Aplicando Processamento Paralelo com GPU ao Problema do Fractal de Mandelbrot

Bruno Pereira dos Santos[1], Dany Sanchez Dominguez[1], Esbel Valero Orellana[1].

[1]Departamento de Ciências Exatas e Tecnológicas – Campus Soane Nazaré de Andrade - Universidade Estadual de Santa Cruz (UESC) -
Km 16 Rodovia Ilhéus-Itabuna - CEP 45650-000 – ilhéus-Bahia – BA – Brasil

bruno.ps@live.com, dsdominguez@gmail.com, evalero@uesc.br

Abstract. *In many areas of science and engineering large computational problems are common. Parallel processing techniques are use to solve this problems. This article show the characteristics of parallel processing using graphics cards from Nvidia, with the help of CUDA framework. The CUDA/GPU techniques will be applied to the generation of Mandelbrot's fractal problem. We will show serial and parallel implementations, as well as validation of codes and comparative performance between approaches.*

Resumo. *Em diversas áreas da ciência e engenharia são comuns problemas computacionais de grande porte e, para a resolução dos mesmos, são necessárias técnicas de processamento paralelo. Este artigo tem como objetivo mostrar as características do processamento paralelo utilizando placas gráficas da Nvidia, com ajuda do framework CUDA, que será aplicado à geração computacional do fractal de Mandelbrot. Serão apresentadas a implementação serial e paralela, assim como a validação dos códigos e comparativo de desempenho entre os mesmos.*

1. Introdução

A redução do tempo necessário para solucionar um problema, bem como a resolução de problemas computacionais mais complexos e de maior porte tem sido o objetivo do processamento paralelo. Com essa finalidade utilizam-se diversos processos que cooperam entre si para a resolução de um problema específico. Nos últimos anos tem se difundido amplamente a utilização de arquiteturas de memória distribuída, como clusters de computadores, muitos deles equipados com modernos processadores que englobam várias unidades de processamento: os *cores*. Contudo uma nova técnica esta crescendo: o processamento paralelo em GPU (Graphical Processing Unit). Esta alternativa tem oferecido vantagens significativas como menor custo de implementação (hardware mais barato) e maior poder de cálculo. As novas GPUs tornaram-se massivamente paralelas e

possibilitam implementações heterogêneas em que as mesmas são utilizadas em conjunto com as tradicionais CPUs (Central Processing Unit).

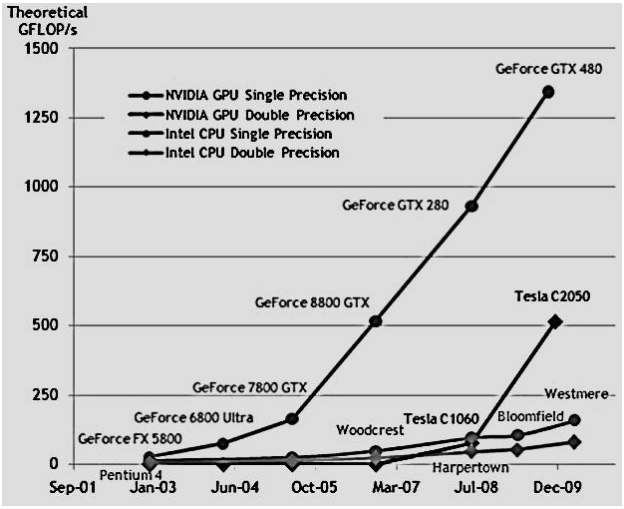

Figura 1 – Gráfico comparativo da evolução do desempenho em operações de ponto flutuante, para precisão simples e dupla, entre GPUs e CPUs. [Nvidia - 1].

Na Figura 1 podem-se observar diferentes gráficos que mostram a evolução do desempenho das GPUs e das CPUs nos últimos anos. Fica evidente que as modernas placas de processamento gráfico tem capacidade de processamento bem superior aos processadores mais modernos. Dada essa disparidade de desempenho e o baixo custo das GPUs, empresas e a comunidade científica tem absorvido a tecnologia. Em 2010, o maior supercomputador do mundo – Tiahne-1A com desempenho de 2,56 PFLOPS – tinha uma arquitetura heterogênea e incluía placas aceleradoras Nvidia Tesla série 2000 [TOP 500 2011]. A comunidade científica vem utilizando computadores, providos de placas GPU, em seus trabalhos nas mais diversas áreas como soluções de problemas de engenharia nuclear, física médica, base de dados para sistemas online, bioinformática, engenharia genética dentre outros[Aiping D, 2011] [Alonso P. 2009], [Goddeke D. 2007].

Os termos GPGPU (acrônimo de General-purpose Computing on Graphics Processing Units) e GPU Computing (computação GPU) se equivalem, e foram definidos por Mark Harris em 2002, ano em que reconheceu o uso das GPUs em aplicações não gráficas [GPGPU.org 2011]. Por serem difíceis de programar para fins genéricos como resolução de equações, foi criado um framework para as unidades gráficas atuais, desenvolvido por diversas empresas como Apple Inc, AMD, Intel, Nvidia, ATI dentre outras. Este framework é conhecido como OpenCL (Open Computing Language) [Framework OpenCL 2011] que visa por em pratica o GPGPU utilizando um padrão de escrita de programas para execução em maquinas heterogêneas.

Por outro lado alguns fabricantes de hardware desenvolveram suas próprias ferramentas para facilitar a utilização das GPUs em aplicações científicas. Dentre estas

implementações destaca-se a arquitetura CUDA (Compute Unified Device Architecture), criada pela NVIDIA [Nvidia - 2]. A arquitetura CUDA pode ser utilizada em conjunto com linguagens de programação bem conhecidas e consolidadas no meio acadêmico, como Fortran, C e C++. CUDA oferece uma API (Application Programming Interface) de baixo nível, chamado de Driver, responsável pela comunicação com o dispositivo. Para programação em mais alto nível tem-se CUDA runtime e bibliotecas que fornecem funções otimizadas de álgebra linear.

Neste artigo vamos dar ênfase na utilização de CUDA para desenvolvimento de aplicações paralelas. O problema computacional adotado para esse trabalho foi a geração do fractal de Mandelbrot, que permitirá compararmos o desempenho de processadores GPU e CPU em aplicações científicas. Para esta finalidade serão geradas imagens do fractal com diferentes resoluções. Neste caso técnicas de processamento paralelo em GPU podem ser aplicadas para aumentar a velocidade com que são geradas estas imagens.

Para realizar as análises serão feitas uma implementação serial em C e uma implementação paralela para GPU utilizando C para CUDA. O objetivo desse trabalho é apresentar os resultados obtidos com estas duas implementações para diferentes resoluções das imagens geradas.

A primeira seção deste artigo abordará os fundamentos do problema, bem como a formulação do fractal de Mandelbrot e o algoritmo em português estruturado que descreve a recorrência. Posteriormente será discutida a aplicação em GPU, introduzindo os conceitos da extensão CUDA e exibição do algoritmo implementado em C e CUDA. Na seção 5 são apresentados os resultados obtidos através de um gráfico comparativo entre os códigos serial e paralelo. Finalmente serão apresentadas as conclusões e as recomendações para trabalhos futuros.

2. Fundamentos do Problema

A teoria fractal tem sua origem na descoberta do matemático alemão Karl Weierstrass que encontrou uma função com a propriedade de ser contínua em todo seu domínio, mas em nenhum ponto diferenciável. As plotagens dessas funções eram difíceis, pois elas são recursivas, então o trabalho manual era praticamente impossível. Com o advento do computador o professor Benoît Mandelbrot foi o primeiro a utilizar a máquina para plotar a função recursiva estudada por Pierre Fatou, hoje chamada de Conjunto de Mandelbrot ou simplesmente Fractal de Mandelbrot.

O conjunto de Mandelbrot é definido como o conjunto específico de pontos do plano complexo de Argand-Gauss que obedecem a distância máxima de 2 da origem do plano, isto é, “não tendem ao infinito” para a sequência definida pela recorrência do número complexo $Z = x + yi$:

$$Z_0 = 0$$
$$Z_{n+1} = Z_N^2 + C$$

Onde Z_0e $Z_{\{n+1\}}$são iterações n e n+1 do numero complexo Z, e $C = a + bi$ fornece a posição de um ponto do plano complexo. Já a parte real e imaginária do complexo Z pode ser desenvolvida até encontrarmos $x_{n+1} = x_n^2 - y_n^2 + a$ e $y_{n+1} = 2x_n y_n + b$. Para calcular os pontos do fractal pode-se utilizar o seguinte algoritmo:

```
int Mandelbrot(complexo c){
int i = 0, ITR = 255;
float x = 0, y = 0, tmp = 0;
enquanto (x²+y² <= 2² && i < ITR) {
tmp = x² - y² + c. real;
y = 2*x*y + c.img;
i++;
}
se(i < ITR)   retorne i;
senão     retorne 0;
}
```

A geração da imagem de um fractal dependerá da quantidade de pontos no domínio, neste caso distância máxima de |2| da origem, e o número máximo de iterações para determinar se o ponto pertence ou não ao conjunto. Para se obter resoluções aceitáveis, isto é, imagens onde é possível observar o padrão de similaridade multi-escala, imagens com resolução maiores que 1200x1200 devem ser utilizadas. Desta forma temos um problema que exige um grande quantidade de operações em função da resolução do fractal que se deseja obter.

3. Programação para GPU

Utilizando as placas aceleradoras gráficas na prática de programação paralela para fins gerais, a NVIDIA, a partir das séries GeForce 8, Quadro e Tesla, traz em seu conjunto software e hardware (CUDA), soluções para o paradigma de programação para placas de vídeo. CUDA é uma extensão do C que permite programar em paralelo aproveitando os diversos multiprocessadores das placas NVIDIA. As GPUs habilitadas com CUDA são enquadradas em um novo modelo de arquitetura chamada de SIMT (Single Instruction Multiple Thread). Com ajuda da arquitetura CUDA é possível programar e gerenciar as linhas de processamento.

Um código produzido em CUDA, quando compilado e executado, é dividido em código host (que será processado na CPU) e código device (que será processado em

paralelo na GPU). Então o fluxo de execução inicia-se, com os comandos do host, até uma chamada para um kernel (código device) que transfere o comando do fluxo para device e inicia a execução em paralelo. Quando termina a execução paralela o device devolve o controle de volta para o host dando seguimento à aplicação. Em resumo, todo código produzido com CUDA, segue uma sequência como mostrado na Figura 2.

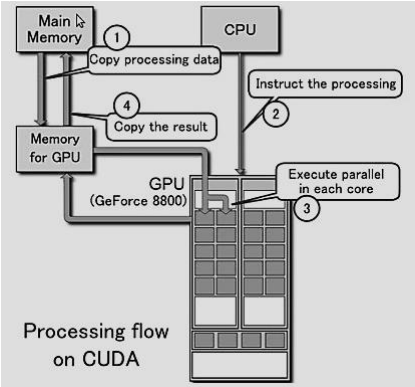

Figura 2 – Sequência de execução do código CUDA [Nvidia - 1].

Para o processamento do kernel, CUDA oferece uma organização para o gerenciamento dos threads da seguinte forma: threads são agrupados formando blocos (menor unidade de processamento) que por sua vez são organizados em um grid como mostrado na Figura 3.

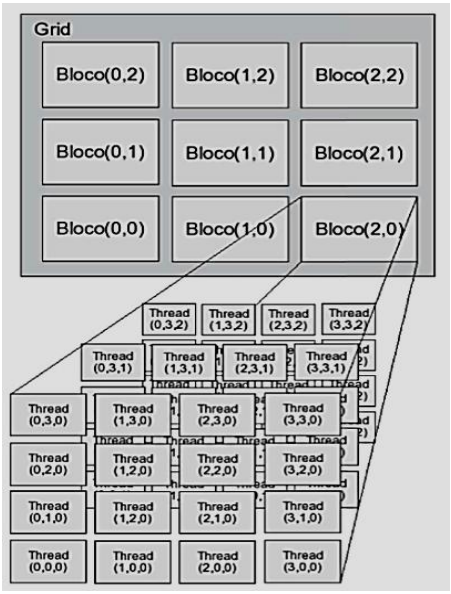

Figura 3 – Organização do Grid, blocos e threads [Almeida, A 2009].

Os blocos de threads podem ser organizados em até 3 dimensões. Já os grids podem ser alocados em até 2 dimensões, como observado na Figura 3. O dispositivo utilizado para a realização do trabalho possui algumas limitações. Cada bloco possui no máximo 512, 512 e 64 threads nos eixos x, y e z respectivamente. Já cada grid pode conter até 65535 blocos

em cada um dos eixos x e y. Também existem 14 multiprocessadores com 8 núcleos, num total de 112 *cores*.

No que diz respeito ao gerenciamento, a API da plataforma CUDA traz diferentes extensões à linguagem C que são:

1) Qualificadores de tipo:
   a) Funções – indicam se o código será executado no host ou device.
   b) Variável – indica em qual memória será armazenada, na CPU ou GPU.
2) Identificadores de variáveis threads – para a identificação dos threads, na organização do kernel lançado.
3) Nova sintaxe para a chamada das funções executadas na GPU.

Para qualificadores de tipo de função existem:

1) __global__, para funções que serão executadas no device (também conhecidas como kernels) e que são chamadas a partir do host.
2) __device__ para rotinas que vão ser executadas no device e somente podem ser chamadas a partir do mesmo.
3) __host__ para funções que serão chamadas a partir da CPU e somente executadas pela CPU.

Já para qualificadores de variáveis tem-se:

1) __device__ define uma variável que reside na memória global da GPU e pode ser acessada por todos os threads do grid e da CPU através da biblioteca runtime.
2) __constant__ difere da __device__ somente em sua localização que é na memória constante da GPU.
3) __shared__ define uma variável shared que reside na memória compartilhada da GPU, esta é acessível por todos os threads do mesmo bloco e tem o tempo de vida do bloco.

No que tange a identificação de cada thread na execução de um kernel a API oferece algumas variáveis:

1) threadIdx.x, threadIdx.y e threadIdx.z, retorna o índice da thread de acordo com a dimensão a que se refere.
2) Para a identificação dos blocos dentro do grid usamos blockIdx.x, blockIdx.y.

Temos ainda identificadores do tamanho das dimensões nos grid e blocos, para isso podem ser utilizadas as variáveis blockDim.'dimensão' até 3 e gridDim.'dimensão' até 2.

Para a chamada de funções executáveis na GPU basta utilizarmos a sintaxes: nome_funcao<<grids, blocos>>(parametros). Existem ainda variáveis dim3 nome_variavel(x, y, z) para definir blocos de 3 dimensões a serem lançados para o kernel.

4. Implementação dos algoritmos

Após o estudo detalhado das características da extensão C para programação de GPUs NVIDIA, utilizando CUDA, foram implementadas duas versões do algoritmo de geração do fractal: uma implementação serial utilizando C e uma implementação paralela utilizando

CUDA. A sub-rotina de cálculo nas duas implementações é apresentada a seguir. A rotina do código C recebe um argumento chamado complexo, que é o ponto do plano em que se verificará a pertinência ao conjunto de mandelbrot. Isto é realizado através de um loop que se repete até que a quantidade máxima de iterações seja ultrapassada ou o modulo do ponto complexo seja maior que dois ($|Z_n| > 2$). Já na rotina em CUDA as variáveis tx e ty identificam um thread num plano imaginário que fará o papel de um ponto para posterior verificação no loop while de forma semelhante ao processamento na versão serial em C.

Os códigos produzidos para a geração do fractal de Mandelbrot seguem o seguinte raciocínio:

1) Alocação de memória: do tipo char por dois motivos, menor custo de memoria e a mesma escala de representação numérica das cores de um pixel numa imagem que é 255. Na versão paralela a alocação é feita na memória global da placa gráfica.
2) Função que recebe um elemento da matriz e retorna a cor do ponto caso não pertença ao conjunto ou ausência de cor caso contrária. Na versão paralela tal função será executada na GPU.

As rotinas em C e em CUDA usam a mesma lógica apresentada no algoritmo do fractal supracitado. O algoritmo utilizado é completamente paralelizável ficando apenas as operações de E/S executadas serialmente. A versão paralela difere nos qualificadores especiais e nos parâmetros, que representam a matriz de char, a distância de um pixel para outro e o tamanho da imagem. Abaixo serão mostrados os códigos das rotinas que calculam o fractal em CUDA e C respectivamente. A esquerda o código paralelo e a direita o serial:

```
__global__  void ponto_fractal(complexo ini,
       char *plano_d,  float dx, float dy, int altura,
       int compr){
   int tx = threadIdx.x + blockIdx.x * blockDim.x;
   int ty = threadIdx.y + blockIdx.y * blockDim.y;
  float real, img;
  real = ini.real + (dx*tx);
  img = ini.img – (dy*ty);
  int i = 0;
  float x = 0; y = 0, tmp;
  while(x*x + y*y  <= 4 &&  i < ITR){
     tmp =  x*x – y*y + real;
     y = 2*x*y + img;
     x = tmp;
     i++;
  }
  if(i<ITR)
    plano_d[tx*comprimento +ty] = i;
  else
    plano_d[tx*comprimento +ty] = 0;
}
```

```
void ponto_fractal(complexo ini){
  int i = 0;
  float x = 0; y = 0, tmp;
  while(x*x + y*y  <= 4 &&  i < ITR){
     tmp =  x*x – y*y + real;
     y = 2*x*y + img;
     x = tmp;
     i++;
  }
  if(i<ITR)
    return = i;
  else
    return = 0;
}
```

5. Resultados e discussão

Para realizar os experimentos numéricos foi utilizada uma estação de trabalho, equipada com uma placa NVIDIA, com as seguintes características: processador intel (R) Core i7 CPU 860 2,8GHz; HD de 250GB; memória RAM 8GB e placa aceleradora gráfica GPU Nvidia GeForce 9800GT com 112 cores, 512 de RAM, 256bits PCI Express 16x.

Para gerar os executáveis a NVIDIA disponibiliza o nvcc [Nvidia - 4], um compilador para códigos que utilizam a biblioteca CUDA. Já o código serial foi compilado utilizando o gcc. Podemos destacar que na compilação foi utilizada a otimização nível dois (-O2), com isso obtém-se um código mais enxuto e eficiente.

Compilados e otimizados os códigos foram submetidos a testes e posteriormente os tempos de cada versão foram tomados para diversos tamanhos de imagem no intervalo de 1.200x1.200 px à 20.000x20.000 px. Para cada tamanho de imagem foram feitas 5 tomadas de tempo e o valor médio destes tempos foi utilizado no cálculo do speedup. Na Figura 4, o gráfico mostra o tempo em segundos para o processamento do fractal em diferentes resoluções (1 – 1.200x1.200 px, 2 - 5.000x5.000 px, 3 – 10.000x10.000 px, 4 – 15.000x15.000 px, 5 – 20.000x20.00 px) . A curva segmentada descreve o processamento do aplicativo serial, já a curva contínua o aplicativo paralelo. Fica evidente a redução de tempo de execução com a utilização da GPU, sobre tudo para imagens de maior resolução.

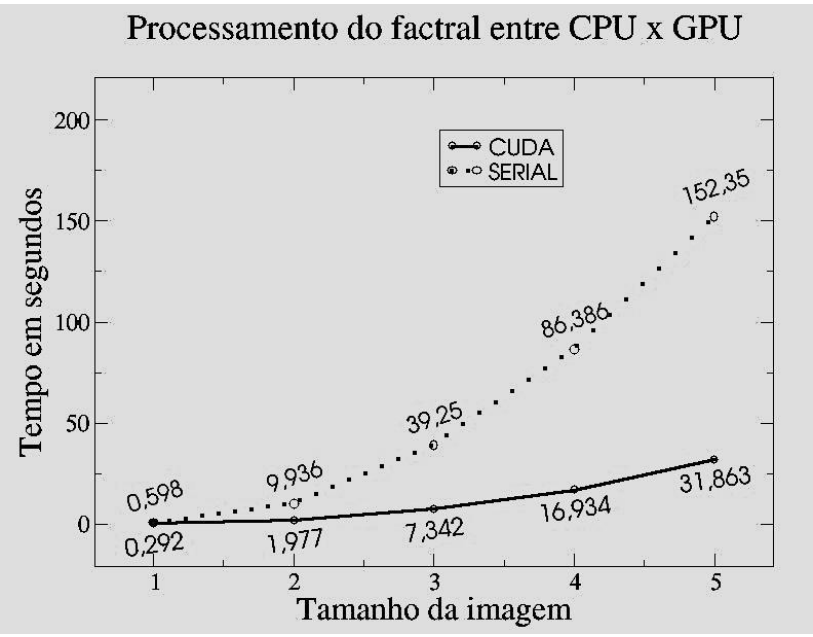

Figura 4 – Comparativo do tempo de processamento entre os códigos serial e paralelo.

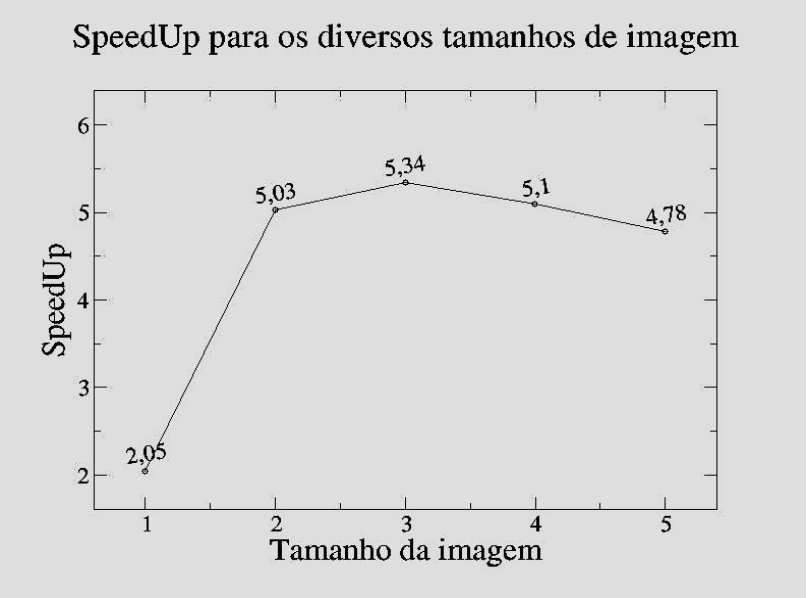

Figura 5 – Speedup para diversos tamanhos de imagem.

O gráfico da figura 5 mostra o speedup calculado para cada resolução utilizando o tempo do aplicativo serial e o tempo obtido com o aplicativo paralelo utilizando a GPU. Os resultados mostram um speedup de aproximadamente 5 para a maior parte dos casos. Apenas a geração da imagem de menor resolução não teve ganho de desempenho tão significativo, o que deve estar relacionado com o fato de que o tempo que se perde transferindo os dados da memória RAM para a GPU e vice versa é grande em relação ao ganho de desempenho. Observa-se também uma tendência da curva de speedup diminuir à medida que as imagens vão ficando maiores. Neste sentido se faz necessário aprimorar a implementação do código paralelo para verificar, se possível, reverter este comportamento.

6. Conclusões e Trabalhos futuros

Após a implementação do algoritmo paralelo de geração do fractal de mandelbrot, podemos concluir que o maior desafio para os programadores de C para CUDA é a absorção das diretivas introduzidas pela biblioteca CUDA, bem como o novo modelo de arquitetura e a programação com threads. Contudo essas barreiras estão sendo vencidas com frameworks para padronização dos códigos, difusão da tecnologia no meio acadêmico e introdução desses recursos em computadores de grande porte.

Quanto ao desempenho do aplicativo pode-se concluir que a implementação do fractal de mandelbrot em CUDA obteve ganho significativo de desempenho quando comparado com a versão serial. Estes resultados ainda podem ser otimizados através de novas implementações em CUDA usando melhores práticas de programação [Nvidia - 3]. Além da melhora de desempenho, a utilização de GPU reduz os custos com hardware e melhora a utilização do espaço físico.

Em trabalhos futuros deve-se utilizar MPI e OPENMP, técnicas tradicionais de processamento paralelo, para implementar outras versões deste código. Com isso espera-se poder realizar um comparativo entre as diversas arquiteturas existentes, isto é, serial *versus* paralela (MPI e OPENMP), paralela (CUDA) *versus* paralela (MPI e OPENMP). Finalmente deve-se trabalhar na criação de um código heterogêneo utilizando a plataforma CUDA agregada às técnicas de MPI e OPENMP, para aproveitar melhor as estruturas híbridas disponíveis hoje nos cluster de computadores.

7. Referências

Aiping Ding, Tianyu Liu, Chao Liang, Wei Ji, and X George Xu (2011) "EVALUATION OF SPEEDUP OF MONTE CARLO CALCULATIONS OF SIMPLE REACTOR PHYSICS PROBLEMS CODED FOR THE GPU/CUDA ENVIRONMENT".

Almeida, A. (2009) "DESENVOLVIMENTO DE ALGORITMOS PARALELOS EM GPU PARA RESOLUÇÃO DE PROBLEMAS NA ÁREA NUCLEAR". Disponivel em: http://www.ien.gov.br/posien/teses/dissertacao_mestrado_ien_2009_07.pdf

Alonso, P., Cortina, R, Martínez-Zaldívar, F. J., Ranilla, J. (2009) "Neville elimination on multi- and many-core systems: OpenMP, MPI and CUDA, J. Supercomputing", in press, doi:10.1007/s11227-009-0360-z, SpringerLink Online Date: Nov. 18.

Goddeke D, Strzodk R, Mohd-Yusof J, McCormick P, H.M Buijssen S, Grajewski M. e Turek S. (2007) "Exploring weak scalability for FEM calculations on a GPU-enhanced cluster".

GPGPU.org (2011). Disponível em: http://gpgpu.org/about/, Março.

NVIDIA Corporation, (2011) "NVIDIA CUDA C ProgrammingGuide 3.1." Disponível em: http://developer.nvidia.com/object/cuda_download.html, Março.

NVIDIA Corporation, (2011) "O que é CUDA". Disponível em: http://www.nvidia.com.br/object/what_is_cuda_new_br.html, Março.

NVIDIA Corporation, (2011) "NVIDIA OpenCL Best Practices Guide". Disponível em: http://developer.download.nvidia.com/compute/cuda/3_2_prod/toolkit/docs/OpenCL_Best_Practices_Guide.pdf, Março.

NVIDIA Corporation, (2011) "The CUDA Compiler Driver NVCC", http://moss.csc.ncsu.edu/~mueller/cluster/nvidia/2.0/nvcc_2.0.pdf, Março.

OpenCL, (2011) "Framework OpenCL". Disponível em: http://www.khronos.org/opencl/, Março.

TOP500, (2011) "Rank dos maiores supercomputadores do mundo". Disponível em: http://www.top500.org/list/2010/11/100, Março.