Paralelização do Problema de Transferência de Calor Utilizando OpenMP e CUDA

(*Relatório Técnico*)

Guilherme Galante

Departamento de Informática – Universidade Federal do Paraná (UFPR)
Caixa Postal 19.081 – CEP 81.531-980 – Curitiba – PR – Brasil

ggalante@c3sl.ufpr.br

1. Introdução

A simulação de fenômenos físicos é de grande importância no desenvolvimento de novas tecnologias em diversos setores da economia. Tais fenômenos podem ser modelados matematicamente e para serem resolvidos com alta qualidade numérica, requerem grande capacidade de processamento.

Uma alternativa para suprir esta demanda é o uso de técnicas de computação de alto desempenho, onde atualmente destacam-se a exploração de *multithreading* utilizando OpenMP em processadores com múltiplos núcleos (*multicore*) e o uso de CUDA em placas gráficas Nvidia. Sob este escopo, este trabalho tem como objetivo apresentar a paralelização do problema da transferência de calor utilizando estas tecnologias, bem como seus respectivos resultados.

2. Solução do Problema da Transferência de Calor

Um problema clássico de aplicação de métodos numéricos em fenômenos físicos é a transferência de calor em uma placa plana homogênea e isotrópica. O processo de transferência de calor em uma placa retangular ocorre pela troca de calor entre partículas do material de um ponto com mais energia para outro com menos energia. Esse processo é conhecido como condução do calor [Lienhard IV e Lienhard V 2008].

Considerando que todos os pontos internos da placa (Figura 1a) estejam a uma temperatura inicial diferente das temperaturas das bordas T_A, T_B, T_C e T_D, o problema é determinar a temperatura em qualquer ponto interno da placa (T_i, j) em um dado instante de tempo.

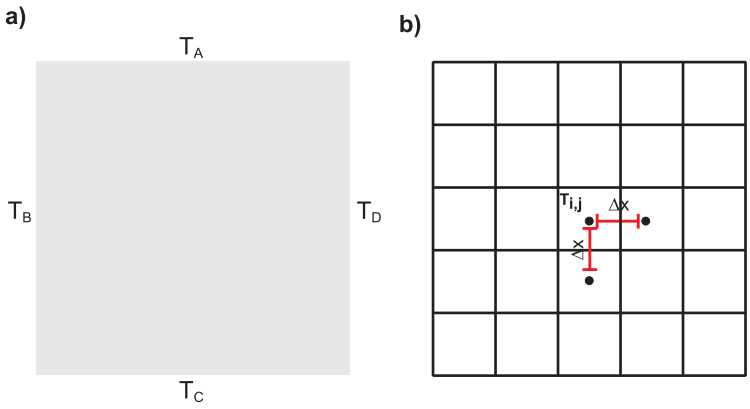

Figura 1. Placa Plana (a) e Domínio Discretizado (b)

Para uma placa isotrópica, as propriedades térmicas do material são iguais em qualquer direção, então, o modelo matemático que relaciona as variáveis do problema é a equação da energia [Borges e Padoin 2006, Lienhard IV e Lienhard V 2008]:

$$\frac{\partial T}{\partial t} = \alpha \left(\frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2} \right)$$

onde:

- T é a temperatura (em graus Celsius - ^oC);
- x e y são as variáveis espaciais (dada em metros - m);
- t é a variável temporal (dada em segundos - s);
- α é a difusividade térmica (dada em metros ao quadrado por segundo - m^2/s);

Considerando os espaçamentos da malha $\Delta x = \Delta y$, como mostrado na Figura 1b, e aproximando a equação diferencial parcial por meio do método das diferenças finitas usando a expansão da Série de Taylor, como descrito em Fortuna (2000), obtém-se:

$$\left(1 + \frac{4\alpha\Delta t}{\Delta x^2} T_{i,j}^{n+1}\right) - \frac{\alpha\Delta t}{\Delta x^2} T_{i-1,j}^{n+1} - \frac{\alpha\Delta t}{\Delta x^2} T_{i+1,j}^{n+1} - \frac{\alpha\Delta t}{\Delta x^2} T_{i,j-1}^{n+1} - \frac{\alpha\Delta t}{\Delta x^2} T_{i,j+1}^{n+1} = T_{i,j}^{n} \qquad (1)$$

Esta equação é convertida em um stêncil (ou molécula) computacional que é aplicada às k células do domínio. Cada célula do domínio gera uma linha do sistema de equações. Uma ilustração deste procedimento é mostrado na Figura 2.

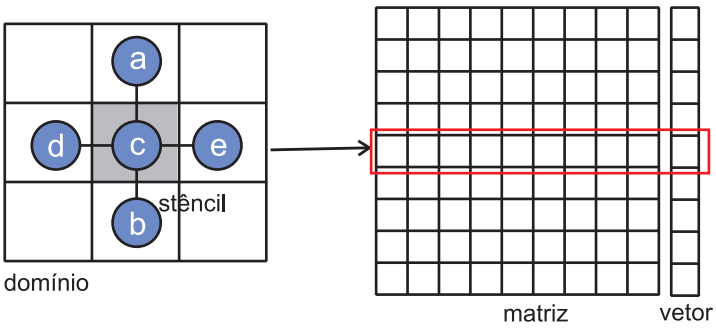

Figura 2. Geração dos sistemas de equações utilizando o stêncil computacional

Ao final deste processo, obtém-se um sistema de equações pentadiagonal (uma diagonal para cada elemento do stêncil) simétrico com k linhas. A solução deste sistema corresponde às temperaturas de cada célula em um passo de tempo.

2.1. Considerações Computacionais

Considerando a formulação matemática apresentada, a simulação de um passo do fenômeno da transferência de calor é formada pelos seguintes passos:

1. Obtenção das temperaturas do domínio (Leitura de dados);
2. Montagem dos sistemas de equações;
3. Resolução dos sistemas de equações;
4. Atualização das temperaturas no domínio.

Para obter uma evolução temporal da difusão da temperatura na placa plana, basta atualizar o sistema com as temperaturas obtidas na iteração anterior (item 4) e resolvê-lo novamente (item 3). Este processo pode ser repetido para a quantidade de passos de tempo desejado.

De acordo com o item 2 da lista anterior, é necessário encontrar a solução do sistema de equações para cada passo de tempo da simulação. Como método de solução foi escolhido o gradiente conjugado (GC) [Shewchuk 1994], dada as características de esparsidade e simetria dos sistemas de equações gerados. O algoritmo GC é composto basicamente de operações vetoriais e matriciais, entre elas, soma e subtração de vetores, multiplicação de matriz por vetor, multiplicação de escalar por vetor e produto escalar, o que simplifica o processo de paralelização. Na Figura 3, apresenta-se o algoritmo do GC.

```
i = 0
r = b - Ax          (subtração de vetores (produto matriz-vetor))
d = r               (cópia de vetor)
δn = r T r          (produto escalar)
enquanto (i<itmax) e (erro>ε)
    q = Ad          (produto matriz-vetor)
    α = δn / d T q  (produto escalar)
    x = x + αd      (soma de vetores (multiplicação escalar-vetor))
    r = r - αq      (subtração de vetores (multiplicação escalar-vetor))
    δv = δn
    δn = r T r      (produto escalar)
    β = δn / δv
    d = r + βd      (adição de vetores (multiplicação escalar-vetor))
    i++
```

Figura 3. Algoritmo do Gradiente Conjugado

Outra influência da esparsidade dos sistemas é a maneira de como a matriz do sistema pode ser armazenada em memória. Neste trabalho utiliza-se o formato de armazenamento diagonal [Saad 1996]. Neste formato utiliza-se um vetor para cada diagonal e um vetor *offset* para armazenar as distâncias entre as diagonais secundárias da principal. Um exemplo é mostrado na Figura 4.

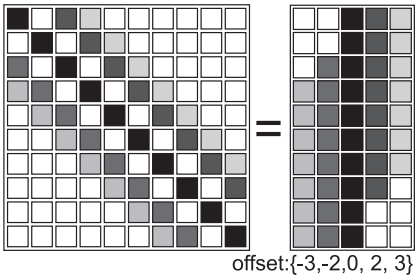

Figura 4. Exemplo de armazenamento de matriz no formato Diagonal

Utilizando este formato de armazenamento utiliza-se no máximo $5N$ elementos, ao invés de N^2 que seriam empregados em um formato denso. Esta abordagem possibilita a resolução de problemas muito maiores e de forma mais otimizada, uma vez que as operações com posições zeradas são quase que totalmente eliminadas.

No caso deste trabalho, o principal benefício de se utilizar este formato é a operação de multiplicação matriz-vetor do algoritmo do gradiente conjugado que se limita a $5N$ operações, para a matriz pentadiagonal, ao invés das N^2, para uma matriz completa. Os detalhes da implementação podem ser vistos na Seção 3.

3. Implementação dos Métodos

Nesta seção apresenta-se as implementações utilizadas na solução do problema de transferência de calor. Foram feitas cinco implementações distintas, sendo uma sequencial, três implementações utili-

zando OpenMP com diferentes abordagens e uma implementação utilizando CUDA.

A implementação sequencial foi feita com a finalidade de verificar a qualidade numérica e verificar os ganhos de desempenho das versões paralelas. As demais implementações são descritas nas Seções 3.1 e 3.2.

3.1. Paralelização usando OpenMP

Foram desenvolvidas três implementações utilizando OpenMP com o objetivo de explorar diferentes abordagem de paralelismo utilizando a API.

Na primeira versão (chamada de OpenMPv1) foi feita apenas explorando o paralelismo de laço (*for*) disponível no OpenMP. Basicamente foram paralelizados os procedimentos de montagem dos sistemas e o método do GC. O procedimento de montagem é simples, onde cada *thread* fica responsável por montar as linhas dos sistemas de um conjunto de células do domínio.

A paralelização do GC foi feita apenas inserindo o pragma `parallel for` dentro de cada procedimento das operações que o compõem, como mostra o Código 1, onde pode ser observada a paralelização dos procedimentos de produto escalar e multiplicação de vetor por escalar, e suas respectivas chamadas.

Código 1. Exemplo de Paralelização - OpenMPv1

```
1  float produto_escalar (float *vetor1, float *vetor2, int tdom)
2  {
3          int i;
4          float resposta=0;
5
6          #pragma omp parallel for   default(none) \
7              private(i) \
8              shared(tdom, vetor1, vetor2)\
9              reduction (+:resposta)
10         for (i=0;i<tdom;i++)
11                 resposta+= vetor1[i] * vetor2[i];
12
13         return (resposta);
14 }
15
16
17 void escalar_vetor (float *vetor, float escalar, float *resposta, int tdom)
18 {
19         int i;
20
21         #pragma omp parallel for   default(none) \
22             private(i) \
23             shared(tdom, resposta, escalar, vetor)
24         for (i=0;i<tdom;i++)
25                 resposta[i]=escalar * vetor[i];
26 }
27 ...
28
29 //chamada dos procedimentos
30 var1=produto_escalar(vet1,vet2,n_elem);
31 escalar_vetor (esc,vet1,vet2,n_elem);
```

Considerando que o OpenMP utiliza o modelo de programação *fork-join* [Chapman *et al.* 2007], no qual em cada região `parallel` as *threads* são criadas e destruídas, esta primeira implementação possui um *overhead* significativo para a criação de *threads* dentro de cada procedimento. Dessa forma, na segunda versão (OpenMPv2) o método do GC foi implementado de maneira diferente.

Nesta segunda versão, criou-se apenas uma região paralela no início do procedimento do GC. Dentro desta região cada *thread* calcula a sua linha inicial e final, baseando-se em sua identificação

(`tid`). Dessa forma, elimina-se o custo de diversas criações e destruições das *threads*, mas exige que alguns pontos de sincronismo sejam adicionados ao código, por exemplo, no cálculo do produto escalar e nas operações envolvendo escalares. Um trecho do código é mostrado no Código 2.

Código 2. Exemplo de Paralelização - OpenMPv2

```
1  float produto_escalar (float *vetor1, float *vetor2, int ini, int fim)
2  {
3          int i;
4          float resposta=0;
5
6          for(i=ini;i<fim;i++)
7                  resposta+= vetor1[i] * vetor2[i];
8
9          return (resposta);
10 }
11
12
13 void escalar_vetor (float *vetor, float escalar, float *resposta, int ini, int fim)
14 {
15         int i;
16
17         for(i=ini;i<fim;i++)
18                 resposta[i]=escalar * vetor[i];
19 }
20
21 //chamada dos procedimentos - região paralela
22 #pragma omp parallel default(none)\
23                 private(ini,fim,tid,i) \
24             shared(vet1,vet2,esc,tam,nth,resto,pesc)
25 {
26     tid=omp_get_thread_num();
27     nth=omp_get_num_threads();
28
29     #pragma omp master
30     {
31         tam=(int)tdom/nth;
32         resto=tdom%nth;
33         pesc=(float *)calloc(nth,sizeof(float));
34     }
35     #pragma omp barrier
36
37     ini=tid*tam;
38     fim=ini+tam;
39 ...
40     //chamada produto_escalar
41     pesc[tid]=produto_escalar(vet1,vet2,ini,fim);
42     #pragma omp barrier
43     #pragma omp master
44     {
45                 for(i=0;i<nth;i++)
46                 var1=pesc[i];
47     }
48
49     //chamada escalar    _vetor
50     escalar_vetor (esc,vet1,vet2,ini,fim);
51 }
```

A terceira versão utiliza um paradigma diferente de programação paralela para métodos numéricos chamado de *decomposição de domínios* [Smith *et al.* 1996]. Decomposição de domínios é normalmente utilizado em computação de memória distribuída mas implementou-se utilizando memória compartilhada para verificar como o algoritmo se comportaria neste tipo de ambiente.

Nessa abordagem divide-se o domínio do problema (a placa plana, neste caso) entre os processadores, que ficam responsáveis por gerar os sistemas de equações relacionados ao seu conjunto de células e resolvê-los. Dessa forma, ao invés de se ter apenas um sistema que é resolvido por p processadores, tem-se p sistemas. Para manter a continuidade da solução nas bordas, a cada passo de

simulação os processadores trocam dados das fronteiras dos domínios e recalculam a solução. Muitas vezes, dependendo do tamanho dos sistemas e das fronteiras, é necessário diversas trocas, o que pode comprometer o desempenho.

Na solução implementada, o domínio é dividido em p faixas e cada uma delas é atribuída a um processador. Cada processador gera o sistema de equações da sua parte do domínio e também calcula quais são as dependências de fronteira, ou seja, quais dados deverão ser obtidos de outros processadores. Na solução do sistema utilizou-se uma abordagem SPMD, onde cada processador executa um GC sequencial sobre seus dados. Após a obtenção da solução parcial, cada processador coloca sua solução em um vetor compartilhado para que possa haver a troca de dados referentes aos valores da fronteira. Uma ilustração do método é mostrada na Figura 6. Este processo deve se repetir até que se obtenha a continuidade da solução ou por um número determinado de iterações (o que pode comprometer a qualidade numérica).

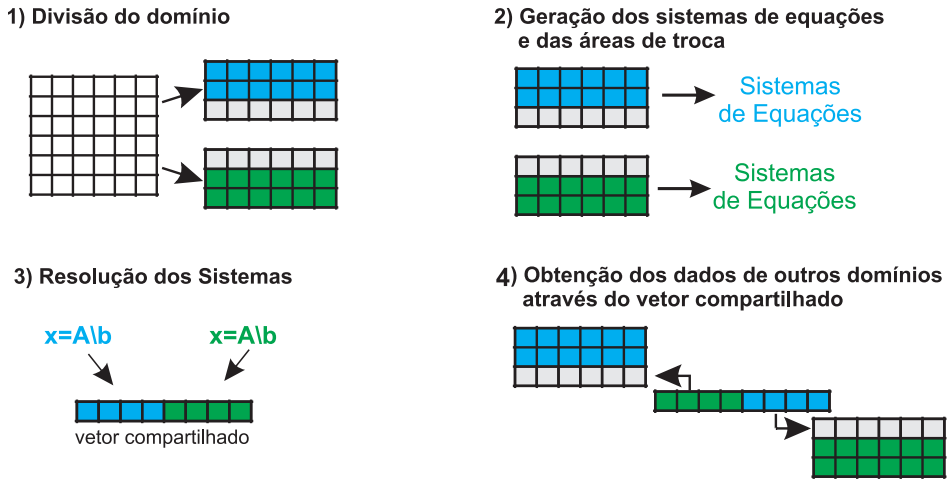

Figura 5. Passos da solução por decomposição de domínios

Os resultados das três implementações apresentadas nesta seção podem ser vistos na Seção 4.

3.2. Paralelização usando CUDA

A implementação do problema de transferência de calor utilizando CUDA explora um outro paradigma de paralelismo: a computação massivamente paralela. Neste paradigma o objetivo é utilizar centenas ou milhares de *threads* disponibilizadas pelas GPUs para melhorar o desempenho de aplicações.

Nesta abordagem, a paralelização foi feita criando-se *kernels* para os procedimentos de montagem dos sistemas de equações e para as diversas operações que compõem o algoritmo do GC. A implementação foi feita de modo que só haja duas transferências de memória entre a CPU e GPU antes das computações e uma de transferência de resultados ao final. Todas os demais *kernels* utilizam variáveis alocadas diretamente na GPU.

No *kernel* de montagem dos sistemas de equações, inicialmente, cada *thread* calcula a célula do domínio pelo qual será responsável. Na sequência, cada uma das *threads* opera sobre esta célula e monta uma linha do sistema de equações.

A paralelização das operações de soma/subtração de vetores, multiplicação de vetor por escalar e multiplicação de matriz por vetor foram feitas de forma simples, onde cada *thread* opera sobre uma posição dos vetores e armazena na posição correspondente do vetor de resultado. No Código 3

são apresentados os *kernels* das operações de multiplicação de matriz por vetor e multiplicação de vetor por escalar.

Código 3. Multiplicação de matriz por vetor e vetor por escalar

```
1  //multiplicação de matriz por vetor
2  __global__ void kernel_mmv(float *matAd1,float *matAd2,float *matAd3,float *matAd4,float *matAd5,int
       *offset,float *vet, float *result,int size)
3  {
4      int i = blockDim.x * blockIdx.x + threadIdx.x;
5      float tmp=0.00;
6
7      if(i<size)
8      {
9        if((i+offset[0])>=0)
10       tmp+=matAd1[i]*vet[i+offset[0]];
11
12       if((i+offset[1])>=0)
13       tmp+=matAd2[i]*vet[i+offset[1]];
14
15       if((i+offset[2])>=0)
16       tmp+=matAd3[i]*vet[i+offset[2]];
17
18       if((i+offset[3])>=0)
19       tmp+=matAd4[i]*vet[i+offset[3]];
20
21       if((i+offset[4])>=0)
22       tmp+=matAd5[i]*vet[i+offset[4]];
23
24       result[i]=tmp;
25     }
26 }
27
28
29 //multiplicação de vetor por escalar
30 __global__ void kernel_escalar_vetor(float *esc, float *vet, float *vet2,int size)
31 {
32     int idx = blockDim.x * blockIdx.x + threadIdx.x;
33
34     if(idx<size)
35       vet2[idx] = *esc * vet[idx];
36 }
37
38 //chamada dos kernels
39 kernel_mmv<<<gridDim,blockDim>>>(matAd1,matAd2,matAd3,matAd4,matAd5,offset,vet1,vet2,size);
40
41 kernel_escalar_vetor<<<gridDim,blockDim>>>(esc, vet1, vet2,size);
```

Já a implementação da operação de produto escalar mostrou-se mais complexa nesta arquitetura, havendo a necessidade de dividir a operação em duas partes. Na primeira, multiplica-se cada parte do vetor e armazena-se o resultado em um vetor em memória global, portanto disponível a todos os blocos. Na segunda parte, cada bloco copia uma porção do vetor da memória global para um vetor em memória compartilhada e faz a redução desta parte do vetor. O Código 4 mostra os kernels necessários na operação de produto escalar. É importante salientar que na operação de redução é necessário uma operação de soma atômica para números de ponto flutuante, não implementada em CUDA até a presente versão.

Dessa forma, buscou-se uma alternativa no fórum da Nvidia[1], onde foi publicado o kernel *atomicFloatAdd*. Neste kernel, utiliza-se da operação *atomicCAS* (*Compare and Swap*) para números inteiros e operações de conversão *__float_as_int* e *__int_as_float* na implementação da soma atômica. Uma outra opção seria copiar o vetor de resultados das multiplicações e executar a redução em CPU e em seguida copiar o resultado de volta à GPU. Embora esta segunda opção seja eficaz, perde em eficiência já que acrescenta o custo das transferências GPU-CPU-GPU ao tempo de execução.

[1] `http://forums.nvidia.com/index.php?showtopic=67691`

Código 4. Kernels para produto escalar

```
1  //multiplicação dos vetores
2  __global__ void kernel_pe_mult(float *a, float *b, float *c,int size)
3  {
4          int idx = blockDim.x * blockIdx.x + threadIdx.x;
5
6          if(idx<size)
7          {
8                  c[idx] = a[idx] * b[idx];
9          }
10 }
11
12 //soma dos elementos do vetor - reducao
13 __global__ void kernel_pe_soma(float* c, float* result,int size)
14 {
15         int tid = threadIdx.x;
16         int idx = blockIdx.x * blockDim.x + threadIdx.x;
17
18         __shared__ float x[THREADS];
19
20         if(idx<size)
21             x[tid] = c[idx];
22         else
23         x[tid]=0;
24
25         __syncthreads();
26
27         for(int s=blockDim.x/2; s>0; s=s/2)
28         {
29             if(tid < s)
30               x[tid] += x[tid + s];
31             __syncthreads();
32         }
33
34         if(tid == 0)
35         {
36             atomicFloatAdd(result, x[0]);
37         }
38 }
39
40 //Soma atomica de float
41 __device__ inline void atomicFloatAdd(float *address, float val)
42 {
43         int i_val = __float_as_int(val);
44         int tmp0 = 0;
45         int tmp1;
46
47         while( (tmp1 = atomicCAS((int *)address, tmp0, i_val)) != tmp0)
48         {
49              tmp0 = tmp1;
50              i_val = __float_as_int(val + __int_as_float(tmp1));
51         }
52 }
```

Um outro problema na implementação do GC é a divisão de escalares. De modo a não utilizar cópias entre CPU e GPU, implementou-se um *kernel* que é executada por uma única *thread* por bloco. Este tipo de operação além de aproveitar mal a arquitetura, faz com que haja a necessidade de sincronização de todas as *threads* para a atualização dos valores.

Os resultados da implementação utilizando CUDA podem ser vistos na Seção 4.2.

4. Testes e Resultados

Nesta seção apresenta-se os resultados obtidos com as implementações desenvolvidas neste trabalho. Na Seção 4.1 são apresentados os resultados das implementações OpenMP, na Seção 4.2 os resultados obtidos com CUDA e na Seção 4.3 os resultados físicos das simulações.

4.1. Resultados OpenMP

Os testes das implementações utilizando OpenMP foram executados na máquina *Alt* do Departamento de Informática da UFPR, cuja configuração é composta por 8 núcleos de processamento Intel Xeon E5345 de 2.33GHz e 7GB de memória RAM. Para as tomada de tempos utilizados domínios quadrados com 512x512, 1024x1024, 2048x2048 e 4096x4096 células de cálculo, com temperatura de 10^oC nas bordas e 1^oC no interior do domínio. Foram simulados 10 iterações do método, com parâmetros $\Delta T = 0,1s$, $\Delta x = 0,1m$ e $\alpha = 0,01$. Cada teste foi repetido cem vezes, com os quais calculou-se o tempo médio de execução, o desvio padrão e o *speedup*.

Nas Tabelas 1 e 2 são mostrados os resultados obtidos com a primeira implementação utilizando OpenMP (OpenMPv1). Com os domínios com tamanho 512x512 e 1024x1024 os maiores ganhos de desempenho ocorreram com a utilização de 4 *threads*. O fato de não haver ganhos maiores com 8 ou 16 *threads* está relacionado do tamanho do domínio, considerado pequeno para estas quantidades de *threads*, fazendo com que não compense o custo da criação das *threads* a cada operação, como explicado na Seção 3.1. No entanto, utilizando domínios maiores os ganhos com 8 processadores ficaram mais significantes, alcançando um *speedup* máximo de 4,319 no domínio 4096x4096. Já a utilização de 16 *threads* não apresentou resultados satisfatórios, alcançando tempos de execução sempre superiores ao obtidos com o uso de 8 *threads*. Isso ocorre devido ao fato da máquina utilizada para os testes ter apenas 8 núcleos de execução, havendo a necessidade de escalonamento das *threads* entre os núcleos disponíveis e por consequência queda no desempenho.

Tabela 1. OpenMPv1 - Tempo de execução e Desvio padrão

Tempo	Tamanho da entrada			
threads	**512**	**1024**	**2048**	**4096**
1	1,213 0,014	4,898 0,055	19,692 0,241	79,702 0,615
2	0,814 0,012	3,109 0,027	12,430 0,109	49,537 0,506
4	0,552 0,034	1,778 0,064	7,481 0,107	29,777 0,113
8	0,913 0,351	3,050 0,887	4,972 0,072	18,453 0,194
16	1,118 0,310	3,320 0,911	5,514 0,060	20,292 0,120

Tabela 2. OpenMPv1 - Speedup

Speedup	Tamanho da entrada			
threads	**512**	**1024**	**2048**	**4096**
2	1,490	1,576	1,584	1,609
4	2,197	2,755	2,632	2,677
8	1,328	1,606	3,960	4,319
16	1,085	1,476	3,571	3,928

Nas Tabelas 3 e 4 são apresentados os tempos de execução da segunda implementação utilizando OpenMP (OpenMPv2). Como apresentado na Seção 3.1 esta implementação procurou-se eliminar parte do custo da criação de *threads*, o que resultou em melhores tempos de execução

comparando-os com os obtidos com a primeira versão (ver Tabelas 1 e 2). Esta solução mostrou-se mais escalável, apresentando bons ganhos de *speedup* com 8 *threads* em todos os tamanhos de domínio. Mais uma vez não foi vantajoso o uso de 16 *threads*.

Tabela 3. OpenMPv2 - Tempo de execução e Desvio padrão

Tempo	Tamanho da entrada			
threads	**512**	**1024**	**2048**	**4096**
1	1,213	4,898	19,692	79,702
	0,014	0,055	0,241	0,615
2	0,640	2,538	10,257	43,003
	0,008	0,064	0,181	0,537
4	0,412	1,577	6,622	27,266
	0,022	0,028	0,038	0,116
8	0,358	1,076	4,504	18,079
	0,027	0,039	0,042	0,105
16	0,474	1,200	4,640	19,382
	0,037	0,027	0,036	0,140

Tabela 4. OpenMPv2 - Speedup

Speedup	Tamanho da entrada			
threads	**512**	**1024**	**2048**	**4096**
2	1,895	1,930	1,920	1,853
4	2,947	3,106	2,974	2,923
8	3,385	4,554	4,372	4,409
16	2,561	4,083	4,244	4,112

Para a implementação que utiliza decomposição de domínios (OpenMPv3) foram feitos dois testes distintos. No primeiro teste, foram feitas 2 trocas de dados entre os domínios. Os resultados de desempenho destes testes podem ser observados nas Tabelas 5 e 6.

Tabela 5. OpenMPv3 - Tempo de execução e Desvio padrão (com duas trocas de dados por iteração)

Tempo	Tamanho da entrada			
threads	**512**	**1024**	**2048**	**4096**
1	1,213	4,898	19,692	79,702
	0,014	0,055	0,241	0,615
2	1,263	5,164	20,998	83,342
	0,025	0,134	0,527	1,038
4	0,735	3,160	13,352	53,230
	0,069	0,277	0,855	2,112
8	0,490	1,879	8,361	34,044
	0,076	0,246	0,148	0,457
16	0,432	1,904	8,320	35,399
	0,014	0,220	0,150	0,467

Como se pode observar, as duas trocas de dados entre os domínios causou uma queda de desempenho significativa, embora a qualidade numérica seja compatível com a obtida com as duas

Tabela 6. OpenMPv3 - Speedup (com duas trocas de dados por iteração)

Speedup	Tamanho da entrada			
threads	**512**	**1024**	**2048**	**4096**
2	0,961	0,949	0,938	0,956
4	1,650	1,550	1,475	1,497
8	2,477	2,606	2,355	2,341
16	2,811	2,572	2,367	2,252

primeiras implementações. Assim, considerando os resultados não satisfatórios, no segundo teste configurou-se o método para que fizesse apenas 1 troca de dados. Os resultados de desempenho destes testes podem ser observados nas Tabelas 7 e 8.

Tabela 7. OpenMPv3 - Tempo de execução e Desvio padrão (com uma troca de dados por iteração)

Tempo	Tamanho da entrada			
threads	**512**	**1024**	**2048**	**4096**
1	1,213 0,014	4,898 0,055	19,692 0,241	79,702 0,615
2	0,655 0,026	2,651 0,082	10,826 0,308	42,800 0,655
4	0,422 0,078	1,715 0,265	6,669 0,096	27,078 0,380
8	0,303 0,070	0,918 0,033	4,314 0,133	17,585 0,162
16	0,268 0,042	1,011 0,058	4,298 0,111	18,472 0,522

Tabela 8. OpenMPv3 - Speedup (com uma troca de dados por iteração)

Speedup	Tamanho da entrada			
threads	**512**	**1024**	**2048**	**4096**
2	1,853	1,847	1,819	1,862
4	2,875	2,856	2,953	2,943
8	4,004	5,337	4,565	4,532
16	4,519	4,844	4,581	4,315

Neste segundo teste, alcançou-se bons tempos de execução, melhores até que os obtidos com a implementação OpenMPv2. No entanto, constatou-se que uma única troca entre os domínios é insuficiente para garantir uma boa qualidade numérica. Uma maneira de melhorar estes resultados é a criação de áreas de sobreposição entre as partes dos domínios, de modo que com menos trocas de dados seja possível melhorar a qualidade numérica [Smith *et al.* 1996, Cai e Saad 1996].

Para resumir os resultados obtidos com as implementações utilizando OpenMP, apresenta-se um gráfico na Figura 6 com os comparativos de tempo na solução do problema da transferência de calor no domínio com 4096x4096 células. Analisando os testes efetuados, a versão que combina melhor qualidade numérica e melhor desempenho é a OpenMPv2.

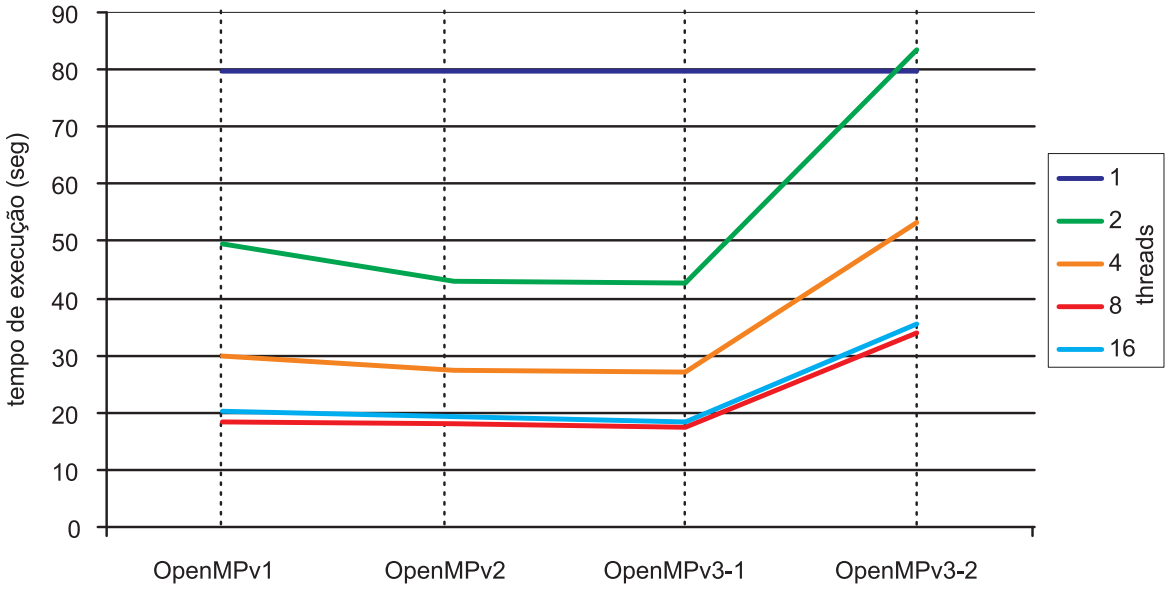

Figura 6. Comparativo de tempo de execução com domínios de 4096 x 4096 células.

4.2. Resultados CUDA

Os testes da implementação em CUDA foram executados na máquina ag03 do *cluster* do LSPAD. A máquina possui um processador Intel Core2 Quad Q6600 de 2.40GHz, 2GB de RAM e uma placa Nvidia 8600GT. Para as tomada de tempos utilizados domínios quadrados com 512x512, 1024x1024 e 2048x2048 células de cálculo, com temperatura de 10^oC nas bordas e 1^oC no interior do domínio. Foram simulados 10 iterações do método, com parâmetros $\Delta T = 0,1s$, $\Delta x = 0,1m$ e $\alpha = 0,01$. Cada teste foi repetido cem vezes, com os quais calculou-se os tempos médios de execução,apresentados na Tabela 9. Variou-se também a quantidade de *threads* por blocos para verificar a influência deste fator sobre o tempo total de execução.

Tabela 9. Comparação de tempo de execução CUDA e CPU

	CPU	512 threads	256 threads	128 threads
	domínio 512x512			
montagem da matriz	0,00891	0,00234	0,00220	0,00219
simulação	0,99339	0,30064	0,38905	0,76484
total	1,00230	0,30298	0,39125	0,76703
	domínio 1024x1024			
montagem da matriz	0,03479	0,00827	0,00767	0,00753
simulação	4,06877	**1,18226**	1,53430	3,05124
total	4,10356	1,19053	1,54197	3,05877
	domínio 2048x2048			
montagem da matriz	0,129848	0,03183	0,029018	**0,028788**
simulação	16,464389	4,48112	4,621018	5,144968
total	16,594237	4,51295	4,65004	5,17376

Como pode ser observado, foi possível alcançar *speedups* máximo 4,5 na montagem dos sistemas de equações e de 3,7 nos ciclos de simulação, mostrados em negrito na Tabela. Esperava-se ganhos superiores com o uso desta arquitetura, no entanto considerando as características do algoritmo do GC e do hardware utilizado, são resultados válidos.

Outro resultado observado é que a variação dos tamanho dos blocos influencia muito nos tempos de execução. Um exemplo disso é a execução da aplicação com domínio de tamanho 1024x1024, onde com o uso de blocos com 128 threads levou-se mais que o dobro da configuração que usa 512.

É muito importante que se conheça bem a aplicação e plataforma na qual se está trabalhando, pois mínimas alterações do código ou em suas configurações de execução podem resultar em ganhos significativos de desempenho. Uma ferramenta que pode ajudar refinamento de código é o *Visual Profiler*, disponibilizado no *toolkit* de desenvolvimento CUDA[2]. Com ele pode-se obter dados importantes da aplicação, tais como uso de mémória, uso do processador, tempo de execução total e dos kernels, entre outros. Na Seção 4.2.1 são apresentados alguns resultados utilizando esta ferramenta.

4.2.1. Resultados obtidos com o VisualProfiler

O NVIDIA CUDA Visual Profiler é uma ferramenta criada pela NVIDIA para ajudar o desenvolvimento de aplicações CUDA. Esta ferramenta analisa aplicações paralelas via contadores como: acessos coalescentes e não-coalescentes à memória global; leituras e escritas à memória local; e desvios divergentes. Nesta seção são analisadas alguns destas métricas. Para estes testes utilizou-se uma placa Nvidia 8500GT.

Na Tabela 10 é possivel observar alguns resultados obtidos com a aplicação relacionados ao uso de registradores (Reg.), uso de memória compartilhada (M. Cmp.) e ocupância dos multiprocessadores (Ocup.).

Tabela 10. Uso de registradores, uso de memória compartilhada e ocupância de multiprocessadores

	Número de threads por bloco								
kernel	512			256			128		
	Reg.	M.Cmp.	Ocup.	Reg.	M.Cmp.	Ocup.	Reg.	M. Cmp.	Ocup.
kernel_div	2	28	67%	2	28	100%	2	28	100%
kernel_gera_matriz	14	56	67%	14	56	67%	14	56	67%
kernel_mmv	6	52	67%	6	52	100%	6	52	100%
kernel_escalar_vetor	4	32	67%	4	32	100%	4	32	100%
kernel_subtrai_vetores	4	32	67%	4	32	100%	4	32	100%
kernel_soma_vetores	4	32	67%	4	32	100%	4	32	100%
kernel_pe_mult	4	32	67%	4	32	100%	4	32	100%
kernel_pe_soma	3	2076	67%	3	1052	100%	3	540	100%

Pode-se observar que os *kernels* da aplicação utilizaram poucos registradores e pouca memória compartilhada, com uma pequena exceção do kernel_pe_soma que utiliza uma quantidade maior de memória compartilhada para fazer uma redução. Em relação à ocupância, conseguiu-se uma boa taxa de utilização dos multiprocessadores, 100% em quase todas as operações (exceto kernel_gera_matriz), nas execuções com blocos com 256 e 128 *threads*. Com blocos de 512 *threads* a taxa de ocupância foi inferior, alcançando 67% para todas as operações.

Outro ponto analisado foram os acessos coalescentes e não-coalescentes da aplicação. Como pode ser visto na Tabela 11, o único kernel que possui acessos não-coalescentes é o kernel_mmv (multiplicação de matriz por vetor), com 93% dos acessos de busca desse tipo. Todos os demais apresentam 100% de acessos coalescentes. Esse tipo de informação pode ser útil na modificação do

[2] `http://developer.nvidia.com/object/cuda_3_0_downloads.html`

código para melhorias do desempenho geral da aplicação, já que esta operação ocupa, em média, 60% do tempo total de execução (Tabelas 12, 13 e 14), sendo executada 40 vezes.

Tabela 11. Acessos coalescentes e não-coalescentes

nome do kernel	gld unc.	gld coal.	gst unc.	gst coal.
kernel_div	0	16384	0	32768
kernel_escalar_vetor	0	73728	0	262144
kernel_gera_matriz	0	133248	0	3678210
kernel_mmv	7340030	524160	0	262144
kernel_pe_mult	0	131072	0	262144
kernel_pe_soma	0	65536	0	0
kernel_soma_vetores	0	131072	0	262144
kernel_subtrai_vetores	0	131072	0	262144

Considerando que para estes testes de *profile* utilizou-se hardware diferente dos testes de tempo de execução da Tabela 9, observa-se que as Tabelas 12, 13 e 14 apresentam resultados diferentes. Comparando estas últimas três, que contém o tempo de execução de cada *kernel*, pode-se notar que os melhores tempos, em média, foram alcançados utilizando-se blocos de 256 *threads.* Este fato mostra, mais uma vez, que as aplicações CUDA são altamente sensíveis ao hardware e a seus parâmetros de execução.

Tabela 12. Quantidade de acessos e tempo de execução dos kernels - 128 threads por bloco

nome do kernel	acessos	tempo GPU (usec)	tempo CPU (usec)	%
kernel_div	60	499,1	570,0	0,80
kernel_escalar_vetor	90	1383,3	1433,2	3,36
kernel_gera_matriz	1	12173,1	12235,0	0,32
kernel_mmv	40	49137,9	49191,4	53,07
kernel_pe_mult	70	1621,8	1666,9	3,06
kernel_pe_soma	70	17483,3	17530,0	33,04
kernel_soma_vetores	70	1602,9	1645,2	3,02
kernel_subtrai_vetores	40	1767,7	1900,9	1,90
memcopy	31	861,4	123	0,72

Tabela 13. Quantidade de acessos e tempo de execução dos kernels - 256 threads por bloco

nome do kernel	acessos	tempo GPU (usec)	tempo CPU (usec)	%
kernel_div	60	250,7	268,7	0,49
kernel_escalar_vetor	90	1460,2	1475,8	4,30
kernel_gera_matriz	1	13787,2	13847,0	0,45
kernel_mmv	40	49122,5	49136,4	64,37
kernel_pe_mult	70	1617,6	1632,1	3,71
kernel_pe_soma	70	8264,5	8305,7	18,95
kernel_soma_vetores	70	1614,0	1633,3	3,70
kernel_subtrai_vetores	40	1783,9	1807,4	2,33
memcopy	31	860,2	122,742	0,87

Tabela 14. Quantidade de acessos e tempo de execução dos kernels - 512 threads por bloco

nome do kernel	acessos	tempo GPU (usec)	tempo CPU (usec)	%
kernel_div	60	126,4	147,0	0,23
kernel_escalar_vetor	90	1796,5	1810,9	4,92
kernel_gera_matriz	1	14331,7	14399,0	0,43
kernel_mmv	40	49327,9	49342,1	60,16
kernel_pe_mult	70	1924,7	1938,8	4,1
kernel_pe_soma	70	10147,2	10167,0	21,65
kernel_soma_vetores	70	2031,2	2045,6	4,33
kernel_subtrai_vetores	40	2108,0	2197,7	2,57
memcopy	31	874,2	122,7	0,82

Para finalizar esta seção, apresenta-se na Figura 7 o perfil de execução da aplicação, onde pode-se ver claramente a predominância da operação de multiplicação de matriz por vetor e o melhor tempo obtido com blocos de 256 *threads*.

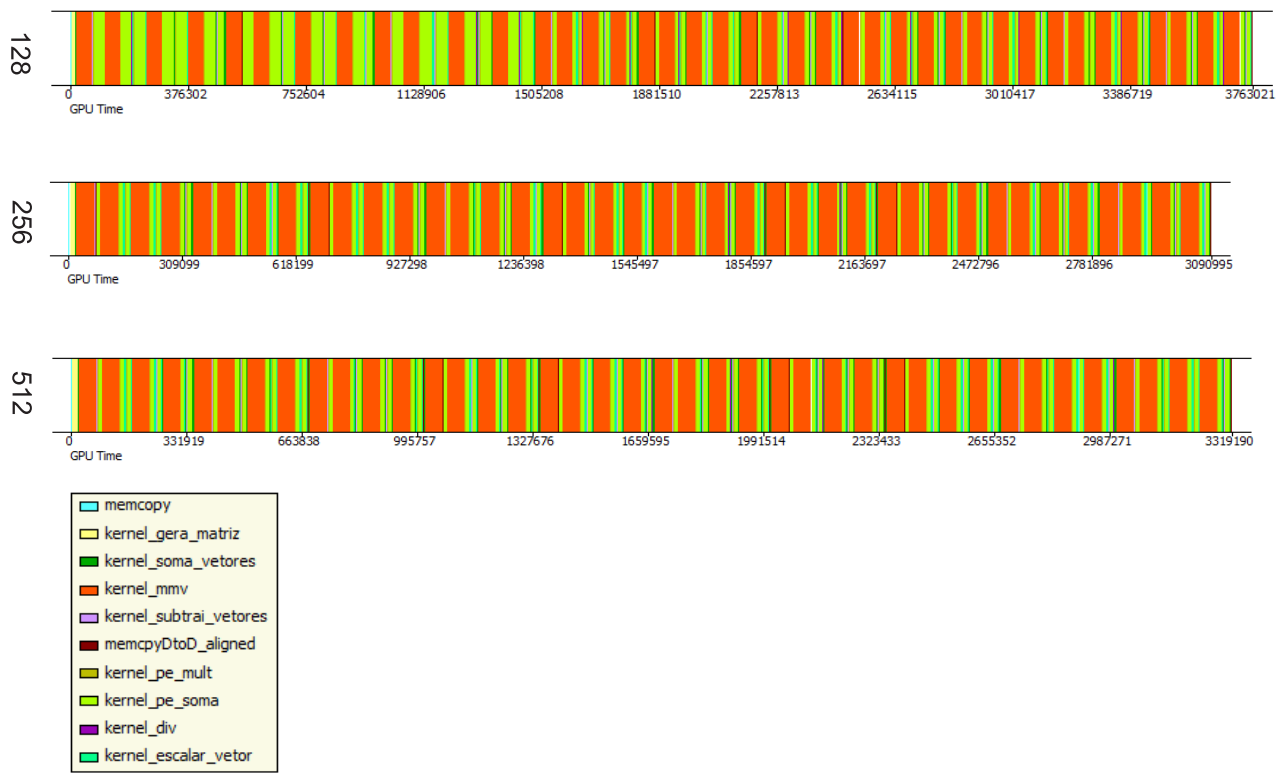

Figura 7. Perfil de execução

4.3. Resultados Físicos

A Figura 8 apresenta o resultado de uma simulação de 180 iterações em um domínio 100x100, com temperatura de 10^oC nas bordas e 1^oC no interior do domínio. Os dados foram obtidos com a execução da implementação OpenMP (versão 2) utilizando 2 *threads*. Os gráficos foram gerados com o a função *surf* (superfície) do software Scilab [Scilab 2009].

O resultado obtido está de acordo com o esperado, onde a maior temperatura flui para as áreas de menor temperatura.

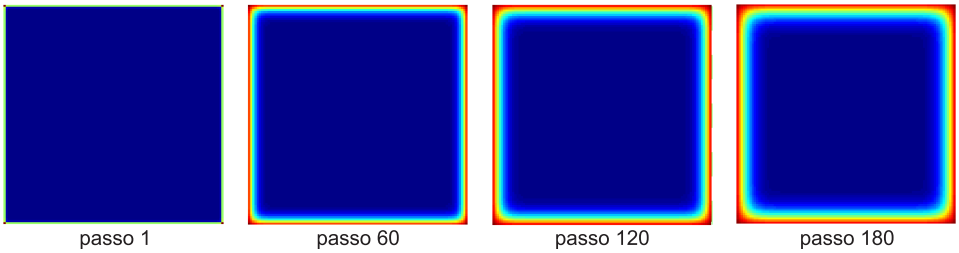

Figura 8. Difusão do calor nos passos de tempo 1, 60, 120 e 180

5. Conclusão

O principal objetivo deste trabalho foi a apresentação da paralelizações do problema de transferência de calor e seus respectivos resultados. Foram implementadas três versões utilizando OpenMP e uma versão utilizando CUDA.

Em relação às implementações, destaca-se a segunda versão (OpenMPv2), que apresentou bons resultados de desempenho e de qualidade numérica. Implementou-se também um método de decomposição de domínios, mas este não apresentou resultados satisfatórios, mostrando-se uma abordagem mais apropriada para memória distribuída.

Com a implementação em CUDA obteve-se resultados de desempenho razoáveis, abaixo da espectativa. Apresentou-se também alguns resultados obtidos com o CUDA Visual Profiler, que se mostrou uma ferramenta muito útil no refinamento de código nesta arquitetura.

De modo geral, o desenvolvimento de aplicações paralelas não é uma tarefa fácil. Exige do programador bons conhecimentos tanto do problema a ser resolvido como da arquitetura (hardware e software) utilizada em sua solução. Analisando as duas ferramentas, OpenMP e CUDA, pode-se dizer que a primeira está em um nível muito maior de desenvolvimento, ou seja, oferece ao programador uma API muito mais completa que a oferecida pelo CUDA, que ainda precisa melhorar em muitos pontos.

Referências

Borges, P. A. P. e Padoin, E. L. (2006). Exemplos de métodos computacionais aplicados a problemas na modelagem matemática. In *Anais Erad*, ERAD - Escola Regional de Alto Desempenho.

Cai, X. e Saad, Y. (1996). Overlapping domain decomposition algorithms for general sparse matrices. Technical report.

Chapman, B., Jost, G., e Pas, R. v. d. (2007). *Using OpenMP: Portable Shared Memory Parallel Programming (Scientific and Engineering Computation)*. The MIT Press.

Fortuna, A. O. (2000). *Técnicas Computacionais para Dinâmica dos Fluidos: Conceitos Básicos e Aplicações*. EDUSP, São Paulo.

Lienhard IV, J. H. e Lienhard V, J. H. (2008). *A Heat Transfer Texbook*. Phlogiston Press, Cambridge.

Saad, Y. (1996). *Iterative Methods for Sparse Linear Systems*. PWS Publishing Company.

Scilab (2009). Manual scilab - versão 5.2.2.

Shewchuk, J. R. (1994). An introduction to the conjugate gradient method without the agonizing pain. Technical report, Pittsburgh, PA, USA.

Smith, B., Bjorstad, P., e Gropp, W. (1996). *Domain Decomposition: Parallel Multilevel Methods for Elliptic Partial Differential Equations*. Cambridge University Pres, Cambridge.