Instituto de Matemática e Estatística da Universidade de São Paulo

Implementação do Método de Integração Numérica Runge-Kutta em GPGPU para Aplicações Científicas

Giancarlo Rigo
Rafael Reggiani Manzo
Supervisor: Prof. Doutor Marcel P. Jackowski

2 de dezembro de 2012

Sumário

Parte I

Objetiva

Capítulo 1

Introdução

1.1 Motivações

O método de integração numérica de Runge-Kutta permite a aproximação da solução de equações diferenciais ordinárias (EDOs), podendo ser generalizado para a reconstrução de trajetórias tridimensionais a partir de campos vetoriais, que será o objeto desta monografia.

Este método é computacionalmente custoso quando a quantidade de pontos iniciais é grande. Isto impossibilita o seu uso em aplicações em tempo real quando programada de forma convencional para CPU. Contudo, já existem implementações que conseguem atingir a performance desejada para processamento em tempo real através do uso de uma placa gráfica dedicada (6.3.6) para realizar o processamento em paralelo.

Hoje o que há de mais moderno para programação de propósito geral na placa gráfica é o uso de *CUDA* ou *OpenCL*. Mas, ao pesquisarmos implementações do método nestas linguagens encontramos apenas um resolvedor de equações diferenciais feito em *OpenCL*(6.3.6) e nada sobre reconstrução de trajetórias tridimensionais foi feito em ambas as linguagens.

Por fim, uma das aplicações para este método é a reconstrução de fibras musculares ou do tecido branco cerebral a partir de imagens por difusão de tensores (conhecido por tractografia). Isto é uma funcionalidade comum em softwares para exploração de imagens médicas, como o *MedSquare* (6.3.6).

1.2 Objetivos

Primeiramente, é preciso criar um protótipo em *C++* capaz de processar o *dataset* de entrada, aplicar o método e exibir seus resultados. Isto será útil para ganhar familiaridade com todos os aspectos que envolvem a implementação.

Após, como este método é computacionalmente custoso porém altamente paralelizável, o segundo objetivo é tê-lo implementado para *GPU*, com *CUDA* e *OpenCL*, de forma que uma aplicação seja capaz de fazer sucessivas chamadas ao algoritmo e cada uma destas responda em um tempo curto o bastante para ser considerado em tempo real.

Com estas três implementações prontas, para ganhar familiaridade com a *VTK*, será igualmente importante fazer um prótipo equivalente ao anterior, mas que use esta biblioteca que é muito poderosa e difundida, mas igualmente complicada.

Além destes protótipos, e tão importante quanto, será a criação de testes de performance para as implementações obtidas para garantir que, de fato, a implementação em *GPU* é mais eficiente que a convencional em *CPU* e também mensurar qual a diferença de eficiência entre elas.

Por fim, quando alcançados estes objetivos, será possível implementar este algoritmo como um *pipeline* para o *VTK*, permitindo a implementação no software livre *MedSquare*[6.3.6] onde um de seus possíveis usos será a adição da funcionalidade de tractografia em tempo real (*real time fiber tracking*[6.2]).

1.3 Desafios

Quando programamos para *GPU* temos que ter em mente certas limitações da linguagem, como complexidade das estruturas de dados, a melhor forma de utilizar toda sua capacidade em paralelo, a forma mais eficiente de usar seus vários níveis de memória e, principalmente, como minimizar a transferência da *RAM* para a *GPU* e vice-versa. Caso contrário, muito provavelmente, a implementação em *GPU* será mais lenta que uma para *CPU*.

Ainda no contexto de programação para *GPU*, um segundo desafio será a implementação em *OpenCL* que é uma linguagem muito menos difundida que o *CUDA* e ligeiramente diferente desta para implementar, pois permite que um mesmo código seja executado tanto em *GPU* quanto *CPU*.

Por fim, a implementação de um *pipeline* para o *VTK* torna a implementação do método ainda mais complexa pois ela deve se adaptar a sua arquitetura.

Capítulo 2

Conceitos e tecnologias estudadas

2.1 Problemas de Valor Inicial

São os problemas nos quais são dados:

- um conjunto de pontos iniciais P;
- o valor de $f(P_0)$ para cada $P_0 \in P$;
- um sistema de uma ou mais EDOs em f;
- e um tamanho de passo h.

Por fim, desejamos obter $f(P_0 + h)$.

2.2 Método de Integração Numérica Runge-Kutta

É um dos métodos para resolução de Problemas de Valor Inicial através da obtenção de uma aproximação para o valor de $f(P_0 + h)$. Ele é uma generalização do Método de Euler através da aplicação de séries de Taylor (6.1). Essa generalização nos permite a obtenção de diversas ordens do método, dentre as quais as mais comuns são 2 e 4, conhecidos como *RK2* (ou, também, *método do ponto médio*) e *RK4*.

Ambos os métodos possuem termos da ordem de uma potência de h. Estes termos são o erro associado ao método e, portanto, quanto menor o tamanho do passo, menor o erro do método.

Logo, dado um Problema de Valor Inicial conforme descrito acima com uma única equação diferencial $g = f'$, temos os seguintes métodos definidos para apenas um ponto inicial, embora sua generalização para mais pontos consista apenas de sucessivas aplicações do mesmo método para cada ponto.

2.2.1 Ordem 2

Sejam k_1 e k_2 variáveis auxiliares, temos a seguinte expressão para o método de ordem 2:

$k_1 = h.\, g(P_0)$
$k_2 = h.\, g(P_0 + \frac{k_1}{2})$
$f(P_0 + h) = f(P_0) + k_2 + O(h^3)$

2.2.2 Ordem 4

Sejam k_1, k_2, k_3 e k_4 variáveis auxiliares, temos a seguinte expressão para o método de ordem 4:

$k_1 = h.\, g(P_0)$
$k_2 = h.\, g(P_0 + \frac{k_1}{2})$
$k_3 = h.\, g(P_0 + \frac{k_2}{2})$
$k_4 = h.\, g(P_0 + k_3)$
$f(P_0 + h) = f(P_0) + \frac{k_1}{6} + \frac{k_2}{3} + \frac{k_3}{3} + \frac{k_4}{6} + O(h^5)$

2.3 Campo vetorial como discretização de EDOs

Se calcularmos o valor de cada EDO em um conjunto finito de pontos P obteremos então um campo vetorial que pode ser compreendido como a discretização destas equações neste conjunto limitado.

2.3.1 Algoritmos de aproximação

Porém esta discretização exige uma aproximação para o caso onde desejamos obter um o valor da equação, em um ponto não definido no campo vetorial. Como, por exemplo, o caso onde as coordenadas de todos os pontos são inteiros e o ponto desejado tem alguma de suas coordenadas racional.

Nearest Neighbour

No caso do exemplo acima (2.3.1), este algoritmo simplesmente aproxima o ponto desejado para o ponto mais próximo a este que esteja definido no campo vetorial.

Ou seja, um algoritmo para uma das coordenadas cujo valor pretendido seja a, que pode ser generalizado para as demais, é:

1. Se $(a - \lfloor a \rfloor) \geq 0.5$ então devolvemos o valor em $\lceil a \rceil$;
2. Caso contrário, então devolvemos o valor de $\lfloor a \rfloor$.

Interpolação trilinear

Antes de definir a interpolação trilinear é preciso definir a **interpolação linear**. Na qual dados dois pontos, P_1 e P_2, para os quais uma função f está definida, o valor desta função em um ponto P, $f(P)$, sobre a reta r que liga estes dois pontos é a ponderação do valor de $f(P_1)$ e $f(P_2)$ pela distância de P a P_1 e P_2, respectivamente.

Este método muito utilizado no $\Re^2$ é dado por:
$f(P) = f(P_1) + \frac{(f(P_2)-f(P_1))}{(P_2-P_1)}.(P - P_1)$

Por sua vez, a interpolação trilinear é inerentemente utilizada no $\Re^3$. Ela pondera os oito pontos P_1, P_2, P_3, P_4, P_5, P_6, P_7 e P_8 definidos no campo para então obter P.

No caso do exemplo 2.3.1, podemos compreender estes oito pontos como os vértices do cubo que contém o ponto de interesse.

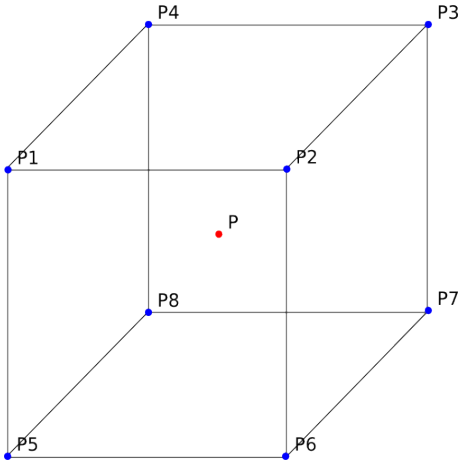

Figura 2.1: Os pontos cujas coordenadas são todas as combinações de chãos e tetos do ponto P formam um cubo unitário que o contém

A idéia do algoritmo é interpolar os pontos de cada aresta dois a dois na mesma coordenada sucessivamente. Se interpolarmos sobre x, depois sobre y e por fim sobre z para obtermos P, teremos as seguintes interpolações a serem calculadas:

1. $X_1 = intp(P_1, P_2)$;
2. $X_2 = intp(P_3, P_4)$;
3. $X_3 = intp(P_5, P_6)$;
4. $X_4 = intp(P_7, P_8)$;
5. $Y_1 = intp(X_1, X_2)$;
6. $Y_2 = intp(X_3, X_4)$;
7. $P = intp(Y_1, Y_2)$.

Onde $intp(A, B)$ é uma função que calcula a interpolação linear de A e B.

2.4 Computação de propósito geral na GPU

Mais conhecida como *General-Purpose computing on Graphics Processing Units (GPGPU)*, trata-se de criar trechos de código que são executados na unidade de processamento gráfico ao invés de fazê-lo na *CPU*. Este recurso é interessante para algoritmos altamente paralelizáveis uma vez que as unidades de processamento gráfico foram feitas para isto.

Por exemplo, uma NVIDIA GeForce GTX 690 possui mais de 3000 núcleos de processamento (*CUDA Cores*) a aproximadamente 900MHz cada e 4GB de memória dedicada (6.3.6). O que representa um pequeno *mainframe* à disposição para a execução de algoritmos paralelos.

Ainda no princípio das placas gráficas dedicadas, percebendo seu poder computacional, teve início a programação de algoritmos gerais (isto é, algoritmos que não sejam de processamento gráfico) em termos de operações gráficas. Ou seja, os algoritmos eram traduzidos em termos de multiplicações de matrizes para poderem ser processados na placa gráfica.

Com o crescimento deste tipo de uso, surgiu a linguagem *Cg* facilitando a confecção de programas para a placa gráfica, mas que no fim das contas era compilado em termos de *DirectX* ou *OpenGL shaders*. Ou seja, ainda era apenas uma abstração para o processamento gráfico.

O que enfim levou a criação de linguagens de mais alto nível como *OpenCL* e *CUDA*. Do ponto de vista do programador, funcionam como uma extensão para linguagens como *C*, *C++*, *Fortran* etc. Permitindo que quase não haja distinção entre programar para *GPU* ou *CPU*.

Antes de apresentar os detalhes das linguagens, é preciso destacar especificidades da programação para GPU que ainda não foram abstrataídas:

- O trecho de código executado na *GPU* é chamado de *kernel*. Cada instância deste é uma chamada de *thread*. Um conjunto de threads

pode ser agrupado no que a NVIDIA chama de *bloco*. E, por sua vez, estes podem ser agrupados no que a NVIDIA chama de *grade*;

- Os dados sobre os quais os algoritmos vão operar precisam ser transferidos da memória principal à memória dedicada da placa gráfica, através do barramento, e o resultado de volta. Esta transferência pode ser muito lenta, pois além do próprio barramento ser um gargalo, este ainda é compartilhado com todos os demais periféricos. Portanto, o primeiro objetivo é minimizar estas transferências a apenas o necessário e, quando elas forem necessárias, que sejam feitas em grandes quantidades de dados para usar toda a banda;

- Os diversos núcleos de processamento da unidade de processamento gráfico na verdade estão fisicamente agrupados no que é chamado de *stream multiprocessors* pela NVIDIA. Isto é importante, pois todas as *threads* de um bloco devem ser executadas no mesmo *stream multiprocessor*. Ou seja, para utilizar toda capacidade da placa gráfica, é preciso ter vários *blocos*;

- A placa gráfica possui diferentes níveis de memória. A memória local contém as variáveis locais da thread, a memória compartilhada é a de acesso mais rápido para leitura, por estar dentro de cada *stream multiprocessor*, e por fim a memória global. Para tornar o acesso à memória o menos custoso possível, é preciso fazer uso da memória compartilhada, embora o seu uso sem os devidos cuidados pode gerar erros de inconsistência semelhantes aos que podem ser vistos em um cache mal implementado. Ou seja, a escrita deve ser cuidadosa;

- É preciso levar em conta durante a programação os limites físicos da placa gráfica. O principal deles diz respeito à memória que possui espaço limitado em todos os níveis, desde a memória global até a quantidade de registradores que cada *thread* em um *bloco* pode utilizar (existe um limite de registradores por *bloco* na verdade).

2.4.1 Linguagem CUDA

Atualmente na versão 4.2, é classificada como uma arquitetura para computação de propósito geral em paralelo introduzida em 2006 pela NVIDIA. Do ponto de vista do programador é uma extensão disponível para as linguagens *C*, *C++* e *FORTRAN*. Os principais elementos do *CUDA* são a hierarquia de agrupamento das threads, os níveis de memória e as barreiras de sincronização.

A hierarquia de agrupamento consiste de dois níveis: grades (*grids*) e blocos (*blocks*). Um bloco agrupa muitas threads e uma grade, por sua vez, agrupa muitos blocos. Além de os blocos afetarem diretamente o escalonamento como descrito em 2.4, cada bloco possui um limite de threads que

pode conter e o mesmo para grades com relação a blocos (este limite depende do hardware). Outro uso interessante para este agrupamento, é organizar processamento sobre dados que possuem três dimensões.

O primeiro nível de memória é a memória local de cada thread privada, depois a memória compartilhada de cada bloco e, por fim, a memória global acessível por todas as threads. A memória global persiste até que seja desalocada pelo programa, enquanto as demais persistem apenas enquanto suas estruturas existem. As memórias local da thread e compartilhada do bloco são as mais rápidas ao custo de seu escopo limitado.

2.4.2 Linguagem OpenCL

Atualmente mantida pelo Khronos Group (6.3.3), um consórcio de várias empresas como AMD, Apple, NVIDIA e outras, é classificada como um padrão aberto multiplataforma de programação heterogênea (*GPU*, *CPU* e outros processadores) em paralelo.

Assim como o *CUDA*, do ponto de vista do programador, pode ser vista como uma extenxão da linguagem, mas apenas disponível para *C*. Sua hierarquia de agrupamento possui apenas quatro níveis: contexto (*context*); programa (*program*); *kernel*; e item de trabalho (*work item*).

Um contexto agrupa diversos dispositivos, suas memórias e as filas de comandos que estes recebem. Análogo a um bloco em CUDA, um programa agrupa diversos *kernels*, que representam o código de um item de trabalho (parecido com as threads em CUDA) que é recebido por uma das filas do contexto para ser executado.

O uso da memória dos disposistivos é abstraída por meio de buffers. Porém, no que diz respeito à hierarquia de memória, a arquitetura implementada pelo OpenCL é análoga a já descrita para a linguagem CUDA. A única diferença está na nomenclatura onde a memória local em OpenCL é equivalente à memória compartilhada em CUDA e a memória privada seria a memória local em CUDA.

2.5 Biblioteca VTK

Atualmente na versão 5.10, a *Visualization Toolkit* (ou *VTK*), é uma poderosa biblioteca de computação gráfica muito utilizada para aplicações científicas (dentre estas aplicações está o *MedSquare*[6.3.6]).

Implementada na linguagem *C++* utilizando orientação a objetos, ela fornece desde as classes capazes de representar trajetórias tridimensionais e campos vetoriais, até classes responsáveis pela interface gráfica com o usuário.

Porém, este poder vem ao custo do desenvolvedor ter de compreender sua arquitetura de *pipelines* e, ao menos, a interface das classes que utilizará.

O que não é simples quando a documentação online é escassa ou já ficou obsoleta e a bibliografia é de difícil acesso.

2.5.1 Arquitetura de *pipelines*

Um pipeline é um conjunto de elementos que ao atuarem em conjunto possibilitam desde a leitura dos dados de entrada até a exibição na tela da transformação feita sobre estes dados. Para isso sua arquitetura divide-se em quatro elementos chave representados pelas classes:

- *vtkInformation* é a classe responsável por armazenar os meta dados do pipeline e encapsular os dados para troca de informação entre as demais classes;
- *vtkDataObject* contém os dados sobre os quais o pipeline deve operar;
- *vtkAlgorithm* representa o algoritmo a ser executado pelo pipeline, como um filtro;
- e *vtkExecutive* representa a lógica necessária para que o pipeline seja utilizado pelo programa, isto é, como se conectar a ele e como executá-lo.

2.5.2 Classes

Para a implementação do método são utlizadas diversas classes da biblioteca que em conjunto abstraem o campo vetorial e os pontos iniciais, aplicam o método e abstraem os pontos que compõe a fibra.

Todos os dados de entrada do algoritmo são armazenados direta ou indiretamente (através de herança) por meio de um objeto da classe *vtkDataObject*. O campo vetorial é armazenado nesta classe, enquanto que a lista de pontos iniciais é armazenada por outro objeto da casse *vtkPolyData* que é subclasse de *vtkPoints* que, por sua vez, é subclasse de *vtkDataSet* que, enfim, é subclasse de *vtkDataObject*. Ou seja, esta é uma classe fundamental para o armazenamento de dados.

A aplicação do método é feita utilizando a classe *vtkStreamLine* que por meio de heranças em cadeia, indiretamente herda de *vtkAlgorithm*. Para operar, esta classe é composta pelos dados acima descritos, a determinação de um tamanho de passo e um integrador. Este integrador, no caso, é a *vtkRungeKutta2* ou a *vtkRungeKutta4*.

Por fim, o resultado do método é novamente obtido em um objeto da classe *vtkPolyData* para ser renderizado na tela.

Capítulo 3

Atividades Realizadas

3.1 Protótipo das implementações do método

A implementação do método foi toda feita utilizando orientação a objetos em *C++*, com excessão das funções específicas ao cálculo que possuem implementações, em *CUDA*, *OpenCL* e *C++*. Estas funções específicas ao cálculo consistem de operações básicas com vetores, aproximações para o campo vetorial e o método de Runge-Kutta em si. O código compartilhado entre as demais implementações vai desde as estruturas de dados até representações dos resultados em *OpenGL*.

O código pode ser encontrado em seu repositório *Git* (em https://github.com/rafamanzo/runge-kutta) é organizado basicamente em quatro pastas:

- **core** contém as três implementações do método nas pastas *c*, *cuda* e *opencl*; além das estruturas de dados que repensentam a entrada (*dataset.cpp*) e a saída do método (*fiber.cpp*).
- **example-factories** contém os scripts que geram arquivos de entrada para o protótipo de campos vetoriais sintéticos como exemplos;
- **includes** contém todos os cabeçalhos necessários, facilitando sua inclusão;
- **io** contém as classes que cuidam da entrada e saída do protótipo. A pasta *gui* contém as abstrações utilizadas para a criação de uma interface gráfica com *Glut* e *OpenGL*.

As interações entre todas as classes que compõe o protótipo podem ser vistas através do seguinte diagrama de classes:

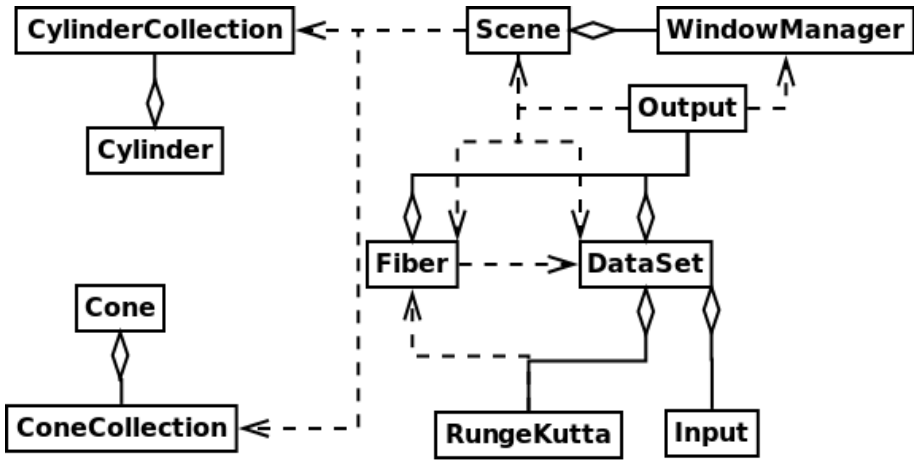

Figura 3.1: Diagrama de classes simplificado para o protótipo

3.1.1 Estruturas compartilhadas em C++

A estrutura mais básica é a *vector* contida em *include/dataset.h*, consistindo de três *doubles*. Sua responsabilidade é representar tanto vetores quanto pontos no $\Re^3$. Neste mesmo arquivo também está a representação da classe *DataSet*, cujo uso é representar as informações de entrada referentes ao campo vetorial e encapsular seu acesso. Analogamente, a classe *Fiber* é responsável por representar os resultados da aplicação do algoritmo.

Além destas abstrações de entrada e saída, existem classes que de fato são responsáveis pela entrada e saída do protótipo. A mais simples é a classe que cuida da entrada, *Input*, que possui os métodos para ler um arquivo de texto que contenha as informações sobre o campo vetorial, pontos iniciais e tamanho de passo (conforme descrito no arquivo *README* que acompanha o protótipo). Outra opção de entrada é o formato Analyze que é suportado pelo biblioteca *CImg* (6.3.6) e portanto o método da classe *Input* que processa este tipo de entrada apenas faz uma chamada a esta biblioteca.

Por fim, a última estrutura compartilhada entre as três implementações é a saída. A forma mais simples de visualizar os resultados é através do *gnuplot* (6.3.6) passando para este o arquivo *rk2-vs-rk4* que é criado ao fim da execução do algoritmo. Por outro lado, a saída pode ser bastante mais complexa que apenas uma classe, como é a entrada, devido à visualização com *Glut* e *OpenGL*.

Esta visualização é composta das classes, aqui denomidas como primitivas, que contém representações de cilindros e cones (classes *Cylinder* e *Cone*)

e as responsáveis por representar coleções destas classes (*CyllinderCollection* e *ConeCollection*). Estas classes primitivas contém principalmente informações sobre como renderizar estes objetos.

Por fim, ainda na interface gráfica, existe a classe *Scene* que é responsável por fornecer métodos que utilizam as primitivas para evitar que a classe *WindowManager* use diretamente as primitivas e tenha que conhecer suas especificidades. Ou seja, ela é como uma camada de abstração para a *WindowManager*, permitindo que esta seja responsável apenas pela interação com a *Glut*.

3.1.2 Implementação do método

A implementação em cada linguagem pode ser encontrada nas subpastas de *core* (*c*, *cuda* e *opencl*), nos arquivos *rk_kernel.** aos quais iremos nos referir apenas como *kernel*. Também cada uma destas pastas contém um arquivo *rk.cpp* reponsável por fornecerer uma interface para seu respectivo *kernel*.

Estas interfaces são utilizadas pois a implementação, por limitação do *CUDA* e do *OpenCL*, não pode ser feita utilizando orientação a objetos. Então a classe (*RungeKutta*) implementada no arquivo *rk.cpp* é a responsável por encapsular o conjunto de funções definidas em seu respectivo *kernel*.

Cada arquivo de *kernel*, além de conter a implementação do método, possui funções auxiliares para se trabalhar com elementos do $\Re^3$ (soma, subtração, módulo, distância e produto por um escalar) e as funções de aproximação necessárias (2.3.1).

Com toda esta estrutura descrita, o método em si é a simples implementação do que é descrito para as ordens 2 e 4 na seção 2.2 com especificidades para as diferentes linguagens utilizadas descritas a seguir.

Antes é preciso destacar que todas as implementações de diferentes ordens são funções independentes umas das outras e que, para evitar que falte memória, os resultados foram limitados a 10000 pontos que podem ser gerados a partir de cada ponto inicial.

Observações sobre o método em C++

Esta implementação, ao contrário das demais, foi feita sequencial e apenas não foi feita orientada a objetos para seguir a arquitetura necessária para as outras duas implementações.

Observações sobre o método em GPU

Nestas implementações em CUDA e OpenCL para *GPU*, além do limite de 10000 pontos que podem ser gerados por cada ponto, é feita uma rechecagem após alocar toda a memória necessária para o *DataSet* calculando novamente a quantidade máxima de pontos que podem ser gerados a partir de cada

ponto inicial baseado na memória dedicada que restou. Então, o limite é o mínimo entre 10000 e este valor calculado.

Ainda são checa as características GPU (quantidade de SMs) de forma a alocar toda sua capacidade de processamento sempre que possível. Similarmente é em CUDA checada a quantidade máxima de memória local que cada bloco pode ocupar para garantir que esta não seja excedida. Isto é algo com que o OpenCL já lida sem que o programador tenha que se preocupar.

3.1.3 Geração de campos vetoriais sintéticos

Campos vetoriais extensos ocupam muito espaço em disco, mesmo, compactados para serem distribuídos de forma prática junto do protótipo. Então, para torná-los disponíveis junto com o protótipo, foram escritos scripts *PHP* que os geram todos eles contidos dentro da pasta *example-factories*.

O campo mais simples gerado possui dimensões 128x128x20, representando uma rotação sobre o eixo z. Nele, o método é aplicado com tamanho de passo 0.1 e pontos iniciais: (0, 16, 10); (0, 32, 10); (0, 48, 10); (0, 64, 10); (0, 80, 10); (0, 96, 10); e (0, 112, 10). O segundo campo é uma inversão periódica no sentido da coordenada y, num campo de tamanho 512x512x128. A aplicação do método sobre ele se dá com tamanho de passo 0.2 e tem um único ponto inicial: (0, 64, 64).

Os dois campos anteriores são úteis para verificar se o método se comporta como esperado, mas são pouco úteis para medir a performance e não reproduzem uma variedade de casos. Portanto, os dois exemplos a seguir procuram suprir estas carências.

O terceiro exemplo é um campo com direções aleatórias com dimensão 256x256x256. Para este campo o método é aplicado com tamanho de passo 0.1 e também são criados 256 pontos iniciais com posições aleatórias.

Por fim, no último campo as direções possuem uma distribuição normal com média 10 e variância 1 e sua dimensão é 256x256x256. Ele possui um único ponto inicial aleatório e tamanho de passo 0.1.

3.2 Protótipo utilizando a biblioteca VTK

Foi utilizada a versão 5.8 da biblioteca *VTK* (6.3.5) novamente com *C++* para gerar um segundo protótipo equivalente ao anterior (3.1), capaz de carregar um campo vetorial no formato Analyze e aplicar os métodos de ordem 2 e 4 já implementados na *VTK* para *CPU*. Assim, permitindo o planejamento das abstrações necessárias para criar implementações em *GPU*.

Neste protótipo, muito do código gerado para o primeiro foi reaproveitado. Seu código fonte pode ser encontrado, assim como o primeiro, em seu repositório Git (https://github.com/rafamanzo/runge-kutta-vtk).

3.2.1 Estrutura

A estrutura deste protótipo pode ser entendida como uma simplificação da primeira, contando com apenas 4 classes utilizadas para interagir com as diversas classes da *VTK*:

- **AnalyzeReader** é responsável por utilizar a bibliteca *CImg* (também utilizada no primeiro) para ler o arquivo no formato Analyze fornecido na entrada e com isto criar uma instância da classe *vtkImageData*, com a qual o *VTK* poderá trabalhar;
- **Input** recebe os argumentos de entrada e os processa, carregando o campo vetorial inclusive;
- **Tracer** realiza a aplicação do método;
- **Renderer** recebe o resultado da aplicação do método e o exibe na tela já com uma interface que permite escala e rotações.

As interações entre estas classes e destas com a *VTK* são ilustradas pelo diagrama de classes simplificado:

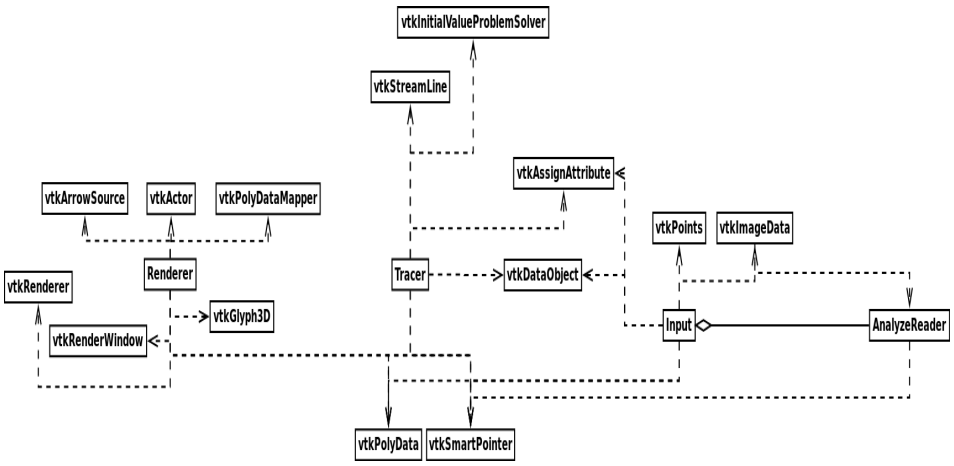

Figura 3.2: Diagrama de classes simplificado para o protótipo com *VTK*

É possível notar que o número de classes necessárias aumentou com relação ao protótipo anterior (3.1). Porém, boa parte das classes envolvidas neste protótipo são da *VTK* e apenas as quatro descritas anteriormente tiveram de ser realmente escritas. E mesmo entre estas quatro classes não existe dependência além de uma composição entre a AnalyzeReader e Input.

Isto demonstra que, apesar de utilizar a *VTK* envolver um número maior de classes sobre as quais o desenvolvedor deve ter conhecimento, ela proporciona um código muito mais enxuto e com baixa dependência entre as classes, representando um protótipo superior em qualidade de código.

3.2.2 Planejamento abstrações da VTK para uso de CUDA e OpenCL

Basicamente, todo processamento consiste de transferência dos dados dados de entrada à memória, processamento destes dados gerando outros dados de saída e, por fim, a transferência destes dados de saída para a memória.

No caso, o objetivo é possibilitar a aplicação do método em *GPU* através da *VTK*. Para tanto, novas implementações das classes da *VTK* que estejam envolvidas nestas três partes do processamento do método descritas anteriormente. Mais especificamente as classes das quais a classe *Tracer* depende.

Nesta classe a entrada é recebida em objetos de duas classes da *VTK*: *vtkDataObject* (campo vetorial); e *vtkPolyData* (pontos iniciais). Então, o método utilizado para integração é uma subclasse de *vtkInitialProblemSolver* para, por fim a classe *vtkStreamLine* realizar o processamento e devolver a saída em um *vtkPolyData*. Sendo estas as classes que devem ser abstraídas para *GPU*.

3.3 Comparação de perfomance entre CPU e GPU

Foram feitas comparações entre as implementações do método para *CPU* utilizando *POSIX threads* com o método para *GPU* em *CUDA*. Estes resultados foram obtidos em duas *CPUs* Intel e *GPUs* NVIDIA diferentes a fim de comprovar o que foi afirmado anteriormente sobre os desempenhos do método em cada um destes ambientes.

A elaboração detes testes contou com os conselhos do mestrando Paulo Carlos Santos no IME-USP e consulta ao conteúdo de sua qualificação: *Ferramentas para Extração de Informações de Desempenho em GPUs NVIDIA*.

3.3.1 Metodologia

Dados de entrada

Para isto foi gerado um campo vetorial sintético com dimensões de 512 pontos no eixo x, 256 pontos no eixo y e outros 256 pontos no eixo z. Todos com o mesmo vetor direção $(0, 1, 0)$ e até 1024 pontos iniciais da forma $(x, 0, z)$, onde $0 \leq x \leq 512$ e $z = 128$ ou $z = 129$.

Os testes foram realizados para os primeiros 16, 32, 64, 128 e 256 pontos iniciais a fim de se obter como que o método escala em cada um dos ambientes.

Todo o código utilizado nos testes pode ser encontrado em no repositório git: https://github.com/rafamanzo/runge-kutta-benchmark

Medições de tempo

A medição de tempo é dividida em tempo consumido por operações em memória (alocação e transferência) e processamento do método (operações de ponto flutuante). Sempre realizando a contagem de *clocks* em cada um dos ambientes.

Casos

Cada medição consistiu em executar trinta vezes cada implementação para estes dados (restringindo a quantidade de pontos iniciais conforme descrito). Então, obtendo a média e desvio padrão do tempo em segundos consumidos em operações de transferência de dados, alocação de memória e processamento dos pontos que compõe a trajetória, bem como um histograma.

3.3.2 Resultados em CPU

Foram realizados os testes em duas *CPUs* Intel diferentes:

- Core 2 Duo E7400, com dois núcleos físicos e lógicos a 2.8GHz cada;
- Core i7-2620M, com quatro núcleos físicos e lógicos a 2.7GHz cada.

O primeiro utiliza dois módulos de memória DDR2 em Dual Channel, enquanto que o segundo utiliza dois módulos de memória DDR2 em Dual Channel.

Runge-Kutta Ordem 2

Média dos tempos de execução para processamento em segundos:

Processadores	Quantidade de pontos iniciais				
	16	32	64	128	256
Core 2 Duo	1.182000	4.101000	17.046000	67.923333	317.030333
Core i7	1.693000	7.443333	34.907333	143.061333	601.070000

Desvio padrão dos tempo de execução para processamento em segundos:

Processadores	Quantidade de pontos iniciais				
	16	32	64	128	256
Core 2 Duo	0.044377	0.143860	0.268398	1.632077	10.779398
Core i7	0.095469	0.612320	0.856380	5.185798	9.779571

Média dos tempos de execução para operações em memória em segundos:

Processadores	Quantidade de pontos iniciais				
	16	32	64	128	256
Core 2 Duo	1.034667	1.037000	1.046333	1.045667	1.048667
Core i7	0.946000	1.071333	1.069667	1.049667	1.048000

Desvio padrão dos tempo de execução para operações em memória em segundos:

Processadores	Quantidade de pontos iniciais				
	16	32	64	128	256
Core 2 Duo	0.007180	0.009000	0.010796	0.007608	0.008844
Core i7	0.042079	0.015217	0.009123	0.026139	0.006000

Runge-Kutta Ordem 4

Média dos tempos de execução para processamento em segundos:

Processadores	Quantidade de pontos iniciais				
	16	32	64	128	256
Core 2 Duo	1.271000	4.013000	17.485000	68.380000	318.383000
Core i7	1.795333	7.721000	35.378667	144.985667	604.026667

Desvio padrão dos tempo de execução para processamento em segundos:

Processadores	Quantidade de pontos iniciais				
	16	32	64	128	256
Core 2 Duo	0.057175	0.469866	0.378724	1.335073	10.443222
Core i7	0.138558	0.460647	0.445853	1.301178	11.964844

Média dos tempos de execução para operações em memória em segundos:

Processadores	Quantidade de pontos iniciais				
	16	32	64	128	256
Core 2 Duo	1.034333	1.060333	1.045333	1.048667	1.048333
Core i7	0.982000	1.068667	1.067333	1.051333	1.050667

Desvio padrão dos tempo de execução para operações em memória em segundos:

Processadores	Quantidade de pontos iniciais				
	16	32	64	128	256
Core 2 Duo	0.006155	0.048682	0.011175	0.013350	0.007341
Core i7	0.064570	0.010242	0.007717	0.010242	0.014591

3.3.3 Resultados em GPU

Foram realizados os testes em duas GPUs NVIDIA GeForce diferentes:

- GTS 450, com 192 CUDA Cores a 1566MHz cada e 1GB de memória GDDR5 à 1804MHz;

- GT 520M, com 48 CUDA Cores a 1480Mhz cada e 1GB de memória DDR3 à 800MHz.

De acordo com o fabricante, o primeiro possui performance média para transferência de dados para memória de 57.7GB/s, enquanto que a segunda possui 12.8GB/s.

Runge-Kutta Ordem 2

Média dos tempos de execução para processamento em segundos:

Processadores	Quantidade de pontos iniciais				
	16	32	64	128	256
GTS 450	0.049661	0.050175	0.057119	0.057070	0.057252
GT 520M	0.054739	0.054733	0.054850	0.061936	0.084663

Desvio padrão dos tempo de execução para processamento em segundos:

Processadores	Quantidade de pontos iniciais				
	16	32	64	128	256
GTS 450	0.002440	0.000243	0.000313	0.000089	0.000220
GT 520M	0.000033	0.000006	0.000019	0.000264	0.000120

Média dos tempos de execução para operações em memória em segundos:

Processadores	Quantidade de pontos iniciais				
	16	32	64	128	256
GTS 450	1.352077	1.488127	1.776701	2.388835	3.597740
GT 520M	1.402293	1.489929	1.680435	2.074177	2.836108

Desvio padrão dos tempo de execução para operações em memória em segundos:

Processadores	Quantidade de pontos iniciais				
	16	32	64	128	256
GTS 450	0.014127	0.010570	0.013345	0.014769	0.011500
GT 520M	0.010907	0.010067	0.011193	0.007211	0.012150

Runge-Kutta Ordem 4

Média dos tempos de execução para processamento em segundos:

Processadores	Quantidade de pontos iniciais				
	16	32	64	128	256
GTS 450	0.089737	0.089613	0.102665	0.102715	0.102927
GT 520M	0.098342	0.098343	0.098618	0.116622	0.164528

Desvio padrão dos tempo de execução para processamento em segundos:

Processadores	Quantidade de pontos iniciais				
	16	32	64	128	256
GTS 450	0.000419	0.000419	0.000263	0.000333	0.000305
GT 520M	0.000006	0.000008	0.000037	0.000137	0.000385

Média dos tempos de execução para operações em memória em segundos:

Processadores	Quantidade de pontos iniciais				
	16	32	64	128	256
GTS 450	1.355632	1.490352	1.773907	2.392552	3.599853
GT 520M	1.404058	1.492534	1.675477	2.077923	2.828944

Desvio padrão dos tempo de execução para operações em memória em segundos:

Processadores	Quantidade de pontos iniciais				
	16	32	64	128	256
GTS 450	0.015841	0.010029	0.010473	0.011441	0.015122
GT 520M	0.008800	0.011051	0.009609	0.007901	0.008453

3.3.4 Distribuições

Com desvio padrão baixo como o encontrontado, em todos os histogramas a concentração se dá toda sobre a média. Esta característica observada nos desvios padrões também são um forte indicativo de que a média é uma boa estatística para ser utilizada.

3.3.5 Conclusão

Os tempos de execução com crescimento exponencial do método em *CPU* refletem claramente o custo de troca de contexto neste ambiente. Ou seja, a *CPU* se adapataria melhor a um algoritmo diferente com número fixo de threads igual ao de núcleos físicos do processador. Também no que diz respeito à *CPU* é evidente o poder de um barramento dedicado para comunicação, onde o tempo gasto em operações na memória praticamente não variou apesar do crescimento exponencial do problema.

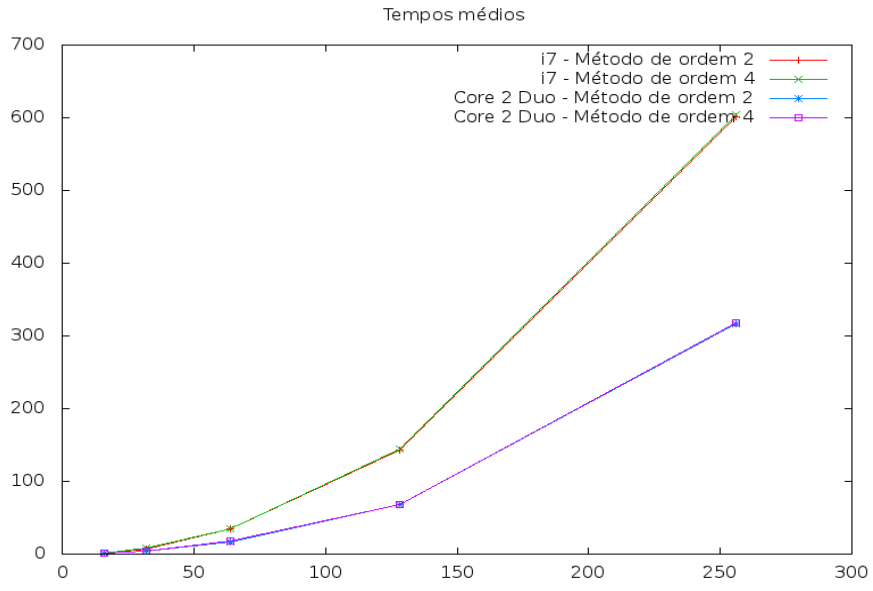

Figura 3.3: Curioso notar que o processador Core 2 Duo bateu em duas vezes um processador Core i7, que é duas gerações posterior ao primeiro.

Este gráfico ainda nos permite notar que em *CPU* não há diferença prática de tempo entre as ordens apesar de uma realizar o dobro de operações de ponto flutuante. Isto reforça a tese de que o fator limitante para esta implementação em *CPU* é o alto custo da troca de contexto.

Por outro lado, o método em *GPU* obteve tempos de processamento muito inferiores aos de ambas as *CPUs*, em alguns casos chegando à ordem de 10^{-4}. Porém, a falta de um barramento dedicado causa altos tempos para as operações em memória.

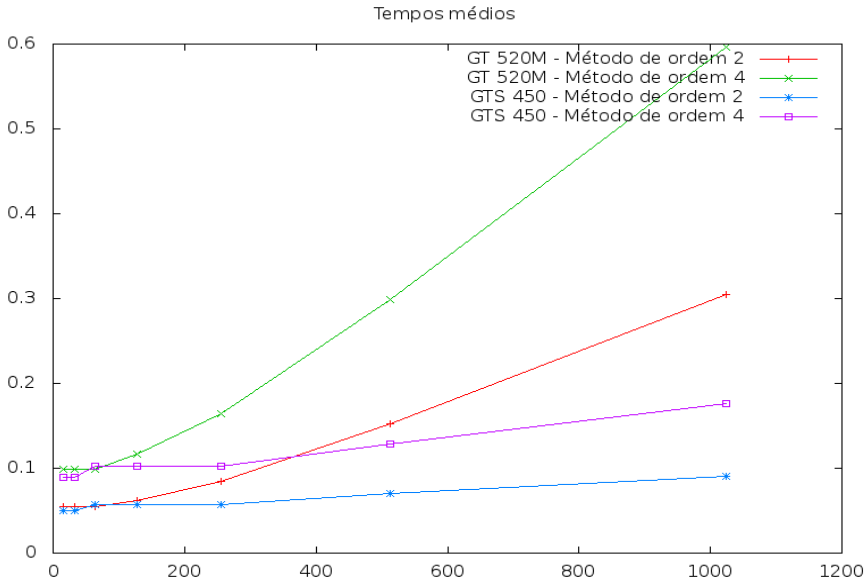

Figura 3.4: A GTS450 começa a apresentar ganho real de performance a partir dos 256 pontos iniciais, como decorrência do seu maior número de *stream multi-proccessors*

Este gráfico ilustra um comportmento esperado para *GPU*, onde no início para um número pequeno de pontos observamos tempo constante, por estar aquém da capacidade do hardware. Então, conforme o problema continua crescendo exponencialmente, o tempo médio começa a crescer, mas linearmente.

O comportamento linear descrito também pode ser observado no que diz respeito às operações de memória.

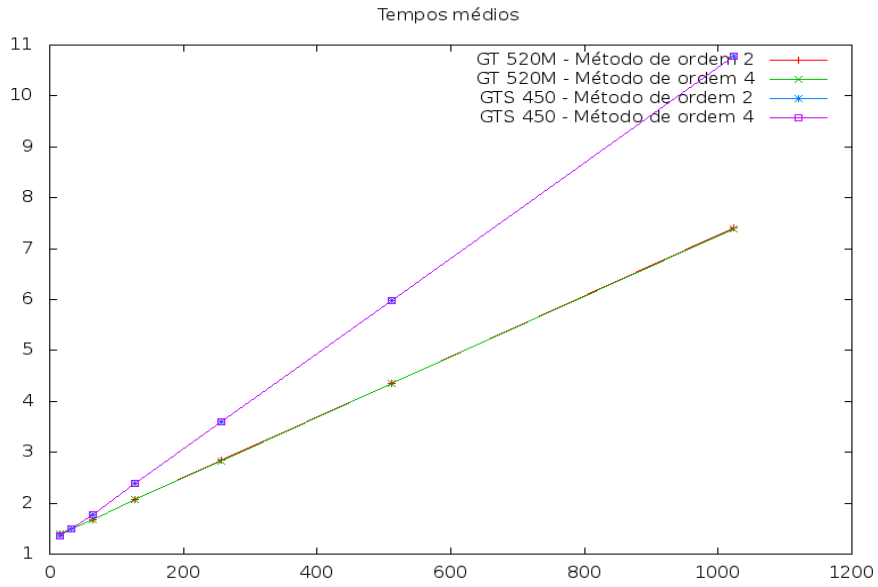

Figura 3.5: Apesar do tamanho do problema crescer exponencialmente, o tempo gasto para operações de memória cresce de forma linear

Com estes dados em mãos fica claro que o método em *GPU* não é apenas melhor em comparação com seu equivalente em *CPU*, mas é muito melhor em qualquer caso para uma quantidade de pontos não tão grande (apartir de 64).

Mesmo com um resultado bastante satisfatório, ainda assim, é possível melhorá-lo abrindo mão de precisão e usando a memória *RAM* em conjunto com a da placa gráfica. Para isto, foram feitas novas melhorias e testes com base no método de ordem 4 para a placa GTS 450.

3.3.6 Utilização de precisão simples ao invés de dupla

Média dos tempos de execução para processamento em segundos:

Precisão	Quantidade de pontos iniciais						
	16	32	64	128	256	512	1024
Dupla	0.089737	0.089613	0.102665	0.102715	0.102927	0.128983	0.176045
Simples	0.074125	0.074505	0.074600	0.091314	0.091356	0.092422	0.094418

Desvio padrão dos tempo de execução para processamento em segundos:

Precisão	Quantidade de pontos iniciais						
	16	32	64	128	256	512	1024
Dupla	0.000419	0.000419	0.000263	0.000333	0.000305	0.000339	0.000384
Simples	0.000283	0.000272	0.000272	0.000208	0.000280	0.000336	0.000111

Média dos tempos de execução para operações em memória em segundos:

Precisão	Quantidade de pontos iniciais						
	16	32	64	128	256	512	1024
Dupla	1.355632	1.490352	1.773907	2.392552	3.599853	5.984084	10.783370
Simples	1.912072	2.047864	2.327917	2.984783	4.182320	6.618064	11.512310

Desvio padrão dos tempo de execução para operações em memória em segundos:

Precisão	Quantidade de pontos iniciais						
	16	32	64	128	256	512	1024
Dupla	0.015841	0.010029	0.010473	0.011441	0.015122	0.016412	0.021922
Simples	0.010561	0.010227	0.008126	0.016713	0.022187	0.030160	0.081429

Conclusão

Com a precisão simples houve ligeiro ganho no tempo de processamento, mas houve uma ligeira perda no tempo de operações na memória que é a grande limitação desta implementação.

3.3.7 Utilização de memória presa

Média dos tempos de execução para processamento em segundos:

Memória	Quantidade de pontos iniciais						
	16	32	64	128	256	512	1024
Global	0.074125	0.074505	0.074600	0.091314	0.091356	0.092422	0.094418
Presa	0.074107	0.074444	0.090507	0.107728	0.107685	0.108624	0.110673

Desvio padrão dos tempo de execução para processamento em segundos:

Memória	Quantidade de pontos iniciais						
	16	32	64	128	256	512	1024
Global	0.000283	0.000272	0.000272	0.000208	0.000280	0.000336	0.000111
Presa	0.000230	0.000118	0.000211	0.000266	0.000235	0.000231	0.000273

Média dos tempos de execução para operações em memória em segundos:

Memória	Quantidade de pontos iniciais						
	16	32	64	128	256	512	1024
Global	1.912072	2.047864	2.327917	2.984783	4.182320	6.618064	11.512310
Presa	1.925960	2.070551	2.380262	3.026972	4.279094	6.787730	11.785125

Desvio padrão dos tempo de execução para operações em memória em segundos:

Memória	Quantidade de pontos iniciais						
	16	32	64	128	256	512	1024
Gloval	0.010561	0.010227	0.008126	0.016713	0.022187	0.030160	0.081429
Presa	0.010899	0.026004	0.016693	0.013016	0.026585	0.024440	0.036797

Conclusão

Utilizando a memória presa não houve ganho de performance nos tempos (inclusive houve ligeira perda de performance), porém ainda é um recurso bastante interessante. Isto pois o uso deste recurso faz que ao invés de ser utilizada a memória da placa gráfica seja utilizada a memória principal da máquina. Visto que a memória da placa gráfica costuma ter menos espaço disponível que a memória principal, isto permite o processamento de problemas com conjuntos de dados ainda maiores.

3.3.8 Avaliação da melhor combinação

O primeiro conjunto de testes em *GPU* nos permite concluir que, de fato, o método de ordem 2 consome metade do tempo de processamento que o de ordem 4 ao custo de perda de precisão. Porém isto pode ser compensado, quando necessário, aumentando o tamanho do passo do método.

Analisando as melhorias, apesar de não terem implicado em mais desempenho, o uso de precisão simples e memória presa trouxeram ganhos do ponto de vista do espaço em memória necessário para o processamento do problema, permitindo o processamento de problemas ainda maiores.

O uso de precisão simples reduziu pela metade a quantidade de memória necessária para a resolução do problema, mas, também, ao custo de se perder precisão sobre a resposta. No entanto esta perda será relevante apenas para campos vetoriais bastante complexos e, mesmo neste caso, pode ser compensada quando é diminuído o tamanho do passo do algoritmo novamente.

A memória presa, por sua vez, não reduz o espaço necessário. Ela realiza a alocação na memória principal e deixa esta disponível para a placa gráfica. Com esta mudança no local dos dados o tamanho do campo vetorial suportado aumenta da mesma forma que a quantidade máxima de pontos gerados por cada ponto inicial.

Capítulo 4

Resultados e produtos obtidos

4.1 Implementações dos métodos

4.1.1 Protótipo básico

O protótipo básico foi uma prova de conceito bem sucedida para a implementação do método implementação em si e suas limitações, bem como para o uso das linguagens CUDA e OpenCL para este fim.

Sua compilação é bastante simples através do comando *make*. Sem nenhum argumento ele compilará a versão em *C++* por padrão. Com os argumentos *cuda* ou *opencl* serão compiladas suas respectivas versões. Abaixo está o processo de compilação

```
$ make cuda
g++ -c -Wall -pedantic -Wextra  -Iinclude main.cpp
g++ -c -Wall -pedantic -Wextra  -Iinclude io/input.cpp
g++ -c -Wall -pedantic -Wextra  -Iinclude core/dataset.cpp
g++ -c -Wall -pedantic -Wextra  -Iinclude core/fiber.cpp
g++ -c -Wall -pedantic -Wextra  -Iinclude io/output.cpp
g++ -c -Wall -pedantic -Wextra  -Iinclude io/gui/primitives/cylinder.cpp
g++ -c -Wall -pedantic -Iinclude io/gui/window_manager.cpp
g++ -c -Wall -pedantic -Wextra  -Iinclude io/gui/scene.cpp
g++ -c -Wall -pedantic -Wextra  -Iinclude io/gui/primitives/cylinder_collection.cpp
g++ -c -Wall -pedantic -Wextra  -Iinclude io/gui/primitives/cone.cpp
g++ -c -Wall -pedantic -Wextra  -Iinclude io/gui/primitives/cone_collection.cpp
nvcc -c -Iinclude core/cuda/rk.cpp -o rk_cuda.o -arch sm_20
nvcc -c -Iinclude core/cuda/rk_kernel.cu -o rk_cuda_kernel.o -arch sm_20
nvcc main.o input.o dataset.o fiber.o output.o cylinder.o window_manager.o scene.o
  cylinder_collection.o cone.o cone_collection.o rk_cuda.o rk_cuda_kernel.o -o rk
  -arch sm_20 -lglut -lGL -lGLU -lm -lpthread -lX11
```

Sua interface é bastante básica, mas permite rotacionar o campo e a fibra, transladá-lo, aproximá-lo ou afasta-lo tudo através de teclas. Além disso, é possível escolher quais informações são exibidas. Ou seja, é possível escolher entre exibir ou ocultar o campo vetorial, as fibras resultantes do RK2 ou as fibras resultantes do RK4.

A única limitação encontrada para esta implementação é lidar com a inderteminação que existe no campo vetorial na região de intersecção de duas fibras. Esta indeterminação faz com que o método possa seguir qualquer uma das fibras na intersecção quando chega a esta região.

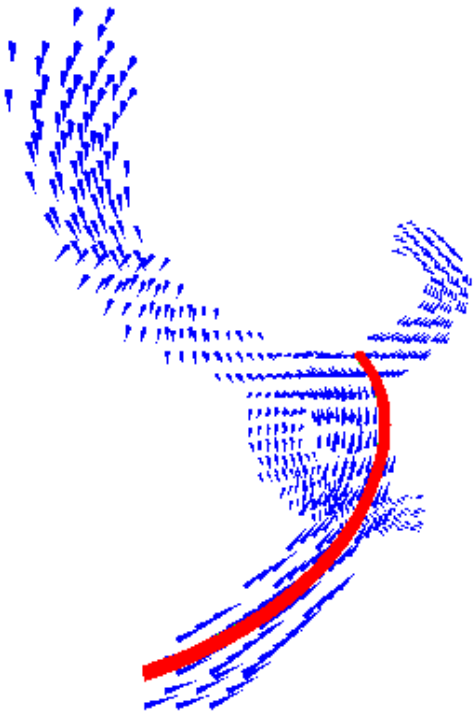

Figura 4.1: Campo 32x32x32 com duas hélices. Ao chegar na intersecção, a fibra se desvia para fora do campo e o método entende que terminou de propagá-la devido à baixa intensidade dos vetores.

Indo além, os testes deste protótipo com ressonâncias reais gerou o mesmo resultado do software de exploração de imagens médicas BioImage Suite (6.3.6), como podemos observar a seguir.

Como podemos observar, o resultado (em vermelho) obtido por ambos é o mesmo, confirmando a corretude do protótipo gerado comparando-o a um software já consolidado.

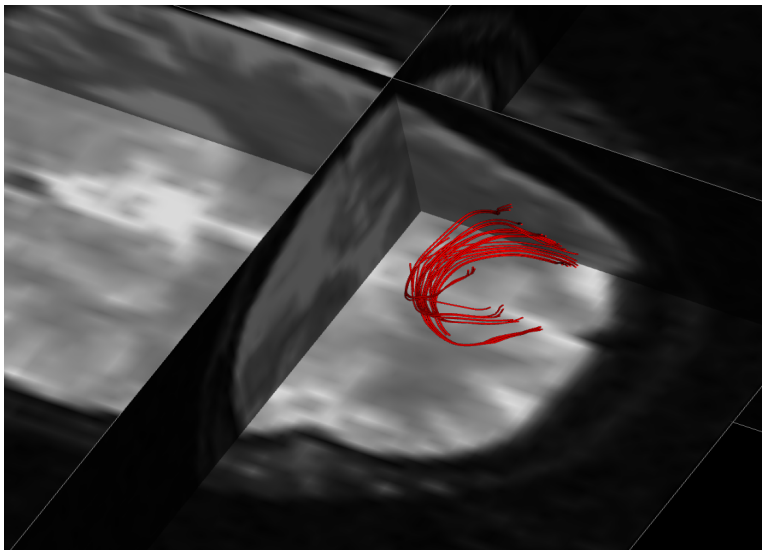

Figura 4.2: Resultado da tractografia realizada pelo BioImage Suite em uma ressonância magnética.

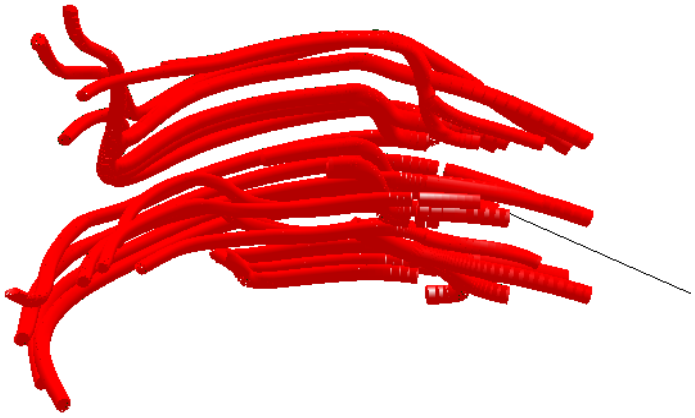

Figura 4.3: Resultado da tractografia realizada pelo pelo protótipo na mesma ressonância magnética da figura 4.2.

4.1.2 Protótipo utilizando a VTK

Este protótipo utilizou a versão 5.6 da biblioteca *VTK* para servir como uma prova de que é possível implementar o método dentro da biblioteca,

posteriormente sendo útil para o planejamento do que deve ser feito para extende-lo em *GPU* no futuro.

A única dependência que existe, é a instalação da biblioteca *VTK* 5.6. Com isto, sua compilação é ainda mais simples que o protótipo anterior utilizando software de compilação multiplataforma *CMake*. Os resultados de uma compilação bem sucedida podem ser vistos a seguir:

```
$ cmake .
-- The C compiler identification is GNU 4.7.2
-- The CXX compiler identification is GNU 4.7.2
-- Check for working C compiler: /usr/bin/gcc
-- Check for working C compiler: /usr/bin/gcc -- works
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Check for working CXX compiler: /usr/bin/c++
-- Check for working CXX compiler: /usr/bin/c++ -- works
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Configuring done
-- Generating done
-- Build files have been written to: ./runge-kutta-vtk
$ make
Scanning dependencies of target RungeKutta
[ 20%] Building CXX object CMakeFiles/RungeKutta.dir/Main.cpp.o
[ 40%] Building CXX object CMakeFiles/RungeKutta.dir/io/input/Input.cpp.o
[ 60%] Building CXX object CMakeFiles/RungeKutta.dir/io/input/AnalyzeReader.cpp.o
[ 80%] Building CXX object CMakeFiles/RungeKutta.dir/core/cpp/Tracer.cpp.o
[100%] Building CXX object CMakeFiles/RungeKutta.dir/io/output/Renderer.cpp.o
Linking CXX executable RungeKutta
[100%] Built target RungeKutta
```

Sua interface possui as mesmas funcionalidades que a do protótipo anterior, permitindo rotação, translação e escala. Da mesma forma este protótipo também não é capaz de lidar com indeterminações em um campo vetorial.

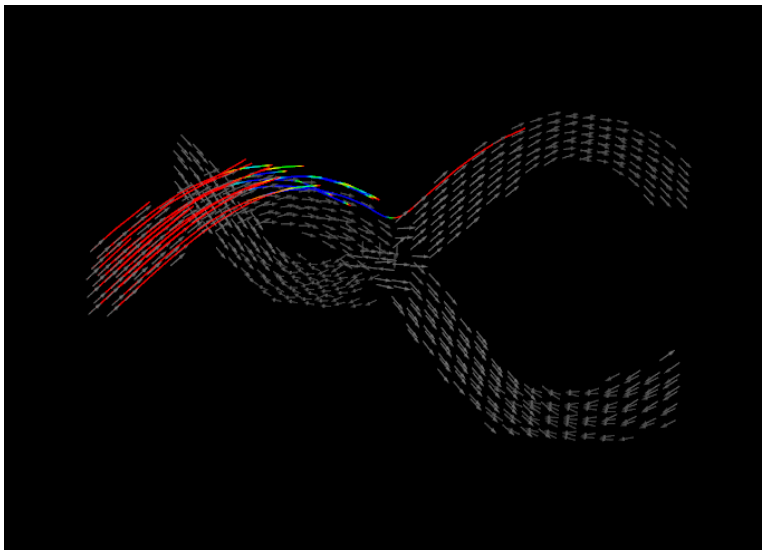

Figura 4.4: Campo 32x32x32 com duas hélices. Ao chegar na intersecção, a fibra se desvia para um fora do campo e o método entende que terminou de propaga-la devido à baixa intensidade dos vetores ou se desvia para outraq hélice devido à indeterminação do campo na intersecção.

Capítulo 5

Conclusão

Quando compreendemos um campo vetorial como a discretização de um sistema de equações diferenciais ordinárias, por meio de interpolações trilineares, é possível adaptar o método de Runge-Kutta para reconstruir trajetórias dentro deste campo a partir de um conjunto de pontos iniciais obtendo resultados visuais dentro do esperado, como foi demonstrado no primeiro protótipo.

Quando a quantidade de pontos iniciais é grande, o tempo de resposta para solucionar o problema também passa a ser grande ao ponto de não ser mais possível realizar o processamento em tempo real na *CPU*. Indo além, podemos entender que os vetores do campo vetorial em questão fornecem apenas a direção e não o sentido, o que ainda duplica a quantidade de pontos iniciais.

Porém, visto que cada ponto e cada direção deste são instâncias independentes do problema, este tempo de processamento pode ser reduzido drasticamente em um hadware com alto paralelismo como uma *GPU*, que foi comprovadamente, capaz de gerar um resultado visual idêntico ao gerado em *CPU* e num tempo que pode chegar a ser até 3000 vezes menor (600s contra 0.2s para 256 pontos iniciais), como observamos nos testes de performance realizados.

Comprovados a viabilidade da implementação em *GPU* e seu ganho de desempenho para este método, foi elaborado um segundo protótipo utilizando a biblioteca *VTK* com as mesmas funcionalidades que o primeiro. Este é útil como base para o planejamento de quais classes desta biblioteca devem ser abstraídas para o uso de *GPU*.

Portanto, a adapatação do método de Runge-Kutta para reconstrução de trajetórias tridimensionais a partir de campos vetoriais pode ser processada em *GPU* com resultado visual idêntico ao obtido pelo processamento em *CPU*, porém com tempo de resposta muito menor.

Capítulo 6

Referências Bibliográficas

6.1 Livros

- PRESS, William H. et al. *Numerical Recipes in C*
- STOER, Josef; BULIRSCH, Roland. *Introduction to Numerical Analysis*

6.2 Artigos

- Mori, S. and van Zijl, P. C. M. (2002), Fiber tracking: principles and strategies – a technical review. NMR Biomed., 15: 468–480. doi: 10.1002/nbm.781;

6.3 Websites

6.3.1 Método de Integração Numérica de Runge-Kutta

- *Método de Euler - http://math.fullerton.edu/mathews/n2003/Euler'sMethodMod.html* (acessado em 10/02/2012);
- *Numerical Recipes In C* - http://apps.nrbook.com/c/index.html (acessado em 10/02/2012);
- Solucionador de EDOs em OpenCL - http://www.cmsoft.com.br/index.php?option=com_content&view=category&layout=blog&id=108&Itemid=162 (acessado em 26/02/2012).

6.3.2 CUDA

- Referência - http://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/CUDA_C_Programming_Guide.pdf (acessado em 25/02/2012)

- Boas práticas - http://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/CUDA_C_Best_Practices_Guide.pdf (acessado em 25/02/2012)

6.3.3 OpenCL

- http://www.khronos.org/assets/uploads/developers/library/overview/opencl-overview.pdf (acessado em 26/02/2012);
- http://developer.amd.com/zones/OpenCLZone/programming/Pages/default.aspx (acessado em 26/02/2012);
- http://developer.download.nvidia.com/compute/cuda/3_2_prod/toolkit/docs/OpenCL_Programming_Guide.pdf (acessado em 26/02/12);
- http://developer.download.nvidia.com/compute/cuda/3_2_prod/toolkit/docs/OpenCL_Best_Practices_Guide.pdf (acessado em 26/02/12);
- http://www.khronos.org/opencl/ (acessado em 26/02/2012).

6.3.4 OpenGL

- Conectar dois pontos com um cilindro - https://github.com/curran/renderCyliner (acessado em 05/07/2012).

6.3.5 VTK

- Documentação oficial para a versão utilizada - http://www.vtk.org/doc/release/5.8/html/ (acessado em 22/10/2012);
- Documentação oficial sobre pipelines - http://www.vtk.org/Wiki/VTK/Tutorials/New_Pipeline (acessado em 22/10/2012).
- Descrição e exemplos de pipelines - http://www.cmake.org/cgi-bin/viewcvs.cgi/*checkout*/Utilities/Upgrading/TheNewVTKPipeline.pdf?revision=1.5 (acessado em 23/10/2012).

6.3.6 Outros

- MedSquare - http://ccsl.ime.usp.br/medsquare/ (Visitado em 07/08/2012)
- Catálogo NVIDIA - http://www.nvidia.com.br/object/geforce-gtx-690-br.html (Visitado em 07/08/2012)
- LAPIX - http://www.lapix.ufsc.br/tractografia-em-tempo-real?lang=pt (Visitado em 07/08/2012)

- CLRungeKutta46 - http://www.cmsoft.com.br/ index.php?option=com_content&view=category &layout=blog&id=108&Itemid=162 (Visitado em 07/08/2012)
- gnuplot - http://www.gnuplot.info/ (Visitado em 17/08/2012)
- CImg - http://cimg.sourceforge.net/ (Visitado em 17/08/2012)
- BioImage Suite - http://bioimagesuite.yale.edu/index.aspx

Parte II

Subjetiva

Capítulo 7

Giancarlo Rigo

Capítulo 8

Rafael Reggiani Manzo

Desenvolver este trabalho de conclusão de concurso foi fundamental para complementar minha formação como cientista da computação. Esta foi a primeira oportunidade de desenvolver um projeto durante todo um ano, com uma preocupação real com sua qualidade e interagindo com um docente frequentemente. Algo oposto aos trabalhos que usualmente são exigidos nas disciplinas onde eu trabalho neles durante no máximo um mês, com a preocupação de que o mínimo necessário esteja funcionando para o momento da correção e praticamente sem interagir com o professor.

Para alcançar este estado do trabalho, diversas disciplinas foram importantes e curiosamente duas delas são o Cálculo IV e a Álgebra Linear, que no momento quando cursei estas disciplinas jamais pensaria em utilizá-las. Estas duas disciplinas são toda a base matemática que foi necessária para ser possível reconstruir as trajetórias.

Igualmente importantes foram disciplinas básicas de computação como Princípios de Desenvolvimento de Algoritmos, onde pude ter o primeiro contato com conceitos como complexidade de algoritmos, estruturas de dados e até programação dinâmica.

Da mesma forma, na parte final do curso a disciplina de Computação Gráfica foi fundamental para tornar possível elaborar os protótipos que são o maior desta monografia.

Por fim, todas as disciplinas merecem o mérito por mostrarem, talvez não da melhor forma, como é possível aprender por conta própria o que é preciso para alcançar seu objetivo.

Parte III

Apêndices

Apêndice A

PME2603 - Tec. e Desenv. Social II

A.1 Introdução

A disciplina de Tecnologia e Desenvolvimento Social II (PME2603) é oferecida pelo departamento de engenharia mecânica da Escola Politécnica como uma disciplina optativa livre para toda a USP. Os docentes responsáveis por ela, Antonio Luis de Campos Mariani e Douglas Lauria fazem parte do projeto PoliCidadã (A.5).

Seu objetivo é desenvolver projetos de tecnologia com cunho social. Ou seja, através da aplicação de tecnologia, antender demandas de membros da sociedade que não são o foco das empresas que desenvolvem tecnologia, por não darem lucro a estas.

Neste contexto, os docentes nos chamaram a atenção para o viés social que esta monografia tem ao viabilizar que softwares livres para exploração de imagens médicas agreguem a funcionalidade de tractografia em tempo real disponível apenas em softwares proprietários e com custo elevado.

A.2 Objetivos

Assim, com esta possibilidade em vista foram levantados dados sobre os custos dos softwares que realizam a exploração de imagens médicas e qual seria o impacto para a sociedade da disponibilização gratuita desta tecnologia.

Além disto, para permitir uma maior compreensão do projeto, foram abordados temas específicos de computação que durante a monografia são assumidos de conhecimento do leitor mas que, neste novo contexto, não são populares mas são fundamentais para a compreensão da monografia.

A.3 Conceitos

A.3.1 Custos de software

Os softwares para exploração de imagens médicas, além da tractografia, oferecem diversos outros recursos para exploração de imagens. Desta forma, os custos aqui expostos dizem respeito à todo o software e não só apenas à funcionalidade.

No Brasil poucas empresas trabalham neste ramo devido à escassez de pessoas capacitadas e também à popularização da ressonância magnética que começa a acontecer apenas agora.

Uma destas empresas é a Artis (A.5), com sedes em São Paulo e Brasília, com a qual conseguimos entrar em contato e levantar um custo total de software mais hardware de R$ 300.000 para a solução deles nacional e outra, de seu concorrente mais forte importada do Canada, com custo aproximado de R$ 1.000.000.

O hardware consiste de um computador, um conjunto de câmeras e guias cirúrgicas. Estimando seus preços, o computador fica em torno de R$ 2000, o conjunto de câmeras mais R$ 2.000, e as guias cirúrgicas por volta de R$ 50.000, totalizando R$ 54.000. Isto leva a um custo estimado de R$ 246.000 para o software da solução nacional.

Uma vez que ninguém no ramo revela seus preços e como eles são compostos, obviamente, isto é apenas uma estimativa, feita com base em conversas informais, mas é suficiente para o propósito de dimensionar o valor do software nestes produtos, que compõe cerca de 80% do custo.

Valor que para países como o Brasil são extremamente elevados, principalmente para o sistema público de saúde. Desta forma um software livre para explorar estas imagens é fundamental para tornar este tipo de recurso médico acessível à toda a população.

A.3.2 Paralelismo

Na parte objetiva da monografia o termo paralelismo foi empregado diversas vezes para descrever uma propriedade do algoritmo responsável pela tractografia que permite seu processamento em tempo real.

Este é um conceito de computação que descreve quando um algoritmo possui trechos que podem ser executados simultaneamente. Ou seja, dois ou mais trechos do algoritmo são executados ao mesmo tempo em processadores diferentes.

Hoje, onde processadores possuem ao menos dois núcleos e alguns chegam até a dezesseis, a execução em paralelo dos algoritmos é de suma importância para se tirar proveito de todo o potencial do hardware.

Teoria

Paralelismo se baseia fortemente na idéia de divisão e conquista. Ou seja, um problema grande que pode ser subdividido em problemas menores e a combinação do resultado detes subproblemas pequenos gera o resultado do primeiro problema maior. Em paralelismo, cada um destes subproblemas é resolvido simultânemente.

Porém, escrever algoritmos paralelos é mais complexo que escrever algoritmos sequenciais. Isso acontece pois ao desenvolver estes algoritmos, quando os subproblemas possuem alguma dependência (um subproblema depende de um resultado intermediário de outro por exemplo) entre si é preciso pensar em como sincronizar as diversas instâncias sendo executadas simultâneamente a fim de garantir uma resposta correta.

Este tipo de problema, conhecido como exclusão mútua, pode ser compreendido como dois individúos que desejam utilizar um recurso ao mesmo tempo. Porém se mais de um indivíduo utilizar este recurso ao mesmo tempo, este se desintegra. Então, uma solução seria o primeiro indivíduo a chegar deixar um aviso de que está utilizando o recurso e quando terminar o retira. Os demais ao chegarem esperam o aviso ser retirado e um a um podem utilizar o recurso garantindo sua integrigade.

Contudo, quando um indivíduo esquece de remover seu aviso, todos os demais esperarão indefinidamente e nunca poderão utlizar o recurso, causando um outro problema que deve ser levado em conta conhecido por *deadlock*.

Exemplo

Uma forma simples de ilustrar o poder do paralelismo é pensando na soma de números. Tomemos oito números dois e calculemos sua soma sequencialmente:

1. $2 + 2 = 4$;
2. $4 + 2 = 6$;
3. $6 + 2 = 8$;
4. $8 + 2 = 10$;
5. $10 + 2 = 12$;
6. $12 + 2 = 14$;
7. $14 + 2 = 16$.

Considerando que cada soma consome uma unidade de tempo para ser executada, o tempo necessário foi sete. Agora resolvamos o mesmo problema, mas compreendendo que os números podem ser agrupados dois a dois sem alterar o resultado:

1. (2+2) + (2+2) + (2+2) + (2+2) = 4 + 4 + 4 + 4
2. (4+4) + (4+4) = 8 + 8;
3. (8+8) = 16.

Desta forma, foi obtido o mesmo resultado mas consumindo apenas 3 unidades de tempo.

A.3.3 Processamento geral na unidade de processamento gráfico

A unidade de processamento gráfico, mais conhecida por sua sigla em inglês *GPU* (*Graphics Processing Unit*), é um hardware desenvolvimento para o processamento de imagens. Como imagens em geral são grandes matrizes, este é um hardware especializado em trabalhar com operações sobre matrizes.

O princípio da computação de propósito geral na unidade de processamento gráfico (*GPGPU - General Puporse computing on Graphics Processing Unit*), ou seja executar algoritmos quaisquer ao invés de apenas os de processamento de imagens, se deu com desenvolvedores realmente convertendo o seus algoritmos em termos de operações com matrizes.

Felizmente com a popularização deste tipo de uso da placa gráfica, surgiram arquiteturas melhores para se trabalhar. Hoje as mais difundidas são CUDA e OpenCL. A primeira é exclusiva para placas gráficas NVIDIA sendo de compreensão mais simples, sendo propriedade desta. Enquanto que a segunda é livre e se propõe a funcionar para qualquer hardware, sendo capaz de realizar processamento.

A.4 Conclusão

Desta forma, o produto da monografia, levando em conta a avaliação de custos de software para exploração de imagens médicas, representa um passo importante para tornar esta tecnologia acessível à toda a população, uma vez que esta tecnologia é pouco comum e sem uma alternativa em software livre.

Igualmente, o detalhamento dos tópicos comuns à computação como paralelismo e *GPGPU* visam tornar estes resultados mais acessíveis a pessoas de outras áreas que possam desejar aproveitar este trabalho.

A.5 Referências

- PoliCidadã - http://policidada.poli.usp.br/
- Artis - http://www.artis.com.br/eximius/index_eximius.php

Apêndice B

MAC0431 - Intro. à Prog. Paralela e Distribuída

B.1 Introdução

Em um dos exercícios programas da disciplina de Introdução à Programação Paralela e Distribuída seu tema foi livre. Os únicos requesitos eram implementar um algoritmo paralelo utilizan o *Apache Hadoop* (B.4). O que foi uma feliz coincidência para testarmos o método desenvolvido em um ambiente distribuído.

B.2 Conceitos e tecnologias estudadas

B.2.1 Apache Hadoop

Hadoop é um arcabouço bastante difundido que permite o processamento distribuído de grandes quantidades de dados. Ele se apóia em dois procedimentos muito comuns em programação funcional: as funções *map* e *reduce*.

A função *map* é aplicada sobre cada elemento de um vetor dado representando o processamento distribuído. Este novo vetor é passado à função *reduce* que agregará os dados deste vetor para produzir o resultado final.

B.3 Resultados

O código produzido com a implementação do método pode ser encontrado em git@github.com:rafamanzo/runge-kutta-hadoop.

Esta implementação do método o vetor inicial consiste de todos os pontos iniciais para os quais queremos aplicar o método. Então a função *map* é programada para aplicar o método de Runge-Kutta de ordem 4. Como este já é o resultado desejado, a função *reduce* foi programada para apenas escrever os resultados do map em disco sem agregar nada.

B.4 Referências

- Hadoop - http://hadoop.apache.org/