Trabalho 1 - Tópicos Especiais em Arquitetura de Computadores I

Leoni Mota Loris

***Resumo* — Para o uso otimizado do processamento paralelo tanto na CPU quanto na GPU, é necessária a determinação de uma boa forma para paralelizar o programa e dividir corretamente os dados, sendo cada vez menos dependente do acesso à memória global (c*ache misses)* e mais dependente do processamento e paralelizado no computador.**

I. INTRODUÇÃO

A computação tem evoluído a partir do processamento exclusivo da CPU para capacidades co-processamento oferecidas pela combinação da CPU e da GPU assim como a paralelização das tarefas e dados em seus núcleos.

Normalmente existem quatro tipos principais de paralelismo: SISD, SIMD, MISD e MIMD. Esta classificação baseia-se nos conceitos de controle do fluxo de instrições (duas primeiras letras, que significam single ou multiple instruction") e de fluxo de dados, sendo as duas últimas letras (onde o D significado Dados).

Neste contexto, utilizaremos o modelo SIMD que viabiliza escrever e executar um mesmo programa trabalhando com dados de forma paralela. Para habilitar este novo paradigma de computação, utilizaremos a arquitetura e arquitetura de processamento paralelo CUDA, hoje incluído nas principais placas gráficas da NVIDIA e para a programação em múltiplos núcleos da CPU usaremos a API OpenMP.

A. Ajuste dos parâmetros

Uma arquitetura que viabiliza o uso do CUDA contém vários SM's (Streaming Multiprocessor) e cada SM tem vários núcleos. Cada um destes núcleos executa as mesmas instruções (ou entra em modo sleep) que permitem a execução de várias threads, estas, tendo acesso à memória a uma memória local de acesso rápido, restrita a cada thread, à memória compartilhada também e rápido acesso, que é restrita a threads de um mesmo bloco e também à memória global que tem um custo maior para o acesso, sendo assim, trabalho do programados conseguir dividir os dados de forma a diminuir o acesso constante à memória global, de acesso mais lento.

Assim, a maioria dos acessos à memória deve ser à compartilha daquele bloco, quando trabalhamos com as diversas threads.

II. METODOLOGIA

A. Hardware disponível

Serão utilizados acessados remotamente os computadores do LCAD com placas de vídeo habilitadas para CUDA cuja arquitetura é a Tesla c2050.

Suas características são mostradas abaixo

Memória global	Núcleos	L2 Cache	Max Threads/ bloco
2687 MBytes	448	786432 Bytes	1024
GFLOPS (float)	GFLOPS (double)	N° de registradores por bloco	Bus width
1030.4	515.2	32768	384 bits

Tabela 1: Características da GPU fornecidas pelo fabricante

Na próxima tabela, vemos informações sobre a CPU:

L1 Cache	L2 Cache	L3 Cache	Núcleos
32 Kbytes	256 Kbytes	8192 KBytes	8

Tabela 2: Características da CPU Intel(R) Xeon(R)

Para otimizar a multiplicação sequencial na CPU a fim de compara-la com a multiplicação paralela na GPU, nos serviremos da localidade espacial dos dados das multiplicações de matrizes e executaremos uma versão da multiplicação em blocos da matriz (o conceito também é usado na multiplicação paralela).

B. Multiplicação em blocos (ou ladrilhada)

A multiplicação por blocos de uma matriz funciona dividindo as matrizes em submatrizes e, em seguida, explorando o fato de que estas submatrizes podem ser manipulados como escalares. Por exemplo, suponha que queremos calcular C = AB, onde A, B, e C são cada 8 × 8 matrizes. Então, podemos particionar cada matriz em quatro submatrizes 4 × 4:

$$\begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} * \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$$

Onde

$$C_{11} = A_{11} * B_{11} + A_{12} * B_{21}$$

$$C_{12} = A_{11} * B_{12} + A_{12} * B_{22}$$
$$C_{21} = A_{21} * B_{11} + A_{22} * B_{21}$$
$$C_{22} = A_{21} * B_{12} + A_{22} * B_{22}$$

Desta forma, podemos trabalhar com partes menores da matriz que podem caber na memória Cache, de rápido acesso, evitando que muitos acessos sejam feitos na memória global.

Assim, o algoritmo que faz esta multiplicação sequencialmente é o que segue:

C. Còdigo sequencial

```
#define N 512
#define TILE_WIDTH 64
for ( i = 0; i < N; i+= TILE_WIDTH )
    for ( j = 0; j < N; j+= TILE_WIDTH)
      for ( k = 0; k < N; k+= TILE_WIDTH)
        for ( y = i; y < i + TILE_WIDTH; y++ )
          for ( x = j; x < j + TILE_WIDTH; x++ )
            for ( z = k; z < k + TILE_WIDTH; z++ )
              C[y][x] += A[y][z] * B[z][x];
```

Os 3 primeiros loops iteram sobre os blocos de forma normal sequencias que conhecemos da multiplicação de matrizes, enquanto que nos 3 loops seguinte, são feitos os cálculos dentro de um único bloco e salvos os resultados intermediários na matriz C[].

Este programa assume a priori, que o tamanho da matriz é um múltiplo inteiro do tamanho do bloco ($TILE\_WIDTH$). Para faze-la de forma genérica, é preciso adicionar na condição dos 3 últimos loops uma função que calcula o mínimo entre $i + TILE\_WIDTH$ e N, para que o acesso à memória não tente acessar um elemento não alocado para a matriz C[].

D. Còdigo paralelo

O método paralelo é, sob um ponto de vista, mais simples pois os índices serão atribuídos por variáveis CUDA que representam o bloco e a thread em que estão.

No código, as matrizes M e N têm o qualificador __shared__ pois serão alocadas na memória compartilhada entre threads de um mesmo bloco, o que representaria o melhor acesso por uma melhor localidade espacial dos dados. Além disto, foi feito uma modificação e utilizado uma única matriz (MN) para comportar as duas e seus acessos são feitos com o correto manuseio dos índices. Foi feito desta forma para se ter uma alocação dinâmica das matrizes na memória compartilhada. Desta forma, será possível chamar o programa diversas vezes definindo o tamanho do bloco a ser alocado em tempo de execução.

Para alocar dinamicamente a memória compartilhada, deve ser adicionado o qualificador 'extern' na variável do Kernel (função a ser executada na GPU), e passar como parâmetro para o Kernel a quantidade de bytes que deseja ser alocado na hora de chamar a função:

```
__global__ void matrixMultiplyPar(double * A, double * B, double * C,
    int numARows, int numAColumns,
    int numBRows, int numBColumns,
    int numCRows, int numCColumns, int TILE_WIDTH) {
extern __shared__ double ds_MN[];
int bx = blockIdx.x, by = blockIdx.y,
    tx = threadIdx.x, ty = threadIdx.y,
    Row = by * TILE_WIDTH + ty,
    Col = bx * TILE_WIDTH + tx;
  float Pvalue = 0;

  for (int m = 0; m < (numAColumns - 1)/TILE_WIDTH + 1; ++m) {
    if (Row < numARows && m * TILE_WIDTH + tx < numAColumns)
      ds_MN[ty * TILE_WIDTH + tx] = A[Row * numAColumns + m * TILE_WIDTH + tx];
    else
      ds_MN[ty * TILE_WIDTH + tx] = 0.0;
    if (Col < numBColumns && m * TILE_WIDTH + ty < numBRows)
      ds_MN[TILE_WIDTH * TILE_WIDTH + ty * TILE_WIDTH + tx] = B[(m * TILE_WIDTH + ty) * numBColumns + Col];
    else
      ds_MN[TILE_WIDTH * TILE_WIDTH + ty * TILE_WIDTH + tx] = 0.0;    __syncthreads();
    for (int k = 0; k < TILE_WIDTH; ++k)
      Pvalue += ds_MN[ty * TILE_WIDTH + k] * ds_MN[TILE_WIDTH * TILE_WIDTH + k * TILE_WIDTH + tx];
    __syncthreads();
  }
  if (Row < numCRows && Col < numCColumns)
    C[Row * numCColumns + Col] = Pvalue;
  }
```

Para chamar esta função do Kernel, faremos:

```
matrixMultiplyPar <<< dimGrid, dimBlock, 2 * TILE_WIDTH * TILE_WIDTH * sizeof(double) >>> (...);
```

Observando a passagem do terceiro parâmetro para o Kernel que permitirá a alocação do recurso compartilhado entre threads de um mesmo bloco.

Observamos que é de fato necessário que as threads sejam sincronizadas entre as operações em um bloco, pois assim podemos garantir que não será escrito na mesma posição de memória mais de uma vez ao mesmo tempo.

Para comportar matrizes de qualquer tamanho, são atribuídos zeros nas posições que excedem a matriz (dentro de um bloco).

E. Implementação

Para otimizar o uso da CPU+GPU, foi feito um programa que realiza as multiplicações de matrizes cujas entradas são aleatórias (entre -0.5 e +0.5) de tamanhos crescentes. E estas multiplicações são realizadas diversas vezes para cada caso dividindo-se as chamadas das multiplicações com o OpenMP. Ao inserir a diretiva do OpenMP no código temos que dizer que as matrizes a serem trabalhadas são do tipo 'firstprivate', o que diz para o compilador que aquelas matrizes devem ter uma cópia privada em cada thread na CPU, e estas, devem ter o valor previamente atribuído a elas.

Para variar o tamanho do ladrilho e a quantidade de vezes que a operação deve ser realizada, foi criado um script 'run.sh' que executa o programa várias vezes em diferentes configurações para uma análise posterior.

III. RESULTADOS E DISCUSSÕES

Com os dados para cada simulação salvos em vários arquivos através do script 'run.sh' podemos avaliar os resultados:

A. Quanto ao tamanho do ladrilho

Foi avaliado quantas vezes mais rápido foi a execução quando comparada com a sequencial ladrilhada e assim variando-se o tamanho da matriz, teremos:

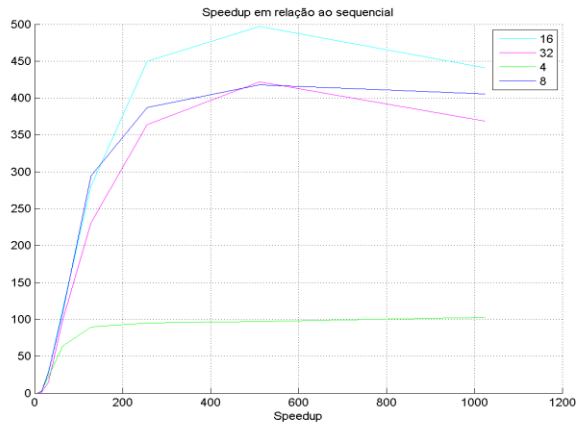

Figura 1: Speedup em relação à execução sequencial para 4 tamanhos de ladrilhos (precisão dupla) executando a multiplicação apenas 1 vez

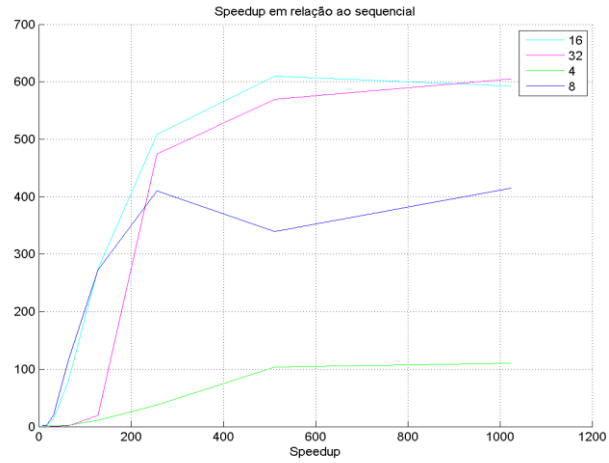

Figura 2: Speedup em relação à execução sequencial para 4 tamanhos de ladrilhos (precisão simples) executando a multiplicação apenas 1 vez

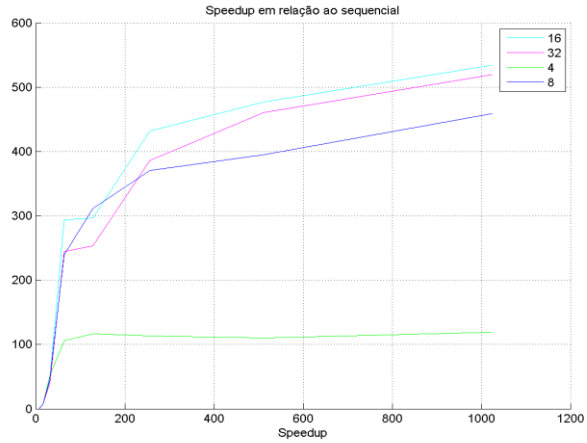

Figura 3: Precisão dupla, executado 50 vezes com OpenMP

Obs.: O conjunto completo de resultados na forma gráfica pode ser encontrado em anexo, pois um uso excessivo de figuras poluiria o relatório escrito.

Para a figura 1, observamos que quando apenas 1 processador requisita a tarefa de multiplicação uma vez, encontramos um valor ótimo para o tamanho da matriz e do ladrilho que correspondem a aproximadamente 550x550 e 16, respectivamente. Verificamos também que podemos utilizar um tamanho maior para a matriz nos outros casos, o que será abordado na sessão III.C.

B. Quanto à quantidade de execuções

Para encontrar a melhor combinação CPU+GPU, variaremos agora a quantidade de vezes que a multiplicação será executada com auxílio do OpenMP começando do tamanho do ladrilho obtido na computação precedente.

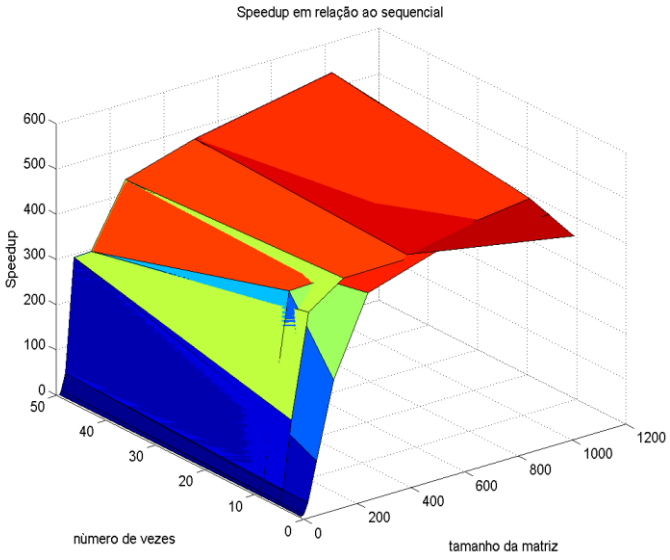

Figura 4: Tamanho do ladrilho: 16 (precisão dupla)

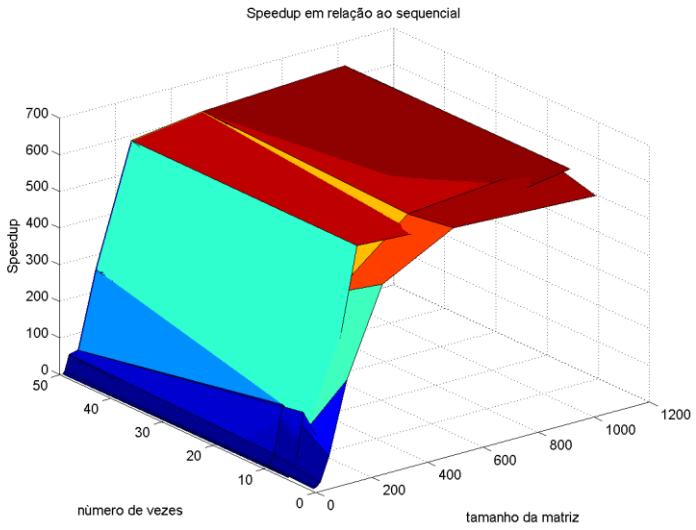

Figura 5: Tamanho do ladrilho: 16 (precisão simples)

Para este caso, será necessário analisar gráficos com configurações diferentes de tamanhos de ladrilhos:

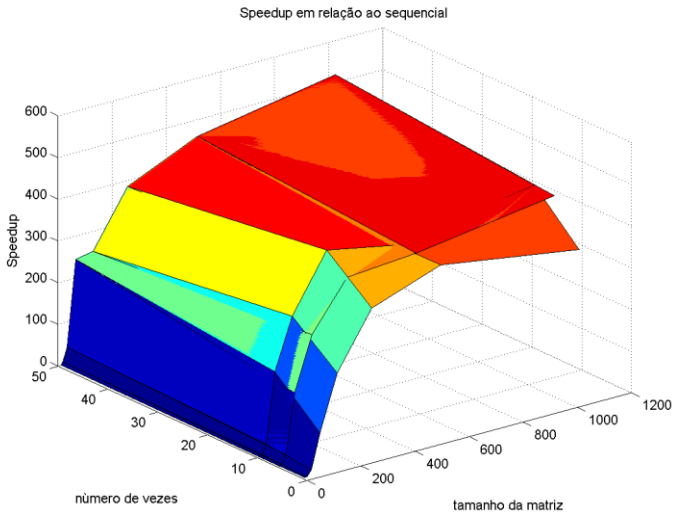

Figura 6:Tamanho do ladrilho: 32 (precisão dupla)

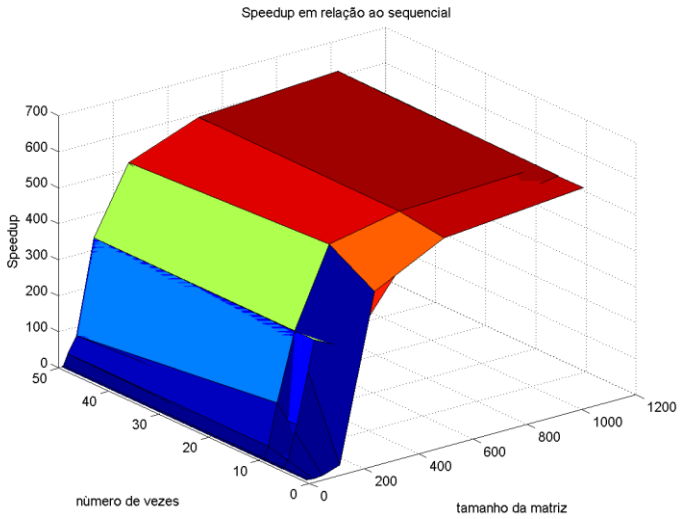

Figura 7:Tamanho do ladrilho: 32 (precisão simples)

Para uma análise mais conclusiva, precisaremos de quantidades maiores de multiplicações, porém, deveríamos reduzir o tamanho das matrizes, pois a partir de 1024x1024, para mais de 100 multiplicações o tempo de processamento do sequencial é extremamente longo, não viabilizando a comparação. Sendo assim, faremos apenas o cálculo de quantas operações de ponto flutuante serão feitas por segundo (gigaflops).

Será modificado o arquivo 'run.sh' e o arquivo 'parmul.cu' para não calcular sequencialmente, mas apenas salvar o tempo gasto no paralelo.

A complexidade da operação em questão é O(n^3), e mais precisamente, serão efetuadas n^3adições e n^3 multiplicações de ponto flutuante (simples ou dupla precisão). Assim teremos:

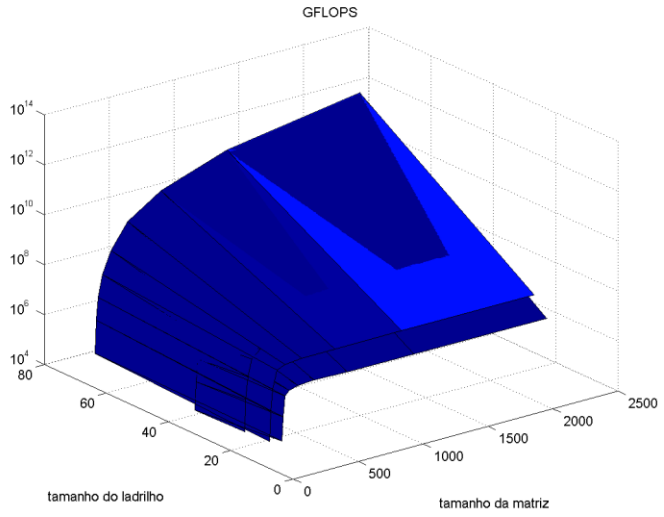

Figura 8: 1000 multiplicações (precisão simples)

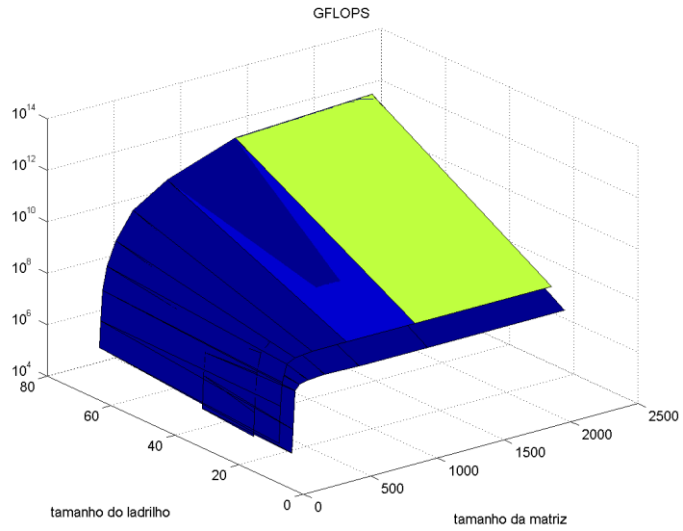

Figura 9:100 multiplicações (precisão simples)

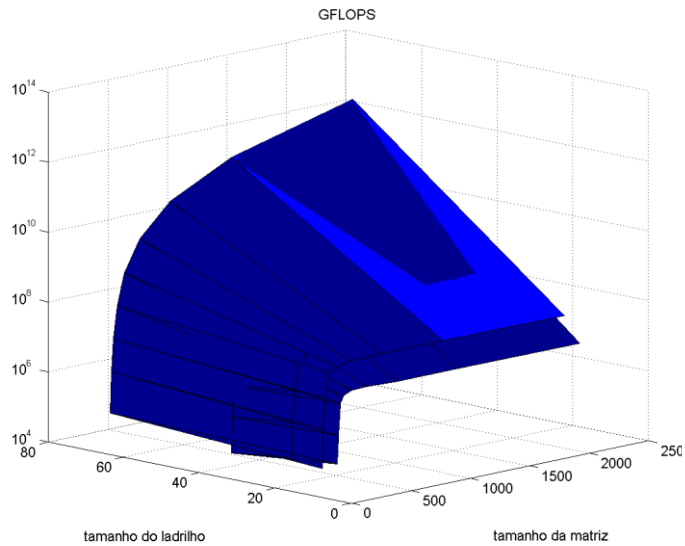

Figura 10:1000 multiplicações (precisão simples)

Observamos que ao escolher um ladrilho de tamanho 64, obtivemos uma performance próxima a aquela fornecida pelo fabricante (1.03 TFLOPS).

Para uma comparação final, faremos uma média de 3 execuções com o tamanho do ladrilho de 64, um tamanho para a matriz de 2048.

Tempo sequencial : 14.1927487 segundos
Tempo em paralelo: 0.029183 segundos
Speedup = 486.336177 Vezes.