MINISTÉRIO DA DEFESA
EXÉRCITO BRASILEIRO
DEPARTAMENTO DE CIÊNCIA E TECNOLOGIA
INSTITUTO MILITAR DE ENGENHARIA
CURSO DE MESTRADO EM SISTEMAS E COMPUTAÇÃO

EDUARDO CAMARGO

UMA FERRAMENTA DE VISUALIZAÇÃO CIENTÍFICA BASEADA NO ALGORITMO DE CÁLCULO DE TRAJETÓRIA DE PARTÍCULAS EM UNIDADES DE PROCESSAMENTO GRÁFICO

Rio de Janeiro
2011

INSTITUTO MILITAR DE ENGENHARIA

EDUARDO CAMARGO

UMA FERRAMENTA DE VISUALIZAÇÃO CIENTÍFICA BASEADA NO ALGORITMO DE CÁLCULO DE TRAJETÓRIA DE PARTÍCULAS EM UNIDADES DE PROCESSAMENTO GRÁFICO

Dissertação de Mestrado apresentada ao Curso de Mestrado em Sistemas e Computação do Instituto Militar de Engenharia, como requisito parcial para obtenção do título de Mestre em Sistemas e Computação.

Orientador: Prof. Sérgio Kostin - D.Sc.

Rio de Janeiro
2011

c2011

INSTITUTO MILITAR DE ENGENHARIA
Praça General Tibúrcio, 80-Praia Vermelha
Rio de Janeiro-RJ CEP 22290-270

Este exemplar é de propriedade do Instituto Militar de Engenharia, que poderá incluí-lo em base de dados, armazenar em computador, microfilmar ou adotar qualquer forma de arquivamento.

É permitida a menção, reprodução parcial ou integral e a transmissão entre bibliotecas deste trabalho, sem modificação de seu texto, em qualquer meio que esteja ou venha a ser fixado, para pesquisa acadêmica, comentários e citações, desde que sem finalidade comercial e que seja feita a referência bibliográfica completa.

Os conceitos expressos neste trabalho são de responsabilidade do autor e do orientador.

C172M Camargo, Eduardo.
Uma Ferramenta de Visualização Científica Baseada no Algoritmo de Cálculo de Trajetória de Partículas em Unidades de Processamento Gráfico/ Eduardo Camargo. – Rio de Janeiro: Instituto Militar de Engenharia, 2011.

107 p.: il.

Dissertação (mestrado) – Instituto Militar de Engenharia, 2011.

1.Transmissão da Informação - Tecnologias. 2. Sistemas de Computação. 3. Tratamento da Informação. I. Título. II. Instituto Militar de Engenharia.

CDD 629.892

INSTITUTO MILITAR DE ENGENHARIA

EDUARDO CAMARGO

UMA FERRAMENTA DE VISUALIZAÇÃO CIENTÍFICA BASEADA NO ALGORITMO DE CÁLCULO DE TRAJETÓRIA DE PARTÍCULAS EM UNIDADES DE PROCESSAMENTO GRÁFICO

Dissertação de Mestrado apresentada ao Curso de Mestrado em Sistemas e Computação do Instituto Militar de Engenharia, como requisito parcial para obtenção do título de Mestre em Sistemas e Computação.

Orientador: Prof. Sérgio Kostin - D.Sc.

Aprovada em 03 de agosto de 2011 pela seguinte Banca Examinadora:

Prof. Sérgio Kostin - D.Sc. do IME - Presidente

Prof. Raquel Coelho Gomes Pinto - D.Sc. do IME

Prof. Anderson Fernandes Pereira dos Santos - D.Sc. do IME

Prof. Lauro Luis Armondi Whately - D.Sc. da UFRJ

Rio de Janeiro
2011

Dedico esta à minha noiva (quase esposa) Nathália e à memória de minha tia Dilmar.

AGRADECIMENTOS

Primeiramente, agradeço à Deus pela oportunidade que me foi dada e por me permitir chegar até aqui.

À toda minha família e, principalmente, aos meus pais por tudo que fizeram por mim e mais do que isso pela formação do meu caráter.

Às pessoas que contribuíram com o desenvolvimento desta dissertação, tenha sido por meio de críticas, idéias, apoio, incentivo ou qualquer outra forma de auxílio. Em especial, desejo agradecer aos professores Sérgio e Raquel por todo trabalho e empenho dedicados para que este trabalho fosse concluído. Agradeço também ao companheiro de estudos e colega de trabalho Marcelo Collares pelas discussões, sugestões, sábados e domingos de estudo no LNCC.

Por fim, a todos os professores e funcionários do Departamento de Engenharia de Sistemas (SE/8) do Instituto Militar de Engenharia.

Eduardo Camargo

"*Lembre-se que as pessoas podem tirar tudo de você, menos o seu conhecimento. É o seu bem mais precioso. Explore; viaje; descubra. Conheça.*"

"*Albert Einstein*"

SUMÁRIO

LISTA DE ILUSTRAÇÕES 9

LISTA DE TABELAS 11

LISTA DE ABREVIATURAS E SÍMBOLOS 12

1 INTRODUÇÃO 15
1.1 Contexto 15
1.2 Motivação 15
1.3 Objetivo 16
1.4 Estrutura do Trabalho 17

2 VISUALIZAÇÃO CIENTÍFICA 18
2.1 Técnicas de Visualização Científica 18
2.2 HeMoLab 19

3 PROGRAMAÇÃO DE PROPÓSITO GERAL EM GPU 25
3.1 Histórico de Evolução das GPUs 25
3.2 Arquitetura das Atuais GPUs 28
3.3 Modelo de Programação CUDA 30
3.4 Interfaces de Programação 32
3.5 Gerenciamento de Threads 33
3.6 Hierarquia de Memória 38
3.7 Considerações sobre Desempenho 40

4 REVISÃO DA LITERATURA 46
4.1 Arquitetura CUDA 46
4.2 Algoritmo de Trajetória de Partículas 49

5 DESCRIÇÃO DA SOLUÇÃO PROPOSTA 51
5.1 Introdução 51
5.2 Implementação Atual 53
5.3 Implementação do CTP Utilizando CUDA 57
5.3.1 Principais Estruturas Utilizadas 58

5.3.2 Posicionamento Inicial da Partículas 59
5.3.3 Divisão das metacélulas 61
5.3.4 Cálculo das Trajetórias 62
5.3.5 Geração dos Arquivos EnSight 67
5.4 Validação da Implementação 67

6 RESULTADOS 69
6.1 Descrição dos experimentos 69
6.2 Objetivos 72
6.3 Análise dos Resultados 72
6.3.1 Experimento 1 72
6.3.2 Experimento 2 74
6.3.3 Experimento 3 76

7 CONCLUSÕES E TRABALHOS FUTUROS 80
7.1 Conclusões 80
7.2 Trabalhos Futuros 80

8 REFERÊNCIAS BIBLIOGRÁFICAS 82

9 APÊNDICES 85
9.1 Apêndice 1: Pseudo-código 86
9.2 Apêndice 2: Divisão das metacélulas 86

LISTA DE ILUSTRAÇÕES

FIG.2.1 Técnicas de visualização. Fonte: (KITWARE, 2009b) 18
FIG.2.2 Módulo Processamento de Imagens 20
FIG.2.3 Árvore Arterial Utilizada pelo Módulo 1D 21
FIG.2.4 Malha de Volume Utilizada pelo Módulo 3D 22
FIG.2.5 Módulo Acoplamento. 22
FIG.2.6 *Pipeline* de Utilização Típico do *HeMoLab* com a fase de visualização em destaque. 23
FIG.2.7 Arquitetura do *ParaView* 24

FIG.3.1 Uso de *Pixel* e *Vertex Shaders* em contraste com *Stream Processors*. 25
FIG.3.2 *Streams* e *Kernel*. Fonte: (KAPASI, 2003) 26
FIG.3.3 Operações de ponto flutuante por segundo. Fonte: (NVIDIA, 2011) 27
FIG.3.4 Distribuição de transistores nas GPUs e CPUs. Fonte: (NVIDIA, 2010) 28
FIG.3.5 Arquitetura G80. Fonte: (KIRK, 2010) 29
FIG.3.6 CUDA - Suporte para várias linguagens. Fonte: (NVIDIA, 2010) 31
FIG.3.7 Arquitetura da GPU e Divisão do código serial e paralelo. Fonte: (NVIDIA, 2010) 32
FIG.3.8 Organização dos *threads* em CUDA. Fonte: (NVIDIA, 2010) 33
FIG.3.9 Pilha de *software* CUDA. Fonte: (NVIDIA, 2010) 34
FIG.3.10 Divisão de uma matriz em mais de 1 bloco. Fonte: (KIRK, 2010) 35
FIG.3.11 Multiplicação de matrizes utilizando múltiplos blocos. Fonte: (KIRK, 2010) 36
FIG.3.12 Escalabilidade transparente para programas CUDA. Fonte: (NVIDIA, 2011) 37
FIG.3.13 Hierarquia de memória. 39
FIG.3.14 Execução do *Kernel* de redução de soma. Fonte: (KIRK, 2010) 41
FIG.3.15 Execução do novo Kernel de redução de soma. Fonte: (KIRK, 2010) 42

FIG.4.1 Simulação de tráfego. Fonte: (CHE, 2008) 46
FIG.4.2 Simulação térmica. Fonte: (CHE, 2008) 47
FIG.4.3 Algoritmo k-médias. Fonte: (CHE, 2008) 48

FIG.5.1 Comparação de técnicas de visualização em um fluxo instável. 52
FIG.5.2 *Pipeline* geral do algoritmo de trajetória de partículas. 54
FIG.5.3 Posicionamento das fronteiras entre metacélulas. 55
FIG.5.4 Sobreposição de fronteiras entre as metacélulas. 56
FIG.5.5 Deslocamento da partícula no interior do tetraedro. 57
FIG.5.6 Fluxograma do algoritmo de CTP em CPU. 58
FIG.5.7 Fluxograma do algoritmo de CTP em GPU. 59
FIG.5.8 *Line Widget.* 60
FIG.5.9 *Sphere Widget.* 60
FIG.5.10 *Points Cloud Widget.* 61
FIG.5.11 Fluxograma do algoritmo de CTP em GPU (detalhamento dos *threads*). 62
FIG.5.12 Comparação da execução das versões serial e paralela do CTP 64
FIG.5.13 Verificação de proximidade entre partícula e triângulo. 65
FIG.5.14 Movendo a partícula. 66
FIG.5.15 Comparação das trajetórias obtidas em CPU (tela a esquerda) e GPU (tela a direita). 68

FIG.6.1 Saída do programa *DeviceQuery*. 69
FIG.6.2 Diagrama da GeForce 9600GT. 70
FIG.6.3 Malha vinda da simulação. 71
FIG.6.4 Tempo médio de execução de 1 até 100 partículas. 73
FIG.6.5 *Speedup* de 1 até 100 partículas. 74
FIG.6.6 Tempo médio de execução de 100 até 1536 partículas. 75
FIG.6.7 *Speedup* de 100 até 1536 partículas. 75
FIG.6.8 Transferência de Dados *vs* Tempo de Processamento de 1 até 1536 partículas. 76
FIG.6.9 Efeito da distribuição de *threads* pelos blocos no experimento 3. 77

FIG.9.1 Malha triangular e seus baricentros. 87
FIG.9.2 Metacélulas de triângulos. 90

LISTA DE TABELAS

TAB.3.1 Comparação entre os *chips* G80, GT200 e Fermi. Fonte: (NVIDIA, 2011) 30
TAB.3.2 Dimensões de grades e blocos 35
TAB.3.3 Qualificadores de variáveis em CUDA 38

TAB.6.1 Configuração dos Experimentos realizados 71
TAB.6.2 Blocos e warps ativos em cada multiprocessador. 77

TAB.9.1 Coordenadas dos triângulos e seus baricentros. 88
TAB.9.2 Lista inicial de baricentros. 89
TAB.9.3 Lista de baricentros ordenada pela coordenada X. 89
TAB.9.4 Divisão após a primeira ordenação. 89
TAB.9.5 Ordenação com base na coordenada Y. 89
TAB.9.6 Divisão após a segunda ordenação. 89

LISTA DE ABREVIATURAS E SÍMBOLOS

ABREVIATURAS

API	-	*Aplication Programming Inteface*
IDE	-	*Integrated Development Environment*
IME	-	*Instituto Militar de Engenharia*
OpenGL	-	*Open Graphics Library*
CUDA	-	*Compute Unified Device Architecture*
CTP	-	*Cálculo de Trajetória de Partículas*
HeMoLab	-	*Hemodynamics Modelling Laboratory*
HPC	-	*High Performance Computing*
GPU	-	*Graphic Process Unit*
GPGPU	-	*General-Purpose computation on the GPU*
SDK	-	*Software Development Kit*
SIMT	-	*Single Instruction Multiple Threads*
GDDR	-	*Graphics Double Data Rate*
API	-	*Application Programming Interface*
QEF	-	*Query Evaluation Framework*
CoDIMS	-	*Configured Data Integration Middleware System*
SP	-	*Stream Processor*
SM	-	*Stream Multiprocessor*
OpenCL	-	*Open Computing Language*
AFP	-	*Audio-Fingerprint*
ECC	-	*Error Correction Codes*
FLOPS	-	*Floating point Operations Per Second*
IEEE	-	*Institute of Electrical and Electronics Engineers*
DRAM	-	*Dynamic random-access memory*

RESUMO

A necessidade de processar grandes volumes de dados associada ao crescente poder computacional contido nas atuais placas gráficas têm incentivado a comunidade acadêmica a produzir técnicas paralelas aplicadas às placas gráficas destinadas ao processamento de aplicações científicas. Neste contexto, o objetivo da presente dissertação é apresentar a implementação de uma ferramenta de visualização científica baseada no algoritmo de cálculo de trajetória de partículas (CTP) em unidades de processamento gráfico utilizando-se, em especial, da *Compute Unified Device Architecture* (CUDA). Essa tarefa baseia-se na paralelização do algoritmo CTP já existente no laboratório de modelagem em hemodinâmica (HeMoLab), instalado no Laboratório Nacional de Computação Científica (LNCC). São ainda avaliados os ganhos de desempenho obtidos em relação a execução em uma única CPU e a combinação de algumas das configurações oferecidas pela CUDA.

ABSTRACT

The need to process large volumes of data associated with the increasing computing power contained in the current graphics cards have encouraged the academic community to produce techniques applied to parallel graphics processing designed for scientific applications. In this context, the goal of this dissertation is to present an implementation of a scientific visualization tool based on the Particle Tracing Algorithm (PTA) on graphics processing units using, in particular, the Compute Unified Device Architecture (CUDA) . This work is based upon the parallelization of the PTA existing at the Hemodynamics Modeling Laboratory (HeMoLab), installed in the National Laboratory for Scientific Computing (LNCC). They are also evaluated the performance gains obtained for the execution on a single CPU and the combination of some of the settings offered by CUDA.

1 INTRODUÇÃO

1.1 CONTEXTO

A utilização de técnicas para modelagem e simulação do sistema cardiovascular humano necessita da utilização de técnicas de visualização científica para a visualização e a interpretação dos resultados obtidos. Tais técnicas têm o objetivo de auxiliar os profissionais de medicina no entendimento do complexo comportamento do fluxo sanguíneo no interior das artérias do corpo humano.

O uso de ferramentas de predição nas áreas de biologia e medicina está se tornando cada vez mais importante pois tais ferramentas permitem a representação de fenômenos biológicos com alto grau de detalhes. A modelagem e a simulação computacional aliadas às técnicas de visualização científica vêm sendo utilizadas para estes fins. As ferramentas desenvolvidas têm potencial para diversas aplicações, das quais destacam-se: (i) predição do resultado de intervenções cirúrgicas; (ii) auxílio e treinamento médico; (iii) simulação e visualização do transporte de fármacos e substâncias pelo fluxo sanguíneo e (iv) predição do risco de ruptura de aneurismas, entre outras (LARRABIDE, 2006).

As técnicas de modelagem e de simulação computacional requerem, em regra, grande poder de processamento. A grande eficiência das atuais placas de vídeo no processamento de dados despertou a atenção da comunidade científica como um meio de suprir suas necessidades por computação de alto desempenho. O custo e o consumo de energia das atuais placas são baixos se comparado a sistemas dedicados a problemas de alto desempenho como por exemplo, aglomerados de computadores e supercomputadores. Além de manterem um desempenho semelhante, por vezes muito superior, aos sistemas dedicados (CHE, 2008). O alto desempenho exigido pelos atuais jogos de computador, cada vez mais realistas, é o combustível para a grande evolução das placas de vídeo.

1.2 MOTIVAÇÃO

Uma técnica de visualização científica normalmente não encontrada em ambientes de visualização de código aberto é o Cálculo de Trajetória de Partículas (CTP) e que, recentemente, foi incorporada ao *HeMoLab* (CAMARGO, 2009). Esta técnica manipula dados oriundos de uma simulação numérica ocasionando um processamento geralmente demorado. Logo, existe

a exigência de se alcançar melhores desempenhos na execução deste tipo de aplicação. Uma solução é a utilização de arquiteturas paralelas. Há implementações voltadas para execução em sistemas distribuídos como pode ser visto em (CAMARGO, 2009), em (PORTO, 2004) e em (DEOLIVEIRA, 2010).

Um importante fator hemodinâmico é o tempo que cada uma das partículas do CTP (CAMARGO, 2009), liberadas para a visualização da simulação do fluxo sanguíneo, leva para atravessar uma região sob análise. Esse fator pode caracterizar regiões de estagnação e auxilar na identificação das condições de formação de aneurismas. Outras técnicas de visualização utilizadas para a caracterização de fluxos de fluidos, como por exemplo *streamlines*, não são capazes de fornecer tais informações (CAMARGO, 2009). Desse modo, o estudo e aperfeiçoamento do CTP mostram-se relevantes sob o ponto de vista da hemodinâmica, da modelagem e da simulação do sistema cardiovascular humano.

O CTP pode ser utilizado para a visualização do transporte de substâncias pelo fluxo sanguíneo, na visualização deste fluxo no interior de um aneurisma ou na identificação de suas condições de formação (LARRABIDE, 2006). Uma característica desejável nessas aplicações computacionais é um baixo tempo processamento. Essa característica mostra-se de grande importância nas aplicações utilizadas em medicina pois nesse tipo de aplicação o fator tempo é importante e pode representar o sucesso no diagnóstico e tratamento de algumas patologias. Desse modo o tempo de processamento de uma aplicação médica mostra-se tão importante quanto o resultado e grau de detalhamento da aplicação em questão.

O CTP pode fazer uso das tecnologias oferecidas pela área de Computação de Alto Desempenho (*High Performance Computing* - HPC). A HPC é uma área do mercado mundial suprida por empresas que utilizam tecnologias como o processador *Cell*, a computação de propósito geral em unidades de processamento gráfico (*General-Purpose computation on the GPU - GPGPU*), e o *cluster*, entre outras arquiteturas, com o intuito de oferecer a melhor solução, para os mais diversos domínios de aplicação, tais como: análise financeira, mineração de dados, processamento de imagens, computação científica, modelagem computacional, visualização científica, etc (ASANOVIC, 2006).

1.3 OBJETIVO

O objetivo da presente dissertação é apresentar a implementação de uma ferramenta de visualização científica baseada no algoritmo de cálculo de trajetória de partículas em unidades de processamento gráfico utilizando-se a arquitetura CUDA. Esta tarefa baseia-se na par-

alelização de CTP já existente no HeMoLab (Laboratório de Hemodinâmica).

São avaliados os ganhos de desempenho obtidos bem como analisadas as diferentes configurações da presente implementação paralela.

1.4 ESTRUTURA DO TRABALHO

O presente trabalho encontra-se organizado da seguinte forma:

- O capítulo 2 aborda os principais conceitos sobre a visualização científica e algumas de suas técnicas, e a descrição do sistema *HeMoLab*.
- O capítulo 3 trata da computação de propósito geral em placas gráficas, incluindo a arquitetura das atuais placas gráficas e o modelo de programação utilizado.
- A literatura associada ao problema estudado e a utilização de placas gráficas para computação de propósito geral é apresentada no capítulo 4
- O capítulo 5 discute a solução proposta e detalha sua implementação e validação.
- Os resultados obtidos são discutidos no capítulo 6.
- Fechando o presente trabalho as considerações finais e trabalhos futuros são apresentados no capítulo 7.

São apresentados ainda os seguintes apêndices:

- Pseudo-código do CTP;
- Detalhamento da divisão das metacélulas.

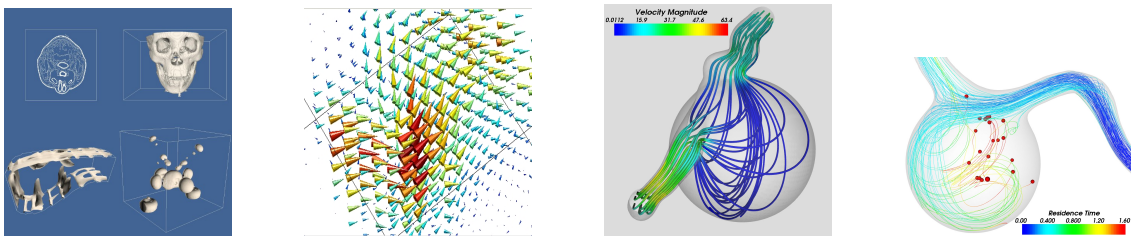
2 VISUALIZAÇÃO CIENTÍFICA

2.1 TÉCNICAS DE VISUALIZAÇÃO CIENTÍFICA

As grandes quantidades de dados para a visualização e interpretação, principalmente em áreas científicas que fazem uso intenso de simulações e de dispositivos de medição, deram origem a técnicas matemático-computacionais para tratamento e apresentação desses dados. A aplicação e desenvolvimento de tais técnicas denomina-se visualização científica (SOUZA, 2003).

Nos últimos anos foram realizadas muitas pesquisas nesta área ocasionando o surgimento de diversas técnicas para visualização de dados como pode ser visto em (DEOLIVEIRA, 2010), (PORTO, 2004), (CAMARGO, 2009) e em (ZIEMER, 2009). O objetivo principal de tais técnicas é possibilitar que o espectador tenha a impressão de estar olhando para os dados reais (SOUZA, 2003). A medida que o *hardware* avança em seu desenvolvimento e agrega maior poder de processamento, novas técnicas são desenvolvidas e as existentes aperfeiçoadas de modo a permitir maior interação e realização de simulações mais complexas. Dentre as diversas técnicas existentes, destacam-se:

- Contorno (*Contour*) - Técnica que utiliza valores escalares anexados à imagem para construir fronteiras nela mesma, chamadas de isolinhas quando exibidas em duas dimensões (2D) e de isosuperfície quando exibidas em três dimensões (3D) (SCHROEDER, 2004).



(a) Contorno (b) Setas (c) Linhas de corrente (d) Trajetória de partículas

FIG. 2.1: Técnicas de visualização. Fonte: (KITWARE, 2009b)

- Setas (*Glyphs*) - Técnica que utiliza valores vetoriais (direção e magnitude). Neste caso, são criadas setas indicando a direção do vetor associado (SCHROEDER, 2004).

- Linhas de corrente (*Streamlines*) - Técnica que utiliza integração numérica para caracterizar um fluxo de fluido usando seu vetor velocidade associado em um espaço tridimensional (SCHROEDER, 2004).

- Trajetória de partículas (*Particle tracing*) - Técnica semelhante a *streamline* mas onde as trajetórias de partículas imersas em um fluído são calculadas em todos os instantes de tempo (SCHROEDER, 2004).

As técnicas de visualização são mostradas na figura 2.1.

2.2 HEMOLAB

O *HeMoLab* - Laboratório de Modelagem em Hemodinâmica - foi criado com o objetivo de reunir em uma só ferramenta as mais recentes técnicas de modelagem e simulação computacional e processamento de imagens médicas.

Os principais módulos do *HeMoLab* são:

- *Módulo de Processamento de Imagens* - Neste módulo é feita a leitura de imagens médicas, tipicamente imagens no padrão DICOM (NEMA, 2009) que é utilizado por equipamentos de aquisição de imagens. Outros formatos (BMP, JPG e PNG) também são suportados. O módulo possui diversos filtros para a remoção de ruído de imagens médicas. O ruído dificulta, por exemplo, a identificação de diferentes estruturas (paredes arteriais, aneurismas, etc.). Após a remoção do ruído utilizam-se filtros específicos para a identificação das diferentes estruturas, este processo chama-se segmentação (LARRABIDE, 2006). Esse módulo, exemplificado na figura 2.2, é composto dos seguintes passos.

  (a) Aplicação de filtros para remoção dos ruídos inseridos pelo equipamento aquisição de imagens médicas.

  (b) Aplicação de filtros para identificação das estruturas de interesse.

  (c) Aplicação de filtros para extração das estruturas de interesse.

- *Módulo 1D* - Neste módulo, as maiores artérias do sistema cardiovascular são representadas por uma árvore, como pode ser visto na figura 2.3, que é parte de uma das telas do *HeMoLab*. Cada segmento representa uma artéria e fornece valores médios de velocidade e pressão, entre outros, para caracterizar o fluxo sanguíneo (LARRABIDE,

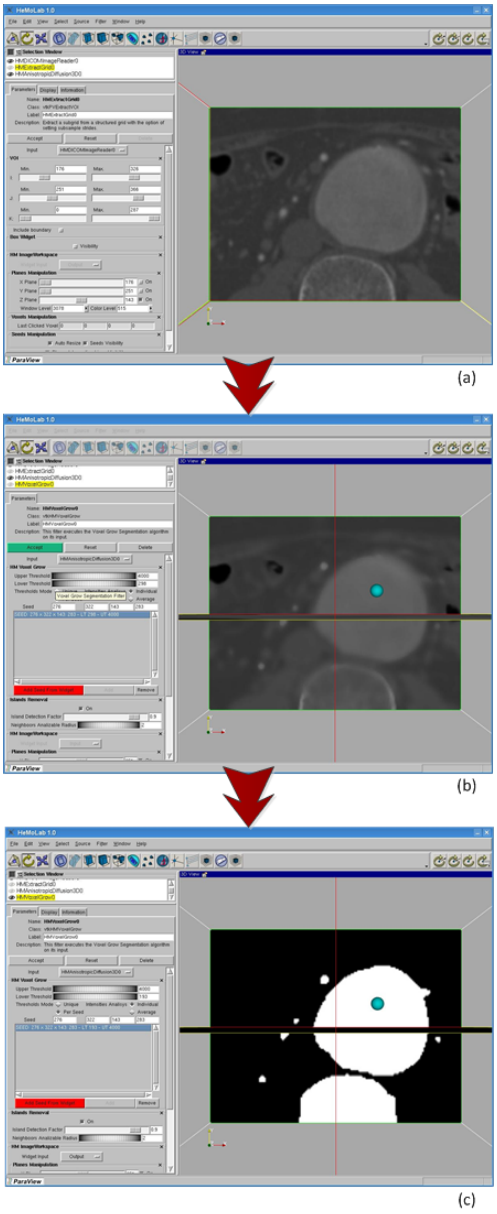


FIG. 2.2: Módulo Processamento de Imagens

2006). Esta árvore é configurável pelo usuário com a utilização de dados de um paciente real. Neste módulo o sistema cardiovascular é tratado de maneira global e sua saída pode ser enviada de ao módulo de Simulação Numérica.

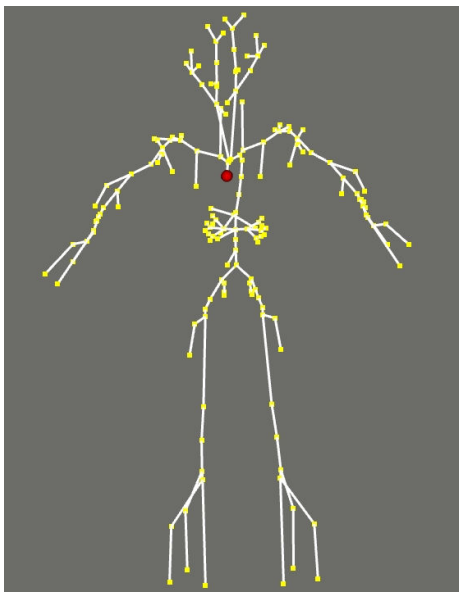


FIG. 2.3: Árvore Arterial Utilizada pelo Módulo 1D

- *Módulo 3D* - As estruturas obtidas com o uso de técnicas de segmentação, utilizadas no módulo de processamento de imagens, são de baixa qualidade e necessitam de tratamento antes de serem configuradas para uma simulação, como pode ser visto na figura 2.4 [1]. O módulo 3D possui filtros para o tratamento e correção da malha de superfície (conjunto de triângulos). Uma malha de boa qualidade para a simulação computacional é aquela onde todos os seus triângulos possuem arestas de comprimentos semelhantes (LARRABIDE, 2006). Uma vez que a condição anterior seja satisfeita é então realizado o preenchimento da malha de superfície com tetraedros, gerando uma malha de volume. A saída deste módulo pode ser enviada ao *módulo de Simulação Numérica*. Este módulo, em particular, faz uso de técnicas e trabalhos já existentes (LENZ, 2009) e/ou originais.

- *Módulo Acoplamento* - Este módulo realiza a união entre a árvore utilizada no módulo 1D e a malha do módulo 3D. Para a realização desta tarefa são identificados os segmentos a ser substituídos e os grupos de triângulos, chamados de tampas, que serão associados aos segmentos correspondentes, como pode ser visto na figura 2.5 onde a esquerda e em amarelo, esta um terminal do modelo 1D sendo acoplado a uma tampa do

[1] Aneurisma em um segmento arterial.

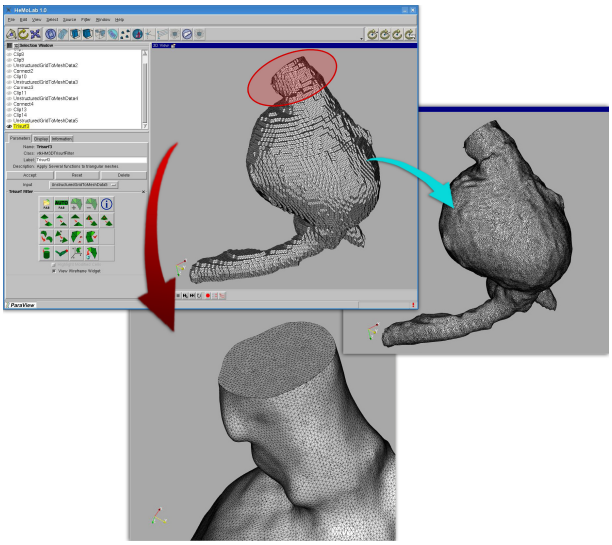


FIG. 2.4: Malha de Volume Utilizada pelo Módulo 3D

modelo 3D, a direita e em vermelho. A saída deste módulo também pode ser enviada diretamente ao módulo de Simulação Numérica.

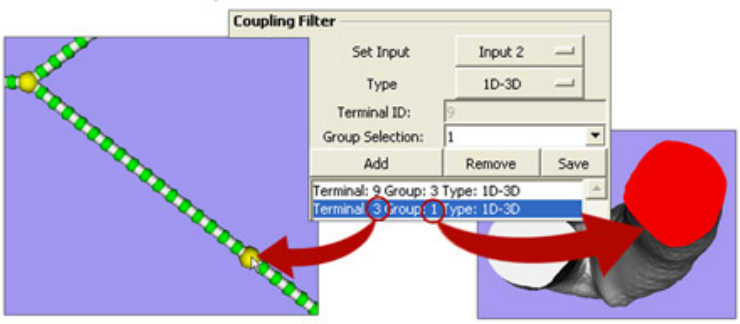


FIG. 2.5: Módulo Acoplamento.

- *Módulo Simulação Numérica* - Este módulo utiliza o Método de Elementos Finitos para resolução do problema (LARRABIDE, 2006) e tem como entrada arquivos formatados, vindos do módulo 1D, do módulo 3D ou do módulo acoplamento. Esses arquivos contém a descrição do problema a ser resolvido (geometria, acoplamentos, parâmetros e condições iniciais, entre outros). Logo, o problema a ser resolvido encontra-se totalmente definido em termos dos arquivos de entrada deste módulo.

- *Módulo Visualização e Análise* - Ao término da simulação numérica a solução do problema modelado é representada pelos valores numéricos dos campos escalares (pressão) e vetoriais (velocidade e deslocamento) em cada ponto do domínio (malha de volume e/ou árvore 1D). Esses resultados representam uma enorme quantidade de dados e não

podem ser analisados sem tratamento adequado (LARRABIDE, 2006).

O funcionamento de todo o sistema *HeMoLab* é divido em etapas lógicas que foram mapeadas nos módulos descritos acima e que devem ser encadeadas de maneira a produzir os resultados esperados, onde a saída de uma etapa é a entrada da seguinte. As mesmas, exemplificadas pela figura 2.6, são descritas a seguir:

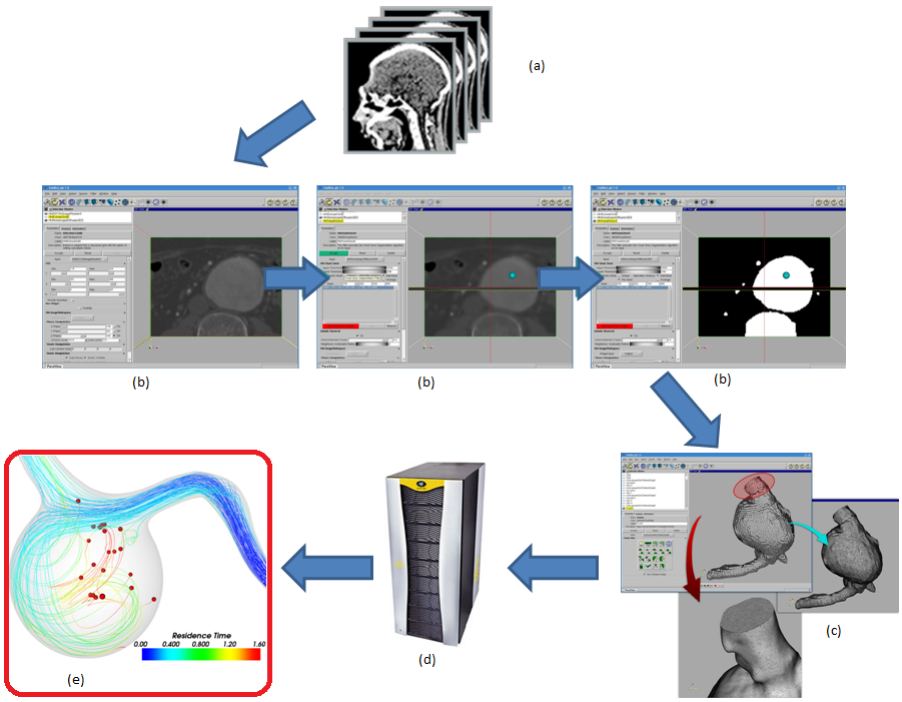


FIG. 2.6: *Pipeline* de Utilização Típico do *HeMoLab* com a fase de visualização em destaque.

(a) Leitura de imagens médicas obtidas através de exames de ressonância magnética (RM) ou tomografia computadorizada (TC);

(b) Utilização de técnicas para tratamento (remoção de ruídos) e segmentação de imagens médicas para extração da geometria de um vaso arterial de interesse;

(c) Tratamento e geração de malhas de superfície e volume.

(d) Simulação numérica utilizando as malhas geradas.

(e) Visualização dos resultados da simulação, foco desta dissertação.

O *HeMoLab* é projetado (contínuo desenvolvimento) como uma extensão do software de visualização científica denominado *ParaView* (KITWARE, 2009a). O *ParaView* foi desenvolvido utilizando a biblioteca gráfica VTK (Visualization ToolKit) (KITWARE, 2009b),

que foi escrita em C++ e utiliza o OpenGL para a renderização. Sua interface com o usuário foi desenvolvida utilizando a linguagem TCL.

O *ParaView* possui um conjunto de características que o fizeram a melhor escolha para o desenvolvimento do *HeMoLab* (LARRABIDE, 2006), dentre as quais pode-se citar:

- Extensibilidade - novas funcionalidades podem ser desenvolvidas e incluídas;
- Código aberto - qualquer grupo ou pessoa pode modificar, distribuir e utilizar o *ParaView* sem a necessidade de pagamentos;
- Multiplataforma - por ter sido desenvolvido utilizando a linguagem ANSI C++, pode ser compilado e utilizado em diferentes plataformas; e
- Arquitetura cliente-servidor - o *ParaView* foi desenvolvido separando sua interface gráfica dos servidores de processamento.

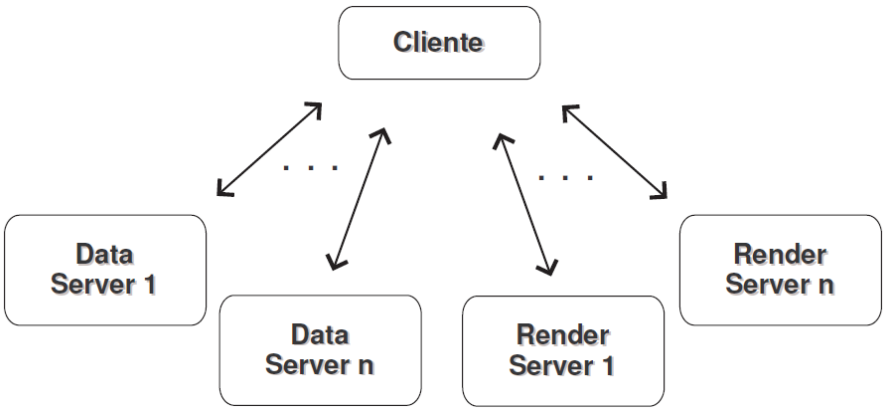


FIG. 2.7: Arquitetura do *ParaView*

O *ParaView* separa clientes, visualização (*render server*) e processamento (*data server)* em processos (ou computadores) diferentes, como pode ser visto na figura 2.7. Essa divisão possibilita que uma simulação seja processada em um supercomputador, sua visualização (renderização) seja feita em um computador de porte intermediário e a apresentação dos dados (interface do usuário e imagem renderizada) seja realizada em um computador sem grandes requisitos de *hardware*.

Maiores detalhes sobre a estrutura, funcionamento ou conceitos envolvidos com o *HeMoLab* e o *ParaView* podem ser obtidos em (LARRABIDE, 2006).

3 PROGRAMAÇÃO DE PROPÓSITO GERAL EM GPU

Este capítulo teve como principais fontes teóricas para sua construção os manuais oficiais da arquitetura CUDA (NVIDIA, 2010) e (NVIDIA, 2011), além do livro (KIRK, 2010). Porém não se limitou à elas.

3.1 HISTÓRICO DE EVOLUÇÃO DAS GPUS

No passado, qualquer aplicação que desejasse utilizar *hardware* gráfico para processamento de propósito geral (não gráfico), deveria ter seus dados passados à GPU na forma de vértices ou pixels (HUANG, 2008) que eram submetidos à GPU e processados por conjuntos de *Pixel Shaders* e *Vertex Shaders*. O grande problema desta arquitetura é que dependendo da aplicação poderia ser necessário mais de um tipo de processamento do que de outro, o que gera ociosidade em parte do *hardware*, como pode ser visto na figura 3.1. No topo da figura 3.1(a) nota-se a grande utilização dos *vertex shaders* para cálculo de vértices e a ociosidade dos *pixel shaders*, o inverso pode ser observado logo abaixo. Esta deficiência gerava um problema para os fabricantes de placas de vídeo que deveriam prever qual seria a combinação ideal nas quantidades de *Pixel Shaders* e *Vertex Shaders* em suas GPUs para maximar o desempenho.

(a) *Shaders* independentes (b) *Shaders* unificados

FIG. 3.1: Uso de *Pixel* e *Vertex Shaders* em contraste com *Stream Processors*.

Para mitigar esta deficiência e suprir as necessidades dos consumidores de jogos de computador e da comunidade científica que sempre buscam desempenho ótimo em suas

aplicações, a NVIDIA unificou *Pixel Shaders* e *Vertex Shaders* em *Stream Processors* programáveis permitindo uma alocação dinâmica e eficiente dos recursos da GPU, como pode ser visto na figura 3.1(b). Esta unificação foi lançada em novembro de 2006, nas GPUs da série 8 que ficaram conhecidas por G80. As GPUs produzidas pela NVIDIA podem ser consideradas o estado-da-arte na implementação de *Stream Processors* programáveis (KAPASI, 2003).

Esta arquitetura deixa o fabricante livre da razão entre *Pixel Shaders* e *Vertex Shaders*, devendo apenas preocupar-se em maximar o número de *Stream Processors* programáveis em suas GPUs. Por serem programáveis, os desenvolvedores podem escolher qual algoritmo utilizar e quantos *Stream Processors* serão dedicados a cada função.

Kernels e *Streams* compõe os princípios básicos do modelo de programação de *Stream Processors*(KAPASI, 2003). Os *kernels* recebem fluxos de dados, chamados de *Streams*, realizam processamento sobre os dados e geram novos dados (*Streams*) de saída que podem, ou não, ser encadeados em um novo *Kernel*. Na figura 3.2 pode-se ver o *stream* de entrada **Input_Image** que passa pelo *kernel* **Convert** e é decomposto nos *streams* **Luminace** e **Chrominance**

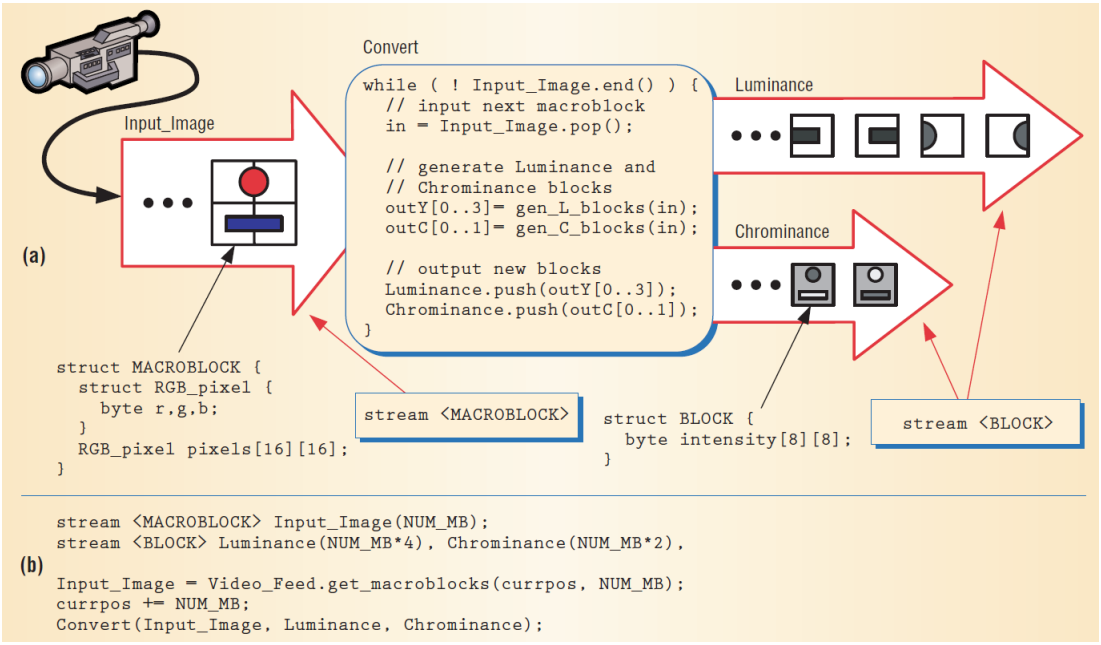


FIG. 3.2: *Streams* e *Kernel*. Fonte: (KAPASI, 2003)

Em novembro de 2006 (NVIDIA, 2010), a NVIDIA apresentou oficialmente a arquitetura CUDA (*Compute Unified Device Architecture*) e expandiu seu público alvo, possibilitando o uso de *hardware* gráfico para processamento de propósito geral (*General-Purpose computa-*

tion on the GPU) através de linguagens de alto nível.

As atuais CPUs possuem limitações quanto as suas frequências de operação pois quanto maior a frequência de operação maior será seu consumo de energia e, consequentemente, maior o calor gerado. Essas limitações produziram avanços na indústria de semicondutores fazendo com que as quantidades de núcleos nas CPUs e nas GPUs dobrassem cada nova geração de semicondutores (KIRK, 2010). Por esse motivo, atualmente existem duas linhas principais para o projeto de microprocessadores: (i) microprocessadores com múltiplos núcleos (*multicore*) que focam em diminuir o tempo de execução das aplicações seriais processando duas, ou mais, tarefas simultaneamente, por exemplo a CPU Intel Core i7 (4 núcleos) e (ii) microprocessadores com muitos núcleos (*many-core*) que focam em diminuir o tempo de execução das aplicações processando uma mesma instrução sobre uma grande massa de dados, por exemplo a GPU GeForce GTX 590 (1024 núcleos).

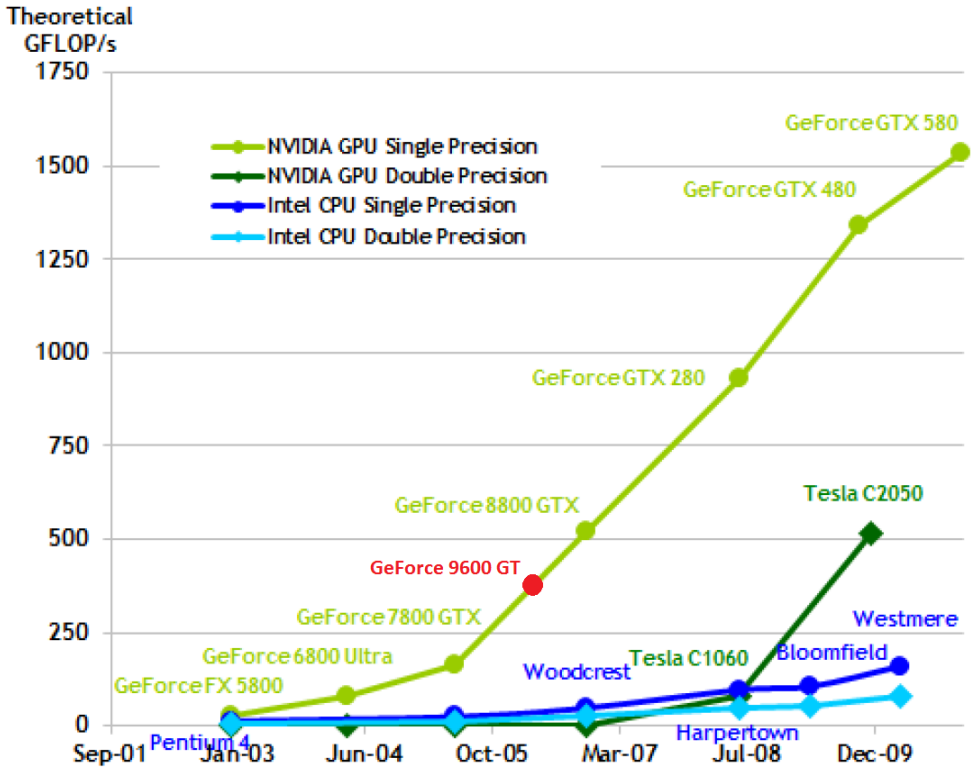


FIG. 3.3: Operações de ponto flutuante por segundo. Fonte: (NVIDIA, 2011)

As quantidades de operações de ponto flutuante por segundo (*Floating point Operations Per Second - FLOPS*) continuam subindo rapidamente nas GPUs quando comparado às CPUs.

As CPUs foram desenvolvidas para tratar de aplicações sequenciais com um controle de fluxo altamente complexo (desvios condicionais) e para armazenar informações em grandes

caches de dados deste modo, escondendo a latência dos acessos à memória. As GPUs, por sua vez, são especializadas no tratamento de aplicações paralelas onde uma mesma instrução é aplicada a um grande volume de dados, não sendo necessário controle de fluxo sofisticado. Sendo assim, a GPU pode dedicar mais recursos (transistores) ao processamento do que ao controle de fluxo e esconder a latência da memória com processamento ao invés de utilizar grandes caches de dados (NVIDIA, 2010). A figura 3.4 demonstra essa distribuição de transistores e caches e unidades de controle.

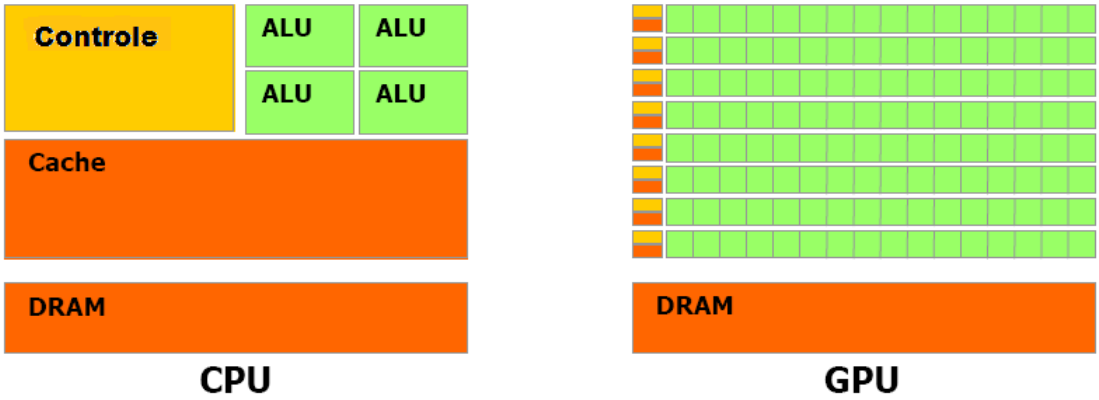


FIG. 3.4: Distribuição de transistores nas GPUs e CPUs. Fonte: (NVIDIA, 2010)

3.2 ARQUITETURA DAS ATUAIS GPUS

A arquitetura desenvolvida e lançada pela NVIDIA a partir de suas placas da série 8 utilizavam o *chip* denominado G80, mostrado na figura 3.5. O G80 e seus sucessores implementam o modelo de *Stream Processors* e levaram a computação de propósito geral em GPU à um novo patamar.

Nas GPUs com suporte para CUDA os processadores de *streams* (SPs) são organizados em multiprocessadores de *streams* (SMs) que são arranjados em uma matriz. A quantidade de SPs por SMs pode variar entre GPUs de gerações diferentes mas o modelo de execução é o mesmo (SIMT - *Single Instruction Multiple Threads*). Os *threads* são executadas em grupos de 32, chamados de *warps*, pelos SPs que compartilham uma memória cache interna ao SM. A cooperação entre *threads* somente pode ocorrer se ambos estiverem alocados em um mesmo SM pois necessitam de sua cache interna neste processo. Atualmente as GPUs possuem até 4 gigabytes de memória GDDR (*Graphics Double Data Rate*) chamada de memória global e de acesso mais lento que a cache interna de cada SM e que pode, ainda, ser acessada simultaneamente por várias *threads*.

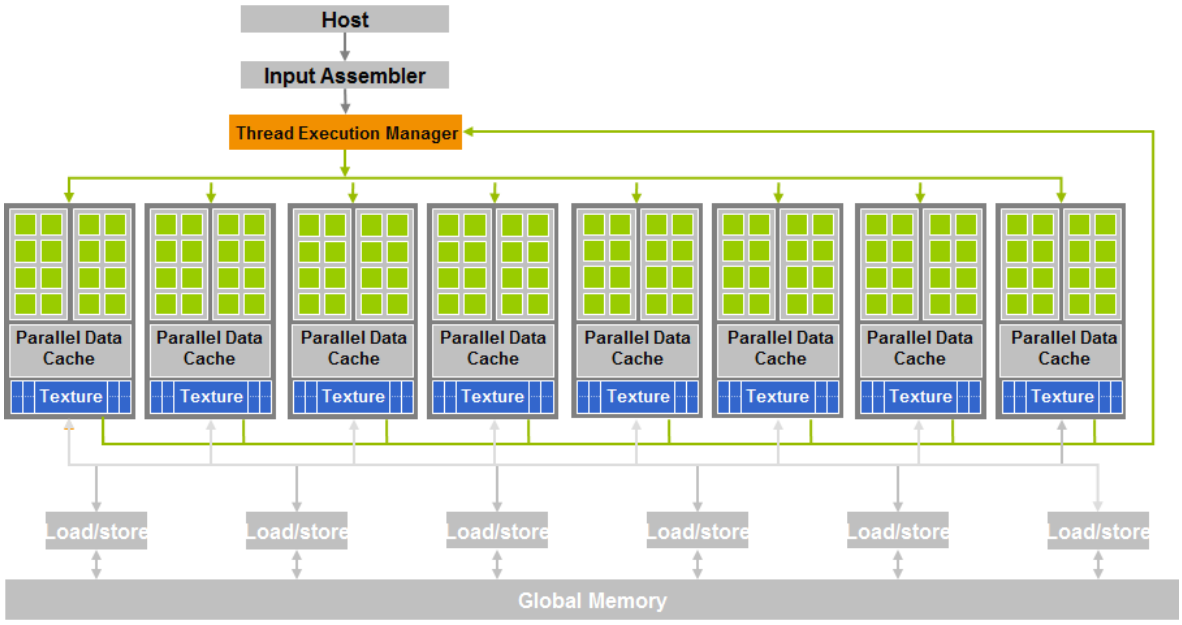


FIG. 3.5: Arquitetura G80. Fonte: (KIRK, 2010)

Atualmente, as GPUs NVIDIA utilizam o *chip* denominado *Fermi* [2]. Este novo projeto de engenharia da NVIDIA é superior a seus antecessores possibilitando um novo salto em GPGPU. A tabela 3.1 compara algumas das novas características da arquitetura *Fermi* (NVIDIA, 2011) com suas antecessoras. Dentre as quais pode-se destacar:

- Adiciona mais um nível de paralelismo à arquitetura através da execução paralela de *kernels*;
- Suporte a ECC (*Error Correction Codes*), método usado para identificar e corrigir erros introduzidos durante o armazenamento ou transmissão de dados. Alguns fabricantes de memória implementam essa técnica em seus *chips*. É utilizado nas *GPUs* para proteção dos dados da memória global, cache e registradores. Essa característica é de grande importância em aplicações de propósito geral.
- Suporte a códigos C++ orientados à objetos;
- Suporte a endereços de memória de 64-bits expandindo a capacidade de memória que pode ser adicionada a placa gráfica e, por consequência, os conjuntos de dados que podem ser armazenados;
- Operações de ponto flutuante em dupla precisão até 8 vezes mais rápidas e em conformidade a especificação IEEE 754-2088;

[2] Homenagem ao físico italiano Enrico Fermi responsável pelo desenvolvimento do primeiro reator nuclear.

- Algumas instruções atômicas 5 a 20 vezes mais rápidas.

TAB. 3.1: Comparação entre os *chips* G80, GT200 e Fermi. Fonte: (NVIDIA, 2011)

GPU	**G80**	**GT200**	**Fermi**
Transistores	681 milhões	1.4 bilhões	3.0 bilhões
CUDA Cores	128	240	512
Operações de ponto flutuante em dupla precisão	Não	30 ops/clock	256 ops/clock
Operações de ponto flutuante em precisão simples	128 ops/clock	240 ops/clock	512 ops/clock
Escalonador de Warp por SM	1	1	2
Memória com suporte ECC	Não	Não	Sim
Kernels concorrentes	Não	Não	Até 16
Comprimento de endereços nas operações de leitura/escrita a memória global	32-bits	32-bits	64-bits
Lançamento	Nov/2006	Jun/2008	Mar/2011

3.3 MODELO DE PROGRAMAÇÃO CUDA

CUDA é uma arquitetura para programação paralela de propósito geral disponível desde as placas gráficas NVIDIA GeForce série 8, Tesla e Quadro. Permite o uso de linguagens de alto nível e de APIs (Application Programming Interfaces) para o desenvolvimento de aplicações, tais como: C, FORTRAN, OpenCL [3] e DirectCompute [4]. A figura 3.6 ilustra as linguagens atualmente suportadas. CUDA está disponível gratuitamente, é multiplataforma e é documentada pela NVIDIA.

Também existem empacotadores (*wrappers*) para CUDA que permitem o desenvolvimento de aplicações em linguagens como Java (jCUDA), C# (CUDA.NET) e Python (PyCUDA) (BURIOL, 2009). CUDA FORTRAN (GROUP, 2011a) é uma extensão comercial para o desenvolvimento de aplicações em CUDA utilizando a linguagem FORTRAN e foi criada pela NVIDIA em parceria com *The Portland Group*(GROUP, 2011b). Com ela o desenvolvedor

[3]Open Computing Language é uma arquitetura para desenvolver programas que funcionem em plataformas heterogêneas, consistindo de CPUs, GPUs e outros processadores

[4]Microsoft DirectCompute é uma interface de programação de aplicativos (API) que tira proveito do poder de processamento paralelo das modernas Unidades de Processamento Gráfico (GPU) para acelerar o desempenho do sistema operacional Microsoft Windows Vista e Windows 7. O DirectCompute faz parte da coleção de APIs DirectX da Microsoft

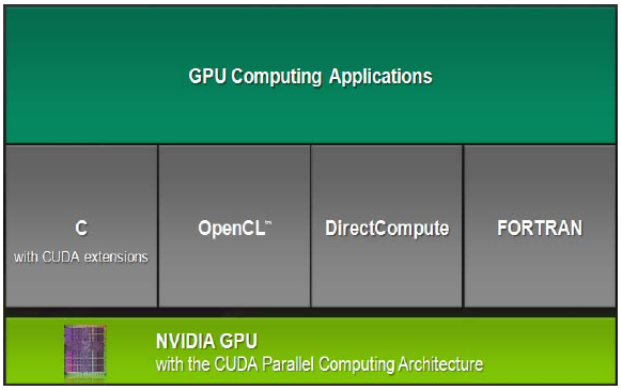


FIG. 3.6: CUDA - Suporte para várias linguagens. Fonte: (NVIDIA, 2010)

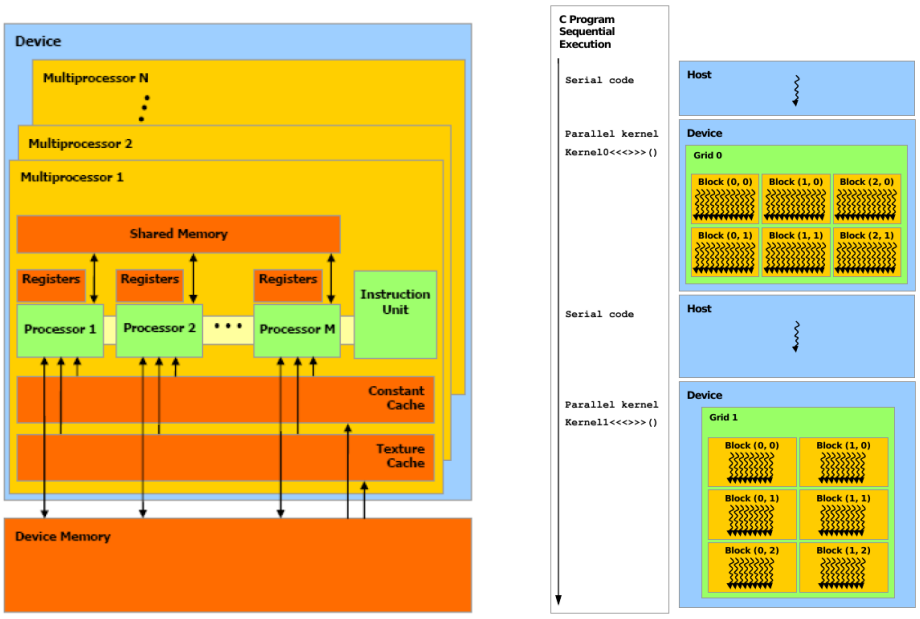
pode escrever suas aplicação em FORTRAN para processamento em GPU sem necessitar conhecimentos avançados sobre CUDA.

Na arquitetura CUDA a GPU é vista como um coprocessador da CPU, onde o desenvolvedor é responsável por escalonar as atividades entre a GPU, chamada de *device*, e a CPU, chamada de *host*. Cabe ao desenvolvedor configurar os acessos aos diversos tipos de memória disponíveis e a quantidade de *threads* (processos leves) que serão utilizados para atingir melhor desempenho.

Para aproveitar o máximo da arquitetura CUDA, o algoritmo a ser processado em GPU não deve possuir muita dependência de dados em cada passo de computação, visto que a GPU paraleliza processamento de dados e não processamento de tarefas. O algoritmo é dividido em subproblemas que serão processados independentemente e em paralelo pela GPU, e estes subproblemas serão divididos em partes menores que serão processados cooperativamente em paralelo. A figura 3.7(b) exemplifica a divisão do algoritmo em partes para execução pelo *host* e pelo *device*.

Na arquitetura CUDA, uma GPU é um *array* de multiprocessadores de *threads*, como pode ser observado na figura 3.7(a). Cada multiprocessador é composto por 8 ou mais processadores de *threads*, registradores de 32 bits, memória compartilhada de alta velocidade, acessível por todos os *threads* de um bloco e duas memórias (constante e textura) somente leitura e mais lentas. Opcionalmente, também é possível acessar a memória principal do *device*, que possui a maior latência.

A extensão da linguagem C para CUDA permite ao programador definir funções, chamadas de *kernel*, que serão executadas N vezes em paralelo pela GPU por N *threads*. Por exemplo, a operação de somar 1 a cada valor de um vetor com 1GB, pode ser transformada em um *kernel*, onde vários *threads* irão executar o incremento nas posições do vetor. Neste caso,



(a) Arquitetura da GPU

(b) Divisão do código serial e paralelo.

FIG. 3.7: Arquitetura da GPU e Divisão do código serial e paralelo. Fonte: (NVIDIA, 2010)

sem sincronia e sem comunicação entre os *threads*.

Os *threads* são distribuídos em blocos de uma, duas ou três dimensões fornecendo, assim, uma maneira natural para o processamento de dados vetoriais e matriciais (BURIOL, 2009). Os *threads* dentro de um bloco podem compartilhar dados usando a memória compartilhada e sincronizar suas execuções para coordenar os acessos a memória. Como o bloco é a unidade de mapeamento dos *threads* para o *hardware*, a quantidade mínima recomendada de *threads* em um bloco será, normalmente, igual a quantidade de processadores de *threads* contidos no multiprocessador utilizado.

Estes blocos de *threads* são organizados em grades de uma ou duas dimensões, como ilustrado pela figura 3.8, onde cada bloco dentro da grade é acessível por um índice que pode ser unidimensional ou bidimensional. A grade é a estrutura completa dos *threads* que executam uma função (*kernel*).

3.4 INTERFACES DE PROGRAMAÇÃO

A arquitetura CUDA disponibiliza ao programador duas interfaces para o desenvolvimento de aplicações CUDA: *C para CUDA* e a *API do driver CUDA*. O programador deve decidir qual delas utilizar pois são mutuamente excludentes (NVIDIA, 2010).

C para CUDA é um pequeno conjunto de extensões da linguagem C que permite ao

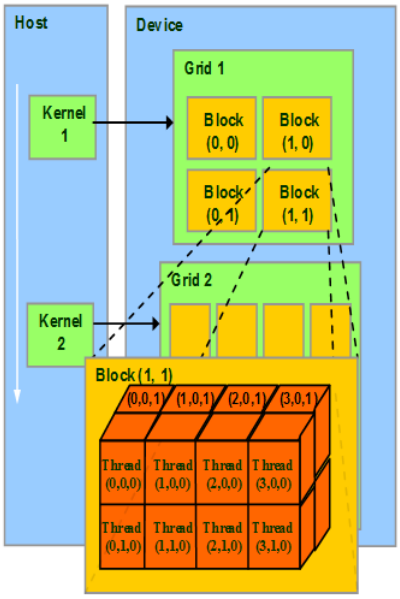


FIG. 3.8: Organização dos *threads* em CUDA. Fonte: (NVIDIA, 2010)

programador acessar o modelo de programação CUDA e definir *kernels* como uma função C, especificando as dimensões das grades e dos blocos cada vez que a função é chamada. Utiliza-se o CUDA *runtime* para a inicialização e gerenciamento de módulos CUDA que tornam-se implícitos[5].

A API do driver CUDA é uma API de baixo nível, requer mais codificação e é mais complexa de programar do que o C para CUDA, mas possibilita ao programador carregar *kernels* como módulos da arquitetura CUDA. Esses módulos (binários ou código assembly) são obtidos pela compilação de *kernels* escritos em C.

A figura 3.9 mostra essa pilha de software e como suas diversas camadas interagem.

3.5 GERENCIAMENTO DE THREADS

As aplicações em CUDA são fundamentadas na execução ordenada e paralela de *threads*. Conforme descrito anteriormente, *threads* são escalonados e executados em grupos, denominados *warps*, compartilhando dados e cooperando entre si na execução do *kernel*. O *kernel* descreve as instruções, em qualquer uma das linguagens suportadas pela arquitetura, que serão executadas pelos *threads*.

Para a permitir a identificação de cada *thread* de maneira única os mesmos são organi-

[5]Até a versão 2.3 do CUDA era suportada a emulação de dispositivos mas este recurso foi retirado das versões subsequentes e justificado pela NVIDIA que as GPUs com suporte a CUDA estão mais acessíveis financeiramente.

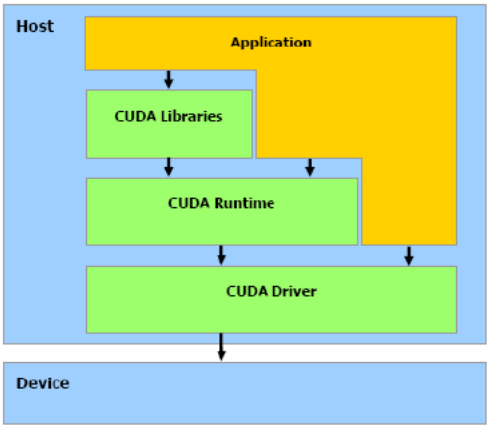


FIG. 3.9: Pilha de *software* CUDA. Fonte: (NVIDIA, 2010)

zados em uma hierarquia de dois níveis representados por duas variáveis do *runtime* CUDA (*blockIdx* e *threadIdx*) e que podem ser acessadas de qualquer parte do *kernel*. A variável *blockIdx* indexa os blocos da grade e a variável *threadIdx* indexa os *threads* de cada bloco ambas do tipo *dim3*. Esse tipo assemelha-se a um *struct* em C e possui 3 componentes inteiros sem sinal: x, y, z.

Normalmente, uma grade será composta por uma matriz bidimensional de blocos mas grades unidimensionais também são permitidas. Sua configuração exata é definida antes da chamada do *kernel* por duas variáveis do tipo *dim3*. O pseudocódigo 3.1 mostra um exemplo de configuração e chamada de um *kernel*. O primeiro parâmetro especifica a quantidade blocos da grade, neste caso em particular o terceiro componente sempre terá o valor 1 pois *grids* não podem ser tridimensionais. O segundo parâmetro especifica a quantidade *threads* em cada bloco. Em seguida ocorre a chamada ao *kernel* e a passagem dos parâmetros obrigatórios entre <<< >>> e dos adicionais, se existirem, e exemplificados pela variável *param*. Neste exemplo, seriam executadas 4096 *threads*, 128 blocos de 32 *threads*.

Pseudo-Código 3.1: Configuração do *kernel*

```
1 dim3 dimGrid(128, 1, 1);
2 dim3 dimBlock(32, 1, 1);
3 KernelFunction<<<dimGrid, dimBlock>>>(param);
```

Os valores máximos para grades e blocos são dependentes da GPU utilizada e podem ser vistos na tabela 3.2. Uma vez que o *kernel* tenha sido chamado todas as dimensões não podem sofrer alterações e serão sempre constantes até o final de seu processamento.

Observando a tabela 3.2 e tendo como referência o processo de multiplicação de matrizes, pode-se notar que, em princípio, matrizes com mais de 512 elementos não poderiam ser

TAB. 3.2: Dimensões de grades e blocos

Especificação técnica	***Compute capability* (versão)**	
	1.x	2.x
Dimensão máxima em x ou y para grades	65535	
Número máximo de *threads* por bloco	512	1024
Dimensão máxima em x ou y para blocos	512	1024
Dimensão máxima em z para blocos	64	

calculadas já que 1 bloco pode ter no máximo 512 *threads*. Essa aparente limitação pode ser contornada dividindo-se a matriz resultado em n blocos de modo que a quantidade de *threads* em cada bloco seja menor ou igual a 512. A figura 3.10 exemplifica esta idéia dividindo uma matriz de 16 elementos em 4 blocos cada um com $2 * 2$ *threads*.

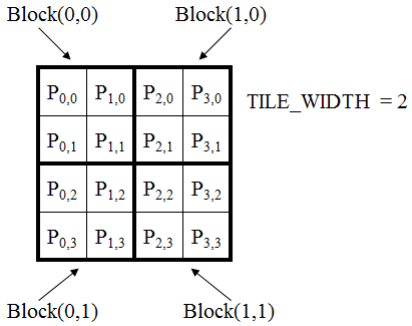


FIG. 3.10: Divisão de uma matriz em mais de 1 bloco. Fonte: (KIRK, 2010)

O pseudocódigo 3.2 de um *kernel* utiliza a divisão da matriz em vários blocos e demonstra os cálculos dos índices de linha e coluna das matrizes para encontrar os elementos que serão operados na realização do produto entre as matrizes **Md** e **Nd**, passadas como parâmetro. Os índices dos blocos são considerados nesta abordagem.

Pseudo-Código 3.2: Multiplicação de matrizes usando múltiplos blocos (KIRK, 2010)

```
1  __global__ void MatrixMulKernel(float* Md, float* Nd, float* Pd, int Width)
2  {
3  // Calcula o índice da linha do elemento de Pd e M
4  int Row = blockIdx.y*TILE_WIDTH + threadIdx.y;
5  // Calcula o índice da coluna do elemento de Pd e N
6  int Col = blockIdx.x*TILE_WIDTH + threadIdx.x;
7
8  float Pvalue = 0;
9  // cada thread calcula um elemento de cada sub-matriz do bloco
10 for (int k = 0; k < Width; ++k)
11   Pvalue += Md[Row*Width+k] * Nd[k*Width+Col];
```

```
12
13 Pd[Row*Width+Col] = Pvalue;
14 }
```

Esta técnica também é apresentada visualmente na figura 3.11 que substituiu as variáveis *blockIdx.x*, *blockIdx.y*, *threadIdx.x* e *threadIdx.y* por *bx*, *by*, *tx* e *ty* respectivamente. Desse modo, mantendo as dimensões dos blocos pequenas têm-se menos de 512 *threads* por bloco.

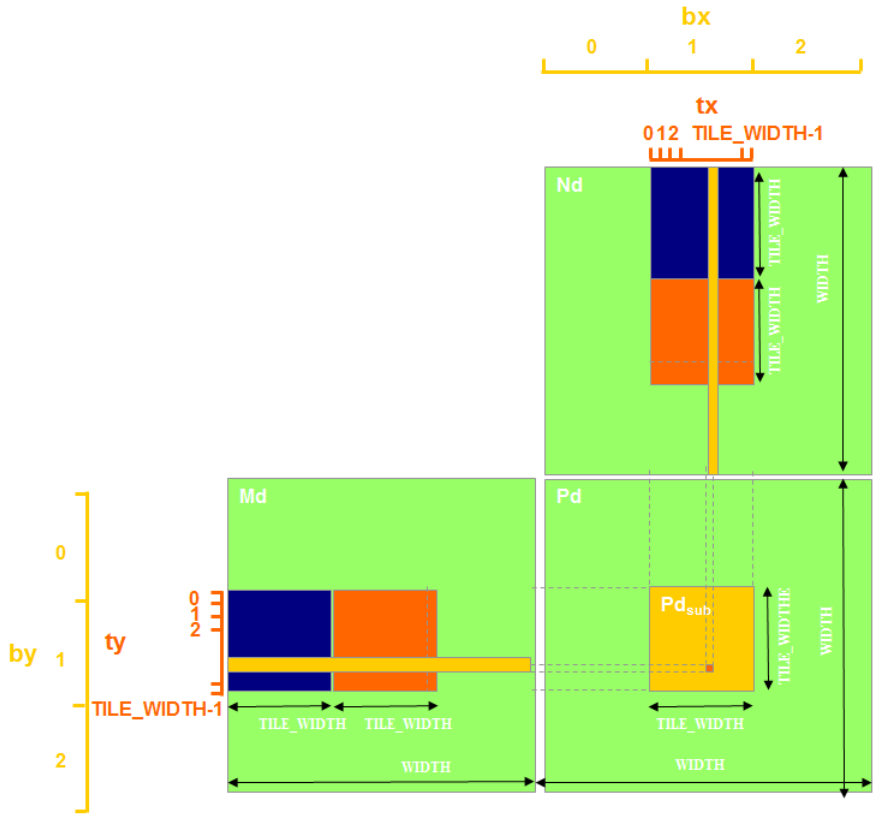


FIG. 3.11: Multiplicação de matrizes utilizando múltiplos blocos. Fonte: (KIRK, 2010)

Nesta implementação da multiplicação de matrizes a sincronização entre os *threads* não se fez necessária mas existem diversos problemas onde tal sincronismo é indispensável, para que os *threads* sinalizem o término de uma tarefa ou a geração de resultados intermediários (GRAMA, 2003).

A sincronização entre *threads* em CUDA é implementada pela função *__syncthreads()*. Um *thread* executa a função de sincronização e aguarda até que todos os *threads* do seu bloco cheguem a este ponto para dar continuidade ao processamento do *kernel*. Isto garante que todos os *threads* tenham atingido o mesmo ponto de execução antes de continuarem o processamento. Somente *threads* de um mesmo bloco podem ser sincronizados.

Caso um *kernel* possua desvios condicionais, todos os *threads* seguirão por todos os possíveis caminhos. Os desvios serão executados um após o outro e algum cálculo será descartado. Isto quer dizer que a instrução de sincronização *__syncthreads()* será executada por todos os *threads* de um bloco ou por nenhum deles, neste caso os *threads* divergem em sua execução. Isto implica que, em alguns casos, algum processamento será perdido já que nem todos os *threads* de um bloco deveriam seguir o mesmo fluxo de processamento. Cabe ao *runtime* CUDA o gerenciamento destes casos.

O fato de não ser permitida a sincronização entre *threads* CUDA de blocos distintos permite a execução de qualquer quantidade de blocos em qualquer ordem. Esta característica, chamada de escalabilidade transparente, permite que a mesma implementação seja executada em GPUs com quantidades diferentes de recursos sem a necessidade de modificações ou re-compilação do código fonte como pode ser visto na figura 3.12.

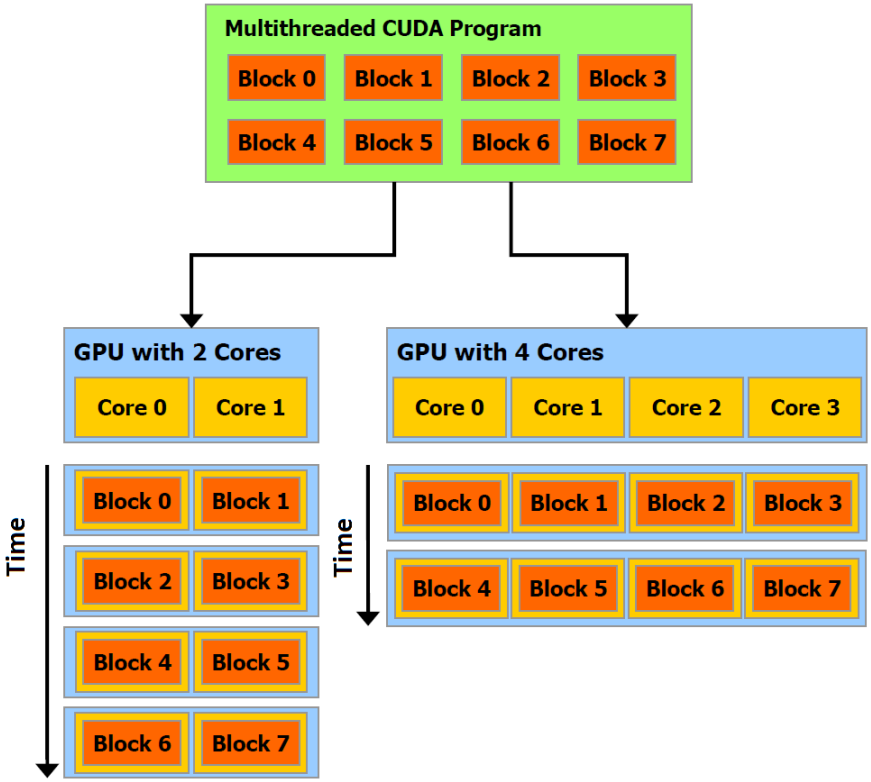


FIG. 3.12: Escalabilidade transparente para programas CUDA. Fonte: (NVIDIA, 2011)

São necessários recursos de *hardware* para manter os identificadores de *threads* e blocos além de acompanhar suas condições de execução. Nos casos que exijam mais recursos do que os disponíveis, o *runtime* CUDA automaticamente diminui a quantidade de blocos simultâneos até que o processamento seja possível.

A escalabilidade transparente confere flexibilidade à arquitetura CUDA possibilitando

a execução do mesmo programa em hardware com diferentes quantidades de recursos. O escalonamento de *threads* é realizado utilizando conjuntos de 32 *threads* denominados *warps*. Quando um bloco é atribuído a um multiprocessador ele é dividido em *warps*. Cada *warp* é o agrupamento de *threads* com IDs consecutivos, os *threads* de 0 a 31 compõe o primeiro *warp*; os *threads* de 32 a 63 compõe o segundo *warp* e assim por diante. Esse agrupamento dos *threads* em *warps* tem por objetivo ocultar a latência do processamento.

Quando um *thread* deve aguardar pelo resultado de uma operação de alta latência, por exemplo um acesso a memória global, o *warp* a qual pertence é retirado de execução e outro que esteja pronto à executar é escalonado para execução. Caso exista mais de um *warp* pronto para executar, um mecanismo de prioridades é utilizado. O escalonamento de *warps* faz com que os recursos nunca fiquem ociosos e tem a vantagem de não introduzir nenhum custo extra com trocas de contexto.

A arquitetura CUDA implementa em *hardware* um *thread manager* para o gerenciamento de *threads*. Esse recurso libera do programador da obrigatoriedade de declarar os *threads* explicitamente, ocultando toda a complexidade inerente à criação e divisão dos blocos que é feita em *hardware*.

3.6 HIERARQUIA DE MEMÓRIA

A arquitetura CUDA provê diversos tipos de memória que devem ser utilizados em conjunto com o objetivo de obter o melhor desempenho da aplicação em desenvolvimento. Estes tipos de memória podem ser vistos na tabela 3.3 e na figura 3.13 e são agrupados em uma hierarquia de modo a facilitar e otimizar seu uso.

TAB. 3.3: Qualificadores de variáveis em CUDA

Declaração da variável	**Memória**	**Escopo**	**Tempo de vida**
Variável automática escalar	Registrador	*Thread*	*Kernel*
Variável automática matricial	Local	*Thread*	*Kernel*
__device__ __shared__ int var	Compartilhada	Bloco	*Kernel*
__device__ int var	Global	Grade	Aplicação
__device__ __constant__ int var	Constante	Grade	Aplicação

As variáveis podem estar armazenadas em memória global, constante, compartilhada, local ou registradores. Dependendo de onde a variável está localizada, o seu tempo de vida e visibilidade por outros *threads*, blocos ou grades são afetados. A visibilidade, ou escopo, de uma variável diz respeito ao intervalo de *threads* que poderão acessá-la. Caso o escopo

de uma variável seja um único *thread*, por exemplo uma variável declarada dentro da função *kernel*, será criada uma cópia privada para cada *thread* que utilizar este *kernel* e seu tempo de vida será o do *thread*.

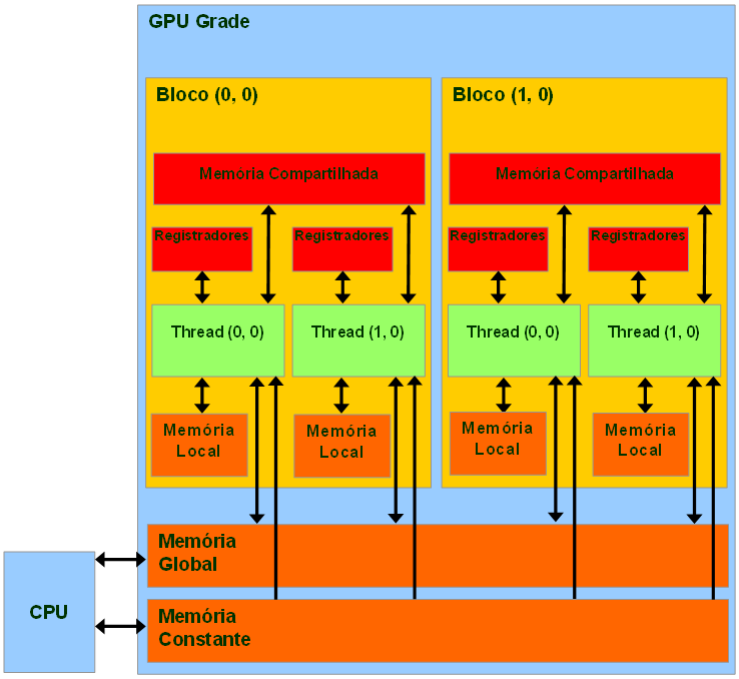


FIG. 3.13: Hierarquia de memória.

Registradores e memória local são utilizados por cada *thread* para armazenar suas próprias variáveis, chamadas de variáveis automáticas. Eles não são visíveis à outros *threads* e só existem enquanto o *thread* que a criou existir. Dados escalares são armazenados nos registradores (mais rápidos), dados matriciais são armazenados na memória local (mais lentos). A memória local é mais um conceito de implementação do que um *hardware* real, visto que os dados são armazenados fisicamente na memória global e o *runtime* CUDA encarrega-se de deixá-los acessíveis somente a seus respectivos *threads*.

A memória compartilhada oferece um meio dos *threads* que compõe um bloco cooperarem e trocarem dados, oferecendo alta velocidade. Os dados contidos neste tipo de memória são visíveis por todos os *threads* do bloco que os criou e irão existir enquanto o bloco existir. Cada *thread* possui seus próprios registradores que não são acessíveis à nenhum outro *thread*. Os registradores e a memória compartilhada são o tipo mais rápido e em menor quantidade na hierarquia de memória CUDA, suportando acessos paralelos a altas velocidades.

As variáveis que devem ser acessíveis por todos os *threads* de todos os *kernels* residem em memória global ou em memória constante. A velocidade da memória global é baixa se comparada ao outros tipos de memória disponibilizadas pela arquitetura. Variáveis resi-

dentes neste tipo de memória normalmente são utilizadas para trafegar dados entre *kernels* durante a execução da aplicação. As variáveis alocadas em memória constante na verdade estão armazenas em memória global. A diferença é que são mantidas em um cache para que o acesso seja mais eficiente.

3.7 CONSIDERAÇÕES SOBRE DESEMPENHO

Existem vários aspectos relativos a desempenho na arquitetura CUDA. Dependendo da aplicação um deles pode exercer maior influência do que outro no desempenho, desse modo, exigindo uma mudança na configuração dos blocos e *threads* ou mesmo a modificação do algoritmo em questão. Dentre os aspectos da arquitetura CUDA relativos a desempenho, destacam-se:

- Divergência na execução dos *threads*;
- Largura de banda de acesso a memória global;
- Pré-busca de dados;
- Particionamento dinâmico dos recursos dos multiprocessadores e
- Desdobramento de laços .

Na seção 3.5 foi dito que os blocos são divididos em grupos de 32 *threads* denominados *warps* e que instruções condicionais potencialmente podem levar a divergência na execução de *threads* de um mesmo *warp*. Uma instrução é executada por todos os *threads* de uma *warp* ao mesmo tempo, antes de passar para a próxima instrução (KIRK, 2010). Este modelo de execução é chamado de *Single Instruction Multiple Threads* (SIMT) e permite que o custo de busca e processamento de uma instrução tenha um menor impacto no desempenho, pois a instrução é executada por um grande número de *threads*. O modelo SIMT só funciona bem quando todas os *threads* seguem o mesmo fluxo de execução (KIRK, 2010), ou seja, quando aparecem as divergências na execução dos warps ocorre degradação no desempenho.

Pseudo-Código 3.3: *Kernel* de redução de soma. Fonte: (KIRK, 2010)

```
1 __shared__ float partialSum[];
2
3 unsigned int t = threadIdx.x;
4 for (unsigned int passo = 1; passo < blockDim.x;  passo *= 2)
5 {
```

```
6     __syncthreads();
7     if (t % (2*passo) == 0)
8             partialSum[t] += partialSum[t+passo];
9  }
```

O pseudo-código 3.3 é ilustrado para demonstrar o problema de divergência. Nele é implementada uma operação de redução de soma, onde cada *thread* utiliza seu índice para acessar um elemento em vetor e realizar a soma deste elemento com outro que terá seu índice calculado. Na primeira iteração os índice múltiplos de 2 serão somados com seus sucessores. Na segunda iteração a variável *passo* é dobrada e apenas os *threads* múltiplos de 4 serão executados e assim continuam as iteração até que ao final tem-se o resultado da redução de soma. Este processo é ilustrado pela figura 3.14.

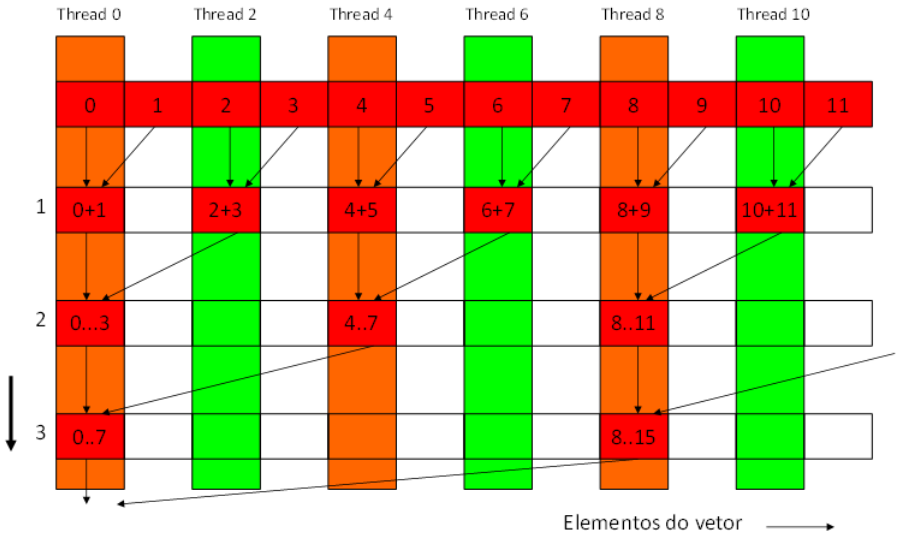


FIG. 3.14: Execução do *Kernel* de redução de soma. Fonte: (KIRK, 2010)

Nesse exemplo cada *thread* de um *warp* segue um caminho diferente no laço *if* do código de exemplo. Apenas os *threads* múltiplos de 2 entram no laço logo, os não múltiplos executam outro caminho o que leva a divergência entre os *threads* de uma mesmo *warp*. Na próxima iteração apenas os *threads* múltiplos de 4 entram no laço e assim continua até o final com cada vez menos *threads* entrando no laço.

Pseudo-Código 3.4: *Kernel* de redução de soma sem divergência. Fonte: (KIRK, 2010)

```
1  __shared__ float partialSum[];
2
3  unsigned int t = threadIdx.x;
4  for (unsigned int passo = 1; passo < blockDim.x;  passo *= 2)
```

```
5 {
6   __syncthreads();
7   if (t % (2*passo) == 0)
8           partialSum[t] += partialSum[t+passo];
9 }
```

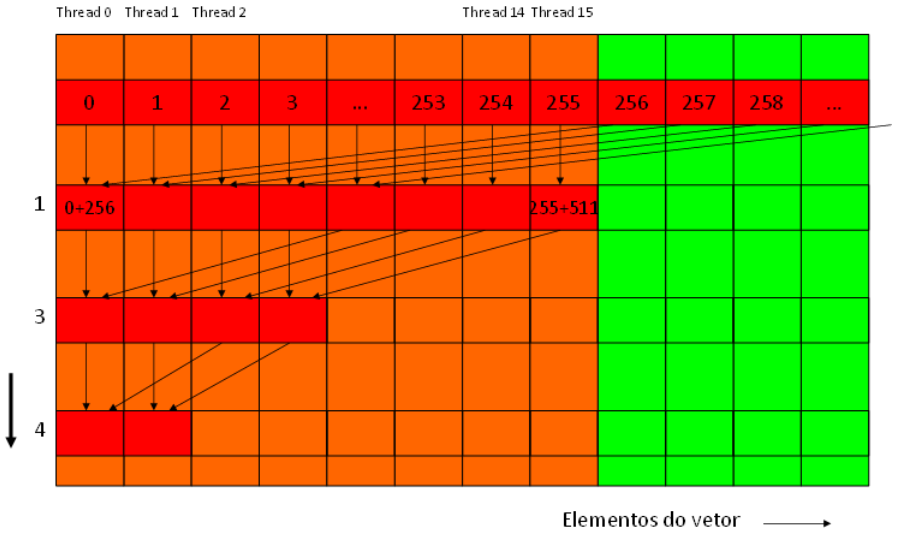


FIG. 3.15: Execução do novo Kernel de redução de soma. Fonte: (KIRK, 2010)

O *kernel* do pseudo-código 3.4 foi modificado para somar dois elementos que estão na metade do bloco em relação um ao outro, ao invés de somar dois vizinhos. Por exemplo, para um vetor de 512 elementos serão criados 512 *threads* onde na primeira iteração o *thread* 0 somará os elementos das posições 0 e 256, armazenando o resultado na posição 0 do vetor como mostra a figura 3.15. Na segunda iteração a variável (*passo*) que controla a iteração do laço *for* terá 1 bit deslocado à direta o que tem o mesmo efeito de uma divisão por 2 e segundo (KIRK, 2010) é um método menos dispendioso computacionalmente de implementar uma divisão por 2 do que uma divisão de inteiro real. Desse modo, todos os *threads* que estiverem na primeira metade da divisão (*threads* 0 a 255) seguirão um caminho e os da segunda metade (*threads* 256 a 511) seguirão outro caminho. Nota-se que a divergência no processamento dos *threads* foi reduzida mas não eliminada pois nas últimas iterações existirão menos de 32 *threads* gerando divergência. A nova versão do *kernel* deverá ter um desempenho melhor do que a primeira pois o efeito negativo no desempenho é mais intenso quando ocorre divergência em *threads* de um mesmo *warp*.

As aplicações em CUDA processam uma grande quantidade de dados utilizando *threads* em paralelo que consomem dados vindo da memória global. Isso obriga a memória global a

possuir largura de banda suficiente para suprir essa demanda, o que não ocorre normalmente, ou a criação de meios para diminuir ou compensar possíveis esperas nos acessos à memória. Um desses meios é o uso da memória compartilhada contida em cada multiprocessador e que pode ter seu conteúdo utilizado de forma simultânea por vários *threads*. A memória compartilhada é mais rápida do que a memória global mas possui tamanho reduzido, algumas GPUs possuem 16KB e as mais recentes GPUs possuem 48KB. Isso implica que os dados a serem copiados para a memória compartilhada devem ser escolhidos com cuidado pois eles devem ser utilizados pelo máximo de *threads* possível, desse modo, justificando o uso deste tipo de memória.

Outro meio de melhorar o desempenho da memória global é aproveitar o fato de que são implementadas utilizando memórias de acesso aleatório dinâmicas (*Dynamic random-access memory* - DRAM) e que estas já utilizam acessos paralelos para aumentar sua taxa de acesso aos dados. Sempre que um dado é requisitado para uma DRAM o endereço do dado solicitado e seus vizinhos são acessados, se uma aplicação puder fazer uso deste padrão de localidade seu desempenho tende a aumentar. Se em um *warp* os *threads* acessam dados da memória global em posições consecutivas, como por exemplo posições sequenciais em uma matriz unidimensional, o *hardware* aglutina todas as solicitações de acesso em uma única solicitação e realiza o acesso de forma consolidada permitindo entrega de dados em taxas próximas do máximo suportado pela memória global.

Outra forma de esconder a latência de acesso a memória global é aumentar o número de instruções de computação entre os acessos a memória fazendo com que menos *warps* fiquem parados aguardando seus dados da memória global. Esta técnica chama-se pré-busca de dados.

Os multiprocessadores possuem recursos limitados para distribuir aos *threads* para suas execuções. Esses recursos são particionados dinâmicamente com o objetivo de serem aproveitados ao máximo. Por exemplo, a GPU GeForce 9600GT pode suportar até 768 *threads* em cada um de seus 8 multiprocessadores pois cada um suporta até 24 *warps*, como cada *warp* possui 32 *threads* logo, tem-se 768 *threads*. Ao se criar uma grade de 8 blocos de 512 *threads* e submeter esta grade para processamento recursos serão desperdiçados pois cada multiprocessador só poderá receber um único bloco pois 2 blocos ultrapassam a capacidade máxima de 768 *threads*. Embora a quantidade de *threads* por bloco esteja em seu limite cada multiprocessador ainda não atingiu o seu limite de 768 threads. Uma maneira de solucionar este problema é criar uma grade de 16 blocos de 256 *threads*, desse modo cada multiprocessador receberá 3 blocos ($256 * 3 = 768$ *threads*) otimizando o uso de recursos. Essa flexibilidade

permite variações na quantidade de *thread* e blocos para maximizar a utilização dos recursos da GPU.

Os registradores de cada multiprocessador também são particionados dinamicamente. São permitidos mais blocos por multiprocessador quando eles utilizarem poucos registradores mas quando a necessidade por registradores aumenta, diminui o número de blocos por multiprocessador. Tomando-se como exemplo o caso anterior, se cada *thread* necessitar de 2 registradores[6] tem-se $2 * 16 * 256 = 8.192$ que é o número exato de registradores disponíveis em cada multiprocessador da GeForce 9600GT. Caso seja realizada alguma modificação no *kernel* fazendo com que suba para 4 a quantidade de registradores necessários por *thread*, isto invalida o cálculo anterior pois $4 * 16 * 256 = 16.384$ é maior do que os 8.192 registradores disponíveis. A solução é diminuir o número de blocos para 8 ($4 * 8 * 256 = 8.192$) e assim não ultrapassar o limite de 8.192 registradores por multiprocessador. Nota-se que após uma mudança no *kernel* a quantidade de *threads* ativos por multiprocessador caiu para 512 representando uma redução de $\frac{1}{3}$ no paralelismo.

Cada instrução processada consome recursos da GPU durante sua execução logo, a quantidade de instruções utilizadas também tem impacto no desempenho de uma aplicação em GPU. Por exemplo, em um laço *for* existem instruções referentes ao controle das iterações e instruções referentes ao processamento do problema em questão. As instruções referente ao controle consomem recursos extras em seu processamento pois o laço precisa ter seu contador de iteração atualizado e realizar um desvio condicional após cada iteração. Além disso, ao se utilizar o contador do laço para indexar qualquer variável no interior do laço serão consumidos recursos para realização de operações de aritmética de endereços. Uma opção para melhorar o desempenho nessas situações é desdobrar da instrução do laço. Essa operação pode ser feita manualmente como mostra o pseudo-código 3.7 ou com auxílio da diretiva **#pragma unroll** como mostra o pseudo-código 3.5. O pseudo-código 3.5 mostra o laço original sem otimização.

Pseudo-Código 3.5: Laço sem desdobramento.

```
1 for (int i = 0; i < n; i++)
2 {
3 Pvalue += vet[i] + vet[i+1];
4 }
```

[6] Ao compilar um *kernel* CUDA utilizando as opções *–ptxas-options=-v* obtêm-se as quantidades utilizadas de registradores, memória local, compartilhada e constante.

Pseudo-Código 3.6: Laço com desdobramento utilizando a direta **#pragma unroll**.

```
#pragma unroll
for (int i = 0; i < n; i++)
{
Pvalue += vet[i] + vet[i+1];
}
```

Pseudo-Código 3.7: Laço com desdobramento manual.

```
// Sendo n=10
Pvalue = Pvalue + vet[0] + vet[1];
Pvalue = Pvalue + vet[1] + vet[2];
Pvalue = Pvalue + vet[2] + vet[3];
Pvalue = Pvalue + vet[3] + vet[4];
Pvalue = Pvalue + vet[4] + vet[5];
Pvalue = Pvalue + vet[5] + vet[6];
Pvalue = Pvalue + vet[6] + vet[7];
Pvalue = Pvalue + vet[7] + vet[8];
Pvalue = Pvalue + vet[8] + vet[9];
Pvalue = Pvalue + vet[9] + vet[10];
```

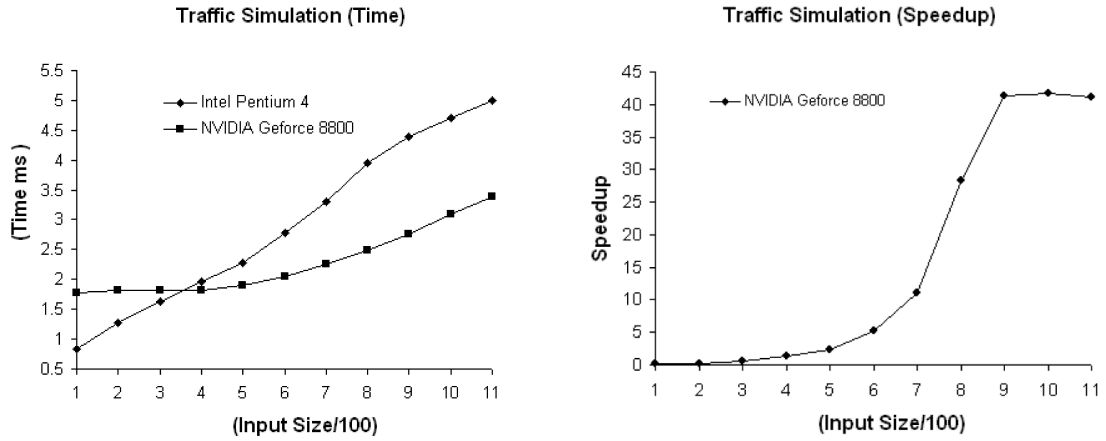
Os relacionamentos entre os recursos e as técnicas de desempenho devem ser estudados e experimentados no intuito de obter os melhores desempenhos. Acredita-se que em uma aplicação apenas 1 ou 2 aspectos são realmente importantes e isso depende da aplicação em questão logo, experimentação e entendimento são necessários para identificar tais aspectos e otimizar sua utilização.

4 REVISÃO DA LITERATURA

Este capítulo destina-se a apresentar alguns trabalhos existentes na literatura que relacionam-se com o algoritmo de trajetória de partículas e com o desenvolvimento e avaliação do desempenho de aplicações que utilizam a arquitetura CUDA. O capítulo está subdividido nos tópicos de arquitetura CUDA e algoritmo de trajetória de partículas.

4.1 ARQUITETURA CUDA

Che *et al* em (CHE, 2008) comparam o desempenho de três aplicações naturalmente paralelas e que demandam grande poder computacional, utilizando implementações em CPU e GPU, são elas: Simulação de tráfego, simulação térmica e o algoritmo k-médias[7]. Os autores afirmam que muitas outras aplicações paralelizáveis podem utilizar o poder computacional contido nas atuas GPUs para atingir ganhos de desempenho de até 40 vezes quando comparados com implementações em CPU. As versões em GPU dos algoritmos escolhidos foram implementadas sem levar em consideração aspectos de otimização que serão implementadas em trabalhos futuros.



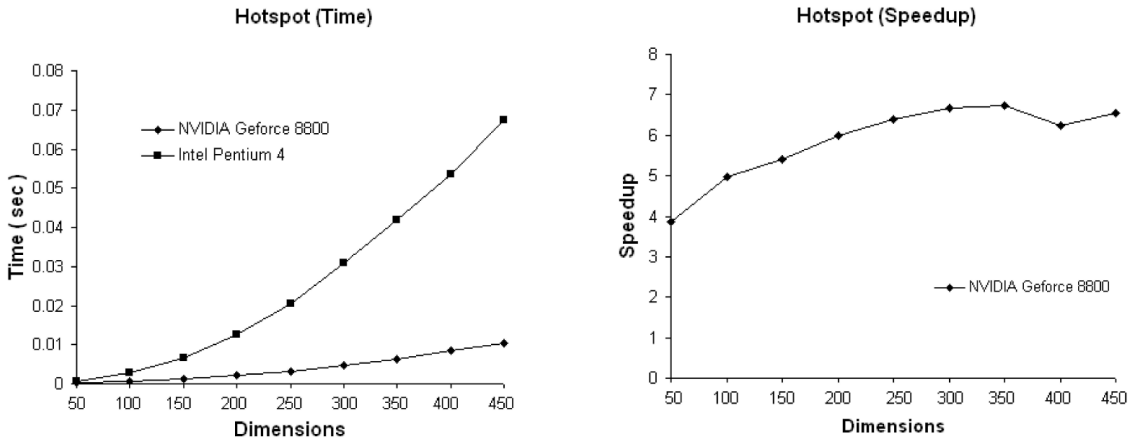
(a) GPU *versus* CPU (b) *SpeedUp* de 40x.

FIG. 4.1: Simulação de tráfego. Fonte: (CHE, 2008)

Pode-se observar no gráfico da figura 4.1(a) que inicialmente a GPU tem seu tempo de processamento superior a implementação em CPU mas quando a entrada de dados é in-

[7]O k-médias é um algoritmo utilizado para reconhecimento e classificação de padrões de forma automática e é utilizado em aplicações de mineração de dados e segmentação de imagens.

crementada pode-se observar que a GPU aumenta seu tempo de processamento em uma proporção menor do que a CPU. Esse comportamento é esperado pois as GPUs foram desenvolvidas para tratar grandes volumes de dados. Nesse experimento, em particular, foi obtido um *speedup*[8] de 40 vezes como demonstrado no gráfico 4.1(b).



(a) GPU *versus* CPU

(b) *SpeedUp* de 6.5x.

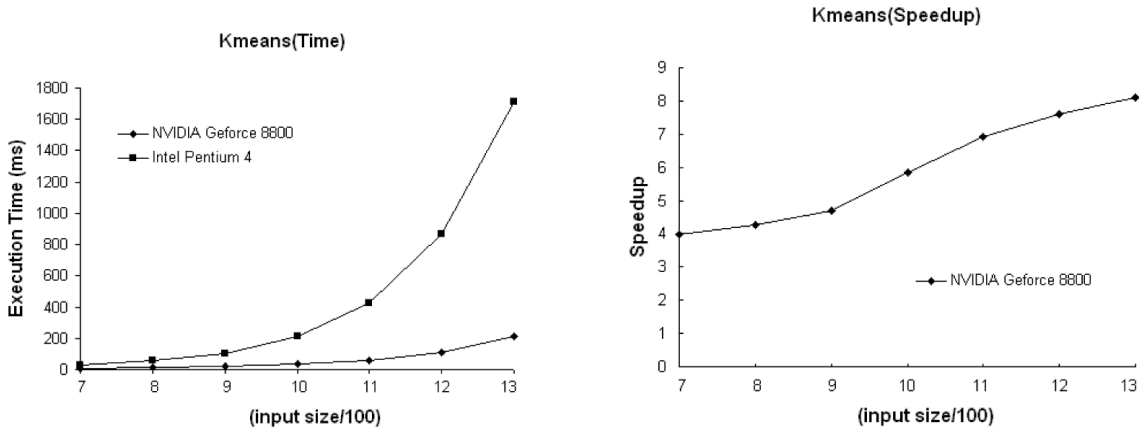
FIG. 4.2: Simulação térmica. Fonte: (CHE, 2008)

Nota-se através dos gráficos das figuras 4.2(a) e 4.2(b) que o ganho de desempenho da aplicação e foi inferior a aplicação de simulação de tráfego. Os autores acreditam que este comportamento é o reflexo da estrutura de dados utilizada para diminuir a comunicação entre os *threads* adicionando uma carga de trabalho extra para a *GPU*.

Segundo (CHE, 2008), o experimento da simulação térmica também teve seu desempenho prejudicado pelo modo como o algoritmo foi implementado. A aplicação processa o conjunto de dados em CPU, remete-os para GPU processar uma outra etapa e termina de processá-los em CPU. Esse procedimento repete-se de forma iterativa onde a cada iteração é inserido uma carga de trabalho extra copiando os dados da CPU para a GPU e vice-versa.

Miranda e Piccoli em (MIRANDA, 2010) apresentam o processo de identificação de áudio e propõe o uso de GPUs para aumentar o desempenho das aplicações que realizam tal processo. A identificação de um sinal de áudio necessita de uma impressão digital de áudio

[8]Quando avaliando sistemas paralelos, o interesse recai em saber qual o ganho de desempenho atingido pela paralelização de uma dada aplicação sobre uma implementação sequencial. Speedup é a medida que captura o relativo benefício de resolver uma problema em paralelo. É definido como a razão entre o tempo requerido para resolver o problema em um único elemento de processamento e tempo requerido para resolver o mesmo problema em um computador paralelo com n elementos de processamento idênticos. Tradução livre de (GRAMA, 2003).



(a) GPU *versus* CPU

(b) *SpeedUp* de 6.5x.

FIG. 4.3: Algoritmo k-médias. Fonte: (CHE, 2008)

(*Audio Fingerprint - AFP*)[9] para a busca e comparação frente a uma coleção de AFPs de sinais, por exemplo aplicações para celulares que identificam a música tocada em sua proximidade. Outra situação possível é a comparação de uma coleção de sinais que não possuem AFPs frente a um sinal conhecido e com sua AFP determinada, por exemplo aplicações que monitoram várias estações de rádio em busca de um sinal em particular que pode ser um comercial ou música. Nesse caso, atualmente é impossível obter todas as AFPs para todos os sinais de entrada utilizando somente uma CPU (MIRANDA, 2010). Os autores implementaram o processo de identificação de áudio para o segundo caso em GPU pois, segundo eles, as atuais GPUs são, em muitos sentidos, um super computador portátil. As implementações alcançaram um *speedup* de 2.5 quando processando um único frame de um sinal de áudio. Foi considerado que um sinal de áudio típico, por exemplo uma música, pode conter 1462 frames. Esta música seria processada em 4165.63 milisegundos utilizando uma CPU, e a correspondente implementação em GPU processaria a mesma música em 0.055 milisegundos, atingindo um *speedup* de 4 a 5 ordens de magnitude (MIRANDA, 2010).

Estes trabalhos compararam implementações em CPU e GPU de aplicações naturalmente paralelas onde foram obtidos bons resultados quanto ao desempenho das versões GPU, justificando maiores pesquisas na área de computação em GPU e o desenvolvimento de aplicações que façam uso das tecnologias obtidas.

[9]Tradução livre de Audio-Fingerprint - AFP. A qualidade de um AFP para um dado sinal é diretamente proporcional ao custo computacional para produzi-lo devido a quantidade de cálculos que devem ser realizados.

4.2 ALGORITMO DE TRAJETÓRIA DE PARTÍCULAS

Porto *et al* em (PORTO, 2004) apresenta o ambiente CoDIMS (*Configured Data Integration Middleware System*) que foi desenvolvido para gerar sistemas de *middleware*[10] de grade que suportem a execução de aplicações científicas. Esse ambiente possui um componente de controle que instancia e configura objetos de um *framework* próprio, visando a criação de um *middleware* customizado.

Oliveira em (DEOLIVEIRA, 2010) utilizou o QEF (*Query Evaluation Framework*) para implementar o cálculo de trajetória de partículas em ambiente de grade. Neste *framework* as técnicas de processamento da visualização são modeladas como operadores algébricos e integradas a um conjunto de operadores de controle que gerenciam e distribuição de dados na grade.

Os pontos mais relevantes em (PORTO, 2004) e em (DEOLIVEIRA, 2010) foram a implementação e processamento do cálculo de trajetória de partículas em ambiente de grade computacional. O algoritmo utilizado foi o mesmo que será exposto no capítulo 5.

Em (DEOLIVEIRA, 2010) afirma-se que a implementação do cálculo de trajetória de partículas utilizando QEF teve desempenho superior ao obtido em (PORTO, 2004). Segundo o autor, esse resultado deveu-se ao fato de não existir transferência de dados entre os nós remotos e o nó mestre a cada passo de computação.

Camargo *et al* em (CAMARGO, 2009) apresentam uma implementação do cálculo de trajetória de partículas conforme descrito no capítulo 5 e que foi incorporado ao HeMoLab. A contribuição do trabalho é o aproveitamento das características do HeMoLab/ParaView, descritas na seção 2.2, para proporcionar à esta implementação um ambiente de alto desempenho. O fluxo de trabalho geral do HeMoLab e seu *pipeline* de execução típico, foram expostos na seção 2.2.

O trabalho (HECHT, 2010) apresenta e discute a implementação de uma técnica, semelhante ao CTP, que também é empregada para visualizar simulações de dinâmica de fluidos com base em seu vetor velocidade. A técnica modela partículas que não podem ser vistas individualmente por serem muito pequenas. Tais partículas só podem ser vistas quando em

[10]Um *middleware* de grade é responsável por integrar os diversos elementos de *hardware* e *software* que compõem uma grade e gerenciar o acesso dos usuários aos recursos computacionais presentes na grade, ocultando toda a sua complexidade e infra-estrutura além de oferecer uma interface uniforme para os recursos presentes na grade. Ele é o componente responsável pela autenticação e autorização de acesso aos recursos, entrada e saída de dados e aplicações, execução de programas, localização de recursos e balanceamento de carga.

grandes quantidades e com auxílio de técnicas de iluminação. A implementação realizada foi desenvolvida para ser processada em CPU e sua maior limitação é não permitir identificar visualmente a direção do fluido em um quadro de vídeo isolado, sendo necessários mais quadros para determinar sua direção.

Mittmann *et al* em (MITTMANN, 2009) descrevem e implementam em GPU a técnica de visualização baseada em linhas de corrente e Kondratieva *et al* em (KONDRATIEVA, 2005) descrevem e implementam em GPU o cálculo de trajetória de partículas. Ambos os trabalhos têm por objetivo localizar e rastrear moléculas de água entre tecidos vivos. Tendo como dados de entrada um exame de ressonância magnética o algoritmo identifica e calcula a trajetória de partículas ou linhas de corrente entre as moléculas de água das fibras do tecido, desse modo, identificando estruturas anatômicas. Mittmann *et al* em (MITTMANN, 2009) também implementam uma versão da referida técnica para aglomerados de computadores (*clusters*) e comparam com a versão GPU, concluindo que a versão GPU é superior em tempo de processamento.

Estes trabalhos abordam o cálculo de trajetória de partículas salientando sua importância quanto técnica de visualização de resultados para simulações numéricas e para processamento de imagens. Os resultados dos trabalhos também apontam a utilização de tecnologias como *cluster* e GPU para obter ganho de desempenho em suas implementações.

5 DESCRIÇÃO DA SOLUÇÃO PROPOSTA

Este capítulo destina-se a apresentar o algoritmo de cálculo de trajetória de partículas, seus pontos fortes e suas deficiências. Também será apresentada em detalhes a implementação do CTP utilizando-se a arquitetura CUDA além de diferenciá-la de sua atual implementação serial.

5.1 INTRODUÇÃO

O cálculo de trajetórias de partículas e de linhas de corrente são conceitos comuns em mecânica de fluidos e servem para caracterizar um fluxo de fluidos. O cálculo de trajetórias de partículas é particularmente útil porque permite o estudo do comportamento do fluido instável enquanto as linhas de corrente normalmente são utilizadas para estudar fluidos estáveis.

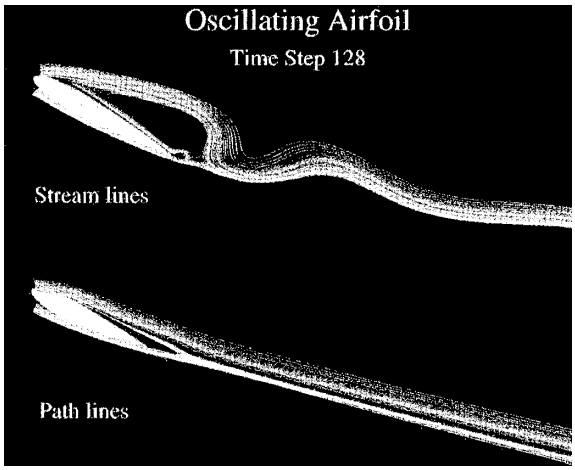
Os fluidos instáveis são aqueles que variam seu escoamento em função do tempo, normalmente variam seu campo velocidade. Suas simulações computacionais geram como saída uma malha de pontos e vários conjuntos numéricos onde cada conjunto numérico representa um passo de tempo da simulação que podem ocupar um espaço de centenas de gigabytes quando somados (KENWRIGHT, 1996). Simulações computacionais de fluidos estáveis, ao contrário, geram uma malha de pontos e um único conjunto de dados numérico pois este tipo de escoamento de fluido não varia em função do tempo (KENWRIGHT, 1996).

Os autores em (KENWRIGHT, 1996) definem três tipos principais de técnicas para visualização de fluidos com escoamentos instáveis.

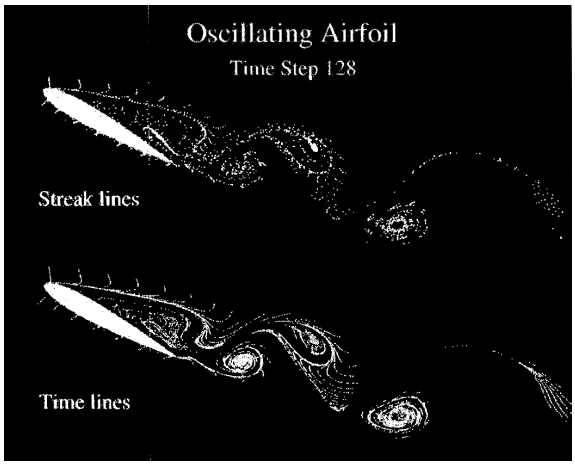
- *path line*: criada pela linha de trajetória de uma única partícula.
- *time line*: criada pelas linhas de trajetórias de várias partículas que são liberadas em um mesmo instante de tempo.
- *streak line*: criada pela lançamento contínuo de partículas em um ponto fixo.

A figura 5.1 mostra as linhas de corrente, as *path lines*, as *streak lines* e as *time lines* para uma simulação de fluxo instável em torno de um aerofólio. A parte de maior importância nessa simulação é o vórtex gerado atrás do aerofólio (KENWRIGHT, 1996) e que foi mostrada

na figura 5.1(b) pelas *streak lines* e *time lines* mas não foi mostrada na figura 5.1(a) pelas linhas de corrente e *path lines*. Essas técnicas falharam pois não foram desenvolvidas para capturar o movimento de fluxo em função do tempo.



(a) Linhas de corrente e *Path Lines*. *Fonte: (KENWRIGHT, 1996)*



(b) *Streak Lines* e *Time Lines*. Fonte: (KENWRIGHT, 1996)

FIG. 5.1: Comparação de técnicas de visualização em um fluxo instável.

Vários algoritmos foram desenvolvidos para tratar o problema do cálculo de trajetória de partículas em fluidos estáveis mas poucos algoritmos foram desenvolvidos para fluidos instáveis devido, principalmente, a complexidade dos cálculos (KENWRIGHT, 1996). Nesse caso normalmente os cálculos das trajetórias são realizados e seus resultados armazenados para somente depois ser gerada a visualização.

O método *Lagrange* para cálculo de trajetória de partículas em fluidos em um escoamento foi concebido em 1760 pelo matemático, físico e astrônomo franco-italiano Joseph Louis Lagrange (1736 - 1813).

(XAVIER, 2006) descreve o CTP como: dada uma partícula de massa desprezível e identificada por sua posição inicial $\mathbf{p}$, um vetor velocidade que descreve o fluxo do fluído ($\mathbf{v_p}$) e um intervalo de tempo (t_0 a $T_\mathbf{p}$), calcular a próxima posição da partícula ($\mathbf{x_p}$) em função da velocidade e do tempo ($\mathbf{v_p}(\mathbf{x_p}, t)$).

$$\begin{cases} \frac{d\mathbf{x_p}}{dt} = \mathbf{v_p}(\mathbf{x_p}, t) \quad t \in [t_0, T_\mathbf{p}] \\ \mathbf{x_p}|_{t=t_0} = \mathbf{p}, \end{cases} \tag{5.1}$$

Na equação 5.1, como dito anteriormente, $\mathbf{p}$ é o identificador (ID) da partícula, que coincide com sua posição inicial em t_0, e $T_\mathbf{p}$ é o instante de tempo final para a trajetória da partícula $\mathbf{p}$ que foi calculada.

5.2 IMPLEMENTAÇÃO ATUAL

A simulação numérica produz grandes volumes de dados, normalmente na ordem de dezenas de *gigabytes* (LARRABIDE, 2006), obrigando qualquer implementação deste algoritmo a ser o mais eficiente possível.

O conjunto de dados de entrada inclui a posição inicial das partículas, a geometria do domínio decomposta em células[11] (tetraedros) e um vetor velocidade associado aos vértices das células (CAMARGO, 2009). As relações existentes, no conjunto de dados, são modeladas como abaixo:

- Pontos (*point-id, coord-x, coord-y, coord-z*): cada ponto de uma geometria possui um identificador (*point-id*) e coordenadas tridimensionais (*coord-x, coord-y* e *coord-z*).
- Geometria (*cell-id, point-list*): no problema em questão é o tetraedro, que é a célula base cada qual possui um identificador (*cell-id*) e um conjunto de vértices associados, representados por uma lista (*point-list*) que contém quatro identificadores de pontos (*point-id*);
- Vetor velocidade (*point-id, time-instant, intensity*): contém o valor da velocidade (*intensity*) para todos os pontos da geometria (*point-id*) em todos os instantes de tempo (*time-instant*);
- Partículas (*part-id, time-instant, point*): cada partícula possui um identificador (*part-id*) e esta posicionada em uma coordenada tridimensional (*point*) em um instante de tempo (*time-instant*);

O primeiro passo do algoritmo, representado como um fluxograma na figura 5.2, é a definição da quantidade de partículas a serem utilizadas e de suas respectivas coordenadas iniciais (x, y e z).

Para calcular a próxima posição da partícula o algoritmo deve encontrar o tetraedro que contém uma dada partícula. Isso se faz necessário uma vez que precisa-se dos valores de velocidade associados aos vértices para calcular a próxima posição da partícula. Após o cálculo da próxima posição, retorna-se ao problema de descobrir qual tetraedro contém a partícula. Essa procura pelo tetraedro correto representa um grande gasto computacional e é diretamente proporcional ao tamanho dos dados de entrada devendo ser otimizada.

[11] Pode ser assumida uma outra forma, por exemplo um cubo.

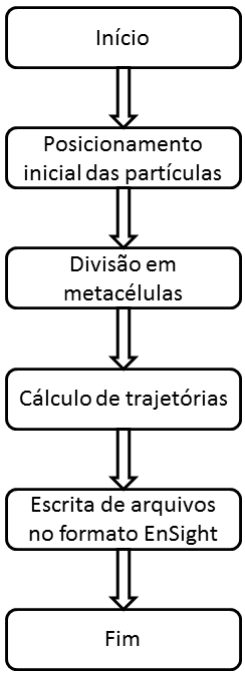


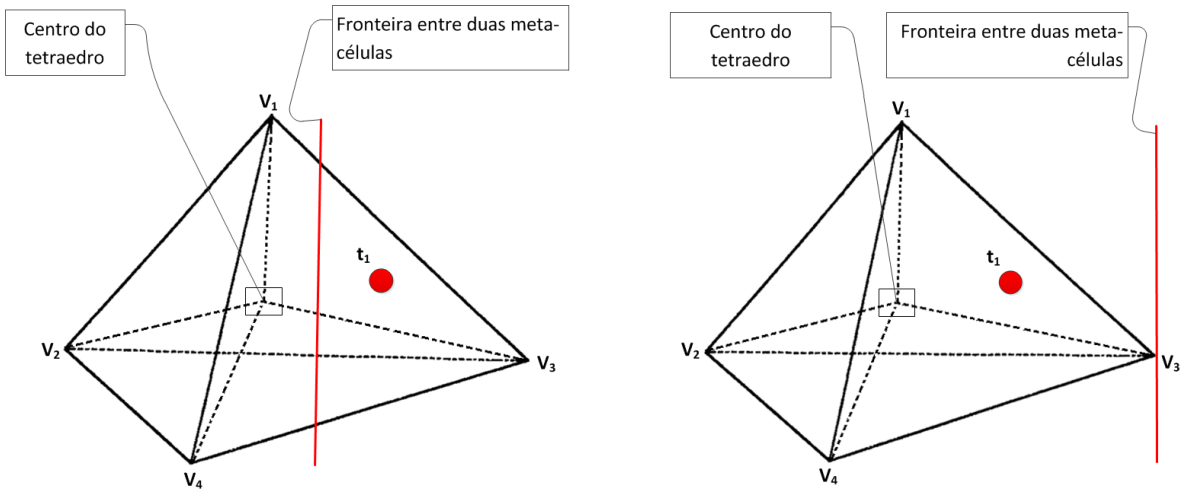
FIG. 5.2: *Pipeline* geral do algoritmo de trajetória de partículas.

Para reduzir o esforço computacional e o tempo despendido na procura do tetraedro correto, a geometria de entrada é ordenada em grupos, chamados de *metacélulas* que são construídas calculando o baricentro de cada tetraedro e, em seguida, ordenando-os como descrito abaixo usando o baricentro como referência:

- Ordenar todas as coordenadas por seus valores do eixo x e particionar em 2 grupos consecutivos;
- Para cada grupo do passo anterior, ordenar todos as coordenadas por seus valores do eixo y e particionar em 2 grupos consecutivos;
- Para cada grupo do passo anterior, ordenar todos as coordenadas por seus valores do eixo z e particionar em 2 grupos consecutivos;

Ao final desta ordenação a geometria de entrada estará dividida em oito metacélulas. Nesse momento cada metacélula terá seus tetraedros percorridos para a identificação dos valores mínimos e máximos para os eixos x, y e z. Esse processo de identificação dos mínimos e máximos baseando-se nos vértices dos tetraedros, ao invés de em seus centros, é importante pois se elimina a possibilidade de situações anômalas onde uma partícula poderia estar no interior de uma tetraedro de fronteira entre duas ou mais metacélulas como mostra a figura 5.3(a) e não ser encontrada pois estaria fora dos limites da metacélula. A figura 5.5(b)

mostra a fronteira englobando todos os vértices do tetraedro. O custo computacional desta etapa é, no pior caso, O(n^2) onde n é o número de tetraedros.



(a) Fronteira posicionada sobre o tetraedro.

(b) Fronteira posicionada fora do tetraedro.

FIG. 5.3: Posicionamento das fronteiras entre metacélulas.

Outra desvantagem decorrente do processo de criação das metacélulas é a sobreposição de suas fronteiras que pode ser vista na figura 5.4 onde a metacélula marcada na cor azul possui $xMin = 1$ e $xMax = 3$ e a metacélula marcada na cor laranja possui $xMin = 2$ e $xMax = 2$. Essa deficiência obriga o algoritmo de busca da próxima etapa (cálculo de trajetória) a procurar em todas as metacélulas pelo tetraedro que contém a partícula. No pior caso (partícula próxima da fronteira) pode ocorrer a busca em todas as metacélulas e no melhor caso (partícula afastada da fronteira) a busca será apenas em uma metacélula. O tamanho dos tetraedros também exerce influência no processo de busca da próxima etapa (cálculo de trajetória) pois quanto menor o tetraedro menor também será a área de fronteira e, por consequência, a possibilidade de um tetraedro de borda conter uma partícula.

Com o conhecimento dos limites x, y e z de cada metacélula pode-se descobrir qual metacélula contém uma partícula apenas comparando suas coordenadas com os valores mínimos e máximos dos eixos das metacélulas. Uma vez encontrada a metacélula que contém a partícula, se existir, inicia-se a busca do tetraedro que está contido na metacélula e que contém a partícula. Uma vez encontrado o tetraedro, se existir, pode-se recuperar os valores de velocidade pertencentes aos seus vértices.

Reconhece-se que a presente técnica utilizada para divisão da malha de entrada e busca pelo tetraedro que contém uma dada partícula não é a mais adequada. Há técnicas como o

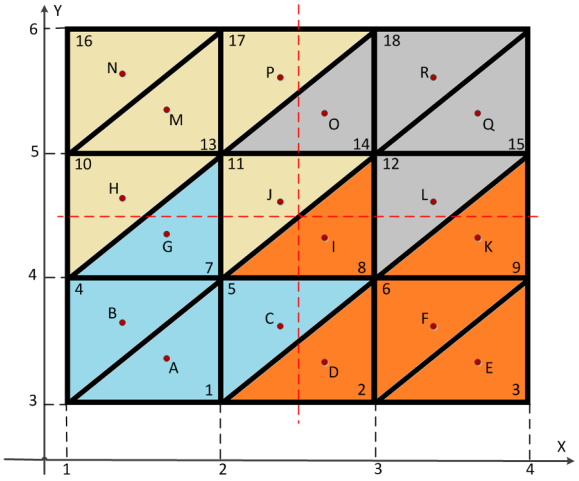


FIG. 5.4: Sobreposição de fronteiras entre as metacélulas.

Octree[12] que possui complexidade igual a O($logn$) (SUNDAR, 2007) e o *Delaunay Searching* (SU, 1994) que possui complexidade igual a O($\frac{\sqrt[3]{n}}{2}$). Entretanto o foco deste trabalho é a paralelização do código referente ao cálculo de trajetória propriamente dito.

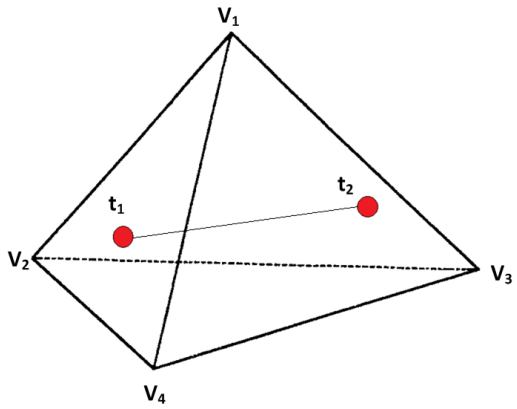
A próxima etapa do algoritmo é calcular a próxima posição das partículas. Cada partícula pode estar posicionada dentro ou fora de algum dos tetraedros, ou células, do domínio. Os valores do vetor velocidade, em cada um dos vértices, exercem influência sobre a posição da partícula resultando em sua movimentação no interior do tetraedro, figura 5.5(a), para tetraedros vizinhos, figura 5.5(b), ou para tetraedros na vizinhança, figura 5.5(c) . As partículas das figuras 5.5(a), 5.5(b) e 5.5(c) sofrem influência das velocidades (V_1, V_2, V_3 e V_4) no instante t_1 o que resulta em uma nova posição da partícula em t_2. A trajetória da partícula é dada pelo conjunto de posições resultantes (CAMARGO, 2009).

Este cálculo é realizado sequencialmente para cada partícula t vezes, onde t é a quantidade de instantes de tempo da simulação, ou seja, toda a trajetória de uma dada partícula p_i é calculada antes de iniciarmos o cálculo da trajetória da partícula p_{i+1}.

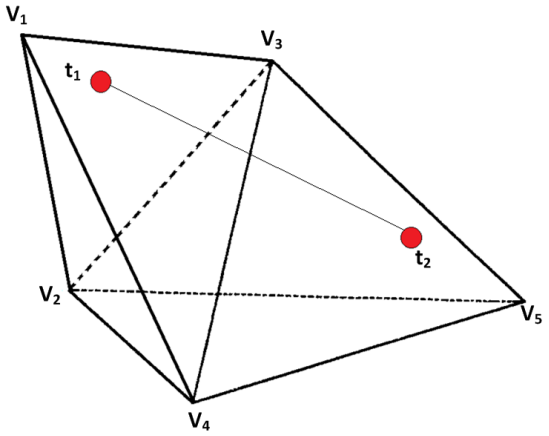
Cada tetraedro pode conter uma, várias, ou nenhuma partícula. As partículas que não se encontram dentro de algum tetraedro ficarão paradas pois não sofrem a influência do vetor velocidade. A figura 5.6 mostra o processo executado pelo algoritmo de CTP em sua versão serial.

Após o cálculo das trajetórias de todas as partículas ocorre a escrita de arquivos no formato EnSight (C.E.I., 2011), contendo os resultados obtidos. Esse formato é utilizado para pós-processamento e visualização de simulações e é suportado pelo HeMoLab/ParaView.

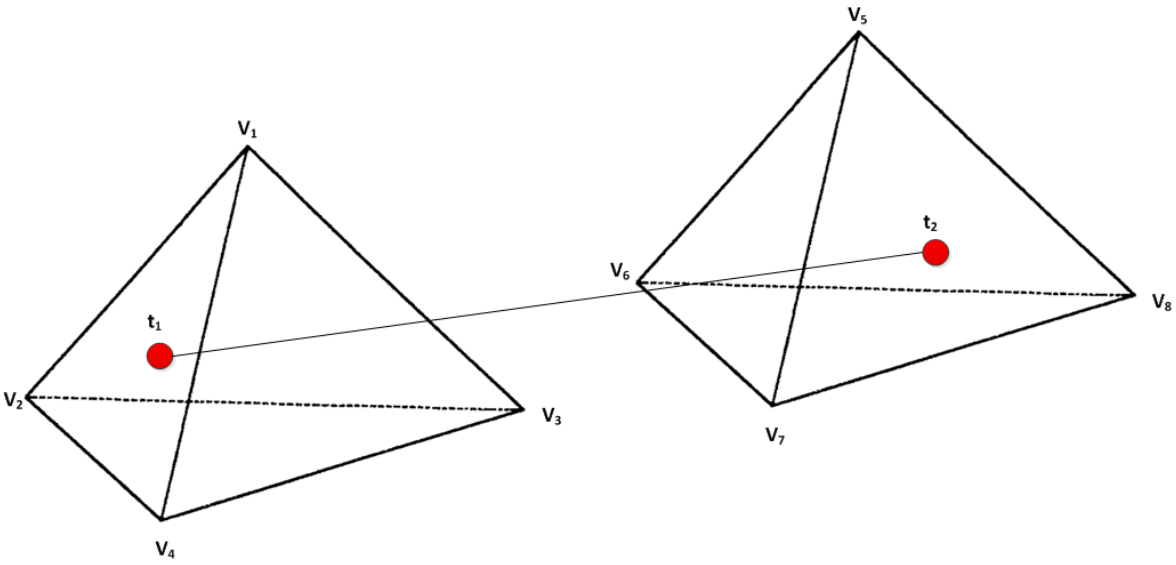
[12]Estrutura de dados em árvore onde os nós filhos possuem zero ou oito filhos.



(a) Deslocamento da partícula no interior do tetraedro.



(b) Deslocamento da partícula para tetraedro vizinho.



(c) Deslocamento da partícula para tetraedro na vizinhança.

FIG. 5.5: Deslocamento da partícula no interior do tetraedro.

Ressalta-se na figura 5.2 que a parte do CTP relacionada especificamente ao cálculo de trajetórias atualmente incorporada ao HeMoLab e que será o foco de implementação em GPU possui complexidade computacional igual a $O(p_n * cell_n)$, onde p_n é a quantidade de partículas e $cell_n$ é a quantidade de células.

5.3 IMPLEMENTAÇÃO DO CTP UTILIZANDO CUDA

A figura 5.7 mostra o fluxograma da versão paralela do CTP que foi implementada utilizando-se CUDA e que será detalhada a seguir.

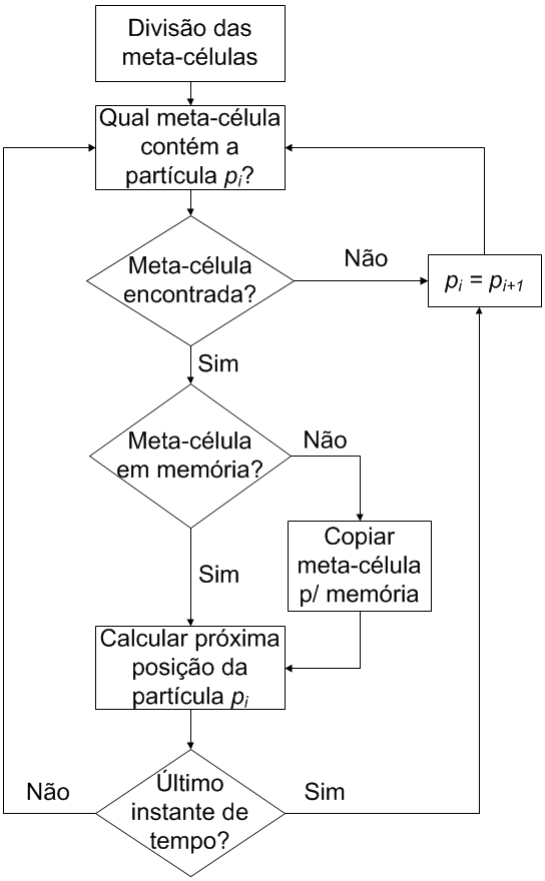


FIG. 5.6: Fluxograma do algoritmo de CTP em CPU.

5.3.1 PRINCIPAIS ESTRUTURAS UTILIZADAS

Ressalta-se que as citações a qualquer linha referem-se às linhas do apêndice 9.1.

Cada ponto da geometria de entrada e os pontos que representam as partículas foram mapeados para a estrutura de dados denominada *point* (linha 2). Esse ponto, ou estrutura, possui um identificador, coordenadas tri-dimensionais, valores para os componentes de velocidade em cada eixo, valor para pressão e um atributo para armazenar o tempo que a partícula levou em seu deslocamento pelo interior da geometria.

Como dito anteriormente, uma célula ou tetraedro é composto por quatro pontos, por este motivo a estrutura que mapeia a célula (linha 20) possui como atributo um vetor $\overrightarrow{v}$ de quatro posições. Essa estrutura também possui um atributo do tipo da estrutura *point* que representa o centro da célula e um atributo para armazenar seu identificador.

Uma metacélula (linha 49) é um conjunto ordenado de células (tetraedros) e, por esse motivo, possui atributos para especificar suas quantidades de pontos e células além de seus limites espaciais. Originalmente, essa estrutura também possuía atributos ponteiros para estruturas do tipo *point* e *cell* a fim de controlar quais eram suas células e pontos. Entretanto,

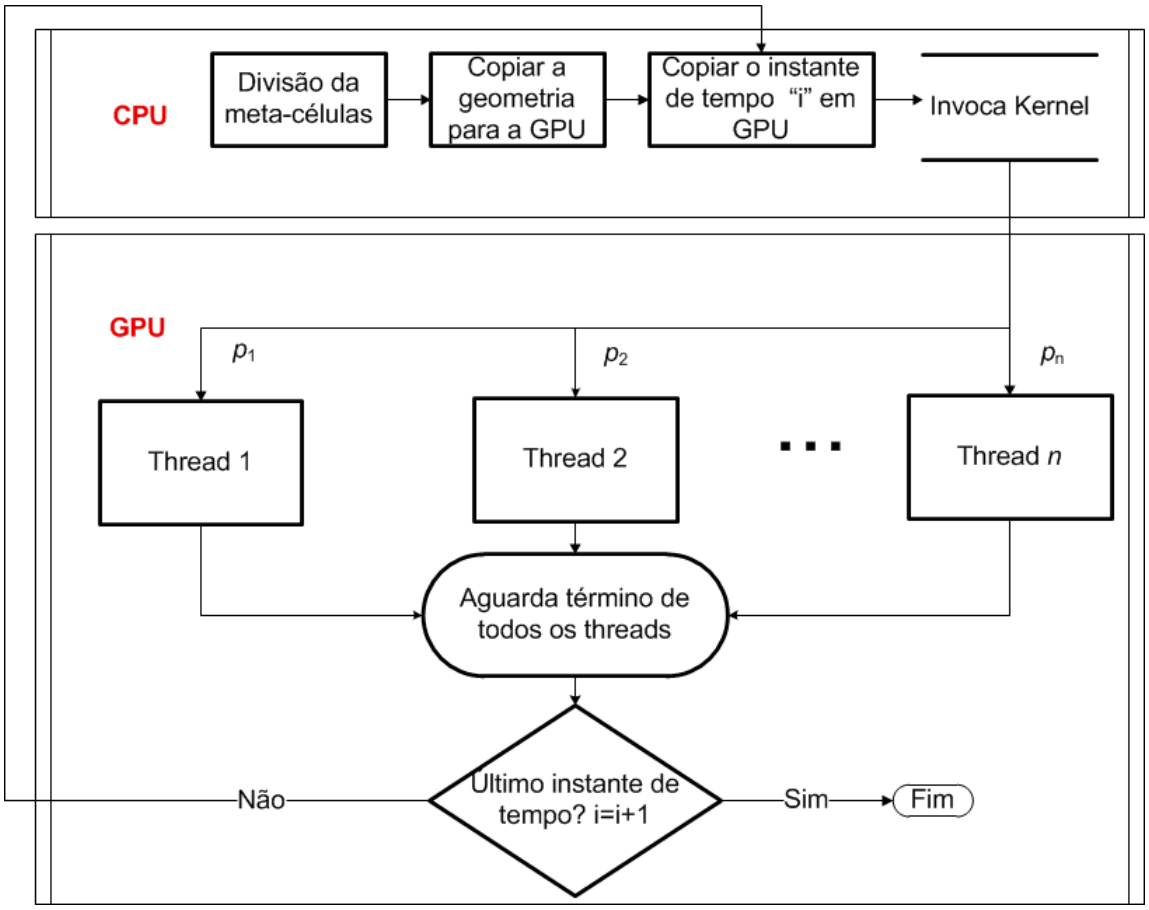


FIG. 5.7: Fluxograma do algoritmo de CTP em GPU.

devido às dificuldades de implementação em GPU, optou-se por remover estes ponteiros internos e mapear esta estrutura sobre um único vetor linear de células do tipo *cell*). As células são armazenadas no vetor seguindo a ordem de sua metacélula, ou seja, todas as células da metacélula 1 são inseridas seguidas por todas as células da metacélula 2 e assim por diante até a última metacélula.

5.3.2 POSICIONAMENTO INICIAL DA PARTÍCULAS

O posicionamento inicial das partículas pode ser feito com o auxílio de três formas geométricas denominadas *widgets*[13] que são posicionadas e dimensionadas pelo usuário para que possam calcular uma quantidade de coordenadas também definida pelo usuário.

A primeira destas *widgets* é a *Line*, mostrada na figura 5.8, onde uma linha é criada

[13]Widget é um termo sem tradução que designa componentes de interface gráfica com o usuário (GUI). Qualquer item de uma interface gráfica é chamada de widget, por exemplo: janelas, botões, menus e itens de menus, ícones, barras de rolagem, etc.

para que seja posicionada e/ou dimensionada pelo usuário. Também possui um parâmetro denominado *Resolution* representando a quantidade de coordenadas (partículas) que serão geradas sobre a linha definida.

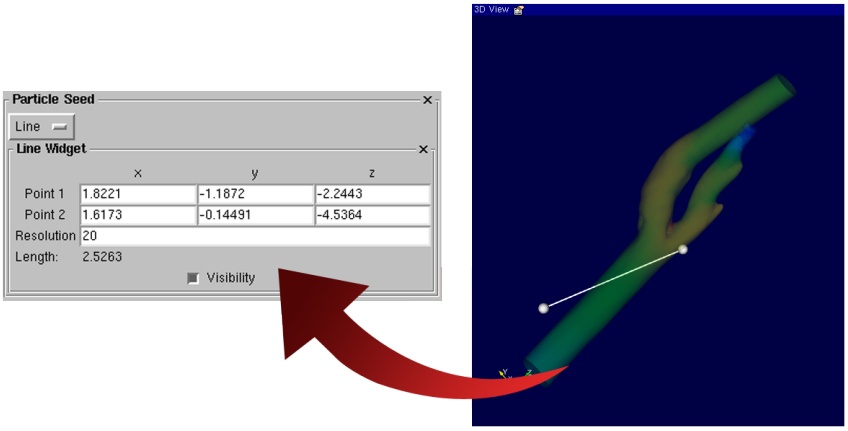


FIG. 5.8: *Line Widget.*

A segunda *widget* é a *Sphere*, exibida pela figura 5.9, onde uma esfera é definida pelo usuário configurando seus parâmetros de raio e centro. Neste caso a quantidade de coordenadas (partículas) geradas aleatoriamente é definida pelo parâmetro *Number of points*, mostrado na interface da *widget*.

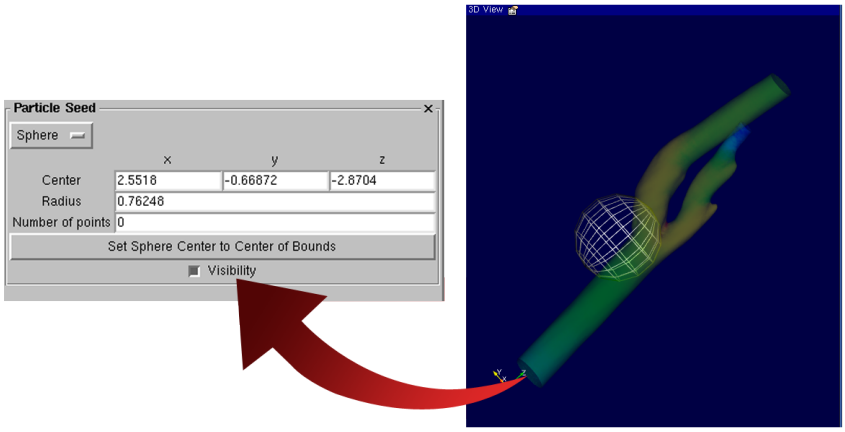


FIG. 5.9: *Sphere Widget.*

A última *widget* é *Points Cloud*, exibida na figura 5.10, onde uma nuvem de pontos é criada em torno do ponto central, definido pelo eixos mostrados na figura, e limitada pelo parâmetro *radius* como no caso da *widget sphere*. O número de coordenadas (partículas) geradas aleatoriamente é definido pelo parâmetro *Number of points*.

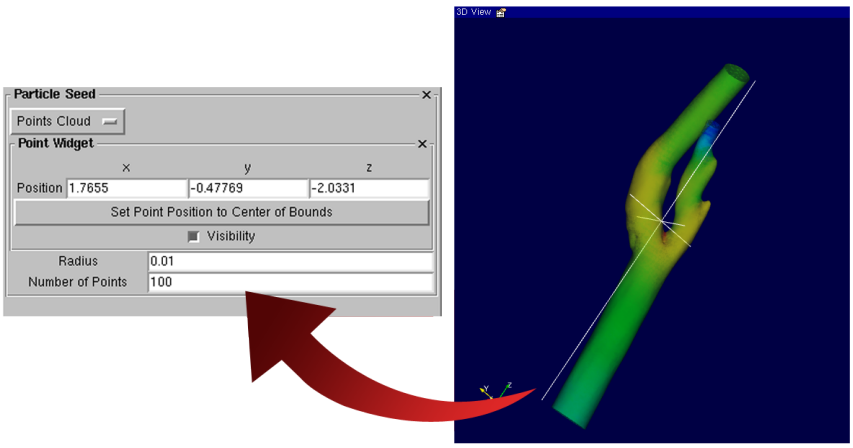


FIG. 5.10: *Points Cloud Widget.*

5.3.3 DIVISÃO DAS METACÉLULAS

Inicialmente são criadas duas listas que irão conter os pontos e células da geometria de entrada como pode ser visto nas linhas 108 e 109 do pseudo-código 9.1. Essas listas são preenchidas conforme as linhas 117 e 126 e no mesmo laço que preenche a lista de células, também são calculados e armazenados o tamanho da maior aresta encontrada na geometria de entrada, na linha 131, bem como o centro de cada célula na linha 137. Esses dois valores serão utilizados na busca da célula que contém uma partícula durante cálculo de trajetórias.

Uma vez que as listas foram preenchidas, os centros de cada célula foram calculados e a maior aresta da geometria encontrada, tem-se início o processo de ordenação e divisão da geometria. O algoritmo de ordenação utilizado é o *quicksort*[14]. A lista inicial de pontos é ordenada pelo componente x do centro de cada célula e dividida ao meio gerando duas sublistas ($l1$ e $l2$), como pode ser visto na linha 145. Cada uma das sublistas $l1$ e $l2$ será ordenada (linhas 149 e 150) pelo componente y do centro de cada célula e divididas ao meio (linhas 153 e 154), dando origem às sublistas $l3$, $l4$, $l5$ e $l6$. Cada uma destas sub-listas será ordenada pelo componente z (linhas 158 à 161) do centro de cada célula e divididas ao meio gerando as 8 metacélulas (linhas 165 à 168). Finalmente, cada uma das oito metacélulas é armazenada em memória secundária (linhas 185 à 192) e o algoritmo está pronto para seguir à próxima etapa que é o cálculo das trajetórias, descrito a seguir. Salienta-se que nenhum tetraedro irá pertencer a mais de uma metacélula pois seu centro foi utilizado na divisão e não seus vértices. Um exemplo detalhado da divisão de uma malha em metacélulas encontra-se no apêndice 9.2

[14]O algoritmo *quicksort* possui complexidade $O(n^2)$ no pior caso e $O(n * logn)$ no caso médio.

5.3.4 CÁLCULO DAS TRAJETÓRIAS

A etapa responsável por efetivamente calcular as trajetórias das partículas será detalhada em nível de implementação. A mesma é exibida na figura 5.11 e compõe-se nos seguintes passos:

1- Inicialização de variáveis, alocação e cópia do vetor de trajetórias para a GPU;

2- Alocação e cópia das metacélulas para a GPU;

3- Alocação e cópia do vetor que conterá os valores de velocidade e pressão de um único instante de tempo para a GPU;

4- Configuração e submissão do kernel para processamento em GPU;

5- Determinação do tetraedro que contém a partículas;

6- Cálculo e armazenamento da próxima posição da partícula.

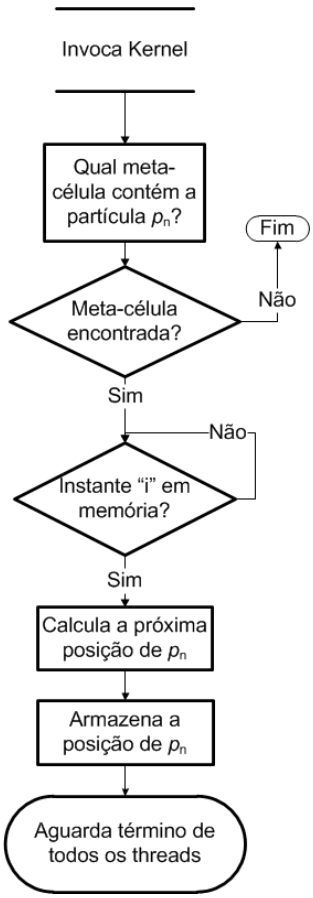


FIG. 5.11: Fluxograma do algoritmo de CTP em GPU (detalhamento dos *threads*).

No passo número 1 ocorre a inicialização de algumas variáveis utilizadas e a seguir ocorre a alocação e inicialização do vetor que irá conter todas as trajetórias (linha 214 e 219) na memória principal. Após a inicialização deve-se copiar as coordenadas (partículas) configuradas pelas *widgets* para o vetor de trajetórias (linhas 222). Nesse instante é alocado o vetor de trajetórias que seguirá para o processamento em GPU (linha 228) e os dados são transferidos para a memória global da GPU.

O passo número 2 consiste em realizar a leitura e a alocação em CPU da geometria (pontos e células) de todas as metacélulas em memória principal (linha 236). Após a leitura, as variáveis que serão utilizadas são alocadas em GPU para representar as metacélulas, seus pontos e células e os dados são copiados para a GPU (linha 240 à 268).

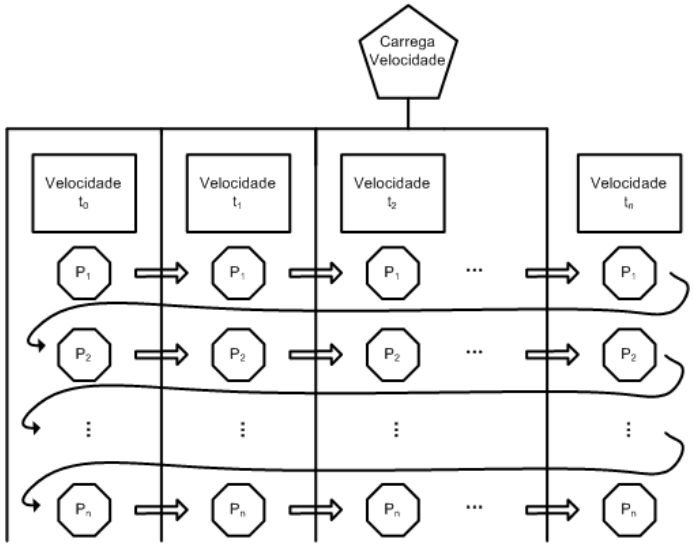
No passo número 3 segue-se a alocação em GPU do vetor (linha 277) que irá conter os valores de velocidade e pressão para cada ponto previamente passado à GPU pelo vetor *Device_fullPoints* em um único instante de tempo. Inicia-se dois laços (linhas 289 e 291) configurados de modo a percorrer todos os instantes de tempo levando em consideração se irá, ou não, ocorrer repetição dos instantes de tempo.

Após a entrada no segundo laço cada instante de tempo será carregado em memória principal (linha 297) e depois enviado à GPU (linha 300). *Essa é a principal diferença entre as versões serial e paralela do CTP* e pode ser vista na figura 5.12. Neste ponto a GPU possui todos os dados necessários para a cálculo do deslocamento das partículas para o instante corrente e então é chamado o método de configuração do *kernel* CUDA (linha 310).

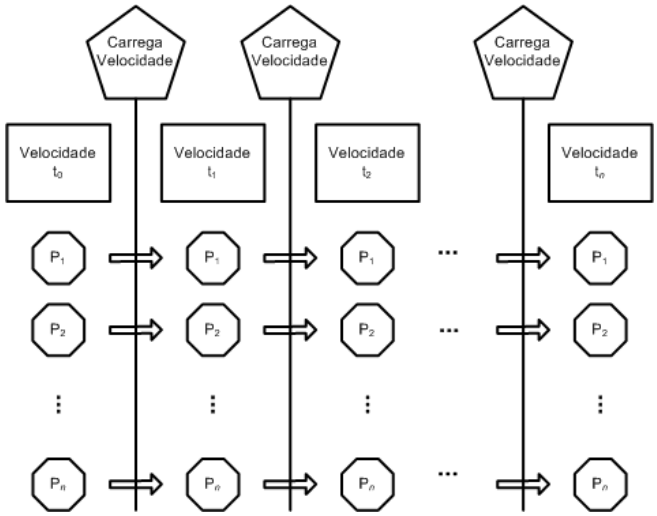
Foi definida a quantidade máxima de 192 *threads* por bloco (linha 470) para a GPU através de experimentação. A quantidade máxima de *threads* por bloco varia em função dos recursos necessários à execução do problema (registradores e memória) e do próprio problema.

No passo de número 4 executa-se um simples cálculo para dividir as partículas pela quantidade de *threads* em um bloco (linha 476) e assim definir a quantidade de blocos que serão executados. Os *threads* e os blocos possuem natureza tridimensional, mas essa característica não se fez necessária, pois apenas o primeiro parâmetro de configuração é utilizado (linhas 479 e 480). Após a configuração das dimensões dos blocos e do *grid*, invoca-se o *kernel* CUDA (linha 483) e todo o processamento, a partir deste ponto, é executado por *threads* assíncronas em GPU.

O passo de número 5 tem início pela localização da metacélula que contém a partícula (linha 518) através da comparação entre os limites da metacélula e a coordenada da partícula. Observa-se a existência de um laço *for* que serve para percorrer todas as metacélulas devido



(a) CPU



(b) GPU

FIG. 5.12: Comparação da execução das versões serial e paralela do CTP

a existência de sobreposição de limites entre elas. Após localizar a metacélula tem início a busca por células candidatas a conter a partícula. Uma célula candidata é aquela onde a distância entre todos os seus vértices e a partícula é menor do que o comprimento da maior aresta (linha 540) e que foi calculado na etapa de divisão das metacélulas. Esse processo tem como fundamento o algoritmo de detecção de colisão entre dois objetos, utilizado na computação gráfica, que antes de determinar se houve colisão entre dois objetos, verifica a possibilidade de colisão através da aplicação da fórmula 5.2 que calcula a distância euclidiana (D) entre o vértice (Q) e a partícula (P).

$$\begin{cases} D = \sqrt{(P_x + Q_x)^2 + (P_y + Q_y)^2 + (P_z + Q_z)^2} \\ P = (P_x, P_y, P_z) \\ Q = (P_x, P_y, P_z) \end{cases} \tag{5.2}$$

Nesse caso, para verificação de possibilidade de colisão utiliza-se um envelope esférico onde o raio da esfera (r) é a distância da maior aresta. Caso a distância entre a partícula e todos os vértices seja menor ou igual ao comprimento da maior aresta significa que existe a possibilidade da partícula estar no interior do tetraedro, a figura 5.13 demonstra este conceito no plano.

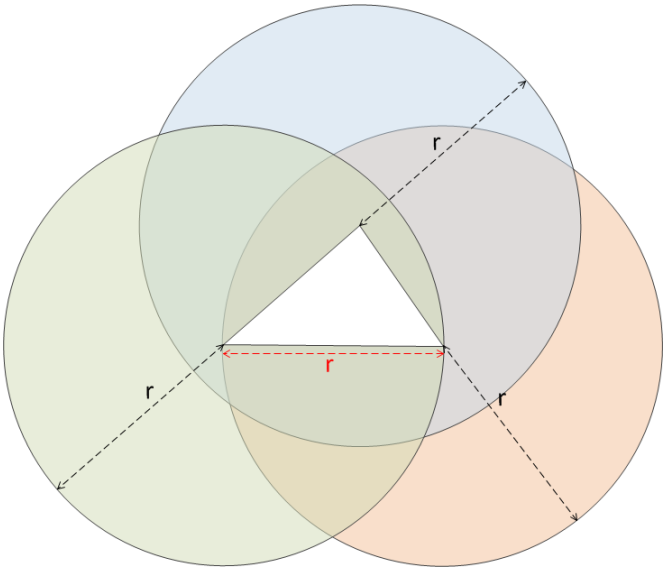


FIG. 5.13: Verificação de proximidade entre partícula e triângulo.

Cada vértice exerce influência na trajetória da partícula logo, quanto mais próximo de um vértice a partícula estiver, maior será sua influência sobre sua trajetória. A figura 5.14 serve como referência para as descrições a seguir. Inicialmente são realizadas as subtrações vetoriais descritas na equação 5.3.

$$\begin{cases} \overrightarrow{V}_1 = P_0 - P_1 \quad \overrightarrow{V}_2 = P_2 - P_1 \\ \overrightarrow{V}_3 = P_3 - P_1 \quad \overrightarrow{V}_4 = P_4 - P_1 \end{cases} \tag{5.3}$$

Após o cálculo dos vetores $\overrightarrow{V}_1$, $\overrightarrow{V}_2$, $\overrightarrow{V}_3$ e $\overrightarrow{V}_4$ calculam-se os produtos vetoriais da equação 5.4 que produzirá as normais N_1, N_2 e N_3.

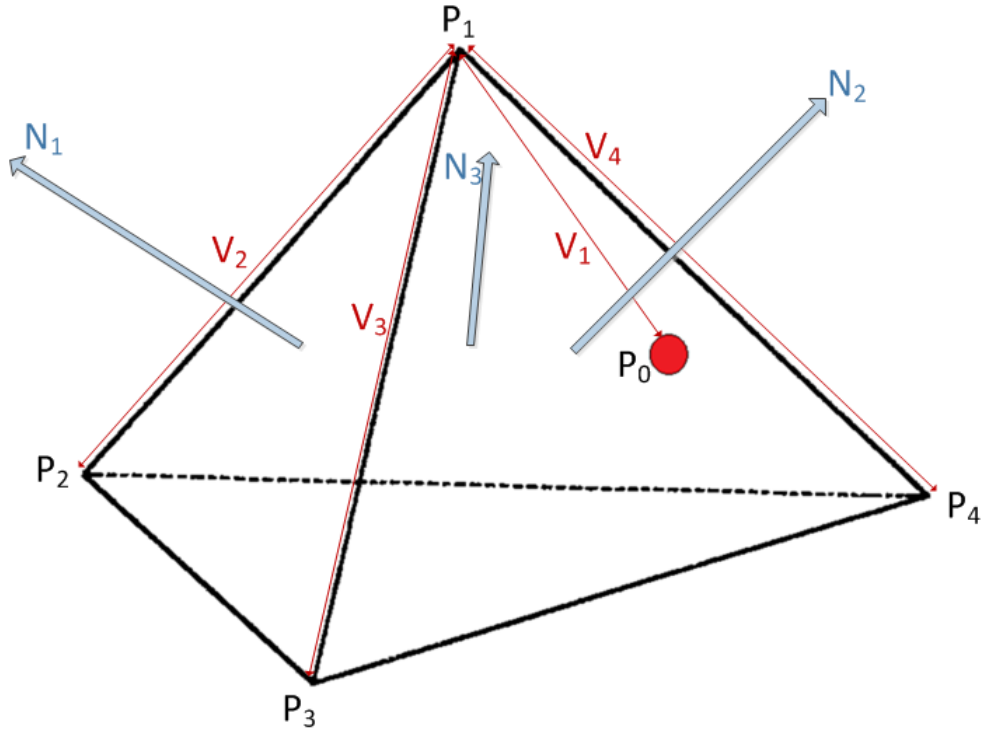


FIG. 5.14: Movendo a partícula.

$$\begin{cases} N_1 = \overrightarrow{V}_2.\overrightarrow{V}_3 \quad N_2 = \overrightarrow{V}_3.\overrightarrow{V}_4 \\ N_3 = \overrightarrow{V}_2.\overrightarrow{V}_4 \end{cases} \tag{5.4}$$

O último passo determina efetivamente se a partícula encontra-se no interior do tetraedro e os valores de contribuição de cada vértice para o movimento da partícula e pode ser visto na equação 5.5. Este peso, ou valor de contribuição, varia entre 0 e 1 para cada vértice desse modo, 1 indica que a partícula encontra-se sobre o vértice verificado e a medida que se afasta o valor tende à 0. Valores maiores do que 1 e menores do que 0 indicam que a partícula encontra-se fora da célula candidata (linha 588).

$$\begin{cases} a1 = 1 - a_2 - a_3 - a_4 \quad a_4 = \frac{Produto(N_1,\overrightarrow{V}_1)}{Produto(N_1.\overrightarrow{V}_4)} \\ a_2 = \frac{Produto(N_3,\overrightarrow{V}_1)}{Produto(N_3.\overrightarrow{V}_2)} \qquad a_3 = \frac{Produto(N_2,\overrightarrow{V}_1)}{Produto(N_3.\overrightarrow{V}_3)} \end{cases} \tag{5.5}$$

Ao final têm-se quatro valores (a_1, a_2, a_3 e a_4) representando a contribuição de cada vértice para o deslocamento da partícula. Caso a partícula esteja situada sobre uma aresta

ou algum dos vértices o algoritmo irá considerar que a mesma esta dentro do tetraedro e dará continuidade ao cálculo, outros tetraedros não serão verificados.

No passo de número 6 tem-se a duração de cada instante de tempo (*delta*), a coordenada de uma dada partícula (*actual_part*) bem como os valores de velocidade em cada vértice da célula ($VecListSearch$) e seus valores de contribuição (a_1, a_2, a_3 e a_4). Esses dados são utilizados para computar a nova posição da partícula utilizando a equação 5.6 e armazenar seu resultado no vetor de trajetórias ($ParticlePath$) (linhas 611 à 627).

$$\begin{cases} next_part.x = actual_part.x + delta * \sum_{i=1}^{4}(a_i * VecListSearch[V_i].x) \\ next_part.y = actual_part.y + delta * \sum_{i=1}^{4}(a_i * VecListSearch[V_i].y) \\ next_part.z = actual_part.z + delta * \sum_{i=1}^{4}(a_i * VecListSearch[V_i].z) \end{cases} \tag{5.6}$$

5.3.5 GERAÇÃO DOS ARQUIVOS ENSIGHT

Após o cálculo de todas as trajetórias tem início a escrita dos arquivos no formato *EnSight* (linha 91) utilizando uma classe do VTK (SCHROEDER, 2004) que implementa esta funcionalidade. Uma descrição completa do formato *EnSight* pode ser obtida em seu manual de usuário disponível em (C.E.I., 2011).

5.4 VALIDAÇÃO DA IMPLEMENTAÇÃO

A saída da aplicação gera um conjunto de arquivos no formato *EnSight* (binários) o que dificulta a comparação e verificação da corretude dos resultados entre a saída processada em CPU e em GPU. Para contornar este obstáculo e facilitar a comparação entre as saídas, o vetor que contém as coordenadas finais é descarregado em um arquivo texto pelo mesmo método (linha 91) que desencadeia a escrita em formato *EnSight*.

Comparando-se os arquivos texto obtidos notam-se diferenças nas casas decimais de alguns valores. Essas divergências eram esperadas pois a arquitetura CUDA não implementava corretamente a norma IEEE-754[15] até o lançamento do novo *chip Fermi*. Ressalta-se que na arquitetura CUDA operações de multiplicação seguidas de adição são implementadas em uma única instrução no *hardware* chamada de FMAD. Instruções FMAD executam duas operações de ponto flutuante em precisão simples por pulso de *clock* porém realizam o truncamento dos valores intermediários o que acarreta em perda de precisão (NVIDIA, 2010)

[15] A norma IEEE-754 foi definida pelo *Institute of Electrical and Electronics Engineers* (IEEE) e estabelece formatos adequados para representar números em precisão simples (32 bits) e precisão dupla (64) bits.

FIG. 5.15: Comparação das trajetórias obtidas em CPU (tela a esquerda) e GPU (tela a direita).

logo, justificando as diferenças encontradas na comparação entre os arquivos texto e que podem ser vistas na figura 5.15.

Essa é uma limitação da GPU utilizada neste trabalho e que poderia provocar comportamentos anômalos nas trajetórias das partículas caso existam muitos instantes de tempo à processar ou o segmento arterial seja muito pequeno. As variações em casas decimais serão acumuladas ao longo do tempo ocasionando erros caso existam muitos instantes de tempo. Também podem gerar erros caso o vaso simulado e visualizado fosse uma arteríola[16] ou um capilar[17], onde uma pequena variação em uma das casas decimais pode ocasionar o deslocamento da partícula para fora dos limites do vaso.

O *chip Fermi* não realiza o truncamento de valores intermediários quando executando a instrução FMAD, logo as discrepâncias encontradas na validação dos resultados não existirão caso utilize-se uma GPU com este novo *chip*.

[16] As arteríolas são vasos sanguíneos de dimensão pequena que resultam de ramificações das artérias

[17] Vasos sanguíneos do sistema circulatório com forma de tubos de calibre bem pequenos. Constituem a rede de distribuição e recolhimento do sangue nas células

6 RESULTADOS

Neste capítulo são apresentados os resultados obtidos com a adaptação do algoritmo de cálculo de trajetória de partículas para processamento em GPU com o uso da arquitetura *CUDA*. Os experimentos realizados têm como objetivo avaliar o desempenho da versão paralela para diferentes quantidades de partículas e para diferentes distribuições de *threads* pelos blocos.

6.1 DESCRIÇÃO DOS EXPERIMENTOS

Para a avaliação de desempenho foi utilizado um computador com processador Intel Quad Core Q9450 @ 2.66 GHz com 8 GB de RAM e uma *GPU* NVIDIA GeForce 9600 GT que possui 8 multiprocessadores de *threads*, cada qual com 8 processadores escalares, resultando em 64 processadores escalares. Essas características da GPU utilizada e mais outras podem ser obtidas através do programa *DeviceQuery* contido no *CUDA SDK*, sua saída pode ser vista na figura 6.1. O diagrama de distribuição de processadores de *threads* desta GPU pode ser vista na figura 6.2. Somente um dos núcleos do processador Quad-core foi utilizado para a execução do CTP.

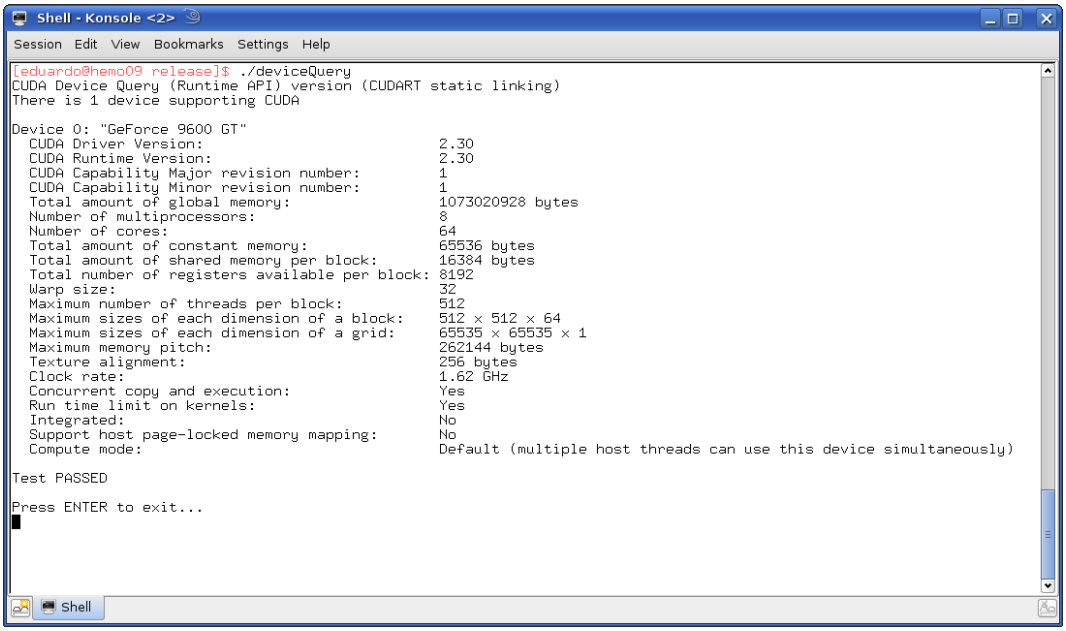


FIG. 6.1: Saída do programa *DeviceQuery*.

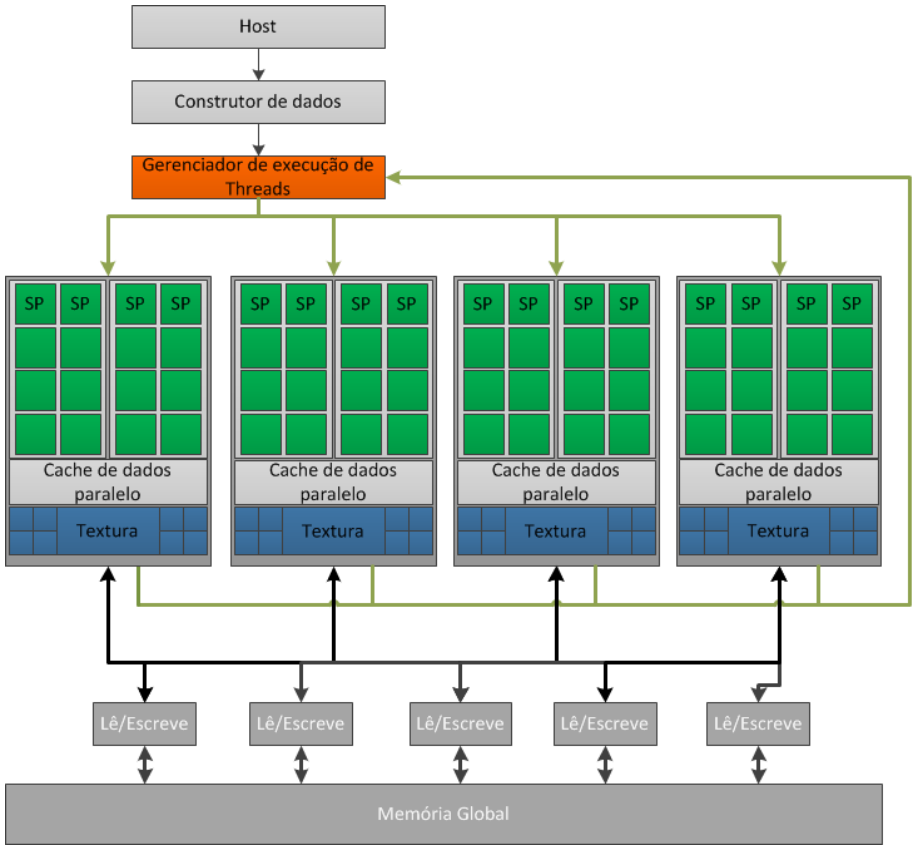


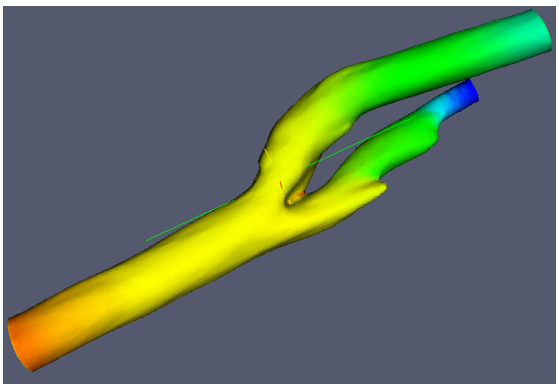
FIG. 6.2: Diagrama da GeForce 9600GT.

Os *softwares* utilizados para o desenvolvimento e para a avaliação de desempenho das versões serial e paralela do CTP foram o Linux OpenSuse 11.0 64 bits, CMake, *VTK*, Eclipse e o *SDK* versão 2.3 da NVIDIA.

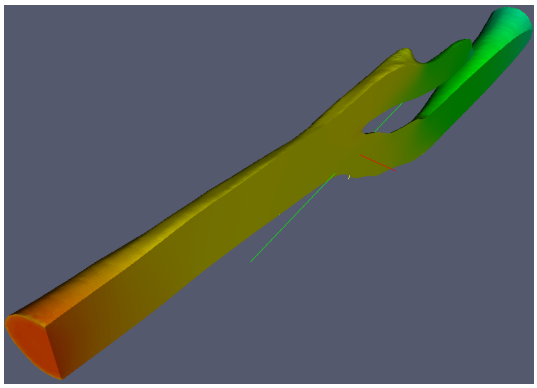
A simulação utilizada como entrada de dados possui as seguintes características:

- Número de pontos igual a 93.517;
- Número de tetraedros igual a 484.495;
- A maior aresta possui comprimento igual a 0,081734 cm;
- Número de instantes[18] de tempo igual a 101;
- O arquivo descritor da geometria possui 10,7 megabytes (MB);
- Os arquivos descritores dos vetores (velocidade, pressão e deslocamento) para todos os instantes de tempo possuem 252,2 megabytes (MB);

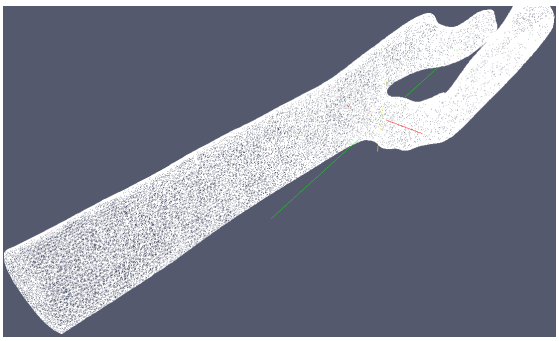
[18]Corresponde a variável i na figura 5.7.



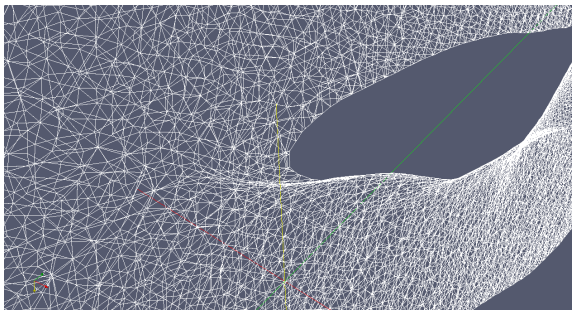
(a) Malha vinda da simulação.



(b) Malha com corte longitudinal.



(c) Arestas da malha em destaque.



(d) Detalhe das arestas da malha.

FIG. 6.3: Malha vinda da simulação.

TAB. 6.1: Configuração dos Experimentos realizados

	Nº de Blocos	Nº de Partículas
Experimento 1	1	Variações de 1 até 100
Experimento 2	2, 4, 8	384, 768 e 1536
Experimento 3	8, 12, 16 e 24	100, 384 e 768

Foram realizados três experimentos como descrito na tabela 6.1 cada qual com 5 repetições. A versão serial executada em *CPU* foi utilizada como referencial para todo o processo de avaliação de desempenho e também foi repetida 5 vezes. Os dados abaixo foram coletados e seus valores médios foram utilizados para a criação dos gráficos indicados.

- Tempo de transferência de dados *CPU* para *GPU*;
- Tempo de transferência de dados *GPU* para *CPU* e
- Tempo de execução total.

6.2 OBJETIVOS

Os objetivos dos experimentos foram:

- Avaliar e quantificar o desempenho para diferentes quantidades de partículas em *CPU* e *GPU*;
- Avaliar o impacto da transferência de dados entre placa mãe e *GPU* sobre o tempo de execução da versão *GPU*;
- Verificar a complexidade computacional do algoritmo implementado e
- Avaliar de que maneira a distribuição de *threads* nos blocos afeta o desempenho da aplicação.

Para isto foram avaliados o *speedup*[19], o tempo médio de execução e o tempo gasto com a transferência de dados em cada experimento.

Os experimentos 2 e 3 são interessantes por fazerem uso do particionamento dinâmico da arquitetura *CUDA* que divide a quantidade de partículas (cada uma associada a um *thread*) pelo número de blocos permitindo, desse modo, avaliar de que maneira a distribuição de *threads* nos blocos afeta o desempenho da aplicação. Para o cálculo do *speedup* foi utilizada a seguinte fórmula.

$$Speedup(p_n) = \frac{CPU(p_n)}{GPU(p_n)} \tag{6.1}$$

Onde na equação 6.1, p_n é o número de partículas, $CPU(p_n)$ é o tempo de execução serial do algoritmo para p_n partículas e $GPU(p_n)$ é o tempo de execução paralela do algoritmo para p_n partículas.

6.3 ANÁLISE DOS RESULTADOS

6.3.1 EXPERIMENTO 1

Como dito anteriormente no capítulo 3 as *CPUs* foram projetadas para o processamento de aplicações de forma sequencial onde devem tratar de um controle de fluxo altamente

[19] *Speedup* é a métrica que captura o relativo benefício de solucionar um problema em paralelo. É definido como a razão entre o tempo utilizado para resolver um problema utilizando um elemento de processamento e o tempo utilizado para resolver o mesmo problema em um sistema paralelo utilizando p unidades de processamento (GRAMA, 2003).

complexo e de grandes caches de dados. As *GPUs*, por sua vez, foram desenvolvidas para o processamento de aplicações altamente paralelas onde uma mesma instrução é aplicada a um grande volume de dados, não sendo necessário um sofisticado controle de fluxo ou grandes caches de dados, pois as *GPUs* escondem a latência da memória com processamento.

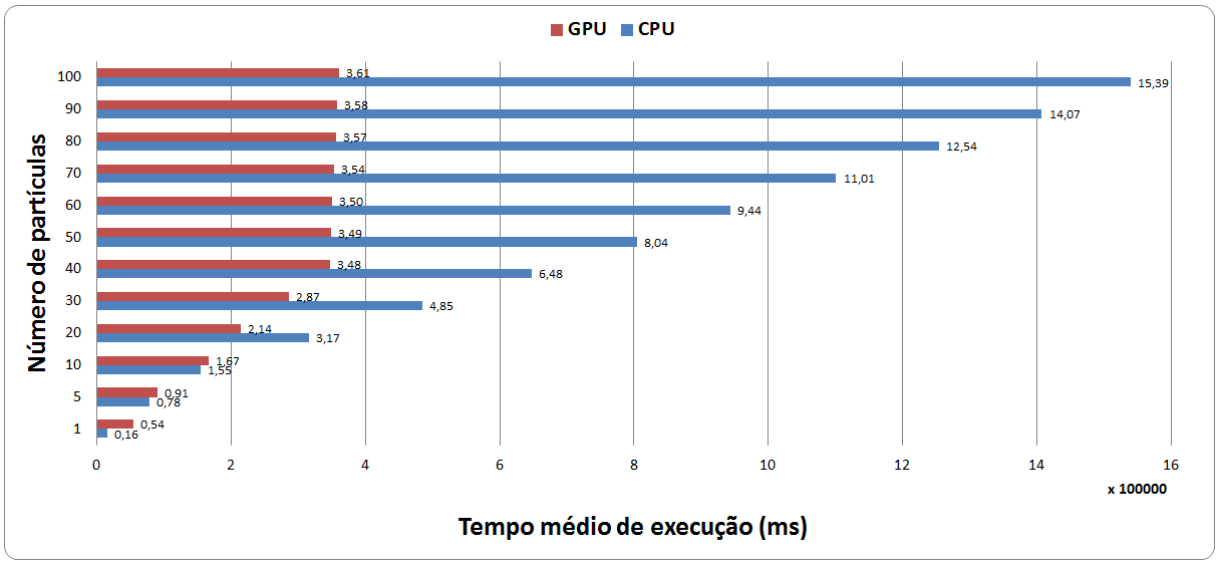


FIG. 6.4: Tempo médio de execução de 1 até 100 partículas.

Desse modo, espera-se que quanto maior a quantidade de dados melhor será o desempenho da *GPU* quando comparada a *CPU*. Essa expectativa foi confirmada pela figura 6.4 que apresenta o tempo de execução em CPU e em GPU variando o número de partículas onde pode-se observar que a *CPU* inicialmente é superior em desempenho, porém a medida que aumenta-se quantidade de partículas, a *GPU* demonstra um melhor desempenho. Observa-se que a tendência da *CPU* é continuar a aumentar o tempo de processamento a medida que a quantidade de partículas aumenta, enquanto que o tempo de execução da *GPU* aumenta mas logo estabiliza, sendo este comportamento o reflexo do paralelismo dos *threads*.

A figura 6.4 também vem confirmar a complexidade do CTP que foi citada no capítulo 5 como sendo igual a $O(p_n * cell_n)$, onde p_n é o número de partículas e $cell_n$ é o número de células (tetraedros).

Analisando a figura 6.5 nota-se que este primeiro experimento não atingiu a máxima capacidade da *GPU* utilizada pois não ocorreu a estabilização, ou diminuição, do *speedup* em função da quantidade de partículas utilizada, o que seria esperado caso o experimento atingisse ou ultrapassasse a capacidade de processamento da *GPU* utilizada. Esse comportamento indica que melhores índices de *speedup* podem ser alcançados aumentado a quantidade de partículas, pelo gráfico, infere-se que o ganho de *speedup* é linear.

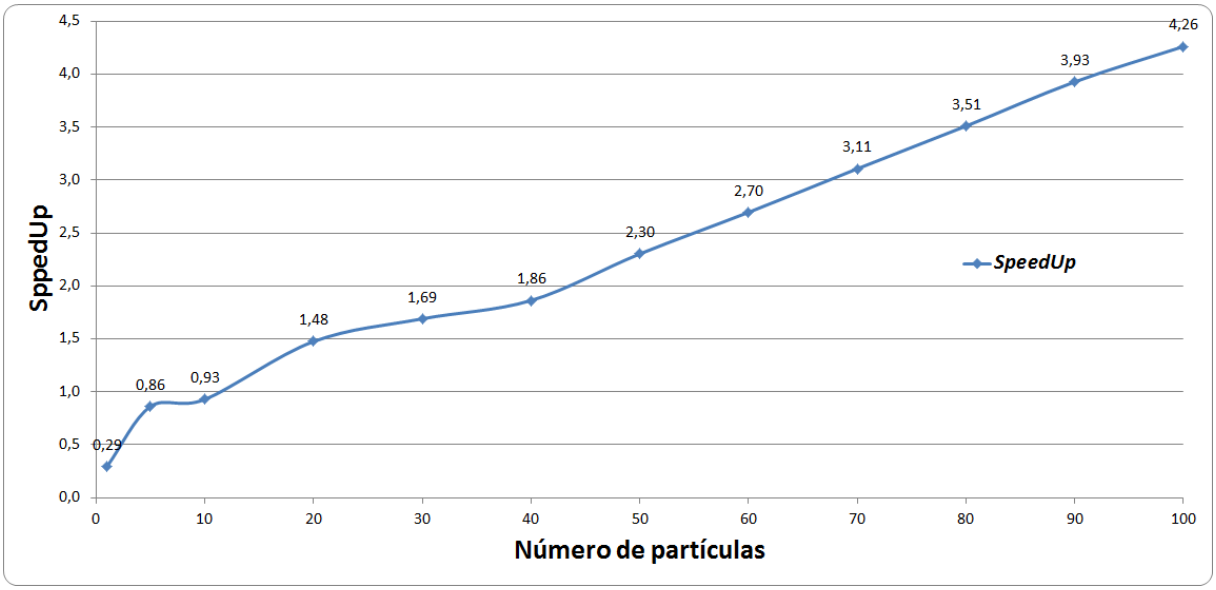


FIG. 6.5: *Speedup* de 1 até 100 partículas.

6.3.2 EXPERIMENTO 2

A medida que a quantidade de partículas aumenta fica ainda mais evidente a disparidade entre o desempenho das versões serial e paralela como pode ser visto na figura 6.6 onde o tempo de execução da versão em *GPU* foi praticamente o mesmo, enquanto que o da versão em *CPU* chegou bem perto de dobrar a cada mudança na quantidade de partículas, ou seja, dobrando a quantidade de partículas o tempo de processamento da *CPU* chega próximo de dobrar porém, em GPU os tempos mantêm-se próximos e com pequenas variações.

Ao observar as figuras 6.5 e 6.7 pode-se notar que a aplicação atingiu um *speedup* de 56 vezes, obtido pela versão *GPU* em relação a versão *CPU* sem, entretanto, estabilizar a curva de *speedup*. Este comportamento continua a indicar que melhores índices de *speedup* podem ser alcançados pelo aumento do número de partículas. É interessante notar na figura 6.6 que utilizando-se 1536 partículas o processamento em CPU termina em aproximadamente 6 horas e 57 minutos enquanto que utilizando-se a mesma configuração porém processando os dados em GPU foi atingido um tempo de aproximadamente 7 minutos.

Observando-se os experimentos 1 e 2 pode-se justificar a vantagem inicial da *CPU* pela pequena quantidade de partículas a processar. Quando não existe um número suficiente de *threads* em cada multiprocessador para ocultar a latência da memória com processamento ocorre ociosidade dos multiprocessadores da *GPU* que devem aguardar até que existam *threads* prontos à executar. Desse modo, quando a aplicação utiliza poucos *threads* o escalonador de *warps* da *GPU* não possui *threads* suficientes para executar enquanto outros

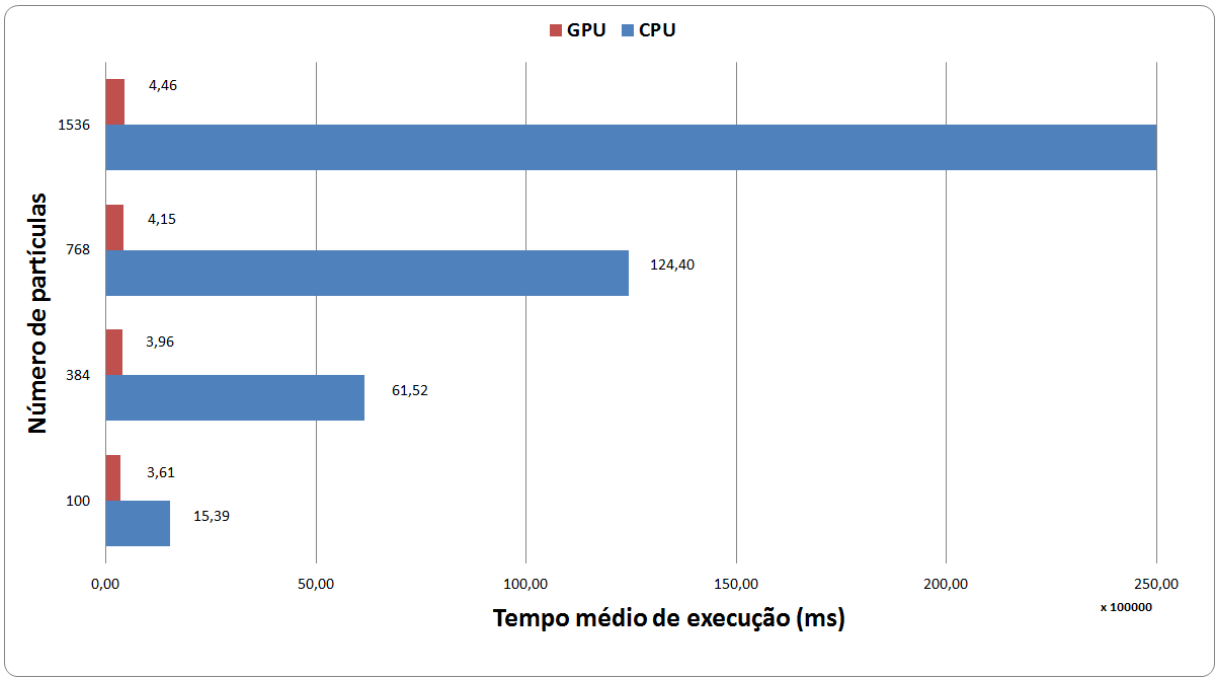


FIG. 6.6: Tempo médio de execução de 100 até 1536 partículas.

acessam a memória, gerando períodos de baixa utilização da *GPU*.

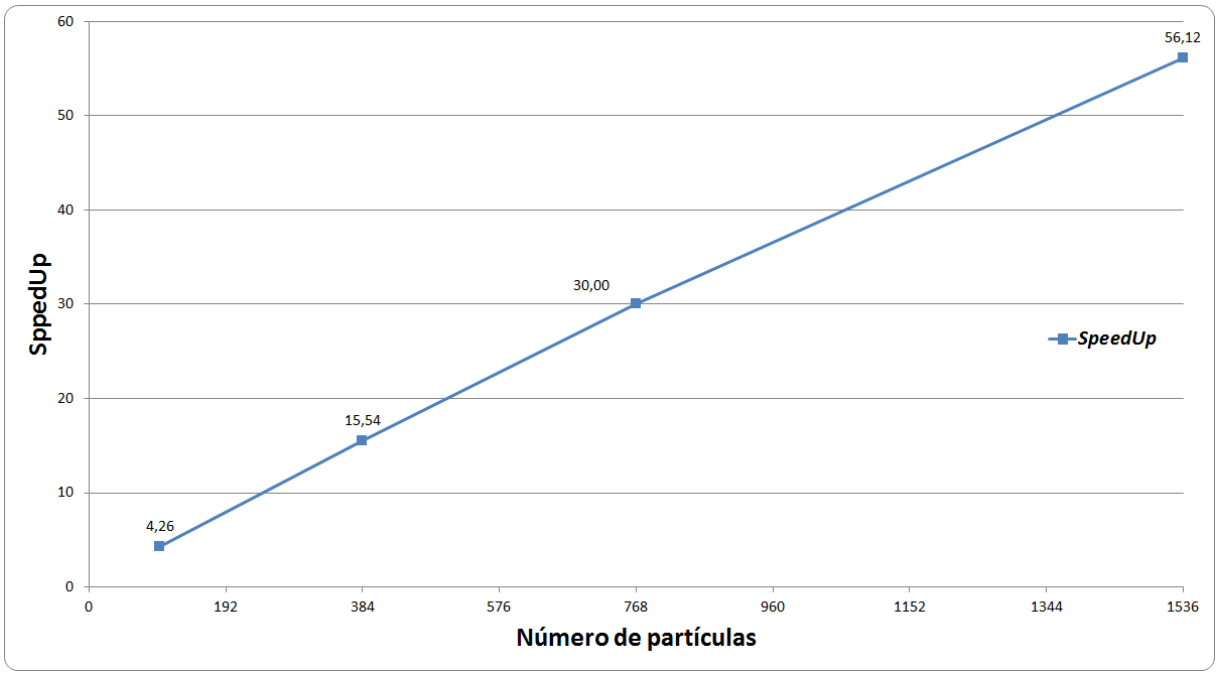


FIG. 6.7: *Speedup* de 100 até 1536 partículas.

Quanto ao tempo dedicado às transferências de dados entre a placa mãe e a memória da GPU nota-se pelo gráfico 6.8 que este representou 1% do tempo total de execução quando utilizadas pequenas quantidades de partículas. Este percentual diminuiu ao aumentar-se a

quantidade de partículas.

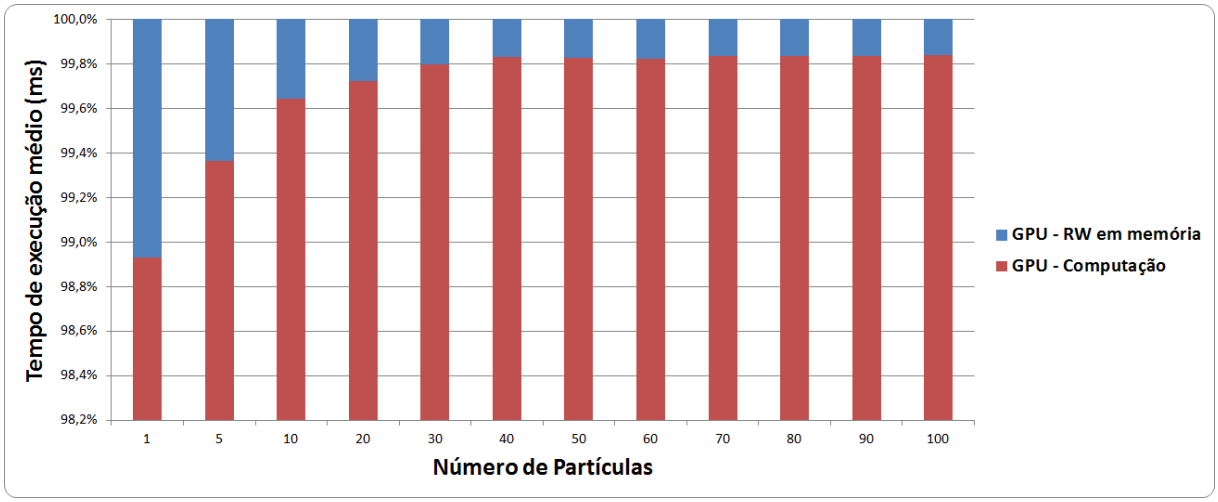


FIG. 6.8: Transferência de Dados *vs* Tempo de Processamento de 1 até 1536 partículas.

Um ponto que chamou a atenção foi o baixo percentual de tempo despendido para a transferência de dados entre placa mãe e a *GPU*. Esperava-se que fosse maior, pois os dados de entrada referentes a cada instante de tempo (resultados da simulação) são transferidos para a *GPU* e processados um a um. Isso indica que o desempenho da aplicação ainda pode ser melhorado com a utilização da memória compartilhada da *GPU* que é mais rápida do que a memória global utilizada entretanto, o algoritmo deveria ser reescrito para utilizar este tipo de memória.

6.3.3 EXPERIMENTO 3

As *GPUs* com suporte a *CUDA* possuem quantidades limitadas de registradores e de memória que podem ser atribuídos a blocos e *threads*. Esses recursos são distribuídos aos *threads* dinamicamente com o objetivo de maximizar seu uso, esta característica foi abordada em maiores detalhes no capítulo 3 e foi explorada no experimento 3. Foram utilizadas diferentes quantidades de blocos e *threads* na tentativa de determinar o impacto desta característica sobre o desempenho da aplicação. Para os testes foram utilizadas as configurações da tabela 6.2 que foram determinadas pelo pseudo-código 6.1.

Pseudo-Código 6.1: Configuração dos blocos e *threads* do experimento 3.

```
1 int r;
2 if( threads % bloco == 0)        r = threads / bloco;
3 else     r = threads / bloco + 1;
```

TAB. 6.2: Blocos e warps ativos em cada multiprocessador.

Partículas (*threads*)	Blocos	Tamanho do bloco	A	B
100	8	13	1	1
	12	9	1,5	1,5
	16	7	2	2
	24	5	3	3
384	8	48	1	2
	12	32	1,5	1,5
	16	24	2	2
	24	16	3	3
768	8	96	1	3
	12	64	1,5	3
	16	48	2	4
	24	32	3	3

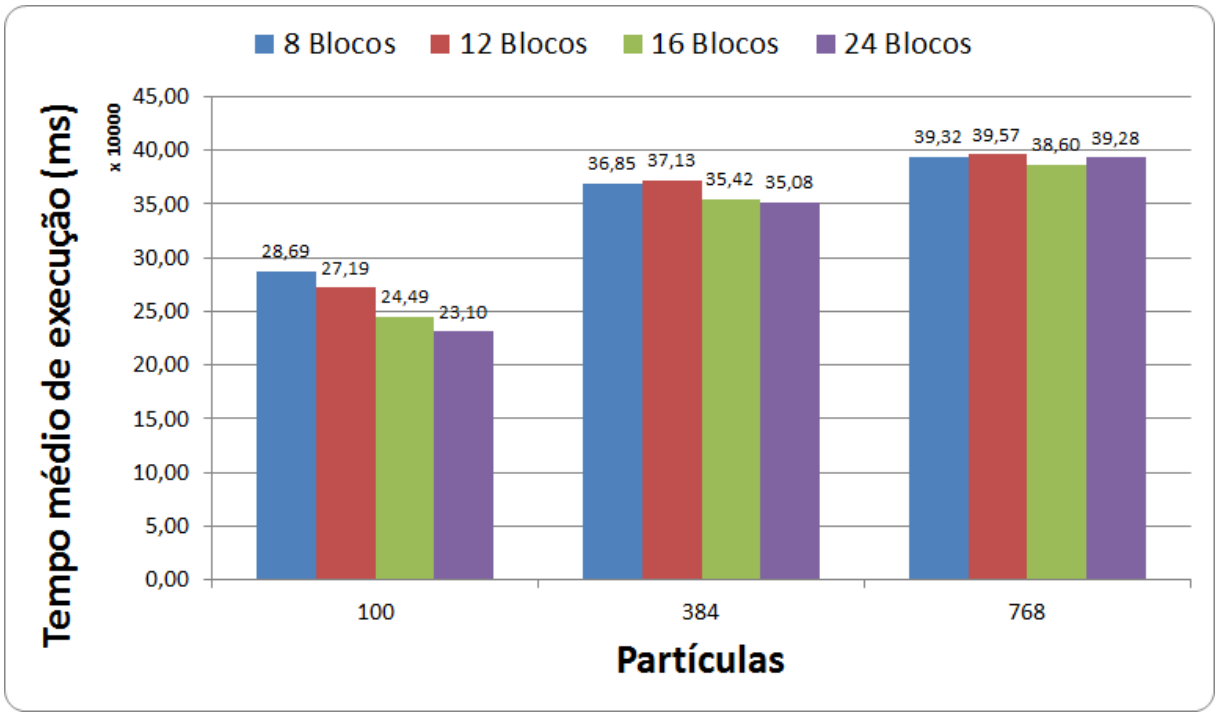


FIG. 6.9: Efeito da distribuição de *threads* pelos blocos no experimento 3.

Na tabela 6.2 a coluna **A** indica os blocos ativos por multiprocessador e a coluna **B** indica os *warps* ativos por multiprocessador.

As especificações das quantidade de recursos de *hardware* em GPUs são expostas de modo padronizado através da chamada *capacidade de computação*. A placa utilizada esta enquadrada como *compute 1.1* como pode ser visto na figura 6.1. Cada nível mais alto de capacidade de computação indica uma geração mais recente de GPU com mais recursos, além de aperfeiçoar os já existentes.

Como expresso na tabela de capacidade de computação encontrada em (NVIDIA, 2010) a GeForce 9600GT pode ter em cada um de seus 8 multiprocessadores até 768 *threads* ativos ou 24 *warps* ativos. A quantidade total de *warps* vezes o tamanho de cada um (32) totaliza 768 *threads* que é maior do que o número máximo de *threads* por bloco (512) permitido logo, conclui-se que não ocorre paralelização no processamento de blocos e sim de seus *warps*.

Na tabela 6.2 pode-se ver as quantidades de blocos ativos que em cada multiprocessador, estas quantidades foram calculadas dividido-se as quantidades de blocos pela quantidade de multiprocessadores existentes na GPU utilizada. Para as configurações que utilizam 12 blocos nota-se que 12 não é um múltiplo de 8 ocasionando que metade dos multiprocessadores tenha 2 blocos ativos e a outra metade tenha 1 bloco ativo.

Para calcular as quantidades de *warps* ativos em cada multiprocessador primeiramente deve-se lembrar que *warps* que não possuem 32 *threads* serão completados com *threads* extras pelo *runtime* CUDA. Desse modo, para identificar a quantidade de *warps* ativos em um multiprocessador deve-se dividir a quantidade de *threads* de um bloco por 32 e realizar um arredondamento para cima, pois o runtime também o realiza, e multiplicar o resultado pela quantidade de blocos ativos em cada multiprocessador. A tabela 6.2 mostra esses valores.

Nota-se que o desempenho ficou prejudicado quando a quantidade de blocos, neste caso 12 blocos, não é múltipla da quantidade de multiprocessadores, neste caso 8. Para a hipótese de 100 partículas nota-se que este fator não influenciou decisivamente o desempenho da aplicação aparentemente devido a pequena quantidade de partículas.

O mecanismo de escalonamento dos *threads* não os escalona diretamente mas sim os *warps*. Toda vez que um dos *warps* realiza um acesso à memória global ocorre o escalonamento, se houver *warps* suficientes, na tentativa de ocultar a latência da memória com processamento. Logo, quanto maior a quantidade de *warps* ativos em um multiprocessador melhor esse mecanismo funcionará. Isso pode ser visto pela comparação da tabela 6.2 com a figura 6.9 para as hipóteses de 384 e 768 partículas, onde nota-se que quanto maior a quantidade de *warps* ativos melhor o desempenho da aplicação. Ao processar 768 partículas em 24

blocos de 32 *threads* observa-se na tabela 6.2 que a quantidade de *warps* ativos, neste caso é de 3. Enquanto que na linha imediatamente acima é de 4 *warps* ativos. Pela comparação destas duas linhas com suas respectivas colunas na figura 6.9, nota-se que com 4 *warps* ativos o tempo de processamento foi melhor do que com 3 *warps* ativos. Justificando a inferência de que o aumento de *warps* ativos pode melhorar o desempenho da aplicação desenvolvida.

7 CONCLUSÕES E TRABALHOS FUTUROS

7.1 CONCLUSÕES

Este trabalho demonstrou como a computação de propósito geral em placas gráficas pode oferecer ganhos de desempenho às aplicações de visualização científica através da implementação de uma ferramenta de baseada no algoritmo de cálculo de trajetória de partículas em unidades de processamento gráfico utilizando-se a arquitetura CUDA.

Para todos os casos avaliados os resultados obtidos com a utilização da arquitetura CUDA foram positivos, incentivando novos experimentos e trabalhos com a utilização desta arquitetura.

As análises realizadas para diferentes configurações demonstraram como as GPUs podem oferecer ganhos de desempenho para a aplicação em questão como demonstram os experimentos apresentados no capítulo 6 onde o tempo de processamento foi reduzido de aproximadamente de 7 horas para 7 minutos, o que representa uma redução de 98%. Outro ponto importante é o aprofundamento dos conhecimentos da arquitetura CUDA o que, com certeza, se faz necessário para o desenvolvimento e otimização de qualquer aplicação que deseje utilizar tal arquitetura.

A dificuldade encontrada e descrita na seção 5.3.1 com relação a implementação de *structs* em ANSI C compostas por outras *structs* provavelmente será facilitado pela NVIDIA, talvez até mesmo extensões para programação orientada à objetos sejam incluídas.

7.2 TRABALHOS FUTUROS

São sugestões de trabalhos futuros diretamente relacionados a esta dissertação:

- Utilização de memória compartilhada e memória constante no presente algoritmo;
- Minimização das divergências relativas ao desvio condicional existente no código;
- Minimização da quantidade de instruções adicionais utilizadas no controle dos laços através da técnica de desdobramento de laços e
- Utilização de outras técnicas de pesquisa espacial, como o *Octree* e o *Delaunay Searching*;

A utilização de GPUs no atual contexto da computação científica mostra-se inexorável, e parece ser o futuro desta área do conhecimento.

Outrossim, sugere-se também a possibilidade de utilizar a computação em GPU para que os filtros disponíveis no HeMoLab atinjam um melhor desempenho, por exemplo o filtro de derivada topológica que foi desenvolvido por Larrabide em (LARRABIDE, 2007). O HeMoLab foi desenvolvido tendo como base o *software ParaView* e a biblioteca VTK como foi descrito em detalhes no capítulo 2. Ambos são focados em visualização científica e processamento de imagens médicas, portanto, nativamente possuem diversos filtros que podem fazer uso da tecnologia de computação em placas gráficas para atingir um melhor desempenho. Neste ponto, todo o estudo e implementação previamente realizados no HeMoLab com o objetivo de incluir a arquitetura CUDA em seu projeto apresentam-se como ponto de partida para futuros trabalhos de portabilidade dos filtros existentes ou inclusão de novos filtros. Uma outra possibilidade é incluir a arquitetura CUDA ao *ParaView* ou ao VTK e portar ou desenvolver filtros para um deles.

Espera-se que no futuro haja a diminuição do tempo de espera do profissional médico pela visualização dos resultados vindos da simulação computacional.

8 REFERÊNCIAS BIBLIOGRÁFICAS

ASANOVIC, K., BODIK, R., CATANZARO, B. C., GEBIS, J. J., HUSBANDS, P., KEUTZER, K., PATTERSON, D. A., PLISHKER, W. L., SHALF, J., WILLIAMS, S. W. e YELICK, K. A. **The Landscape of Parallel Computing Research: A View from Berkeley**. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, Dec 2006. URL `http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-183.html`.

BURIOL, T. M. e ARGENTA, M. A. **Acelerando o Desenvolvimento e o Processamento de Análises Numéricas Computacionais Utilizando Python e Cuda.** Em *Métodos Numéricos e Computacionais em Engenharia - CMNE CILAMCE 2009*, Búzios, Brazil, november 2009.

CAMARGO, E., BLANCO, P. J., FEIJÓO, R. A. e SILVA, R. L. S. **Efficient Implementation for Particle Tracing in Computational Hemodynamics.** Em *Métodos Numéricos e Computacionais em Engenharia - CMNE CILAMCE 2009*, Búzios, Brazil, november 2009.

C.E.I. **Computational Engineering International**. http://www.ensight.com/table/about-us, 2011.

CHE, S., BOYER, M., MENG, J., TARJAN, D., SHEAFFER, J. W. e SKADRON, K. **A performance study of general-purpose applications on graphics processors using CUDA**. *Journal of Parallel and Distributed Computing*, 68(10):1370 – 1380, 2008. ISSN 0743-7315. General-Purpose Processing using Graphics Processing Units.

DE OLIVEIRA, D. E. M. **Otimização das Aplicações de Visualização Científica Usando o QEF**. Dissertação de Mestrado, Instituto Militar de Engenharia, 2010.

GRAMA, A., KARYPIS, G., KUMAR, V. e GUPTA, A. ***Introduction to Parallel Computing (2nd Edition)***. Addison Wesley, 2 edition, janeiro 2003. ISBN 0201648652.

GROUP, T. P. **CUDA Fortran**. http://www.pgroup.com/resources/cudafortran.htm, 2011a.

GROUP, T. P. **The Portland Group, Inc.** http://www.pgroup.com, 2011b.

HECHT, F., MUCHA, P. J. e TURK, G. **Virtual Rheoscopic Fluids**. *IEEE Transactions on Visualization and Computer Graphics*, 16:147–160, 2010. ISSN 1077-2626.

HUANG, Q., HUANG, Z., WERSTEIN, P. e PURVIS, M. **GPU as a General Purpose Computing Resource**. Em *Parallel and Distributed Computing, Applications and Technologies, 2008. PDCAT 2008. Ninth International Conference on*, págs. 151–158, Dec. 2008.

KAPASI, U., RIXNER, S., DALLY, W., KHAILANY, B., AHN, J. H., MATTSON, P. e OWENS, J. **Programmable stream processors**. *Computer*, 36(8):54 – 62, aug. 2003. ISSN 0018-9162.

KENWRIGHT, D. N. e LANE, D. A. **Interactive Time-Dependent Particle Tracing Using Tetrahedral Decomposition**. *IEEE Transactions on Visualization and Computer Graphics*, 2:120–129, June 1996. ISSN 1077-2626. URL `http://dx.doi.org/10.1109/2945.506224`.

KIRK, D. B. e HWU, W.-M. W. ***Programming Massively Parallel Processors: A Hands-on Approach***. Morgan Kaufmann, 1 edition, fevereiro 2010. ISBN 0123814723.

KITWARE. **ParaView**. http://www.paraview,org, 2009a.

KITWARE. **VTK - Visualization Toolkit**. http://www.vtk.org, 2009b.

KONDRATIEVA, P., KRÜGER, J. e WESTERMANN, R. **The Application of GPU Particle Tracing to Diffusion Tensor Field Visualization**. Em *IEEE Visualization*, 2005.

LARRABIDE, I. ***Processamento de imagens via derivada topológica e suas aplicações na modelagem e simulação computacional do sistema cardiovascular humano***. Doutorado em modelagem computacional, Laboratório Nacional de Computação Científica, 2007.

LARRABIDE, I. e FEIJÓO, R. A. **HeMoLab: Laboratório de Modelagem em Hemodinâmica**. Technical Report 13/2006, Laboratório Nacional de Computação Científica, 2006.

LENZ, R., CAVALCANTE-NETO, J. B. e VIDAL, C. A. **Optimized Pattern-Based Adaptive Mesh Refinement Using GPU**. Em *Computer Graphics and Image Processing (SIBGRAPI), 2009 XXII Brazilian Symposium on*, págs. 88–95, Oct. 2009.

MIRANDA, N. e PICCOLI, F. **Using GPU to Speed Up the Process of Audio Identification**. Em *9th International Information and Telecommunication Technologies Symposium (I2TS)*, 2010.

MITTMANN, A., DANTAS, M. e VON WANGENHEIM, A. **Design and Implementation of Brain Fiber Tracking for GPUs and PC Clusters**. Em *Computer Architecture and High Performance Computing, 2009. SBAC-PAD '09. 21st International Symposium on*, págs. 101 –108, 28-31 2009.

NEMA. **Medical Imaging & Technology Alliance**. http://medical.nema.org/, 2009.

NVIDIA. ***CUDA Programming Guide***. NVIDIA Corporation, 2701 San Tomas Expressway, Santa Clara, CA 95050, 3.2 edition, 10 2010.

NVIDIA. ***CUDA Programming Guide***. NVIDIA Corporation, 2701 San Tomas Expressway, Santa Clara, CA 95050, 4.0 edition, 5 2011.

PORTO, F., GIRALDI, G. A., DE OLIVEIRA, J. C., SILVA, R. L. S. e SCHULZE, B. **CoDIMS: an adaptable middleware system for scientific visualization in Grids**. *Concurrency - Practice and Experience*, 16(5):515–522, 2004.

SCHROEDER, W., MARTIN, K. e LORENSEN, B. ***The Visualization Toolkit, Third Edition***. Kitware Inc., 2004. ISBN 1930934122.

SOUZA, C. A. S. D. **Implementação de uma estrutura de dados para visualização científica**. Dissertação de Mestrado, USP, 2003.

SU, P. ***Efficient parallel algorithms for closest point problems***. Tese de Doutorado, Dartmouth College, 1994.

SUNDAR, H., SAMPATH, R. S., ADAVANI, S. S., DAVATZIKOS, C. e BIROS, G. **Low-constant parallel algorithms for finite element simulations using linear octrees**. Em *Proceedings of the 2007 ACM/IEEE conference on Supercomputing*, SC '07, págs. 25:1–25:12, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-764-3. URL `http://doi.acm.org/10.1145/1362622.1362656`.

XAVIER, A. V. **Animação de Fluidos via Atômatos Celulares e Sistemas de Partículas**. Dissertação de Mestrado, Laboratório Nacional de Computação Científica, 2006.

ZIEMER, P. G. P., COLLARES, M. M., CAMARGO, E., DE FREITAS, I. C., BLANCO, P. J., FEIJÓO, R. A. e SILVA, R. L. S. **ImageLab: Um sistema Multi-Orientado na Visualização e Processamento de Imagens Médicas**. Em *Métodos Numéricos e Computacionais em Engenharia - CMNE CILAMCE 2009*, Búzios, Brazil, november 2009.

9 APÊNDICES

9.1 APÊNDICE 1: PSEUDO-CÓDIGO

Pseudo-Código 9.1: Pseudo-código do CTP

```
for(int i=0; i < nSteps; i++) {
  StatusLoadVelocityVectorForSteps[i] = false;}

point * ParticlePath = new point [nPart * nSteps];

for(int i=0; i < nPart; i++){
  double point[3];
  this->InitialPositionOfParticles->GetPoint(i, point);
  SetInitialValues(i, ParticlePath, point, 0.0, 0.0, 0.0);}

Error = CopyParticlePathToGPU( ParticlePath, nPart, nSteps);
Error = CopyGeometryToGPU();
Error = LoadVelocityVectorForSteps( StatusLoadVelocityVectorForSteps );

if( !Error ){
  DispatcherError = DispatcherThreadsGPU( );}

DeleteGeometryFromGPU();
CopyParticlePathToCPU( );
```

9.2 APÊNDICE 2: DIVISÃO DAS METACÉLULAS

Para exemplificar o processo de divisão em metacélulas será utilizada uma malha de células triangulares mostrada na figura 9.1, onde os baricentros de cada triângulo são identificados por um ponto vermelho e uma letra e os triângulos são numerados. As coordenadas dos vértices que compõe cada triângulo, os baricentros e seus identificadores estão descritos na tabela 9.1.

Inicialmente é criada uma lista não ordenada de todos os baricentros que pode ser vista na tabela 9.2. Será usado apenas o Id dos baricentros nas outras tabelas para economizar espaço.

Após a criação da lista de baricentros inicia-se a sua ordenação tendo como base a coluna da coordenada X e utilizando-se o algoritmo *quicksort* o que gera a tabela 9.3. Após a ordenação divide-se a lista ao meio, originando duas novas listas com aproximadamente a mesma quantidade de elementos como mostra a tabela 9.4.

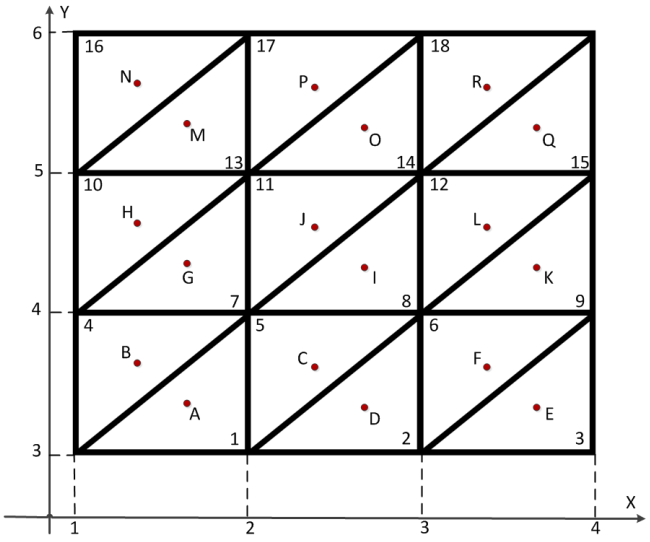


FIG. 9.1: Malha triangular e seus baricentros.

Uma vez terminada a etapa anterior tem-se duas listas que serão ordenadas pela coluna da coordenada Y como mostra a tabela 9.5 e divididas ao meio de forma independente gerando quatro listas que podem ser vistas na tabela 9.6.

Cada lista de baricentros é identificada na imagem 9.2 por uma cor de modo a facilitar a visualização. Os 4 grupos de triângulos agora são chamados de metacélulas.

Neste ponto os vértices de cada metacélula são consultados para a identificação dos valores mínimos e máximo das coordenadas X e Y. Dessa consulta tem-se:

- Metacélula 1 (baricentros A, B, C e G) com $xMin = 1.000$, $xMax = 3.000$, $yMin = 3.000$, $yMax = 5.000$;
- Metacélula 2 (baricentros H, J, M, N e P) com $xMin = 1.000$, $xMax = 3.000$, $yMin = 4.000$, $yMax = 6.000$;
- Metacélula 3 (baricentros D, E, F, I e K) com $xMin = 2.000$, $xMax = 4.000$, $yMin = 3.000$, $yMax = 5.000$ e
- Metacélula 4 (baricentros L, O, Q e R) com $xMin = 2.000$, $xMax = 4.000$, $yMin = 5.000$, $yMax = 6.000$.

TAB. 9.1: Coordenadas dos triângulos e seus baricentros.

Triângulo	X	Y	Id	Baricentro		Triângulo	X	Y	Id	Baricentro	
1	1.000	3.00	A	1.667	3.333	11	2.000	4.000	J	2.333	4.667
	2.000	3.000					2.000	5.000			
	2.000	4.000					3.000	5.000			
4	1.000	3.000	B	1.333	3.667	9	3.000	4.000	K	3.667	4.333
	1.000	4.000					4.000	4.000			
	2.000	4.000					4.000	5.000			
5	2.000	3.000	C	2.333	3.667	12	3.000	4.000	L	3.333	4.667
	3.000	4.000					3.000	5.000			
	3.000	4.000					4.000	5.000			
2	2.000	3.000	D	2.667	3.333	13	1.000	5.000	M	1.667	5.333
	3.000	3.000					2.000	5.000			
	3.000	4.000					2.000	6.000			
3	3.000	3.000	E	3.667	3.333	16	1.000	5.000	N	1.333	5.667
	4.000	3.000					1.000	6.000			
	4.000	4.000					2.000	6.000			
6	3.000	3.000	F	3.333	3.667	14	2.000	5.000	O	2.667	5.333
	3.000	4.000					3.000	5.000			
	4.000	4.000					3.000	6.000			
7	1.000	4.000	G	3.333	3.667	17	2.000	5.000	P	2.333	5.667
	2.000	4.000					2.000	6.000			
	2.000	5.000					3.000	6.000			
10	1.000	4.000	H	1.333	4.667	15	3.000	5.000	Q	3.667	5.333
	1.000	5.000					4.000	5.000			
	2.000	5.000					4.000	6.000			
8	2.000	4.000	I	2.667	4.333	18	3.000	5.000	R	3.333	5.667
	3.000	4.000					3.000	6.000			
	3.000	5.000					4.000	6.000			

TAB. 9.2: Lista inicial de baricentros.

Id	X	Y
A	1.667	3.333
B	1.333	3.667
C	2.333	3.667
D	2.667	3.333
E	3.667	3.333
F	3.333	3.667
G	1.667	4.333
H	1.333	4.667
I	2.667	4.333
J	2.333	4.667
K	3.667	4.333
L	3.333	4.667
M	1.667	5.333
N	1.333	5.667
O	2.667	5.333
P	2.333	5.667
Q	3.667	5.333
R	3.333	5.667

TAB. 9.3: Lista de baricentros ordenada pela coordenada X.

Id	X	Y
B	1.333	3.667
H	1.333	4.667
N	1.333	5.667
A	1.667	3.333
G	1.667	4.333
M	1.667	5.333
C	2.333	3.667
J	2.333	4.667
P	2.333	5.667
D	2.667	3.333
I	2.667	4.333
O	2.667	5.333
F	3.333	3.667
L	3.333	4.667
R	3.333	5.667
E	3.667	3.333
K	3.667	4.333
Q	3.667	5.333

TAB. 9.4: Divisão após a primeira ordenação.

Id	X	Y	Id	X	Y
B	1.333	3.667	D	2.667	3.333
H	1.333	4.667	I	2.667	4.333
N	1.333	5.667	O	2.667	5.333
A	1.667	3.333	F	3.333	3.667
G	1.667	4.333	L	3.333	4.667
M	1.667	5.333	R	3.333	5.667
C	2.333	3.667	E	3.667	3.333
J	2.333	4.667	K	3.667	4.333
P	2.333	5.667	Q	3.667	5.333

TAB. 9.5: Ordenação com base na coordenada Y.

Id	X	Y	Id	X	Y
A	1.667	3.333	D	2.667	3.333
B	1.333	3.667	E	3.667	3.333
C	2.333	3.667	F	3.333	3.667
G	1.667	4.333	I	2.667	4.333
H	1.333	4.667	K	3.667	4.333
J	2.333	4.667	L	3.333	4.667
M	1.667	5.333	O	2.667	5.333
N	1.333	5.667	Q	3.667	5.333
P	2.333	5.667	R	3.333	5.667

TAB. 9.6: Divisão após a segunda ordenação.

Id	X	Y	Id	X	Y	Id	X	Y	Id	X	Y
A	1.667	3.333	H	1.333	4.667	D	2.667	3.333	L	3.333	4.667
B	1.333	3.667	J	2.333	4.667	E	3.667	3.333	O	2.667	5.333
C	2.333	3.667	M	1.667	5.333	F	3.333	3.667	Q	3.667	5.333
G	1.667	4.333	N	1.333	5.667	I	2.667	4.333	R	3.333	5.667
			P	2.333	5.667	K	3.667	4.333			

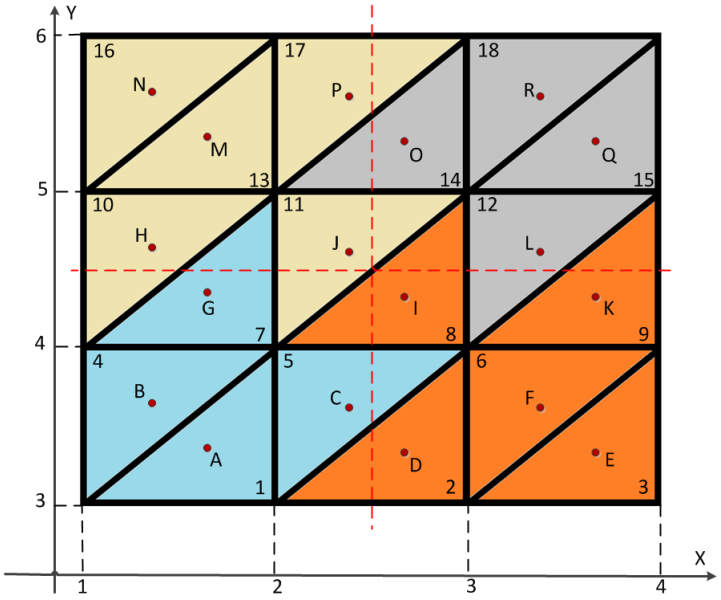



FIG. 9.2: Metacélulas de triângulos.