O Problema do Fractal de Mandelbrot como Comparativo de Arquiteturas de Memória Compartilhada – GPU vs OpenMP

Bruno P. dos Santos, Dany S. Dominguez, Esbel V. Orellana

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 work 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. In addition we generate results using OpenMP techniques. 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 aparecem 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. Foram desenvolvidos códigos paralelos usando CUDA e OpenMP, estes códigos são comparados em relação a seu desempenho o que permite caracterizar as técnicas de GPU e de programação multi-thread.*

1. Introdução

O objetivo principal do processamento paralelo é a redução do tempo necessário para solucionar um problema computacional quer seja complexo ou não. Com essa finalidade utilizam-se diversos processadores que cooperam entre si para a resolução do dado problema específico. Nos últimos anos tem se difundido amplamente a utilização de arquiteturas de memória compartilhada, onde um único processador engloba vários núcleos de processamento: os *multi-cores*. Contudo uma nova técnica esta crescendo: o processamento paralelo em GPU (GraphicalProcessing Unit). Esta alternativa tem oferecido vantagens significativas como menor custo de implementação (hardware mais barato) e maior poder de cálculo e reduzido espaço de ocupação. As novas GPUs tornaram-se massivamente paralelas (englobando, neste hardware, até centenas de processadores, *many-*

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

É notória a evolução do desempenho das GPUs e das CPUs nos últimos anos, com destaque as GPU's que recentemente alcançam a casa dos 1200 GFLOPs de processamento [Nvidia 1]. Ficando evidente que as modernas placas de processamento gráfico tem capacidade de processamento bem superior aos processadores comuns 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. 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) 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 uma aplicação paralela, para a resolução do problema computacional, fractal de Mandelbrot, que permitirá compararmos o desempenho de processadores GPU e CPU nesta aplicação científica. Para esta finalidade serão geradas imagens do fractal com alta

resolução. Para tanto técnicas de processamento paralelo em GPU e CPU podem ser aplicadas para aumentar a velocidade com que são geradas estas imagens.

Para realizar as análises foram feitas uma implementação paralela usando C e técnicas de programação multi-thread OpenMP e para GPU utilizando CUDA. Vale ressaltar que os códigos construídos foram executados em hardwares distintos, segundo o framework utilizado. Uma versão paralela para resolver o problema do Fractal de Mandelbroot utilizando CUDA foi apresentada e caracterizada no trabalho (Santos B., 2011). O objetivo desse trabalho é aprimorar os resultados obtidos no comparativo das versões CUDA e SERIALCUDA, e utilizar a versão em OpenMP e SERIALOpenMP para apresentar os resultados com estas duas implementações paralelas a fim de compara-las quanto ao desempenho na geração de imagens de alta resolução.

A próxima 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 exibindo o algoritmo implementado em CUDA. Na seção 5 são apresentados os resultados obtidos através de gráficos de speedup comparativo para os códigos paralelos usando GPU e OpenMP. 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 uma 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. A plataforma CUDA nada mais é que 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 InstructionMultiple 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 essa sequência.

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 1.

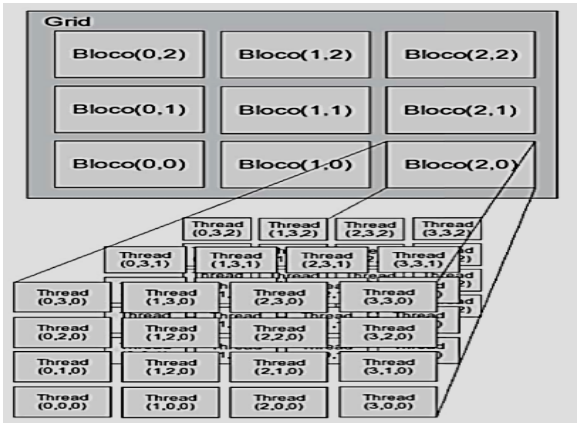

Figura 1 – Organização do Grid, blocos e threads [Almeida, A 2009].

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 2. 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 tange as extensões da linguagem C, a plataforma CUDA oferece diversos qualificadores para indicar qual função será executada na GPU\CPU, bem como a localização das variáveis no hardware gráfico e identificadores para localização das threads dentro da organização do kernel acima citado. Para funções temos os qualificadores __global__, __device__ e __host__ indicando respectivamente, as funções que serão executadas na GPU, as que são chamadas de dentro da GPU e finalmente funções que somente serão executadas na CPU. Já para variáveis tem-se __device__, __constant__ e__shared__ para respectivamente variáveis que residirão na memória principal da GPU, variáveis que serão alocadas na memória constante do dispositivo e shared serão colocadas a disposição numa memória mais rápida acessíveis somente por threads dentro de um bloco. Para identificar os threads utiliza-se threadidx.x, threadidx.y e threadidx.z para localiza-las dentro de um bloco e para localizar os blocos dentro de um grid usa-se blockidx.x e 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. Implementaçãodos algoritmos

Após o estudo detalhado das características da extensão C para programação de GPUs NVIDIA, utilizando CUDA, foi implementada a versão paralela em CUDA e uma em OpenMP para a geração do fractal. Nesta seção damos ênfase a sub-rotina de cálculo da versão paralela em CUDA. A rotina recebe um argumento chamado complexo, que é um ponto do plano em que se verificará a pertinência ao conjunto de Mandelbrot usou-se as variáveis tx e ty identificam um thread num plano imaginário que fará o papel de um ponto para posterior verificação no loop while 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).

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 em CUDA 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 paralelizáveis 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 é mostrado o código da rotina que calcula o fractal em CUDA:

```
__global__voidponto_fractal(complexo ini,
char *plano_d,  float dx, float dy, intaltura,
intcompr){
inttx = threadIdx.x + blockIdx.x * blockDim.x;
intty = 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;
}
```

5. Resultados e discussão

Para realizar os experimentos numéricos na versão em CUDA 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. Destacamos que das placas Nvidia disponíveis no mercado com recursos GPU está é a que apresenta menos recursos. Para os experimentos numéricos em OpenMP usou-se uma estação de trabalho com as seguintes especificações: 8 nos Genuine Intel ia-64, modelo Madison com 9M cachê e 16GB de memória RAM compartilhada.

Para gerar os executáveis a NVIDIA disponibiliza o nvcc [Nvidia - 4], um compilador para códigos que utilizam a biblioteca CUDA. Já o código em OpemMP foi compilado utilizando a versão 4.4.6 do gcc. Podemos destacar que na compilação foi utilizada a otimização nível dois (-O2), com isso obtém-se um código mais enxuto e eficiente.

Feita a compilação os executáveis obtidos foram submetidos a diversas tomadas de tempo para a geração de imagens de alta resolução. Para a aplicação paralela em GPU foram feitas imagens de diversos tamanhosvariando de 1200 x 1200 até 2000 x2000 como mostradona figura 2 e em seguida, na figura 3, o speedup para os diversos tamanhos. Destacamos que nestes experimentos o número máximo de iterações para um ponto do fractal foi escolhido em 256.

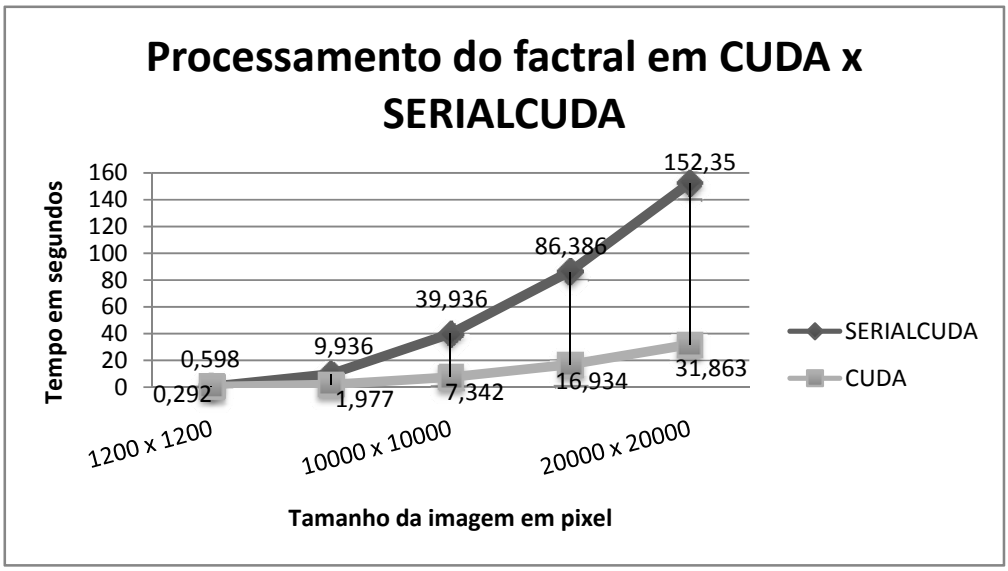

Figura 2 – Comparativo do tempo de processamento entre os códigos serialcuda e paralelo.

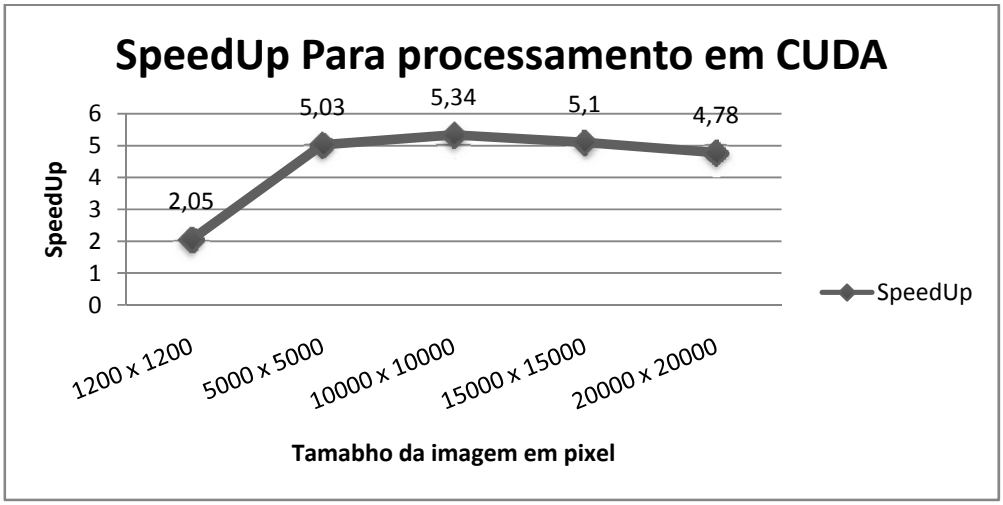

Figura 3 – Speedup para processamendo em CUDA para diversos tamanhos de imagem.

O gráfico da figura 5 mostra o speedup calculado para cada resolução utilizando o tempo do aplicativo serialcuda 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 desempeno. 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.

Para a versão em OpenMP e sua respectiva implementação SERIALOpenMP foram realizados testes com uma resolução fixa de 16384 x 16384 fixando a quantidade máxima de iterações de 4096. Na figura 4, observa-se o speedup obtido na geração da imagem do fractal com alta resolução e a variação da quantidade de threads utilizadas na execução da implementação paralela tradicional utilizando a cláusula Schedule do tipo Dynamic.

Da analise da figura 4 podemos constatar que o código OpenMP alcança um speedup próximo do linear e na medida que aumentamos o número de threads nos afastamos do speedup linear.

Para efetuarmos o comparativo entre as implementações paralelas executamos o algoritmo em CUDA e seu respectivo SERIALCUDA com a resolução de 16384 x 16384 e quantidade máxima de iterações de 4096 no algoritmo de geração da imagem. Para cada um dos códigos calculou-se o speedup e o comparativo é apresentado na no gráfico da figura 5.

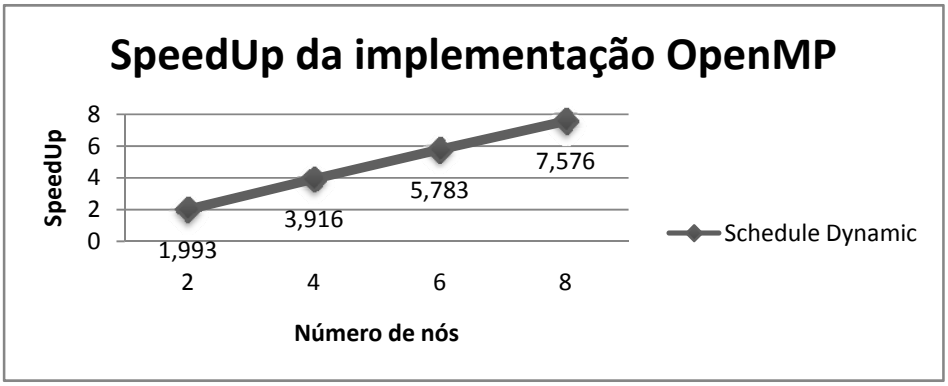

Figura 4 – SpeedUp para a implementação do algoritmo em OpenMP em imagens de resolução 16384 x 16384 para diferentes números de threads.

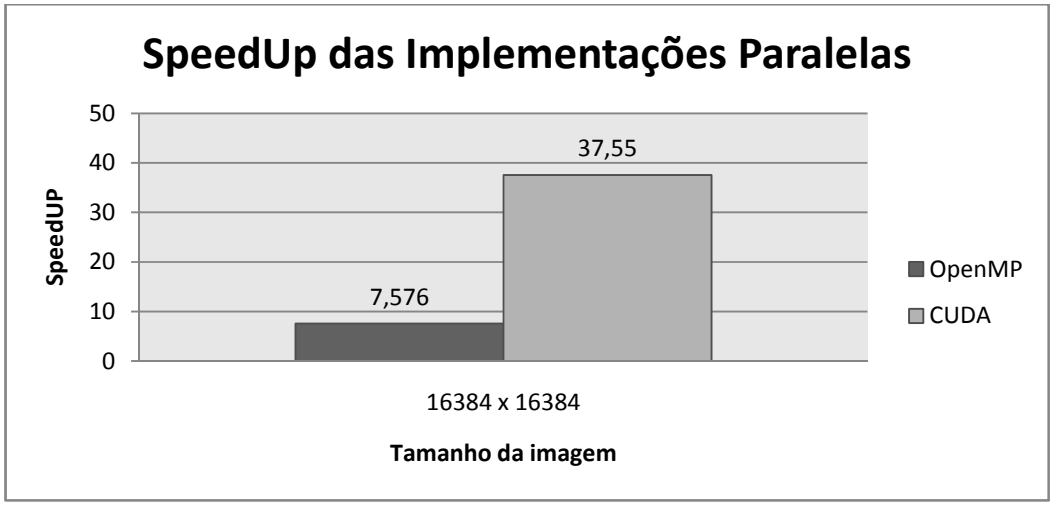

Figura 5 – Speedup comparativo entre os códigos paralelos GPU e OpenMP para imagens de 16384 x 16384 e numero de iterações de 4096.

Observamos na figura 5 que para o caso analisado o speedup conseguido pela implementação CUDA é muito superior que o speedup da versão OpenMP, pelo que podemos verificar a superioridade das aplicações paralelas que utilizam GPU em relação as máquinas de memória compartilhada convencionais.

6. Conclusões e Trabalhos futuros

Após a implementação dos algoritmos paralelos 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 alto número de 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.

No que tange o desempenho, o software produzido com a técnica CUDA obteve ganho significativo quando comparado com a versão em OpenMP o que manifesta a superioridade do paradigma GPU sobre a programação multi-thread convencional. Observa-se ainda que o paralelismo em placas gráficas deva ser ainda mais estudado e dissipado tanto no meio acadêmico quanto no empresarial, pois além de seu custo ser muito inferior a um hardware paralelo tradicional, a alocação de espaços para armazenamento da máquina é drasticamente reduzido.

Para trabalhos futuros, devemos considerar a implementação de uma versão usando MPI e uma versão híbrida que combine MPI e OpenMP. Para então comparar o desempenho destas versões com relação à implementação que utiliza GPU-CUDA.

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) “Nevilleelimination 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.

Santos Bruno P.; Dany S. Domínguez ; Esbel T. V. Orrellana . Aplicando processamento paralelo com GPU ao problema do Fractal de Mandelbrot. Anais da XI Escola Regional de Computação Bahia Alagoas Sergipe - ERBASE, Salvador, 2011.