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

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

2 de dezembro de 2012

Introdução

Na prática

- Aplicações para visualização de imagens por difusão de tensores (*DTI - Diffusion Tensors Images*) costumam oferecer uma funcionalidade chamada de tractografia (*fiber tracking*);
- Por trás desta funcionalidade estão diversas instâncias independentes de problemas de valor inicial (ou integração numérica de uma equação diferencial ordinária);
- A solução destes problemas pode ser aproximada pelo algoritmo de Runge-Kutta;
- Porém, quando temos centenas de instâncias do problema, os cálculos podem levar também centenas de segundos. O que torna impossível sua implementação em tempo real;
- Para tonar possível os cálculos em tempo real implementações em GPU são uma solução já utilizada.

Problemas de Valor Inicial[1]

ou *IVP (Initial Value Problem)*

Dados:

- Equação Diferencial Ordinária (EDO): $f(x, y(x)) = \frac{dy}{dx}$
- Lista de pontos iniciais: $(x_0, y_0), ..., (x_n, y_n)$
- Tamanho de passo h

Para cada ponto inicial (x_i, y_i) queremos calcular o valor de y em $x_i + h$

[1] PRESS, William H. et al. *Numerical Recipes in C*

Método de Integração Numérica Runge-Kutta[2]

Ordem 2

- O algoritmo é uma generalização do método de Euler para aproximação de solução de EDOs usando séries de Taylor
- Para ordem 2 temos a seguinte expressão, onde k_1 e k_2 são variáveis auxiliares:

$$k_1 = h.f(x_n, y_n)$$
$$k_2 = h.f(x_n + \frac{h}{2}, y_n + \frac{k_1}{2})$$
$$y_{n+1} = y_n + k_2 + O(h^3)$$

[2]PRESS, William H. et al. *Numerical Recipes in C*

Método de Integração Numérica Runge-Kutta[3]

Ordem 4

- Para ordem 4 temos a seguinte expressão, onde k_1, k_2, k_3 e k_4 são variáveis auxiliares:

$$k_1 = h.f(x_n, y_n)$$
$$k_2 = h.f(x_n + \frac{h}{2}, y_n + \frac{k_1}{2})$$
$$k_3 = h.f(x_n + \frac{h}{2}, y_n + \frac{k_2}{2})$$
$$k_4 = h.f(x_n + h, y_n + k_3)$$
$$y_{n+1} = y_n + \frac{k_1}{6} + \frac{k_2}{3} + \frac{k_3}{3} + \frac{k_4}{6} + O(h^5)$$

[3]PRESS, William H. et al. *Numerical Recipes in C*

Campos Vetoriais

- Representam a discretização de uma ou mais EDOs
- $x_i + h$ pode não estar definido no campo
- Podem ser muito complexos (várias EDOs misturadas no mesmo campo)

GPGPU

General Purpose Computing on Graphics Processing Units

Computação de propósito geral na unidade de processamento gráfico.

- As GPUs são utilizadas principalmente para processamento gráfico, mas descobriu-se seu poder de processamento é muito interessante também para outros tipos de processamento que envolvam cálculos e sejam altamente paralelizáveis.
- Uma NVIDIA GeForce GTX 690 possui mais de 3000 núcleos de processamento (*CUDA Cores*) a aproximadamente 900MHz cada e 4GB de memória dedicada[4].

Então este é um ambiente bastante propenso para implementarmos nosso algoritmo e obtermos uma implementação do método de Runge-Kutta em tempo real.

[4] Catálogo NVIDIA

Implementação de algoritmos gerais em GPU

- No inicío era feito em termos de operações gráficas (produto de matrizes de textura por exemplo)
- Com o surgimento da linguagem Cg isso se tornou mais plausível, mas ela ainda é uma linguagem baseada no C que é convertida em termos de DirectX ou shaders do OpenGL.
- Percebendo a necessidade de algo mais próximo às linguagens de propósito geral, surgiram outras duas linguagens CUDA e OpenCL que podem ser utilizadas como extensões das linguagens C, C++ e Fortran.

Comparação teórica

CUDA

- Propriedade da NVIDIA.
- Apenas para GPUs NVIDIA (apesar de seguir o padrão LLVM, só existe compilador para GPUs NVIDIA).
- Alto acoplamento ao código (é possível mesclar CUDA com outras linguagens no mesmo arquivo fonte).
- Tem maior conhecimento do hardware permitindo otimizações específicas para este.

OpenCL

- A marca é propriedade da Apple (desenvolveu a primeira versão), mas é desenvolvido pelo Khronos Group, um consórcio de empresas que atualmente inclui AMD, ARM, Intel e NVIDIA e outras empresas.
- Executado em GPUs e CPUs de qualquer fabricante desde que com os drivers apropriados.

Características importantes da arquitetura da GPU

Geral

- O trecho de código executado na GPU é chamado de *kernel*;
- Um mesmo *kernel* possui várias instâncias sendo executadas concorrentemente. Estas instâncias podem ser agrupadas em até dois níveis;
- Existem quatro níveis diferentes de memória nas GPUs com diferentes escopos e velocidades de acesso;
- A GPU se encarrega do escalonamento.

Características importantes da arquitetura da GPU
CUDA e NVIDIA

- Cada instância de um *kernel* é chamada de *thread* que por sua vez podem estar agrupadas em blocos (*blocks*) que, por fim, se agrupam em grades (*grids*);
- Seus níveis de memória são: **local** (exclusiva de cada *thread* e muito rápida), **compartilhada** (rápida e todas as *threads* no mesmo bloco têm acesso a ela), **constante** (todas as *threads* de todos os blocos leêm, mas é lenta) e **global** (todas as *threads* de todos os blocos leêm e escrevem, muito lenta);
- Os núcleos da GPU estão agrupados em SMs (*stream multiprocessors*) cada um com uma memória compartilhada dedicada;
- Todas as threads de um bloco são necessariamente escalonadas para o mesmo SM;
- Sua menor unidade de escalonamento é um *warp*, que corresponde a um conjunto de 16 *threads*.

Características importantes da arquitetura da GPU

OpenCL

- Cada instância de um *kernel* é chamada de item de trabalho (*work-item)* que podem ser agupadas em grupos de trabalho (*work-groups*);
- Por ser multiplataforma, existe um nível superior que é o contexto (*context*). Ele, além de agrupar grupos de trabalho, agrupa todos os dispositivos e suas memórias;
- Seus níveis de memória são: **privada** (exclusiva de cada *thread* e muito rápida), **local** (rápida e todas as *threads* no mesmo bloco têm acesso a ela), **constante** (todas as *threads* de todos os blocos leêm, mas é lenta) e **global** (todas as *threads* de todos os blocos leêm e escrevem, muito lenta);

Motivação

Objetivos

- Protótipos em C++, CUDA e OpenCL;
- Realizar testes para comprovar se os desempenhos são os esperados;
- Protótipo utilizando a biblioteca VTK.

Metodologia

Adapatação a campos vetoriais
Quanto as EDOs

- Temos vetores que podemos interpretar como a direção da reta tangente na direção da função que desejamos aproximar.
- O que nos leva a seguinte expressão para o RK4, por exemplo:
  $k_1 = h.(\alpha, \beta, \gamma)$
  $k_2 = \frac{k_1}{2} + h.(\alpha, \beta, \gamma)$
  $k_3 = \frac{k_2}{2} + h.(\alpha, \beta, \gamma)$
  $k_4 = k_3 + h.(\alpha, \beta, \gamma)$
  $(x_{n+1}, y_{n+1}, z_{n+1}) = (x_n, y_n, z_n) + \frac{k_1}{6} + \frac{k_2}{3} + \frac{k_3}{3} + \frac{k_4}{6}$
- Onde (x_n, y_n, z_n) é um ponto inicial, (α, β, γ) é o vetor associado a este ponto e $(x_{n+1}, y_{n+1}, z_{n+1})$ é o ponto resultante do método

Adapatação a campos vetoriais
Quanto a natureza dos campos vetoriais

- Em pontos nos quais não há um vetor associado, fazemos a interpolação (*Trilinear interpolation*).
- No limite do campo não teremos todos os pontos necessários para a interpolação. Então usamos o mais simples possível que é buscar o vizinho mais próximo (*Nearest Neighbour*).

Aplicações

Tractografia
Fibra

Definição de fibra

- Existem imagens de ressonância magnética por difusão de tensores que resultam em campos vetoriais que representam caminhos nervosos ou musculares;
- Reconstrução tridimensional de trajetórias do trato como extensão natural do campo vetorial obtido através de uma ressonância magnética[5].

[5] 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

Tractografia
Quais fibras calcular

- Geralmente são calculadas a partir de um ou mais pontos iniciais em ambos os sentidos (o mesmo processo é feito para o campo vetorial oposto).
- Este conjunto de um ou mais pontos é chamado de região de interesse[6].

[6] 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

Paralelizável

- Cada ponto inicial e cada direção são instâncias independentes (o ponto de uma fibra, idealmente, não é influenciado por outra fibra) do problema que podem ser resolvidas concorrentemente.
- Então temos um problema altamente paralelizável que envolve múltiplas operações de ponto flutuante ideal para ser tratado na GPU.

O que foi feito

- Implementações do RK2 e RK4 em C++, CUDA e OpenCL;
- Testes comparativos do método em GPU e CPU;
- Visualização do resultado do método em OpenGL;
- Visualização do resultado do método através da VTK;
- Resultado do método pode ser exportado para o GNUPlot.

Kernel RK4
CUDA

```
__global__ void rk4_kernel(vector *v0, int count_v0, double h, int n_x, int n_y, int n_z,
                           vector_field field, vector *points, int *n_points, int max_points)
{
  vector k1, k2, k3, k4, initial, direction;
  int i, n_points_aux;

  n_points_aux = 0;

  i = threadIdx.x;

  set( &initial, v0[i] );
  set( &direction, field[cuda_offset(n_x, n_y, initial.x, initial.y, initial.z)] );

  while(floor(module(direction)) > 0.0 && n_points_aux < max_points){
    n_points_aux++;

    set( &(points[cuda_offset(count_v0, 0, i, n_points_aux - 1, 0)]), initial );

    set( &k1, mult_scalar( direction, h ) );
    set( &k2, sum( mult_scalar(k1, 0.5), mult_scalar( direction, h ) ) );
    set( &k3, sum( mult_scalar(k2, 0.5), mult_scalar( direction, h ) ) );
    set( &k4, sum( k3, mult_scalar( direction, h ) ) );

    set( &initial, sum( initial, sum( mult_scalar( k1, 1.0/6.0 ), sum( mult_scalar( k2, 1.0/3.0 ),
         sum( mult_scalar( k3, 1.0/3.0 ), mult_scalar( k4, 1.0/6.0 ) ) ) ) ) );
    set( &direction, trilinear_interpolation(initial, n_x, n_y, n_z, field) );
  }

  n_points[i] = n_points_aux;
}
```

Kernel RK4
OpenCL

```
__kernel void rk4_kernel(__global vector *v0, __global unsigned int* count_v0,__global double* h,
  __global int* n_x,__global int* n_y,__global int* n_z, __global vector* field,
  __global vector *points,__global unsigned int *n_points, __global unsigned int* max_points){

  vector k1, k2, k3, k4, initial, direction;
  unsigned int i, n_points_aux;

  n_points_aux = 0;

  i = get_global_id(0);

  set( &initial, v0[i] );
  set( &direction, field[opencl_offset(*n_x, *n_y,initial.x,initial.y,initial.z)] );

  while(module(direction) > 0.0 && (n_points_aux < (*max_points) && n_points_aux < MAX_POINTS)){
    n_points_aux++;

    points[opencl_offset((*count_v0), 0, i, n_points_aux - 1, 0)] =  initial;

    set( &k1, mult_scalar( direction, *h ) );
    set( &k2, mult_scalar( trilinear_interpolation(sum(initial, mult_scalar( k1, 0.5 )), n_x, n_y, n_z,
                                                   field), *h) );
    set( &k3, mult_scalar( trilinear_interpolation(sum(initial, mult_scalar( k2, 0.5 )), n_x, n_y, n_z,
                                                   field), *h) );
    set( &k4, mult_scalar( trilinear_interpolation(sum(initial, k3), n_x, n_y, n_z, field), *h) );

    set( &initial, sum( initial, sum( mult_scalar( k1 , 0.166666667 ), sum( mult_scalar( k2, 0.333333333
                  sum( mult_scalar( k3, 0.333333333 ), mult_scalar( k4, 0.166666667 ) ) ) ) ) );
    set( &direction, trilinear_interpolation(initial, n_x, n_y, n_z, field) );
  }

  n_points[i] = n_points_aux;

}
```

Testes de performance
Processamento

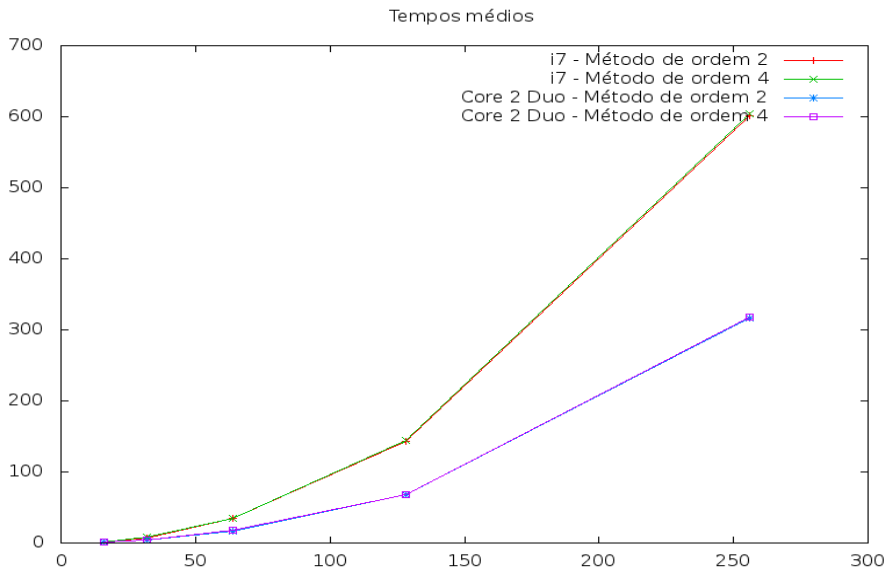

Figura: Média dos tempos de processamento em CPU pela quantidade de pontos iniciais

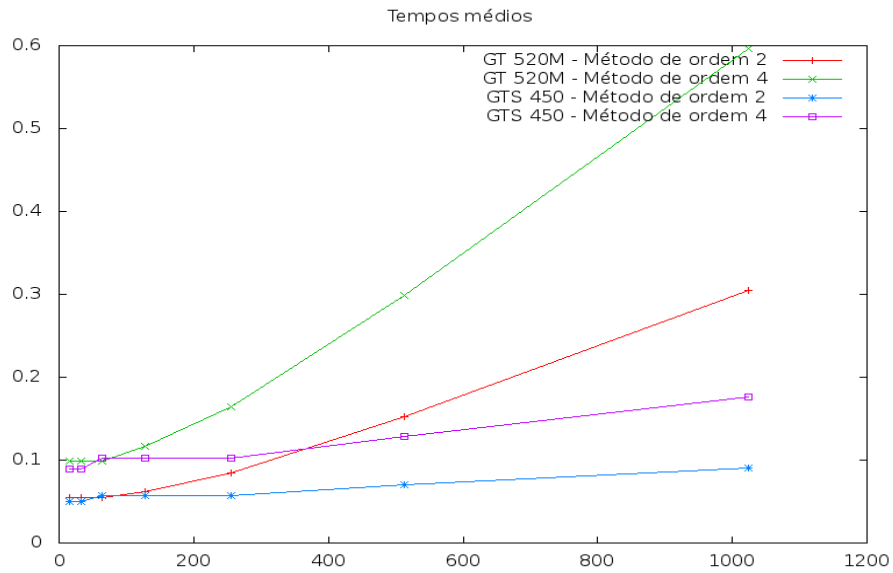

Figura: Média dos tempos de processamento em GPU pela quantidade de pontos iniciais

Testes de performance
Memória

- Em CPU, apesar de o problema crescer exponencialmente o tempo gasto em operações em memória é constante, devido ao se barramento dedicado;
- Por outro lado, na GPU este é o maior consumo de tempo, com crescimento linear (gráfico ao lado);
- Porém a soma dos tempos de processamento com o de operações em memória ainda é muito mais vantajoso para a GPU.

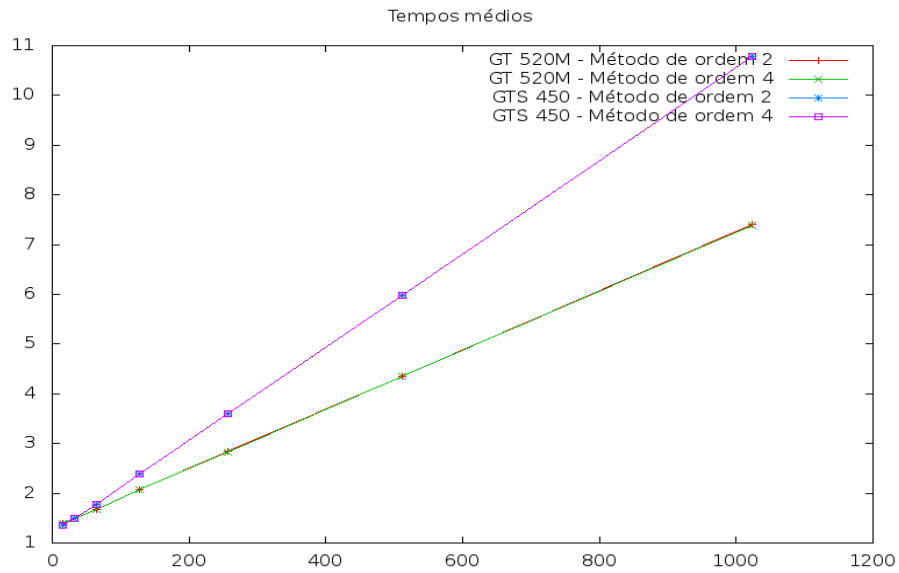

Figura: Média dos tempos de operações memória para GPU pela quantidade de pontos iniciais

Visualização
GNUPlot

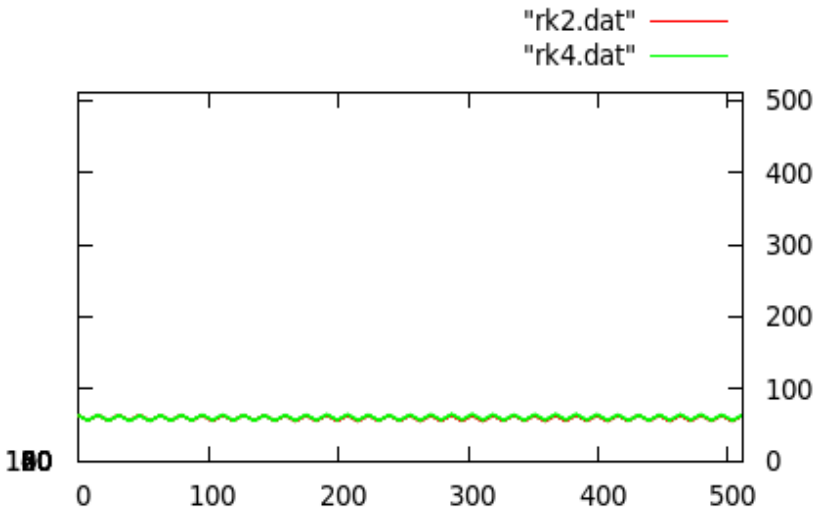

Figura: Visualização no OpenGL do resultado dos algoritmos para um campo vetorial 512x512x128, com ponto inicial (0,64,64) e tamanho de passo 0.2, que representa a direção de retas mudando periodicamente.

Visualização
OpenGL

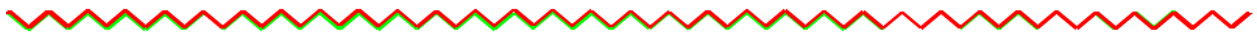

Figura: Visualização no OpenGL do resultado dos algoritmos para um campo vetorial 512x512x128, com ponto inicial (0,64,64) e tamanho de passo 0.2, que representa a direção de retas mudando periodicamente. Podemos ver que a resolução é maior que com o gnuplot e que depois de muitas iterações o RK2 diverge mais rápido.

Visualização
VTK

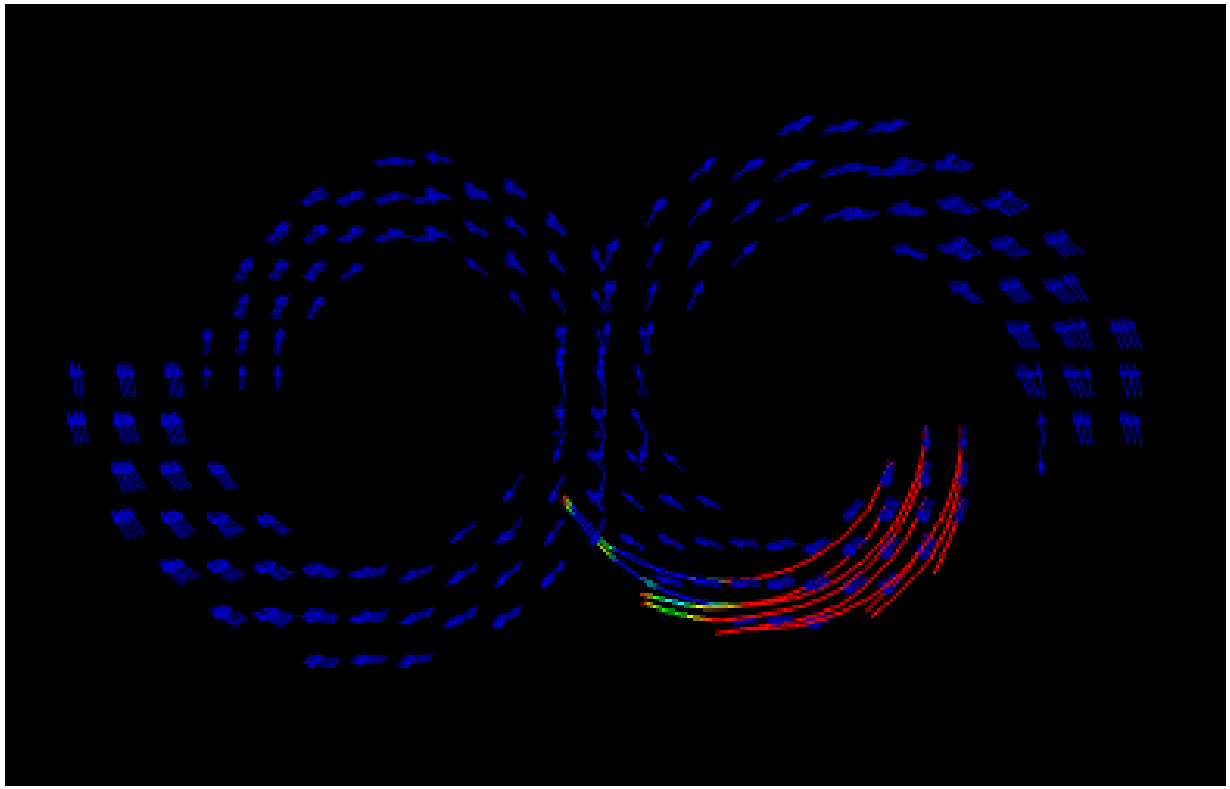

Figura: Visualização através da implementação com a biblioteca VTK para um campo 32x32x32 representando a intersecção de duas hélices, com pontos iniciais na base de uma das hélices. Esta biblioteca permite facilmente visualizar o resultado (em vermelho) e os vetores do campo (em azul)

Conclusões

- A discretização de EDOs como campo campo vetorial e a adaptação do método de Runge-Kutta não comprometem, visualmente, a precisão do resultado;
- A implementação de um *kernel* em CUDA e OpenCL que aplique o método foi comprovada como possível;
- Também foi comprovada como sendo muito mais rápida que uma implementação equivalente em C++ para CPU;
- Portanto, a tractografia em tempo real é possível através de CUDA e OpenCL.