Asociación Argentina de Mecánica Computacional

Mecánica Computacional Vol XXIX, págs. 7003-7018 (artículo completo)
Eduardo Dvorkin, Marcela Goldschmit, Mario Storti (Eds.)
Buenos Aires, Argentina, 15-18 Noviembre 2010

PROCESSAMENTO BASEADO EM UNIDADES DE PROCESSAMENTO GRÁFICAS (GPU) PARA CÁLCULO DE PARÂMETROS FÍSICOS E GEOMÉTRICOS DO TECIDO ÓSSEO TRABECULAR A PARTIR DE DADOS DE MICRO TOMOGRAFIA

Marco A. Argenta[a], Tiago M. Buriol[b], Mildred B. Hecke[a], Sergio Scheer[b]

[a]*Grupo de Bioengenharia, Universidade Federal do Paraná, Centro Politécnico, Jardim das Américas, Curitiba/PR, Brasil, marco.argenta@gmail.com, mildred@ufpr.br*
http://www.grupo.bioengenharia.ufpr.br

[b]*Grupo de Visualização do PPGMNE, Universidade Federal do Paraná, Centro Politécnico, Jardim das Américas, Curitiba/PR, Brasil, tiagoburiol@gmail.com, scheer@ufpr.br*
http://rbv.cesec.ufpr.br/

Palavras-Chave: Supercomputação Pessoal, Osso Trabecular, Micro tomografia.

Resumo. A modelagem numérica de estruturas biológicas como ossos e dentes representa um desafio científico na área de simulações de estruturas, devido à dificuldade em se obter amostras para ensaios laboratoriais, como é o caso do estudo do remodelamento ósseo e fenômenos relacionados. Porém, dados de tomografia podem ser usados para estimar parâmetros usados no ajuste de modelos teóricos conhecidos de outras áreas da engenharia adaptando-os para a bioengenharia. Uma das dificuldades, nesse caso, ocorre devido à demanda de processamento computacional para tratar grandes volumes de dados tridimensionais. Por outro lado, dispõe-se hoje, de um alto poder de processamento, massivamente paralelo fornecido pelas unidades de processamento gráficas (GPU), modernas. Isso permite a realização de análises complexas em uma fração do tempo demandado pelas CPUs. Neste trabalho, é relatada uma experiência de processamento de dados de micro tomografia utilizando a GPU. Os algoritmos desenvolvidos têm como objetivo a obtenção dos parâmetros como volume de osso e vazios, densidade aparente e do pixel, grau de anisotropia, direções anisotrópicas e dimensão fractal tridimensional. Todo desenvolvimento foi feito utilizando um conjunto de ferramentas de distribuição livre, algumas de código aberto, tanto para o processamento como para a visualização dos dados. Dentre essas ferramentas utilizadas estão o Visualization Toolkit (VTK) e arquitetura de computação paralela nVidia CUDA. Além da descrição detalhada do ambiente de desenvolvimento utilizado, são discutidos aspectos práticos da implementação e do desempenho no processamento computacional obtido, apresentando-se os principais benefícios e dificuldades encontradas, bem como, as soluções propostas.

Copyright © 2010 Asociación Argentina de Mecánica Computacional http://www.amcaonline.org.ar

1 INTRODUÇÃO

Em engenharia, é comum descrever o comportamento de estruturas utilizando modelos matemáticos que fazem uso de parâmetros previamente estabelecidos, os quais caracterizam a propriedades dos materiais que compõem tais estruturas. Também é comum obter esses parâmetros por meio de ensaios utilizando-se corpos de prova de dimensões padronizadas e geometria simples. Entretanto, no caso de estruturas biológicas como ossos e dentes, a obtenção das propriedades dos materiais representa um desafio científico devido à dificuldade em se obter amostras para ensaios laboratoriais com dimensões padronizadas, por causa da geometria irregular. Nesse contexto, os dados de micro tomografia computadorizada podem ser úteis para estimar os parâmetros usados no ajuste de modelos teóricos conhecidos de outras áreas da engenharia, adaptando-os para a bioengenharia.

A obtenção de parâmetros físicos e geométricos de ossos, tais como, volume de vazios, volume de osso, densidade aparente, grau de anisotropia, direções anisotrópicas e a dimensão fractal a partir de imagens de micro tomografia, são utilizados, por exemplo, para diagnosticar doenças como a osteoporose, quantificar fenômenos como a remodelação óssea ou, ainda, gerar modelos matemáticos aproximados para descrever o comportamento mecânico dessas estrutura (Harrison, et al. 2008), (Nagaraja, Couse e Guldberg 2005). Esse tipo de computação, em geral, requer algoritmos semelhantes aos utilizados em processamento de imagens, tendo em vista que os dados gerados por micro tomografias são distribuídos em uma grade tridimensional, regular e ortogonal, armazenados na forma de imagens digitais.

A demanda computacional para o processamento desses dados é alta, pois os modernos equipamentos de micro tomografia são capazes de gerar grandes volumes de dados, na ordem de Gigabytes, resultando centenas de imagens com resolução de até 16 Megapixels e detalhes na ordem de 3~5 micrômetros. A falta de recursos computacionais para processar rapidamente esse volume de dados, a dificuldade e a complexidade da programação de alto desempenho, são alguns dos principais fatores que limitam cientistas e pesquisadores no processo de desenvolvimento de aplicativos para análise e processamento de grandes volumes de dados.

Além disso, as linguagens de programação tradicionais, tais como Fortran e C, são pouco convenientes e produtivas para cientistas não programadores. Por outro lado, programas comerciais gerados em pacotes voltados a computação científica, tais como Maple, MatLab e Maxima, não são portáteis, ou seja, não podem ser executados em outras máquinas, a menos que a máquina-destino também possua o aplicativo gerador do programa, cujo custo da licença muitas vezes inviabiliza a sua utilização.

Em termos de aplicativos comerciais, frequentemente, cientistas se deparam com problemas para os quais as ferramentas adequadas não existem. O desenvolvimento feito pelos próprios cientistas (físicos, matemáticos, biólogos, etc., que não são programadores profissionais), geralmente ocorre de forma colaborativa através do

Copyright © 2010 Asociación Argentina de Mecánica Computacional http://www.amcaonline.org.ar

compartilhamento de códigos fonte. Esse processo impõe aos pesquisadores uma situação em que eles têm de buscar uma formação improvisada em programação para produzirem programas minimalistas, que contêm o menor número de linhas de código possível para resolver o problema em questão (Coelho 2007).

Hoje, dispõe-se de um alto poder de processamento, massivamente paralelo, fornecido pelos processadores gráficos GPUs modernos. Isso possibilita, de certa forma, a realização de análises complexas em uma fração do tempo demandado pelas CPUs. Também dispõe-se hoje de linguagens de programação bastante convenientes, as quais possuem de vastas bibliotecas e diferentes recursos que minimizam o esforço de programação. A linguagem Python, por exemplo, projetada para ser de fácil aprendizado, suporta diferentes paradigmas de programação, incluindo Orientação a Objetos (OO), além de ser multiplataforma, possuir uma vasta biblioteca padrão e ter inúmeros módulos disponibilizados por terceiros. Por meio do *wrapper* pyCUDA, por exemplo, é possível acessar a GPU via Python e escrever programas de forma bastante conveniente.

Este trabalho é parte de um projeto maior que objetiva estabelecer uma metodologia para a determinação de parâmetros físicos e geométricos do tecido trabecular ósseo, obtidos em função de imagens de micro tomografia. O projeto possui três frentes distintas e complementares, a primeira delas visa estabelecer e formalizar metodologias na forma de algoritmos para obtenção dos parâmetros em questão. A segunda é a implementação desses algoritmos utilizando recursos computacionais inovadores, tais como, processamento massivamente paralelo utilizando placas gráficas e linguagens de programação de alto nível, interpretadas e multiplataforma, de tal maneira que se componha um ambiente produtivo e eficiente. A terceira frente é a que busca gerar ferramentas de visualização científica que provêm interfaces não convencionais para exploração de grandes volumes de dados tais como campos tensoriais tridimensionais.

Neste artigo, especificamente, será tratada a questão da implementação dos algoritmos na GPU. Todo desenvolvimento foi feito em Python utilizando um conjunto de ferramentas de distribuição livre ou código aberto, tanto para o processamento como para a visualização dos dados. Dentre essas ferramentas utilizadas estão o Visualization Toolkit (VTK) e arquitetura de computação paralela nVidia CUDA. Além da descrição detalhada do ambiente de desenvolvimento utilizado, são discutidos aspectos práticos da implementação e do desempenho no processamento computacional obtido, apresentando-se os principais benefícios e dificuldades encontradas, bem como, as soluções propostas. Como exemplificação da eficiência da metodologia é discutida a implementação do algoritmo para o cálculo das direções anisotrópicas e grau de anisotropia por meio dos momentos de inércia de massa.

Copyright © 2010 Asociación Argentina de Mecánica Computacional http://www.amcaonline.org.ar

2 USANDO DADOS DE MICRO TOMOGRAFIA PARA ESTIMAR PARÂMETROS FÍSICOS E GEOMÉTRICOS DO OSSO TRABECULAR

A micro tomografia consiste em uma técnica não destrutiva para obtenção de dados do interior de amostras em escala micrométrica. O princípio físico básico da tomografia computadorizada é a interação de radiação com matéria. A geração de imagens de micro tomografia começa com a obtenção de projeções de raios X. Esse processo gera uma sequência de imagens radiológicas, capturadas pela rotação da amostra em 360º. Então, essas imagens passam por um processo computacional de reconstrução que resulta em fatias transversais da amostra. Uma vez reconstruída, cada fatia tomográfica transversal da amostra é representada em forma de uma matriz digital N x M pixels, onde N representa o número de pixels na vertical e M na horizontal. As fatias são calculadas a certa distância uma das outras que, em alguns casos, é exatamente igual ao tamanho do pixel da imagem. Um esquema do processo de aquisição é mostrado na Figura 1.

A cada pixel da imagem, designa-se um valor de cinza que é proporcional ao coeficiente de absorção do detector. Assim, tem-se que a região mais densa aparece mais escura e consequentemente à região que é menos densa aparece mais clara. Como a maioria dos monitores apresenta 256 níveis de cinza com um byte, a escala é montada com 0 para o preto e 255 para o branco e os valores que são intermediários a estes equivalem aos níveis de cinza propriamente dito.

Supondo que se tenha uma amostra de 200 fatias com resolução de 512 por 512 (N x M pixels), então, o volume de dados irá corresponder a uma matriz tridimensional com mais de 50 milhões de valores. Essa quantidade de dados, embora considerada pequena, pode ser suficiente para tornar seu processamento lento e oneroso, quando realizado em série por um computador pessoal. O aparelho de micro-CT utilizado para a aquisição das fatias tomográficas transversais é modelo SkyScan 1172 de alta resolução, com capacidade de detalhar as imagens em até 3 micrômetros com resolução na casa dos 16 Megapixels.

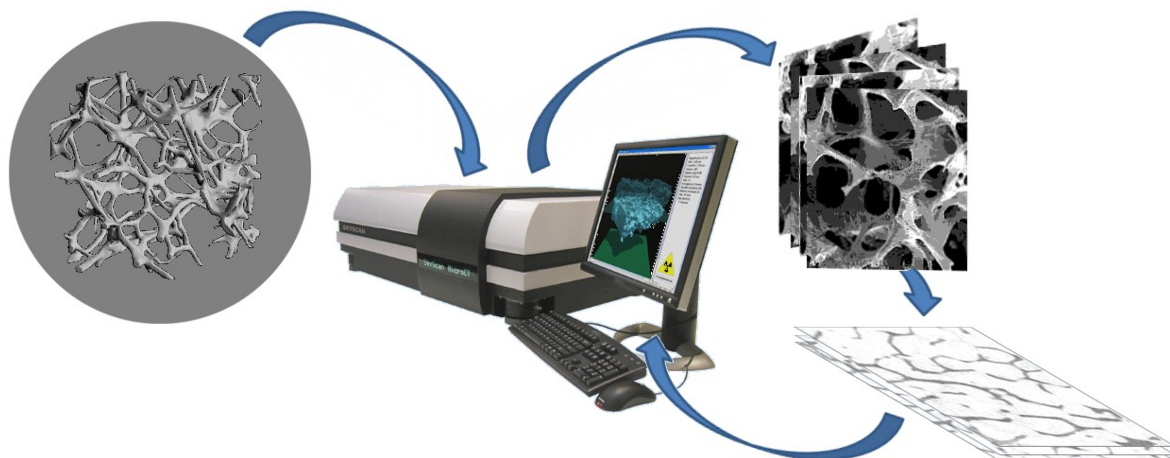

Figura 1: Esquema de aquisição de seções transversais da amostra usando micro tomografia.

Copyright © 2010 Asociación Argentina de Mecánica Computacional http://www.amcaonline.org.ar

2.1 DESCRIÇÃO DOS ALGORITMOS

A partir dos dados obtidos com as fatias tomográficas transversais, são aplicados algoritmos para a obtenção de parâmetros físicos e geométricos da amostra como volume de vazios, volume de osso, densidade aparente, densidade do pixel, grau de anisotropia, direções anisotrópicas e dimensão fractal tridimensional.

Os algoritmos para o cálculo dos volumes de vazio e de osso são baseados em características e quantidade de pixels. É feita uma varredura em cada pixel de todas as imagens (fatias tomográficas transversais) e compara-se o valor de cinza do pixel com um valor limiar pré-estabelecido. Como cada uma dessas imagens representa certa seção da amostra e como, no caso deste micro tomógrafo, a distância entre as fatias é igual ao tamanho do pixel, ou seja, cada fatia representa uma porção volumétrica da amostra com espessura igual ao tamanho do pixel, cada pixel é considerado como um cubo, conhecido como voxel. Uma visualização aumentada da fatia é ilustrada na Figura 2. Uma vez conhecida a quantidade de voxels que representam osso e a quantidade que representa vazio, basta integrar os elementos ao longo de todo volume de dados para obter-se o volume total de osso e vazio.

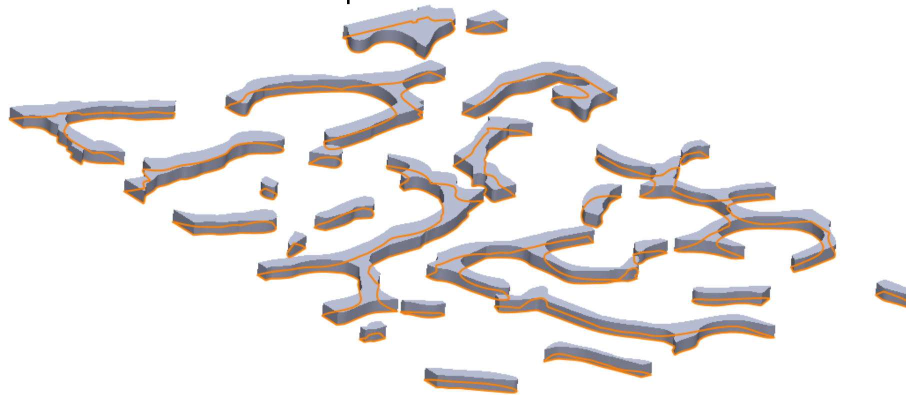

Figura 2: Seção extrudada da amostra a partir da fatia tomográfica transversal, com espessura do tamanho do pixel (visualização com espessura aumentada).

A densidade aparente é calculada com o peso total das amostras dividida pelo volume de osso calculado anteriormente. A densidade do pixel é estimada em cada ponto do osso trabecular correlacionando-se a escala de Hounsfield, identificada pelos valores de cinza das fatias tomográficas, com valores de cinza de fatias tomográficas de materiais com densidades conhecidas, colocados juntos com a amostra durante a aquisição.

O grau de anisotropia e as direções anisotrópicas podem ser calculados utilizando-se o método do comprimento médio de interceptação (Tabor e Rokita 2007), função de uma esfera de influência, e do momento de inércia das partes das trabéculas constantes em volumes representativos cúbicos dessa amostra. A dimensão fractal (Mandelbrot 1982), usada para quantificar a irregularidade das trabéculas, é calculada utilizando-se o método de box counting (Stoyan e Stoyan 1994).

Estes algoritmos são descritos de uma forma mais ampla no primeiro trabalho do projeto, intitulado: Metodologia para a obtenção de parâmetros físicos e geométricos

Copyright © 2010 Asociación Argentina de Mecánica Computacional http://www.amcaonline.org.ar

do osso trabecular função de imagens de micro tomografia (Argenta, Buriol e Hecke 2010).

Neste trabalho apenas é mostrada a implementação na GPU, usando pyCUDA, do algoritmo de cálculo das direções anisotrópicas de e grau de anisotropia, com momentos de inércia, da amostra como um todo e de volumes de referência representativos da mesma, tendo como objetivo ilustrar a eficiência do uso da placa gráfica para o processamento dos dados.

2.2 DIFICULDADES DE IMPLEMENTAÇÃO

Os algoritmos citados possuem a característica de realizar operações matemáticas relativamente simples sobre cada voxel da amostra, de forma independente umas das outras, o que caracteriza um processo que poderia ser realizado de forma paralelizada. Portanto, o problema da demanda de alto processamento computacional poderia ser amenizado por um processamento massivamente paralelizado. Em sistemas paralelos, essas operações independentes entre si, são realizadas simultaneamente por centenas ou milhares de microprocessadores, por meio de *threads*. A principal dificuldade nesse sentido é a falta de conhecimento, por parte de cientistas não programadores, em desenvolvimento de aplicativos para computação paralela.

Em termos de aplicativos comerciais de alto desempenho, frequentemente, cientistas se deparam com problemas para os quais as ferramentas adequadas não existem. O desenvolvimento feito pelos próprios cientistas (físicos, matemáticos, biólogos, etc., que não são programadores profissionais), geralmente ocorre de forma colaborativa através do compartilhamento de códigos fonte. Esse processo impõe aos pesquisadores uma situação em que eles têm de buscar uma formação improvisada em programação para produzirem programas minimalistas, que contêm o menor número de linhas de código possível para resolver o problema em questão.

É fato que muitos cientistas não possuem formação/habilidades e disponibilidade para escrever aplicativos tão sofisticados quanto os pacotes comerciais, ou programas para computação de alto desempenho, além disso, frequentemente cientistas dispõem de pouco tempo entre suas atividades de pesquisa para dedicar-se à programação. Outro complicador é que as linguagens de programação tradicionais foram projetadas e desenvolvidas por programadores para programadores e voltadas ao desenvolvimento de softwares profissionais com dezenas de milhares de linhas de código. Este cenário foi percebido por empresas de software científico, o que fez surgir uma nova classe de software, voltado para a demanda de cientistas que precisavam implementar métodos computacionais específicos e que não podiam esperar por soluções comerciais (Coelho 2007).

Nesta nova classe de aplicativos científicos, a qual inclui softwares como MATLAB, Mathematica e Maple, os programas são escritos em uma linguagem de alto nível, proprietária, por meio da qual os cientistas podem implementar seus próprios algoritmos, sem ter esforçar-se em transmitir a um programador profissional o que,

Copyright © 2010 Asociación Argentina de Mecánica Computacional http://www.amcaonline.org.ar

exatamente, ele deseja. Dessa maneira, os programas são escritos e executados no próprio ambiente, não podendo ser executados fora dos mesmos. Portanto, os programas gerados não são portáteis, ou seja, não podem ser executados em outras máquinas, a menos que a máquina-destino também possua o aplicativo gerador do programa, cuja licença pode custar milhares de dólares. Algumas vezes os programas nem podem ser utilizados em outro sistema operacional. Portanto, o programa produzido pelo cientista não lhe pertence, pois necessita do código proprietário do ambiente de desenvolvimento comercial para ser executado (Coelho 2007).

3 USO DA GPU

A indústria de jogos e a demanda de aplicativos em tempo real, com gráficos 3D de alta definição, fez com que as placas gráficas programáveis (GPUs) evoluíssem para sistemas massivamente paralelos, com alto poder de processamento e grande largura de banda de memória. A Figura 3 mostra uma comparação da evolução ocorrida no últimos anos entre os desempenhos de alguns modelos de GPUs da fabricante NVIDIA e alguns dos mais modernos processadores do mercado.

As GPUs, inicialmente designadas ao processamento gráfico, hoje, contêm o mais poderoso chip disponível em uma estação de trabalho de alto desempenho (Luebke 2008). No entanto, permanece sendo um desafio desenvolver aplicativos que possam distribuir adequadamente as tarefas entre os vários núcleos de processamento para, assim, conseguir um ganho real de desempenho (NVIDIA 2010).

Ciente desse fato, a fabricante de placas gráficas NVIDIA, desenvolveu uma extensão da linguagem C para a arquitetura CUDA que permite utilizar os processadores da placa de vídeo para outros fins, como processamento numérico e computação científica. Além da linguagem C, é possível programar as GPUs que suportam a arquitetura CUDA, utilizando algum dos muitos *wrappers* disponíveis, por exemplo, para as linguagens Java (jCUDA), C# (CUDA.NET), Python (pyCUDA), entre outros.

O modelo de programação em CUDA define funções em C, chamadas kernels que, quando chamadas, serão executadas N vezes em paralelo por N diferentes threads na GPU. Blocos de threads podem ter uma, duas, ou três dimensões, fornecendo um meio natural para ser em utilizados no cálculo de campos escalares uni, bi ou tridimensionais (NVIDIA 2010).

Por serem desenhadas para aumentar o desempenho de aplicações gráficas, as GPUs são capazes de realizar grande quantidade de operações aritméticas simultaneamente. Dedicam mais transistores ao processamento de dados, em vez de cache e controle de fluxo, como ocorre nas CPUs. São, portanto, poderosas ferramentas para processar grandes quantidades de dados paralelamente o que é ideal para muitas aplicações como processamento de imagens e dados de tomografia computadorizada. Processar dados na GPU é, primeiramente, uma maneira de “aliviar” a CPU e, além disso, o aumento de desempenho pode ser significativo por um preço bem menor se comparado a outros sistemas.

Copyright © 2010 Asociación Argentina de Mecánica Computacional http://www.amcaonline.org.ar

Diversos pesquisadores já vêm utilizando os benefícios da programação em paralelo oferecido pela GPU, conseguindo ganhos de 130x (Lu, et al. 2009) até 300x (Fang e Boas 2009).

Dentre as várias linguagens de programação que possibilitam acessar funções da GPU, a linguagem Python apresenta algumas características que podem fazer dela a melhor escolha. Por exemplo, Python é uma linguagem Orientada a Objetos (OO), poderosa e escrita para ser de fácil aprendizado. Possui uma sintaxe clara e concisa, que favorece a legibilidade do código e a geração de programas livres de erros. Além disso, inclui diversas estruturas de alto nível (listas, tuplas, dicionários, data/hora, complexos e outras).

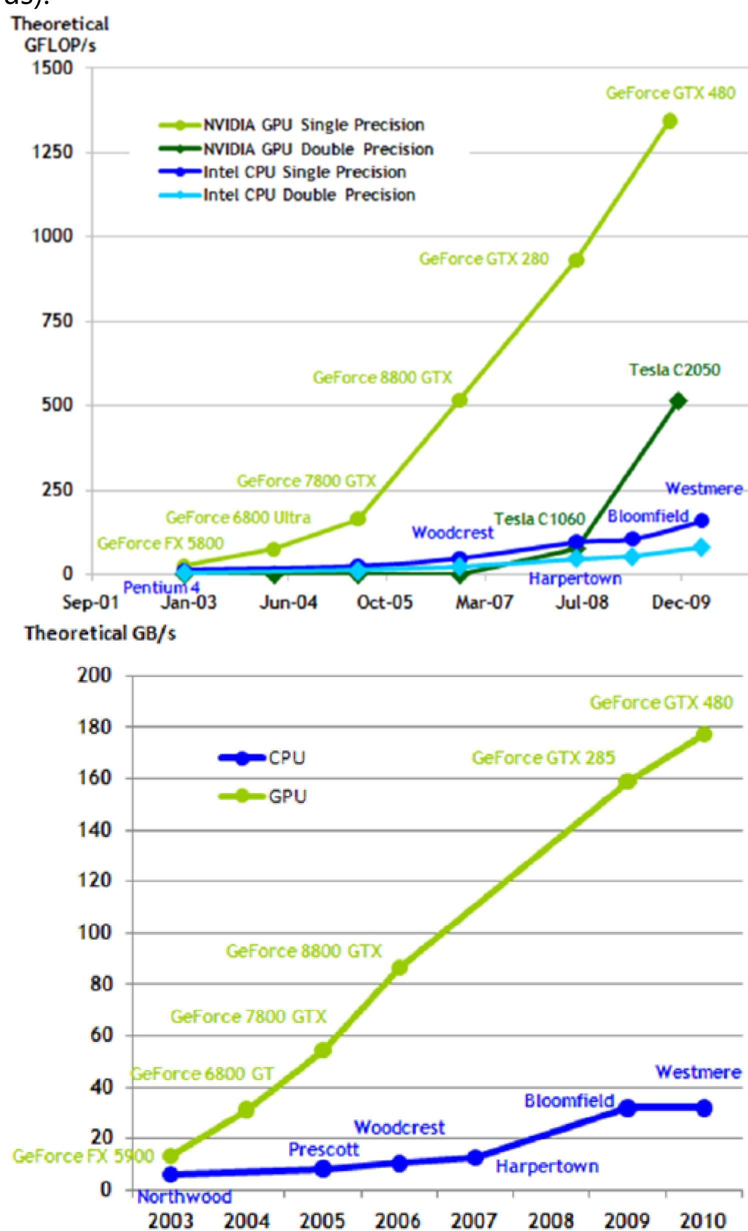

Figura 3: Comparação entre CPUs e GPUs de operações como ponto flutuante e largura de banda de memória (NVIDIA 2010).

De maneira geral, as linguagens interpretadas tais como Python (Python 2010)

Copyright © 2010 Asociación Argentina de Mecánica Computacional http://www.amcaonline.org.ar

alcançam um desempenho inferior em comparação a linguagens como C e Fortran, no entanto, permitem um desenvolvimento mais rápido. A combinação de Python com o processamento na GPU usando CUDA oferece uma conveniente combinação da produtividade de Python com o desempenho da GPU. Para acessar as funções da API de CUDA com o Python pode-se usar o *wrapper* pyCUDA (A. Klöckner 2010) que apresenta algumas vantagens em relação ao uso da extensão C (C for CUDA) da NVIDIA.

4 USANDO PYTHON, CUDA E VTK PARA CRIAR UMA PLATAFORMA EFICIENTE E PRODUTIVA

Tendo em vista vencer algumas dificuldades e limitações, geralmente encontradas ao se desenvolver programas de processamento paralelo e, também, buscando atingir um equilíbrio entre esforço de programação e desempenho final do aplicativo, optou-se pela linguagem de programação Python e o *wrapper* para acesso ao processador gráfico via arquitetura CUDA chamado pyCUDA.

```
__global__ void VecAdd(float* A, float* B, float* C, int N
{
    int i = blockDim.x * blockIdx.x + threadIdx.x;
    if (i < N)
        C[i] = A[i] + B[i];
}
```

Figura 4: Kernel CUDA para soma de dois vetores.

Copyright © 2010 Asociación Argentina de Mecánica Computacional http://www.amcaonline.org.ar

```
int N = ...;
size_t size = N * sizeof(float);

// Alocar vetores de entrada h_A e h_B na memória RAM
float* h_A = (float*)malloc(size);
float* h_B = (float*)malloc(size);

// Alocar vetores na memória da placa de vídeo
float* d_A;
cudaMalloc(&d_A, size);
float* d_B;
cudaMalloc(&d_B, size);
float* d_C;
cudaMalloc(&d_C, size);

// Copiar os vetores da memória RAM para memória da placa de vídeo
cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);

// Executar o kernel
int threadsPerBlock = 256;
int blocksPerGrid =
(N + threadsPerBlock – 1) / threadsPerBlock;
VecAdd<<<blocksPerGrid, threadsPerBlock>>>(d_A, d_B, d_C, N);

// Copiar os resultados da memória da placa de vídeo para memória RAM
// h_C contém os resultados na memória RAM
cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);

// Liberar a memória da placa
cudaFree(d_A);
cudaFree(d_B);
cudaFree(d_C);
```

Figura 5: Código em C para somar dois vetores na GPU usando o kernel mostrado na Figura 4.

A linguagem de programação Python foi projetada com a filosofia de enfatizar a importância do esforço do programador sobre o esforço computacional. Dessa maneira, prioriza a legibilidade do código sobre a velocidade ou expressividade. Ou seja, tem uma curva de aprendizagem menor, permitindo maior produtividade ao custo de gerar programas possivelmente mais lentos, em comparação a linguagens como Fortran e C. O uso do Python é frequentemente associado com grandes ganhos de produtividade e ainda, com a produção de programas de alta qualidade e de fácil manutenção (Coelho 2007). A arquitetura de computação paralela CUDA permite utilizar os recursos das unidades de processamento gráfico (GPUs) possibilitando acelerar o processamento numérico e resolver problemas computacionais complexos em uma fração do tempo necessário utilizando uma CPU (NVIDIA 2010).

No processamento via GPU, pelo menos três passos são necessários: (1) transferir os dados da memória RAM para a memória da GPU, (2) Processar na GPU e (3) retornar os resultados para a memória RAM da CPU. Utilizando CUDA com a linguagem C, é necessário, também, alocar memória, definir tamanho de blocos e grids e, ainda, liberar a memória ao final do processo, explicitamente. Com o

Copyright © 2010 Asociación Argentina de Mecánica Computacional http://www.amcaonline.org.ar

pyCUDA, algumas dessas etapas são desnecessárias pois o *wrapper* faz automaticamente. As Figura 4 e Figura 5 mostram como somar dois vetores utilizando C para CUDA, a Figura 6 mostra como somar dois vetores utilizando o pyCUDA.

```
VecAdd = ElementwiseKernel("float* A, float* B, float* C",
                           "C[i] = A[i] + B[i]",
                           "VecAdd")
A_gpu = gpuarray.to_gpu(A)
B_gpu = gpuarray.to_gpu(B)
C_gpu = gpuarray.to_gpu(C)
VecAdd(A_gpu, B_gpu, C_gpu)
C = C_gpu.get()
```

Figura 6: Código para somar dois vetores utilizando o pyCUDA.

PyCUDA é um encapsulador escrito em Python, de código aberto, que fornece acesso facilitado aos recursos da API CUDA. Existem muitos outros encapsulamentos para CUDA disponíveis, mas alguma vantagens fazem de pyCUDA uma escolha bastante conveniente. Por exemplo, a limpeza automática de objetos fora de escopo, o que torna mais fácil escrever códigos corretos, livres de vazamentos de recursos e outras falhas (Stroustrup 2001). Além disso, pyCUDA gerencia as dependências por isso não elimina um contexto antes que toda a memória alocada seja liberada. Abstrações como *pyCUDA.driver.SourceModule* e *pyCUDA.gpuarray.GPUArray* tornam a programação mais simples e intuitiva em comparação à extensão da linguagem C fornecida pela NVIDIA. Outra vantagem é que permite acessar integralmente todos os recursos da API CUDA, verificando erros automaticamente convertendo-os em exceções Python (A. Klöckner 2010). Como descreve (Klöckner, et al. 2009), pyCUDA possibilita uma grande redução nos esforços de implementação da maioria dos códigos de alto desempenho em diversas aplicações de computação científica. Uma comparação entre um código escrito em Python e escrito usando o pyCUDA com o Numpy é mostrado nas Figura 7 e Figura 8 respectivamente.

Tradicionalmente, escreve-se um programa para que realize uma determinada tarefa. Em metaprogramação escreve-se um programa que escreverá outro programa para realizar a tarefa. Em pyCUDA, a metaprogramação realiza automaticamente as tarefas como decidir o número de threads por bloco, a quantidade de dados a serem processados ao mesmo tempo e quais dados devem ser carregados na memória compartilhada.

O Numpy (Scipy 2008) é o pacote básico da linguagem Python para trabalhar com arranjos, vetores e matrizes de N dimensões, de uma forma comparável e com uma sintaxe semelhante ao software proprietário Matlab, mas com muito mais eficiência, e com toda a expressividade da linguagem. Provê diversas funções e operações sofisticadas, incluindo: objeto *array* para a implementação de arranjos multidimensionais; objeto *matrix* para o cálculo com matrizes; ferramentas para álgebra linear; transformadas de Fourier básicas; ferramentas sofisticadas para geração de números aleatórios, entre outros. Além disso, as classes criadas podem

Copyright © 2010 Asociación Argentina de Mecánica Computacional http://www.amcaonline.org.ar

ser facilmente herdadas, permitindo a customização do comportamento (por exemplo, dos operadores típicos de adição, subtração, multiplicação, etc.). O módulo é implementado em linguagem C, o que dá uma grande velocidade às operações realizadas.

```
for n in range(0,N):
    for i in range(0,W):
        for j in range(0,H):
            Ix = Ix+M[i][j][n]*(c+(my[i][j][n]-ycg)**2+(mz[i][j][n]-zcg)**2
            Iy = Iy+M[i][j][n]*(c+(mx[i][j][n]-xcg)**2+(mz[i][j][n]-zcg)**2
            Iz = Iz+M[i][j][n]*(c+(my[i][j][n]-ycg)**2+(mx[i][j][n]-xcg)**2
```

Figura 7: Cálculo dos momentos de inércia usando Python.

```
inercia = ElementwiseKernel("float k, float *m1, float *m2, float *dm, float *momI"
                            "momI[i] = dm[i] * (k + m1[i]*m1[i] + m2[i]*m2[i])",
                            "momI")

inercia(k, my_gpu, mz_gpu, M_gpu, out_gpu)
Ix = np.sum(out_gpu.get())

inercia(k, mx_gpu, mz_gpu, M_gpu, out_gpu)
Iy = np.sum(out_gpu.get())

inercia(k, mx_gpu, my_gpu, M_gpu, out_gpu)
Iz = np.sum(out_gpu.get())
```

Figura 8: Cálculo do momento de inércia usando Numpy + pyCUDA.

O VTK, Visualization ToolKit, (Kitware 2009) é um software gratuito, de código aberto, para computação gráfica 3D, processamento de imagem e visualização. Consiste numa biblioteca de classes, implementada e desenvolvida em linguagem C++ utilizando os princípios de programação orientada ao objeto e é independente da plataforma de desenvolvimento (MS Windows, Unix, Mac). Possuí também várias interfaces para linguagens interpretadas, que permitem desenvolver aplicações noutras linguagens para além de C++, como Tcl/Tk, Java e Python. Com o seu código fonte em C++ disponibilizado livremente, foi desenvolvido para ser facilmente expansível. Novas classes podem ser facilmente implementadas com o apoio da vasta documentação disponibilizada no código e nas páginas do manual on-line.

5 DISCUSSÃO E RESULTADOS

Cada fatia tomográfica é considerada como representativa de uma porção volumétrica da amostra, com a espessura do tamanho idêntico ao do lado do pixel da imagem, Figura 2. Empilhando-se essas fatias no espaço, obtêm-se um sólido tridimensional representativo da amostra de osso.

O primeiro passo para a montagem do sólido tridimensional representativo da amostra de osso é a identificação de quais pixels de cada fatia tomográfica fazem parte da estrutura óssea e quais pixels representam vazios. Esse processo, chamado de segmentação, foi realizado aqui tomando-se, primeiramente, uma pequena região de uma das fatias, contendo parte do osso e parte de vazio. Então, comparando-se os

Copyright © 2010 Asociación Argentina de Mecánica Computacional http://www.amcaonline.org.ar

valores dos níveis de cinza com a imagem, optou-se por assumir o valor 200 como limiar. Ou seja, pixels com valor de cinza maior que 200 são considerados vazios e valores menores que 200 são considerados osso, conforme ilustra a Figura 9.

O processo de segmentação, portanto, envolveu varrer todos os pixels de todas as imagens e verificar se o pixel corresponde ao osso ou ao vazio. Por fim, obteve-se duas matrizes tridimensionais, a matriz com os dados originais composta pelos valores de cinza e a matriz de valores binários, a qual indica se cada voxel do volume corresponde a osso ou a vazio, conforme mostra a Figura 9.

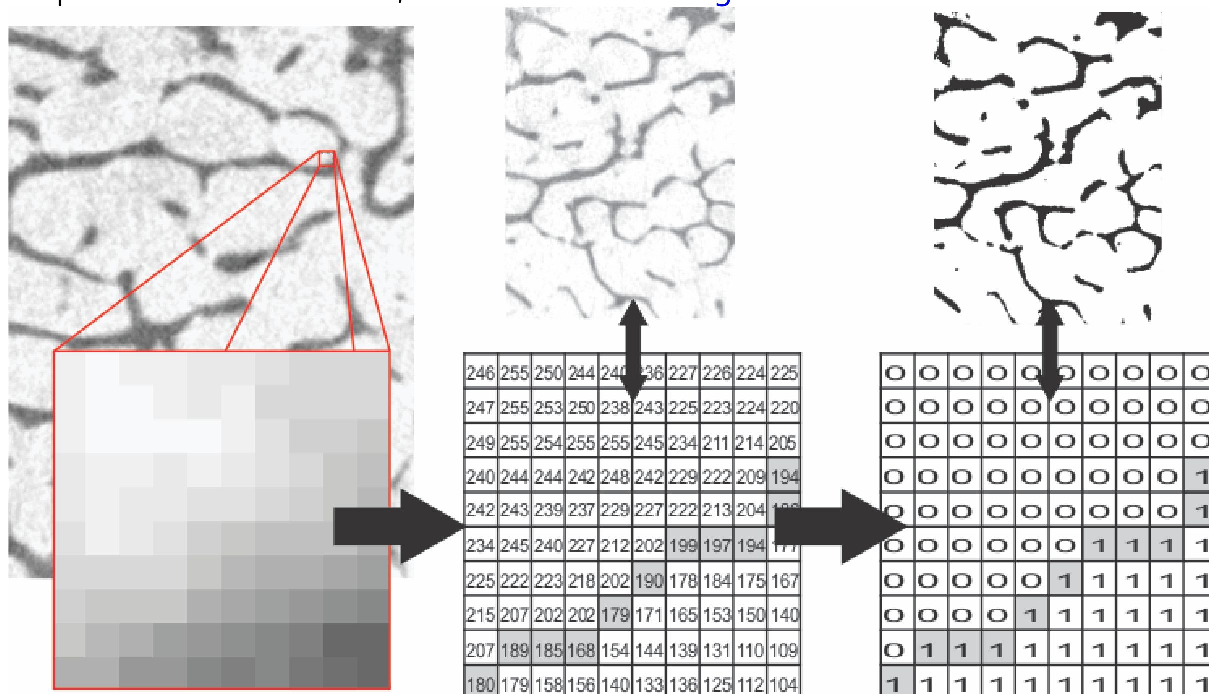

Figura 9: Processo de segmentação.

Para verificar o desempenho dos algoritmos desenvolvidos, utilizou-se, inicialmente, uma amostra relativamente pequena. Os dados utilizados inicialmente foram 189 imagens no formato bitmap (BMP) de tamanho 280 por 380 pixels cada. Isso resulta em um volume de dados correspondente a uma matriz tridimensional de 189x280x380, ou seja, 20.109.600 valores.

5.1 DIREÇÕES ANISOTRÓPICAS

A ilustração do uso da GPU para o processamento dos dados é feita, neste trabalho, com o uso do algoritmo para o cálculo das direções anisotrópicas e grau de anisotropia, baseado no momento de inércia de massa.

As direções anisotrópicas são definidas pelas direções principais do tensor de inércias calculado em função de eixos x, y e z, paralelos aos eixos X, Y e Z globais, que passam pelo centro de massa do volume de referência. As coordenadas do centro de massa CM(X,Y,Z) do volume de dados, foi feita com a integração do produto da massa de cada voxel pela distância x, y, ou z até a origem do sistema global, dividido o resultado pela massa total da amostra osso. Dessa maneira, foram necessárias cerca

Copyright © 2010 Asociación Argentina de Mecánica Computacional http://www.amcaonline.org.ar

de 20 milhões de operações de subtração, seguidas de produto e soma e, além disso, foi necessário avaliar em cada pixel se o valor corresponde a osso ou vazio.

Após a localização do centro de massa, foram calculados os momentos de inércia em cada um dos três eixos com origem no centro de massa, I_x, I_y e I_z, assim como os produtos de inércia P_{xy}, P_{yz} e P_{zx}, para a montagem do tensor de inercia do volume de referência (Argenta, Buriol e Hecke 2010). Os momentos de inércia de massa e os produtos de inércia do volume de referência são calculados com a soma dos momentos de inércia ou produtos de inércia dos voxels que representam osso, calculados em relação aos eixos que passam no centro de massa.

Com o tensor de inércias para o volume de referência calculado, as direções principais de inércia são obtidas com o cálculo dos autovalores e autovetores. Uma visualização rápida das direções principais pode ser feita utilizando-se o VTK, com a classe *vtkTensorGlyph*, e é ilustrada na Figura 10.

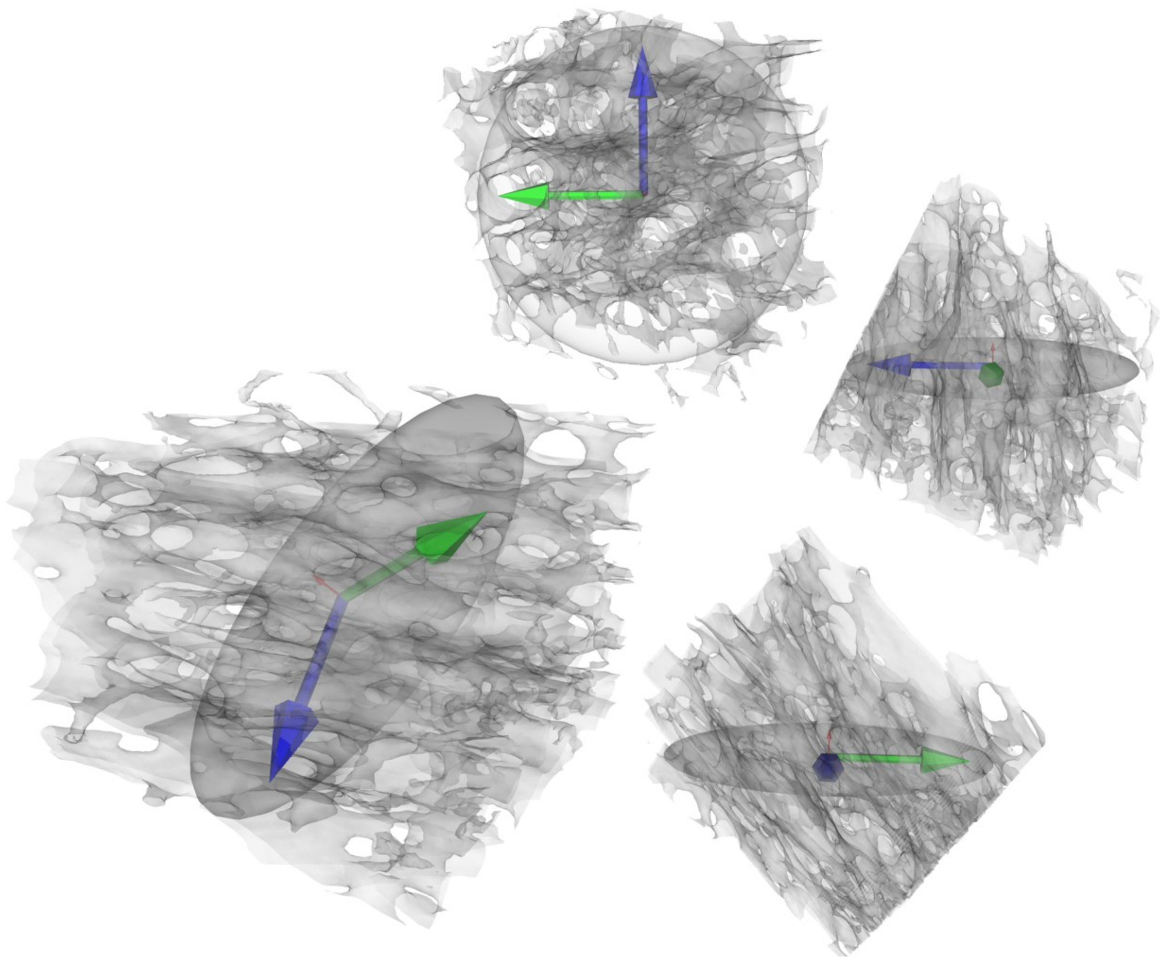

Figura 10: Eixos principais de inércia.

Todos esses parâmetros foram calculados usando, primeiramente, somente a linguagem Python e em seguida usando Numpy + pyCUDA, para comparação. Foram utilizadas duas plataformas, baseadas em PC, com configurações diferentes, a primeira (PC1) é um Intel Core 2 Quad, 3.2 GHz e 3.2GB de memória RAM com uma Nvidia GeForce 9600GT rodando sob Ubuntu 9.04 32bit. A segunda (PC2) é um Intel

Copyright © 2010 Asociación Argentina de Mecánica Computacional http://www.amcaonline.org.ar

Xeon Quad Core 3.8 GHz e 3.2 GB de memória RAM como uma Nvidia GTX 280 rodando sob o Ubuntu 9.04. A Tabela 1 ilustra uma comparação de tempos para as diferentes implementações nas duas plataformas.

	Python	Numpy + pyCUDA	Ganho de desempenho
PC1	44 min 25,163 s	2 min 50,325 s	15,65x
PC2	73 min 36,345s	11min 45,876 s	6,25x

Tabela 1: Comparação dos tempos de resolução do algoritmo.

Os resultados indicaram um ganho de desempenho elevado em ambas as plataformas para esse algoritmo.

6 CONCLUSÕES

A programação na GPU é complexa e demanda tempo e disponibilidade para o seu conhecimento e domínio. Facilitadores como o pyCUDA ajudam a executar essas tarefas de uma forma mais simples.

O fato de ser possível executar os cálculos em cada voxel da imagem para a obtenção de propriedades locais, que podem ser somadas para se obter uma propriedade global da amostra, faz com que esse tipo de análise seja uma ótima candidata para ser executada em paralelo. A conciliação de uma linguagem de altíssimo nível e de fácil implementação com técnicas inovadoras de programação em paralelo utilizando as GPUs, é, sem dúvida, uma excelente forma de se aliar produtividade com desempenho.

REFERENCIAS

Argenta, Marco André, Tiago Martinuzzi Buriol, e Mildred Ballin Hecke. "Metodologia para a obtenção de parâmetros físicos e geométricos do osso trabecular função de imagens de micro tomografia." *XXXI Iberian-Latin-American Congress on Computational Methods in Engineering*, 15-18 de Novembro de 2010: Artigo Proposto.

Coelho, Flávio Codeço. *Computação Científica com Python: Uma introdução à programação para cientistas.* 1. Petrópolis, RJ, 2007.

Fang, Qianqian, e David A. Boas. "Monte Carlo simulation of photon migration in 3D turbid media accelerated by graphics processing units." *Optics Express* 17, n. 22 (2009): 20187-20190.

Harrison, Noel M., Pat F. McDonnell, Denis C. O'Mahoney, Oran D. Kennedy, Fergal J. O'Brien, e Peter E. McHugh. "Heterogeneous linear elastic trabecular bone modelling using micro-CT attenuation data and experimentally measured heterogeneous tissue properties." *Journal of Biomechanics*, 2008: 2589–2596.

Kitware, Inc. "VTK Documentation." *VTK The Visualization Toolkit.* 2009. http://www.vtk.org/VTK/help/documentation.html.

Copyright © 2010 Asociación Argentina de Mecánica Computacional http://www.amcaonline.org.ar

Klöckner, A., T. Warburton, J. Bridge, e J. S. Hesthaven. "Nodal discontinuous Galerkin Methods on Graphics Processors." *Journal of Computational Physics* 228, n. 21 (Novembro 2009): 7863-7882.

Klöckner, Andreas. "PyCUDA v0.94rc documentation." *PyCUDA's documentation.* 19 de Junho de 2010. http://documen.tician.de/pycuda/.

Lu, Peter J., Hidenkazu Oki, Catherine A. Frey, Gregory E. Chamitoff, e et al. "Orders-of-magnitude performance increases in GPU-accelerated correlation of images from the International Space Station." *Journal of Real-Time Imaging Processing*, 2009.

Luebke, D. "CUDA: Scalable parallel programming for high-performance scientific computing." *5th IEEE International symposium on Biomedical Imaging: From Nano to Macro*, 14-17 de Maio de 2008: 836-838.

Mandelbrot, Benoit B. *The Fractal Geometry os Nature.* W. H. Freeman and Company, 1982.

Nagaraja, Srinidhi, Tracey L. Couse, e Robert E. Guldberg. "Trabecular bone microdamage and microstructural stresses under uniaxial compression." *Journal of Biomechanics*, 2005: 707–716.

NVIDIA. "CUDA C Programming Guide Version 3.1.1." *NVIDIA Corporation.* 2010. www.nvidia.com.

Python, Software Foundation. "Python documentation." *Python.* 12 de Setembro de 2010. http://docs.python.org/.

Scipy, Community. "Numpy and Scipy Documentation." *Scipy.org.* 2008. http://docs.scipy.org/doc/ (acesso em 15 de Junho de 2010).

Stoyan, Dietrich, e Elga Stoyan. *Fractals, Random Shapes and Point Fields: Methods of Geometrical Statistics.* John Wiley & Sons, 1994.

Stroustrup, B. *Exception Safety: Concepts and Techniques. Lecture Notes in Computer Science.* Edição: Christophe Dony, Joergen Lindskov Knudsen, Alexander Romanovsky e Anand Tripathi. 2001.

Tabor, Zbisław, e Eugeniusz Rokita. "Quantifying anisotropy of trabecular bone from gray-level images." *Bone*, 2007: 966-972.

Copyright © 2010 Asociación Argentina de Mecánica Computacional http://www.amcaonline.org.ar