL *devicePointer) {
        int xid = blockIdx.x * blockDim.x + threadIdx.x;
        int yid = blockIdx.y * blockDim.y + threadIdx.y;
        int id = yid * gridDim.x * blockDim.x + xid;

        devicePointer[id].type                  = 0;
        devicePointer[id].speed                 = 0;
}

__global__ void initParticleGroups(CELL *devicePointer, float leftSpeedx, float leftSpeedy, float
leftMass, float rightSpeedx, float rightSpeedy, float rightMass, float speedParam)
{
        int xid = blockIdx.x * blockDim.x + threadIdx.x;
        int yid = blockIdx.y * blockDim.y + threadIdx.y;
        int id = yid*2 * gridDim.x * blockDim.x*2 + xid*2;
```

```
		if (sqrt (pow(((float)yid*2 - 200),2) + pow(((float)xid*2 - 125),2)) < 50){
			devicePointer[id].type			= 1;
			devicePointer[id].vector.x		= leftSpeedx / speedParam;
			devicePointer[id].vector.y		= leftSpeedy / speedParam;
			devicePointer[id].mass			= leftMass;
			devicePointer[id].flag			= 0;
			devicePointer[id].speed = sqrt(pow(leftSpeedx,2) + pow(leftSpeedy,2));

			//sets the counter
			double powx, powy;
			powx = pow(devicePointer[id].vector.x, 2);
			powy = pow(devicePointer[id].vector.y, 2);
			devicePointer[id].movimentCounter = 1 / (sqrt(powx + powy));

		} else if(sqrt (pow(((float)yid*2 - 200),2) + pow(((float)xid*2 - 250),2)) < 50){
			devicePointer[id].type			= 1;
			devicePointer[id].vector.x		= rightSpeedx / speedParam;
			devicePointer[id].vector.y		= rightSpeedy / speedParam;
			devicePointer[id].mass			= rightMass;
			devicePointer[id].flag			= 0;
			devicePointer[id].speed = sqrt(pow(rightSpeedx,2) + pow(rightSpeedy,2));

			//sets the counter
			double powx, powy;
			powx = pow(devicePointer[id].vector.x, 2);
			powy = pow(devicePointer[id].vector.y, 2);
			devicePointer[id].movimentCounter = 1 / (sqrt(powx + powy));
		}
}

__global__ void initGroupAndObstacle(CELL *devicePointer, float speedx, float speedy, float mass, float speedParam) {
		int xid = blockIdx.x * blockDim.x + threadIdx.x;
		int yid = blockIdx.y * blockDim.y + threadIdx.y;
		int id = yid*2 * gridDim.x * blockDim.x*2 + xid*2;

		if (sqrt (pow(((float)yid*2 - 170),2) + pow(((float)xid*2 - 125),2)) < 50){
			devicePointer[id].type			= 1;
			devicePointer[id].vector.x		= speedx / speedParam;
			devicePointer[id].vector.y		= speedy / speedParam;
			devicePointer[id].mass			= mass;
			devicePointer[id].flag			= 0;
			devicePointer[id].speed			= sqrt(pow(speedx,2) + pow(speedy,2));

			//sets the counter
			double powx, powy;
			powx = pow(devicePointer[id].vector.x, 2);
			powy = pow(devicePointer[id].vector.y, 2);
			devicePointer[id].movimentCounter = 1 / (sqrt(powx + powy));
		} else if (xid == 150 && yid < 150){
			devicePointer[id].type			= 2;
			devicePointer[id].vector.x		= -1;
```

```
                devicePointer[id].vector.y          =  0;

                devicePointer[id+1].type            = 2;
                devicePointer[id+1].vector.x        = -1;
                devicePointer[id+1].vector.y        =  0;

                devicePointer[id+512].type          = 2;
                devicePointer[id+512].vector.x   = -1;
                devicePointer[id+512].vector.y   =  0;

                devicePointer[id+513].type          = 2;
                devicePointer[id+513].vector.x   = -1;
                devicePointer[id+513].vector.y   =  0;

        } else if (yid == 150 && xid < 152 ){
                devicePointer[id].type              = 2;
                devicePointer[id].vector.x          = 0;
                devicePointer[id].vector.y          = -1;

                devicePointer[id+1].type            = 2;
                devicePointer[id+1].vector.x        = 0;
                devicePointer[id+1].vector.y        = -1;

                devicePointer[id+512].type          = 2;
                devicePointer[id+512].vector.x   = 0;
                devicePointer[id+512].vector.y   = -1;

                devicePointer[id+513].type          = 2;
                devicePointer[id+513].vector.x   = 0;
                devicePointer[id+513].vector.y   = -1;
        }
}



__global__ void eraseFutureDomain(CELL *devicePointerNext, size_t pitch) {
        int xid = blockIdx.x * blockDim.x + threadIdx.x;
        int yid = blockIdx.y * blockDim.y + threadIdx.y;
        int id = yid * gridDim.x * blockDim.x + xid;

        devicePointerNext[id].type          = 0;
        devicePointerNext[id].vector.x   = 0;
        devicePointerNext[id].vector.y   = 0;
        devicePointerNext[id].mass          = 0;
        devicePointerNext[id].flag          = 0;

        devicePointerNext[id].vector.x   = 0;
        devicePointerNext[id].vector.y   = 0;
}
```

```
//moves the particle into the next step space
__device__ void moveParticle(CELL *origin, int originId, CELL *destiny, int destinyId, double counter){
	destiny[destinyId].type 		= origin[originId].type;
	destiny[destinyId].mass 		= origin[originId].mass;
	destiny[destinyId].vector.x = origin[originId].vector.x;
	destiny[destinyId].vector.y = origin[originId].vector.y;
	destiny[destinyId].speed = sqrt(pow(origin[originId].vector.x, 2)+
	pow(origin[originId].vector.y, 2));
	destiny[destinyId].flag = origin[originId].flag;
	if (counter != 0) {
		destiny[destinyId].movimentCounter = counter;
	} else {
		double powx, powy;
		powx = pow(origin[originId].vector.x, 2);
		powy = pow(origin[originId].vector.y, 2);
		destiny[destinyId].movimentCounter = 1 / (sqrt(powx + powy));
	}
}//moveParticle

//chooses the best cell for the particle to move
__device__ void chooseParticleCell(CELL *devicePointer, CELL *devicePointerNext, int rootX, int rootY,
int cellX, int cellY, int myId ) {
	//int myId  = cellY * gridDim.x * blockDim.x *2 + cellX;
	float cos = fabsf(devicePointer[myId].vector.x) / devicePointer[myId].speed;
	int novaCoordX , novaCoordY;

	if (devicePointer[myId].vector.x != 0){
		novaCoordX = cellX + (devicePointer[myId].vector.x)/
	fabsf(devicePointer[myId].vector.x);
	}else{
		novaCoordX = cellX ;
	}
	if (devicePointer[myId].vector.y != 0){
		novaCoordY = cellY + (devicePointer[myId].vector.y)/
	fabsf(devicePointer[myId].vector.y);
	} else {
		novaCoordY = cellY ;
	}

	if ( 1 >= cos && cos > 0.86602) {
		novaCoordY = cellY ;
	} else if (0.5 > cos && cos >= 0) {
		novaCoordX = cellX ;
	}

	//choses the best cell to move the particle
	int coord;
	float maxDist = 10;//greater than any distance inside a neighborhood
	int bestX = novaCoordX;
	int bestY = novaCoordY;
```

```
	for (int disty = rootY; disty <= rootY + 1; disty++) {
		for (int distx = rootX; distx <= rootX + 1; distx++) {
			float auxdist = pow(((float)novaCoordY - disty),2) +
	pow(((float)novaCoordX - distx),2);

			if (devicePointerNext[disty* gridDim.x * blockDim.x *2 +
			distx].type == 0 && (auxdist < maxDist  || (auxdist ==
maxDist && disty == cellY && distx == cellX)))
			{
				maxDist = auxdist;
				bestX = distx;
				bestY = disty;
			}
		}
	}

	coord = bestY * gridDim.x * blockDim.x *2 + bestX;
	//move the particle into the future domain
	if (coord != myId && (devicePointer[myId].flag == 1 ||
	devicePointer[myId].movimentCounter == 0)) {
		//if the particle moves to a new cell
		devicePointer[myId].flag = 0;
	} else {
		//if particle tries to move to a prohibited cell and stays still it'll move in the next step, it's
counter stay untouched.
		coord = myId;
		if (devicePointer[myId].movimentCounter == 0){
			devicePointer[myId].flag = 1;
		}
	}
	moveParticle(devicePointer, myId, devicePointerNext, coord,
	devicePointer[myId].movimentCounter);

}//chooseParticleCell

/*//////////////////////////////////////////////////////////////////
</Cellular Automaton Functions>
//////////////////////////////////////////////////////////////////*/

/*//////////////////////////////////////////////////////////////////
<Collisions Functions>
//////////////////////////////////////////////////////////////////*/

__device__ void collideParticles(CELL *devicePointer, int index1, int index2, float e, float angle) {
	fComplex newSpeed1, newSpeed2, V0, R;

	V0.x = ((devicePointer[index1].mass * devicePointer[index1].vector.x) +
	(devicePointer[index2].mass * devicePointer[index2].vector.x)) /  (devicePointer[index1].mass + devicePointer[index2].mass);
	V0.y = ((devicePointer[index1].mass * devicePointer[index1].vector.y) +
	(devicePointer[index2].mass * devicePointer[index2].vector.y)) /  (devicePointer[index1].mass + devicePointer[index2].mass);
```

```
        float escalar1 = e * (devicePointer[index2].mass / (devicePointer[index1].mass +
        devicePointer[index2].mass));

        float escalar2 = e * (devicePointer[index1].mass / (devicePointer[index1].mass +
        devicePointer[index2].mass));

        angle = angle * PI / 180 ;
        R.x = cos(angle);
        R.y = sin(angle);

        fComplex speedDif;
        speedDif.x = devicePointer[index1].vector.x - devicePointer[index2].vector.x;
        speedDif.y = devicePointer[index1].vector.y - devicePointer[index2].vector.y;

        newSpeed1.x = V0.x+(escalar1)*sqrt(pow(speedDif.x,2)+pow(speedDif.y, 2)) * R.x;
        newSpeed1.y = V0.y+(escalar1)*sqrt(pow(speedDif.x,2)+pow(speedDif.y, 2)) * R.y;
        newSpeed2.x = V0.x-(escalar2)*sqrt(pow(speedDif.x,2)+pow(speedDif.y, 2)) * R.x;
        newSpeed2.y = V0.y-(escalar2)*sqrt(pow(speedDif.x,2)+pow(speedDif.y, 2)) * R.y;

        devicePointer[index1].speed = sqrt(pow( newSpeed1.x, 2) + pow( newSpeed1.y, 2));
        devicePointer[index2].speed = sqrt(pow( newSpeed2.x, 2) + pow( newSpeed2.y, 2));
        devicePointer[index1].vector.x = newSpeed1.x;
        devicePointer[index1].vector.y = newSpeed1.y;
        devicePointer[index2].vector.x = newSpeed2.x;
        devicePointer[index2].vector.y = newSpeed2.y;
        devicePointer[index1].flag = 1;
        devicePointer[index2].flag = 1;
}

__device__ void collideParticleObstacle (CELL *devicePointer, int partIndex, int                                    obstIndex,   float   e)
{
        float prodEscalar = (devicePointer[partIndex].vector.x *
        devicePointer[obstIndex].vector.x) + (devicePointer[partIndex].vector.y * devicePointer[obstIndex].vector.y);
                Complex newSpeed1;

        newSpeed1.x = devicePointer[partIndex].vector.x - (1 + e) * (prodEscalar)*devicePointer[obstIndex].vector.x;
        newSpeed1.y = devicePointer[partIndex].vector.y - (1 + e) * (prodEscalar)*devicePointer[obstIndex].vector.y;
        devicePointer[partIndex].speed   = sqrt(pow( newSpeed1.x, 2) + pow( newSpeed1.y, 2));
        devicePointer[partIndex].vector.x = newSpeed1.x;
        devicePointer[partIndex].vector.y = newSpeed1.y;
        devicePointer[partIndex].flag = 1;
}

/*//////////////////////////////////////////////////////////////////
</Collisions Functions>
//////////////////////////////////////////////////////////////////*/

/*//////////////////////////////////////////////////////////////////
<Step Functions>
//////////////////////////////////////////////////////////////////*/
```

```
//Erases the next step domain, decrements counters, moves obstacles into
//the next step domain and control border conditions
__global__ void domainStep(CELL *devicePointer, CELL *devicePointerNext, int start) {
	int xId = blockIdx.x * blockDim.x + threadIdx.x;
	int yId = blockIdx.y * blockDim.y + threadIdx.y;
	int myId = yId * gridDim.x * blockDim.x + xId;

	//This program look into the future to decide de particles best position to move,
	//so, the next step must be empty
	devicePointerNext[myId].type = 0;
	devicePointerNext[myId].flag = 0;

	if ( devicePointer[myId].type == 1 ) {
		//decrements the particles counter
		if (devicePointer[myId].movimentCounter >= 1){
			devicePointer[myId].movimentCounter--;
		}

		if (start == 1 && devicePointer[myId].movimentCounter < 1 &&
	devicePointer[myId].movimentCounter > 0)
		{
			devicePointer[myId].movimentCounter = 1 ;
		}
	} else if ( devicePointer[myId].type == 2 ) {
		//moves obstacles to the next domain
		devicePointerNext[myId].type = devicePointer[myId].type;
		devicePointerNext[myId].vector.x = devicePointer[myId].vector.x;
		devicePointerNext[myId].vector.y = devicePointer[myId].vector.y;
	}
	//border condition
	if (xId == 0 || xId == 511 || yId == 0 || yId == 511) {
		devicePointer[myId].type = 0;
		devicePointer[myId].flag = 0;
	}
}

//computes collisions through the neighborhood
__global__ void collisionStep(CELL *devicePointer, CELL *devicePointerNext, int start, float ePart, float eObst, int blockSize, uint2 *state,
int *res, int num_blocks, int A0, int A1, int C0, int C1) {
	int xId = blockIdx.x * blockDim.x + threadIdx.x;
	int yId = blockIdx.y * blockDim.y + threadIdx.y;
	int myId = yId * gridDim.x * blockDim.x + xId;
	int rootX, rootY;	//coordinates for the root cell
	//RNG params
	uint2 A, C;
	A.x = A0; A.y = A1;
	C.x = C0; C.y = C1;

	//acquire the block's root cell index
	if (start == 0){ //if it's a even step
		rootX	= xId *2;
```

```
        rootY       = yId * 2;
} else { // or a odd step
        rootX       = xId *2+1;
        rootY       = yId * 2+1;
}
int root = rootY * blockSize + rootX;

//Computes the Neighborhood
int myNeighborhood = devicePointer[root].type + devicePointer[root + 513].type +                devicePointer[root
        +1].type + devicePointer[root + 512].type;
if (myNeighborhood > 1) {
        //if there's more than one cell filled (particles and/or obstacles)
        int coord[4];
        int cells[4];
        int index = 0;
        int aux = 0;
        int count = 0;
        cells[0] = root;
        cells[1] = root + 1;
        cells[2] = root + 512;
        cells[3] = root + 513;
        for (int test = 0; test < 4; test++) {
                if (devicePointer[cells[test]].type != 0){
                        count++;
                } else {
                        cells[test] = -1;
                }
        }
        if (count == 2 ){
                while (aux < count) {
                        index = RNG_rand48_get_int((uint2 *)state, (int *)res, num_blocks, A, C, myId)%5-1;
                        if (index < 0){
                                index++;
                        }
                        if (cells[index] > -1  && devicePointer[cells[index]].type != 0 )    {
                                coord[aux] = cells[index];
                                cells[index] = -1;
                                aux++;
                        }
                }
        }  else  {
                while (aux < count) {
                        index = RNG_rand48_get_int((uint2 *)state, (int *)res,
                                num_blocks, A, C, myId)%5-1;

                        if (index < 0){
                                index++;
                        }
                        if (cells[index] > -1 ){
                                coord[aux] = cells[index];
                                cells[index] = -1;
```

```
                                        aux++;
                                }
                        }
                }

                int j = count-1;
                for (index = 0; index <count-1;index++){
                        int i= index;
                        if(devicePointer[coord[i]].type*devicePointer[coord[j]].type== 1){
                                //2 * 2 = 4 => 2 particles
                                float difx=devicePointer[coord[i]].vector.x-
                                        devicePointer[coord[j]].vector.y;

                                float dify = devicePointer[coord[i]].vector.y -
                                        devicePointer[coord[j]].vector.y;

                                if (difx != 0 || dify != 0 ){
                                        float angle = RNG_rand48_get_int((uint2 *)state,
                                                (int *)res, num_blocks, A, C, root)%360;

                                        collideParticles(devicePointer, coord[i], coord[j],
                                                ePart, angle);
                                }
                        } else if (devicePointer[coord[i]].type == 2 &&
                                        devicePointer[coord[j]].type == 1 &&
                                        devicePointer[coord[j]].flag == 0)
                        {
                                // 3 % 2 == 1 => obstacle and particle
                                collideParticleObstacle(devicePointer, coord[j], coord[i],
                                        eObst);

                        } else if (devicePointer[coord[i]].type == 1 &&
                                        devicePointer[coord[j]].type == 2 &&
                                        devicePointer[coord[0]].flag == 0)
                        {
                                // 2 % 3 == 2 => particle and obstacle
                                collideParticleObstacle(devicePointer, coord[i], coord[j],
                                        eObst);
                        }
                        if (j - 1 > index+1){
                                j--;
                        }
                }
        }
}

//Moves all particles into the next step domain acordingly to its speed, counter, and neighbors
__global__ void movementStep(CELL *devicePointer, CELL *devicePointerNext, int start) {
        int xId = blockIdx.x * blockDim.x + threadIdx.x;
        int yId = blockIdx.y * blockDim.y + threadIdx.y;
        int size = gridDim.x * blockDim.x * 2;
```

```
int rootX, rootY;//coordinates for the root cell
int oppX, oppY;//coordinates for the cell opposite to the root
int cwX, cwY;          //coordinates for the cell clockwise from the root
int ccwX, ccwY;//coordinates for the cell counter clockwise from the root

//acquire the block's cells index
if (start == 0){ //if it's a even step
        rootX       = xId *2;
        rootY       = yId * 2;
        cwX         = rootX+1;
        cwY         = rootY;
        oppX        = rootX+1;
        oppY        = rootY+1;
        ccwX        = rootX;
        ccwY        = rootY+1;
} else { // or a odd step
        rootX       = xId *2+1;
        rootY       = yId * 2+1;
        cwX         = rootX+1;
        cwY         = rootY;
        oppX        = rootX+1;
        oppY        = rootY+1;
        ccwX        = rootX;
        ccwY        = rootY+1;
}
int root = rootY * size  + rootX;
int opp  = oppY * size + oppX;
int cw   = cwY  * size + cwX;
int ccw  = ccwY * size + ccwX;
int myNeighborhood = devicePointer[root].type + devicePointer[opp].type +  devicePointer[cw].type + devicePointer[ccw].type;
if (myNeighborhood == 1) {
        if (devicePointer[root].type == 1) {
                if (devicePointer[root].movimentCounter >= 1 &&
                        devicePointer[root].flag == 0)
                {

                        moveParticle(devicePointer,root, devicePointerNext, root,
                devicePointer[root].movimentCounter);

                } else {
                        chooseParticleCell(devicePointer, devicePointerNext, rootX,
        rootY, rootX, rootY, root);
                }
        } else if (devicePointer[cw].type == 1) {
                if (devicePointer[cw].movimentCounter >= 1 &&
                devicePointer[cw].flag == 0)
                {
                        moveParticle(devicePointer,cw, devicePointerNext, cw,
                devicePointer[cw].movimentCounter);
                } else {
```

```
                        chooseParticleCell(devicePointer, devicePointerNext, rootX,
        rootY, cwX, cwY, cw);
                }
        } else if (devicePointer[opp].type == 1) {
                if (devicePointer[opp].movimentCounter >= 1 &&
                devicePointer[opp].flag == 0)
                {
                        moveParticle(devicePointer,opp, devicePointerNext, opp,
                devicePointer[opp].movimentCounter);
                } else {
                        chooseParticleCell(devicePointer, devicePointerNext, rootX,
        rootY, oppX, oppY, opp);
                }
        } else if (devicePointer[ccw].type == 1) {
                if (devicePointer[ccw].movimentCounter >= 1 &&
                devicePointer[ccw].flag == 0)
                {
                        moveParticle(devicePointer,ccw, devicePointerNext, ccw,
                devicePointer[root].movimentCounter);
                } else {
                        chooseParticleCell(devicePointer, devicePointerNext, rootX,
        rootY, ccwX, ccwY, ccw);
                }
        }

} else if (myNeighborhood > 1) {
        //arrange particle priorities for moviment and moves obstacles to the
        //future neighborhood
        int priority[4];
        int index = 0;
        for (int ini = 0; ini <=3;ini++){
                priority[ini] = -1;
        }
        for (int y = rootY; y <= rootY+1; y++) {
        //looks for particles in the neighborhood
                for (int x = rootX; x <= rootX+1; x++) {
                        if(devicePointer[y*gridDim.x*blockDim.x*2 + x].type == 1) {
                                float movQuant = devicePointer[y*gridDim.x *
                        blockDim.x*2 + x].mass *
                devicePointer[y*gridDim.x * blockDim.x*2 +
        x].speed;

                                index = 0;
                                while (priority[index] > -1 && movQuant <
                                (devicePointer[priority[index]].mass
                *devicePointer[priority[index]].speed))
                                {
                                        index++;
                                }

                                for (int k=3; k>index; k--){
```

```
                                                priority[k] = priority[k-1];
                                        }
                                        priority[index] = y*gridDim.x * blockDim.x *2 + x;
                                }
                        }
                }
                for (int i = 0; i < 4; i++) {
                        if (priority[i] == root){
                                chooseParticleCell(devicePointer, devicePointerNext, rootX,
                rootY, rootX, rootY, root);
                        } else if (priority[i] == ccw){
                                chooseParticleCell(devicePointer, devicePointerNext, rootX,
                rootY, ccwX, ccwY, ccw);
                        } else if (priority[i] == opp){
                                chooseParticleCell(devicePointer, devicePointerNext, rootX,
                rootY, oppX, oppY, opp);
                        } else if (priority[i] == cw){
                                chooseParticleCell(devicePointer, devicePointerNext, rootX,
                rootY, cwX, cwY, cw);
                        }
                }
        }
}
/*////////////////////////////////////////////////////////////////////
</Step by step Functions>
////////////////////////////////////////////////////////////////////*/

/*////////////////////////////////////////////////////////////////////
<Simulation in time domain>
////////////////////////////////////////////////////////////////////*/
void runMargolus(int step) {
        dim3 margolusBlocks(NUMBEROFMARGOLUSBLOCKS, NUMBEROFMARGOLUSBLOCKS);        //16*16
        dim3 blocksPerGrid(NUMBEROFBLOCKS, NUMBEROFBLOCKS);
        //32*32
        dim3 threadsPerBlock(THREADNUMPERBLOCK, THREADNUMPERBLOCK);                 //16*16
        int size = NUMBEROFBLOCKS*THREADNUMPERBLOCK;
        int rngSize = NUMBEROFMARGOLUSBLOCKS*THREADNUMPERBLOCK;

        if (step >= 0) {
                int start = step % 2;
                // Update colors of the visual grid according to the current simulation
                // grid
                if (graphics && start == 0){
                        //printf("step: %d \n", step);
                        CUDA_SAFE_CALL(cudaD3D9MapVertexBuffer((void**)&visualPointer,
                        g_pVB));

                        updateVertexColors<<<blocksPerGrid,
                        threadsPerBlock>>>(visualPointer, devicePointer);

                        CUDA_SAFE_CALL(cudaD3D9UnmapVertexBuffer(g_pVB));
```

```
                }

                unsigned int timer = 0;
                cutilCheckError( cutCreateTimer( &timer));
                cutilCheckError( cutStartTimer( timer));
                /* Kernel calls */
                //1 thread / cell
                domainStep<<<blocksPerGrid,threadsPerBlock>>>(devicePointer,
                        devicePointerNext,start);

                //1 thread / margolus block
                //initializing the RNG - Just Like srand() in C.
                collisionStep<<<margolusBlocks,threadsPerBlock>>>(devicePointer,
                        devicePointerNext,start,ePart,eObst,size,(uint2*)state,res,  rngSize, A0, A1, C0, C1);

                cudaThreadSynchronize();
                movementStep<<< margolusBlocks, threadsPerBlock >>>(devicePointer,
                        devicePointerNext, start);

                cutilCheckError( cutStopTimer( timer));
                printf("Processing time: %f (ms)\n", cutGetTimerValue( timer));

                devicePointerAux = devicePointer;
                devicePointer = devicePointerNext;
                devicePointerNext = devicePointerAux;
        } else {
                //initialize the system
                double powx = pow(bigSpeed, 2);
                double powy = pow((float)0, 2);
                double moduloVelocidade          = sqrt( powx + powy);
                printf("initializing the cellular automata domain. \n");
                initCellularSpace <<< blocksPerGrid, threadsPerBlock >>>(devicePointer);
                initParticleGroups<<< margolusBlocks, threadsPerBlock >>>(devicePointer,
        bigSpeed, 0, bigMass, smallSpeed, 0, bigMass, moduloVelocidade);
                //initGroupAndObstacle<<<margolusBlocks,threadsPerBlock>>(devicePointer,bigSpeed,0,bigMass,
                        moduloVelocidade);
                initRNG(time(0), rngSize);
        }
        cudaThreadSynchronize();
        CUT_CHECK_ERROR("Kernel execution failed\n");
        checkCUDAError("Error");
}
/*////////////////////////////////////////////////////////////////////////
</Simulation in time domain>
////////////////////////////////////////////////////////////////////////*/

/*////////////////////////////////////////////////////////////////////////
<Visualization Functions>
////////////////////////////////////////////////////////////////////////*/
LRESULT WINAPI MsgProc(HWND handleWindow, UINT msg, WPARAM wParam, LPARAM lParam) {
        switch( msg ) {
```

```
                    case WM_DESTROY:
                            CUDA_SAFE_CALL(cudaD3D9UnregisterVertexBuffer(g_pVB));
                            CUDA_SAFE_CALL(cudaD3D9End());
                            Cleanup();
                    PostQuitMessage( 0 );
                            return 0;
            }
            return DefWindowProc( handleWindow, msg, wParam, lParam );
}

//////////////////////////////////////////////////////////////////////////////
void Render(int step) {
   // Clear the backbuffer
   g_pd3dDevice->Clear( 0, NULL, D3DCLEAR_TARGET, D3DCOLOR_XRGB(255,255,255),1.0f,0);
   // Run CUDA to calculate next FDTD step
   runMargolus(step);
   // Begin the scene
   if( SUCCEEDED( g_pd3dDevice->BeginScene() ) ) {
                    // Render the visual grid
                    g_pd3dDevice->SetStreamSource( 0, g_pVB, 0, sizeof(CUSTOMVERTEX) );
                    g_pd3dDevice->SetIndices( g_pIB );
                    g_pd3dDevice->SetFVF( D3DFVF_CUSTOMVERTEX );
                    g_pd3dDevice->DrawIndexedPrimitive(D3DPT_TRIANGLELIST,0,0,RESX*RESY,0,
                    (RESX-1)*(RESY-1)*2 );
                    // End the scene
                    g_pd3dDevice->EndScene();
            }
            // Present the backbuffer contents to the display
            g_pd3dDevice->Present( NULL, NULL, NULL, NULL );
}

//////////////////////////////////////////////////////////////////////////////
int WINAPI WinMain(HINSTANCE hInstance,
                    HINSTANCE hPrevInstance,
                    LPSTR lpCmdLine,
                    int nShowCmd)
{
            // Register the window class
            WNDCLASSEX windowClass = { sizeof(WNDCLASSEX), CS_CLASSDC, MsgProc, 0L, 0L,
                     hInstance, NULL, LoadCursor(NULL,IDC_ARROW), NULL, NULL, "CUDA SDK", NULL };

            RegisterClassEx( &windowClass );
            // Create the application's window
            HWND handleWindow = CreateWindow( windowClass.lpszClassName, "Direct3D visualization of CUDA-based
                    Sierpinski",WS_OVERLAPPEDWINDOW, 0, 0, RESX+4, RESY+30, NULL, NULL, windowClass.hInstance,
                    NULL);
             // Initialize Direct3D
             if( SUCCEEDED( InitD3D( handleWindow ) ) ) {
             // Create the scene geometry
                    if( SUCCEEDED(InitGeometry())) {
                            // Initialize interoperability between CUDA and Direct3D
```

```
                                CUDA_SAFE_CALL(cudaD3D9Begin(g_pd3dDevice));
                                // Register vertex buffer with CUDA
                                CUDA_SAFE_CALL(cudaD3D9RegisterVertexBuffer(g_pVB));
                                // Show the window
                                ShowWindow(handleWindow, nShowCmd);
                                UpdateWindow( handleWindow );
                                // Enter the message loop
                                MSG msg;
                                ZeroMemory( &msg, sizeof(msg) );
                                int step = -1;
                                while( msg.message!=WM_QUIT ) {
                                        if( PeekMessage( &msg, NULL, 0U, 0U, PM_REMOVE ) )
                                        {
                                                TranslateMessage( &msg );
                                                DispatchMessage( &msg );
                                        } else {
                                                Render(step);
                                                step++;
                                        }
                                }
                                UnregisterClass( windowClass.lpszClassName, windowClass.hInstance );
                                 return 1;
                        }else printf("Error: Scene geometry could not be created, most likely due to   insufficient video memory.");
                }else printf("Error: Direct3D device could not be created.");
                printf(" Graphical output is disabled.\n");
                UnregisterClass( windowClass.lpszClassName, windowClass.hInstance );
                return 0;
}
/*////////////////////////////////////////////////////////////////////
</Visualization Functions>
////////////////////////////////////////////////////////////////////*/

/*////////////////////////////////////////////////////////////////////
<MAIN Function>
////////////////////////////////////////////////////////////////////*/

int main(int argc, char** argv) {
                printf("A CUDA Cellullar Automaton based Particles Dinamics system \n");
                printf("using 2D arrays allocated with cudaMallocPitch. \n");
                printf("---------------------------- \n");

                printf("Allocating 2D array directly in the device memory with cudaMallocPitch,                                 \n");
                printf("so the allocation is appropriately padded to meet the alignment                                 requirements \n");
                printf("for global memory access \n");

                printf("Allocating the 2 step cellular automata spaces  \n");
                size_t width = RESX * sizeof(CELL);         //512
                size_t height = RESY;                                                   //512
                cudaMallocPitch((void**)&devicePointer, &pitch1, width, height);
                cudaMallocPitch((void**)&devicePointerNext, &pitch2, width, height);
                printf("----------------------------\n");
```

```
            printf("Generating visualization window\n");
            if (!WinMain(GetModuleHandle(NULL), NULL, "", SW_SHOWNORMAL)) {
                        // Check for any CUDA errors
                        checkCUDAError("Error");
            }
            Cleanup();
            cutilExit(argc, argv);
   printf("\nPress ENTER to exit...\n");
   getchar();
}


////////////////////////////////////////////////////////////////////////////
//vis_kernel.cu
////////////////////////////////////////////////////////////////////////////
#ifndef _VIS_KERNEL_
#define _VIS_KERNEL_
////////////////////////////////////////////////////////////////////////////
//HOST CODE FOR INTERFACE
////////////////////////////////////////////////////////////////////////////
HRESULT InitD3D(HWND handleWindow) {
   // Create the D3D object.
   if( NULL == ( g_pD3D = Direct3DCreate9( D3D_SDK_VERSION ) ) )
      return E_FAIL;
            // Set up the structure used to create the D3DDevice
   D3DPRESENT_PARAMETERS d3dpp;
   ZeroMemory( &d3dpp, sizeof(d3dpp) );
   d3dpp.Windowed = TRUE;
   d3dpp.SwapEffect = D3DSWAPEFFECT_DISCARD;
   d3dpp.BackBufferFormat = D3DFMT_UNKNOWN;

   // Create the D3DDevice
   if ( FAILED( g_pD3D->CreateDevice( D3DADAPTER_DEFAULT, D3DDEVTYPE_HAL, handleWindow,
                              D3DCREATE_HARDWARE_VERTEXPROCESSING,
                              &d3dpp, &g_pd3dDevice ) ) )
      return E_FAIL;

   // Turn off culling, so we see the front and back of the triangle
   if( FAILED( g_pd3dDevice->SetRenderState( D3DRS_CULLMODE, D3DCULL_NONE ) ) )
      return E_FAIL;

   // Turn off D3D lighting, since we are providing our own vertex colors
   if( FAILED( g_pd3dDevice->SetRenderState( D3DRS_LIGHTING, FALSE ) ) )
      return E_FAIL;

   return S_OK;
}

////////////////////////////////////////////////////////////////////////////
HRESULT InitGeometry() {
```

```
        unsigned int sizeV = RESX * RESY * sizeof(CUSTOMVERTEX),
                                        sizeI = 6*(RESX-1)*(RESY-1) * sizeof(DWORD);
        myVertices = (CUSTOMVERTEX*) malloc(sizeV);
        myIndices = (DWORD*) malloc(sizeI);

        // Prepare vertex buffer in host memory
        for (unsigned int i=0; i<RESY; ++i)
                for (unsigned int j=0; j<RESX; ++j) {
                        myVertices[i*RESX+j].color = D3DCOLOR_XRGB(0,0,0);
                        myVertices[i*RESX+j].w = 1.0f;
                        myVertices[i*RESX+j].z = 0.5f;
                        myVertices[i*RESX+j].x = (float)j;
                        myVertices[i*RESX+j].y = (float)i;
                }
        // Prepare index buffer in host memory
        for (unsigned int i=0; i<RESY-1; ++i)
                for (unsigned int j=0; j<RESX-1; ++j) {
                        //myIndices[(i*(RESX-1)+j)]   = i;
                        myIndices[(i*(RESX-1)+j)*6]   = (i+1)*RESX+j;
                        myIndices[(i*(RESX-1)+j)*6+1] = i*RESX+j;
                        myIndices[(i*(RESX-1)+j)*6+2] = i*RESX+j+1;
                        myIndices[(i*(RESX-1)+j)*6+3] = (i+1)*RESX+j+1;
                        myIndices[(i*(RESX-1)+j)*6+4] = (i+1)*RESX+j;
                        myIndices[(i*(RESX-1)+j)*6+5] = i*RESX+j+1;
                }

        // Create vertex buffer
        if (FAILED(g_pd3dDevice->CreateVertexBuffer(sizeV, NULL, D3DFVF_CUSTOMVERTEX,
        D3DPOOL_DEFAULT, &g_pVB, NULL)))
                return E_FAIL;

        // Create index buffer
        if (FAILED(g_pd3dDevice->CreateIndexBuffer(sizeI, NULL, D3DFMT_INDEX32,
        D3DPOOL_DEFAULT, &g_pIB, NULL)))
                return E_FAIL;

        // Fill vertex buffer
        VOID* pBufferPtr;
        if(FAILED(g_pVB->Lock(0, sizeV, &pBufferPtr, 0))) return E_FAIL;
        memcpy(pBufferPtr, myVertices, sizeV);
        g_pVB->Unlock();
        // Fill index buffer
        if(FAILED(g_pIB->Lock(0, sizeI, &pBufferPtr, 0))) return E_FAIL;
        memcpy(pBufferPtr, myIndices, sizeI);
        g_pIB->Unlock();

    return S_OK;
}

//////////////////////////////////////////////////////////////////////
//DEVICE CODE FOR INTERFACE
```

```
////////////////////////////////////////////////////////////////////
// Maps float to D3DCOLOR
// vmin, vmax: lower and upper boundary of the range of float values
//         v: the actual value to be mapped, should be within [vmin,max]
////////////////////////////////////////////////////////////////////
__device__ unsigned int ColorFromFloat(float vmin, float vmax, float v) {
            // Apply color mapping */
            float R, G, B;
            if (v == 0) {
                        R = 255.0f;
                        G = 255.0f;
                        B = 255.0f;
            } else {
                        if (v == 1) {
                                    R = 0.0f;
                                    G = 0.0f;
                                    B = 0.0f;
                        } else {
                                    R = 0.0f;
                                    G = 255.0f;
                                    B = 0.0f;
                        }
            }
            // Convert to unsigned int*/
            return 255<<24 | (unsigned char)R<<16 | (unsigned char)G<<8 | (unsigned char)B;
}

__global__ void updateVertexColors(float* vb, CELL* C) {
            // Calculate offset
            unsigned int kn = gridDim.x * blockDim.x * blockDim.y * blockIdx.y + gridDim.x * blockDim.x * threadIdx.y + blockDim.x *
blockIdx.x + threadIdx.x;
            // Change color of the associated vertex
            vb[kn*5+4] = __int_as_float(ColorFromFloat(-0.1f, 0.1f, C[kn].type));
}

#endif
//////////////////////////////////////////////////////////////////////////////
//cudacap_globals.h
//////////////////////////////////////////////////////////////////////////////
#ifndef _ACFDTD_GLOBALS_H_
#define _ACFDTD_GLOBALS_H_
#ifdef __CUDACC__
   typedef float2 fComplex;
#else
   typedef struct{
      float x;
      float y;
   } fComplex;
#endif

#include <d3dx9.h>
```

```
typedef struct{
	int x;
	int y;
} COORDVEC;

struct CELL {
//used for representing an empty cell, a particle or obstacle in the Margolus Neighborhood
	int	type;			//type of particle in the cell
	float	mass;			//particle mass
	fComplex	vector;			//used for particle velocity simulation or obstacle normal vector
	float	movimentCounter;		//counter for number of steps for a particle to move
	float	speed;			//speed of the particle OR lenght
	int	flag;			//flag for already processed particles
};

extern "C" {
	extern float ePart;	//resilience coeficient between particles
	extern float eObst;	//resilience coeficient between particle and obstacle
	extern int	NUMBEROFBLOCKS;
	extern int	NUMBEROFMARGOLUSBLOCKS;
	extern int	THREADNUMPERBLOCK;
	extern size_t pitch1;
	extern size_t pitch2;
	extern CELL *hostPointer;
	extern CELL *devicePointer;
	extern CELL *devicePointerNext;
	extern CELL *devicePointerAux;
	extern float *visualPointer; //device matrix for viewing
	//RNG
	extern int	threadsX;
	extern int	blocksX;
	extern void *state;
	extern int	*res;
	extern unsigned int	A0;
	extern unsigned int	A1;
	extern unsigned int	C0;
	extern unsigned int	C1;
	extern int		size;
	extern int		num_blocks;
}

/* ================== Processing component ================== */
extern "C" {
	extern CELL cell;
}
// Custom vertex structure, each vertex represents one cell of the simulation grid
struct CUSTOMVERTEX {
	FLOAT x, y, z, w;	// vertex position in screen space
	DWORD color;			// vertex color
};
```

```
// Custom FVF corresponding to the above-defined vertex structure
#define D3DFVF_CUSTOMVERTEX (D3DFVF_XYZRHW|D3DFVF_DIFFUSE)

extern "C" {
	// Window dimensions in pixels (the visual grid is always scaled to exactly fit in,
	// so W/H and RESX/RESY should be the same for an undistorted image)
	extern unsigned int RESX, RESY;
	extern float		bigSpeed, smallSpeed;
	extern float		bigMass, smallMass;
	extern float		moduloVelocidade;

	extern LPDIRECT3D9 g_pD3D;
	extern LPDIRECT3DDEVICE9 g_pd3dDevice;

	extern LPDIRECT3DVERTEXBUFFER9 g_pVB;	// Stores vertex positions of the visual grid
	extern LPDIRECT3DINDEXBUFFER9 g_pIB;	// Stores the topology of the vertices
	extern CUSTOMVERTEX *myVertices;		// The "vertex buffer" on the host
	extern DWORD *myIndices;				// The "index buffer" on the host

	extern unsigned char graphics;	// Flag to enable or disable graphical output
}

#endif // #ifndef _ACFDTD_GLOBALS_H_

//////////////////////////////////////////////////////////////////////////////
//cudacap_globals.cpp
//////////////////////////////////////////////////////////////////////////////

#include "cudacap_globals.h"
unsigned int RESX	= 512;
unsigned int RESY	= 512;

int NUMBEROFBLOCKS		= 32;
int NUMBEROFMARGOLUSBLOCKS	= 16;
int THREADNUMPERBLOCK		= 16;//One thread per cell

float	bigSpeed	= 10;
float	smallSpeed = -10;
float	bigMass	= 4;
float	smallMass = 1;
float	ePart	= 1; //resilience coeficient between particles
float	eObst	= 1; //resilience coeficient between particle and obstacle
float	speedParam= 0; //relative max speed for initializing the domain

// pointer for device memory
size_t	pitch1	= NULL;	//pitch pointer to the device memory
size_t	pitch2	= NULL;	//pitch pointer to the device memory

CELL *hostPointer = NULL;
CELL *devicePointer = NULL; //Pointer to the current domain
```

```
CELL *devicePointerNext = NULL; //Pointer to the next step domain
CELL *devicePointerAux = NULL;
Float *visualPointer = NULL; //device matrix for viewing
Int threadsX = NULL;
Int blocksX = NULL;
Void *state = NULL;
Int *res = NULL;
unsigned int A0 = NULL;
unsigned int A1 = NULL;
unsigned int C0 = NULL;
unsigned int C1 = NULL;
int size = NULL;
int num_blocks = NULL;
LPDIRECT3D9 g_pD3D = NULL;
LPDIRECT3DDEVICE9 g_pd3dDevice = NULL;
LPDIRECT3DVERTEXBUFFER9 g_pVB  = NULL;
LPDIRECT3DINDEXBUFFER9 g_pIB      = NULL;
CUSTOMVERTEX *myVertices = NULL;
DWORD *myIndices = NULL;
unsigned char graphics  = 1;
```