Comparação entre CUDA, OpenMP e MPI para Produto Matriz-Vetor

Diego Paiva

Novembro 2020

1 Introdução

Dada uma matriz $A_{n\times n} = \begin{pmatrix} a_{1,1} & a_{1,2} & \dots & a_{1,n} \\ a_{2,1} & a_{2,2} & \dots & a_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n,1} & a_{n,2} & \dots & a_{n,n} \end{pmatrix}$ e um vetor-coluna $B_{n\times 1} = \begin{pmatrix} v_1 \\ v_2 \\ \vdots \\ v_n \end{pmatrix}$, o produto $C = A \cdot B$ pode ser escrito como:

$$C_{n\times 1} = \begin{pmatrix} a_{1,1}v_1 + a_{1,2}v_2 + \dots + a_{1,n}v_n \\ a_{2,1}v_1 + a_{2,2}v_2 + \dots + a_{2,n}v_n \\ \vdots \\ a_{n,1}v_1 + a_{n,2}v_2 + \dots + a_{n,n}v_n \end{pmatrix} \tag{1}$$

Neste trabalho serão examinadas as performances de bibliotecas de computação paralela para o algoritmo do produto matriz-vetor.

2 Algoritmo em C

Abaixo consta uma implementação do algoritmo de multiplicação matriz-vetor em C. A única diferença para a equação (1) é que a matriz de entrada é representada linearmente e os índices variam de 0 a $n-1$ em vez de 1 a n.

```
1  void matrix_vector_multiplication(float *mat, float *vec, float *out, int n)
2  {
3      for (int i = 0; i < n; i++) {
4          float sum = 0;
5
6          for (int j = 0; j < n; j++)
7              sum += mat[i * n + j] * vec[j];
8
9          out[i] = sum;
10     }
11 }
```

Para aumentar o tempo de computação para realização de experimentos computacionais, o laço da linha 14 foi envolvido em um outro laço que executa a operação aritmética k vezes.

3 Experimentos Computacionais

Além da versão sequencial, o algoritmo foi implementado para as bibliotecas CUDA, OpenMP e MPI fazendo as adaptações necessárias no código apresentado na Seção 2. Foram testados quatro valores diferentes de n: 2048, 4096, 8192 e 16384. Para cada valor de n, também foram testados dois valores distintos de k (o número de vezes que a conta paralela é realizada). Os resultados são apresentados nas Seções a seguir, e os valores representam a média de cinco execuções para cada valor de n.

Todas as implementações foram feitas em sistema operacional Linux. O algoritmo sequencial, bem como os algoritmos em OpenMP e MPI, foram executados em CPU Intel Xeon E5620. Já o algoritmo em CUDA foi executado em uma GPU NVIDIA Tesla M2075. Foi empregado um total de 8 *threads* para a execução dos algoritmos paralelos em OpenMP e MPI.

3.1 $k = 4096$

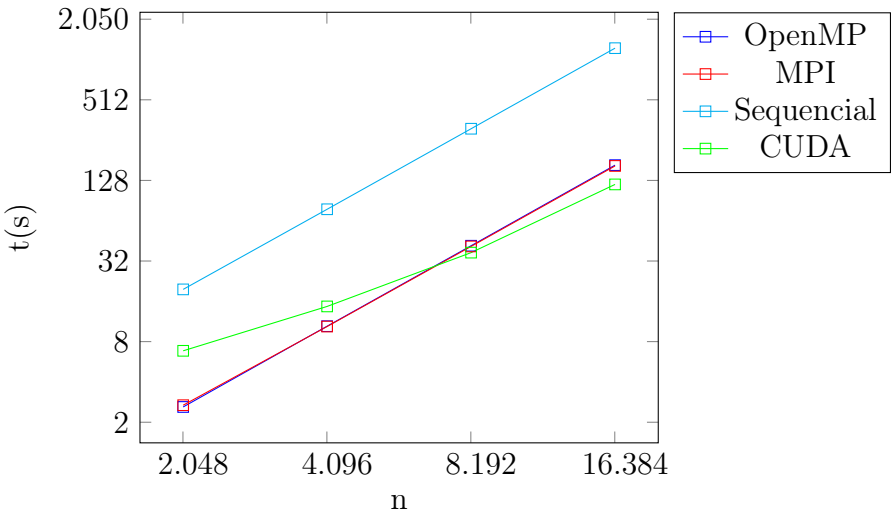

Figura 1: Relação entre tamanho da entrada n e tempo (em segundos) para $k = 4096$.

Pelo gráfico é possível observar que, como esperado, as bibliotecas de programação paralela fornecem ganho de desempenho significativo em relação à implementação sequencial. Na prática, não houve diferença considerável entre OpenMP e MPI, cujas retas ficaram sobrepostas na Figura 3.1. O mais interessante a ser notado é que para este problema, CUDA só superou OpenMP e MPI a partir de $n = 8192$, quando registrou uma redução de cerca de 10,65% do tempo de computação em relação àquelas bibliotecas. Já para $n = 16384$, a redução foi um pouco mais expressiva, em torno de 27,71%. Isto indica que a opção pela utilização de CUDA deve ser pensada para problemas que são suficientemente grandes. Na tabela 1 são reportados os *speedups* que cada biblioteca obteve em relação à execução sequencial.

n	Sequencial		CUDA		OpenMP		MPI	
	t	S	t	S	t	S	t	S
2048	19,64	1,0	6,83	2,88	2,60	7,55	2,68	7,33
4096	77,90	1,0	14,65	5,32	10,43	7,47	10,37	7,51
8192	311,07	1,0	36,95	8,42	41,60	7,48	41,11	7,57
16384	1243,45	1,0	119,41	10,41	166,32	7,48	164,06	7,58

Tabela 1: Tempo t em segundos e *speedup* S obtido por cada implementação para $k = 4096$.

3.2 $k = 65536$

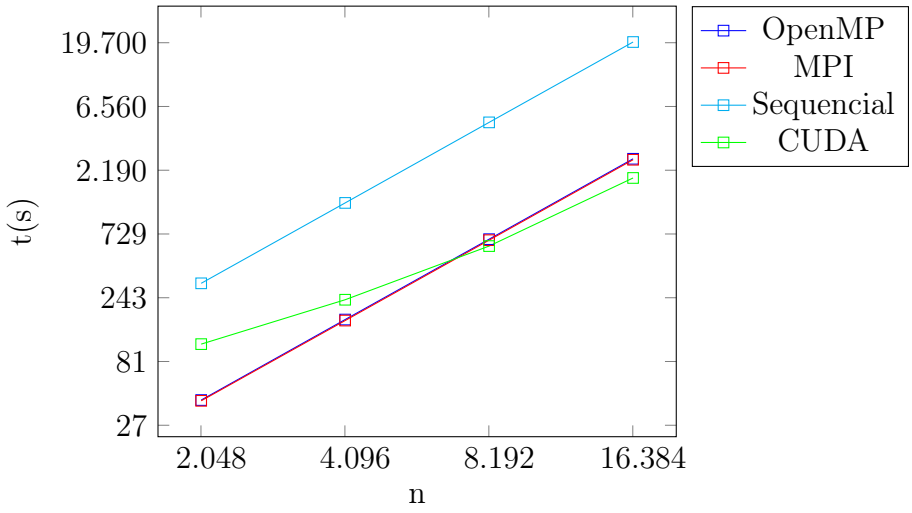

Figura 2: Relação entre tamanho da entrada n e tempo (em segundos) para $k = 65536$.

Para o caso de $k = 65536$, foi observado o mesmo comportamento da Seção 3.1. Somente para $n = 8192$ e $n = 16384$ a implementação em CUDA superou o desempenho das demais implementações. Nestes casos, as reduções do tempo de computação foram de 10,42% e 27,55%, respectivamente. Novamente, a diferença entre as bibliotecas OpenMPI e MPI foi irrisória. Contudo, para este valor de k, é interessante observar como o tempo demandado pela implementação sequencial é demasiadamente longo. Para $n = 8192$, o tempo demandado foi de 1 hora e 23 minutos, enquanto que as versões paralelas exigiram algo em torno de 10 a 11 minutos. Esta diferença é ainda mais perceptível para $n = 16384$: a versão sequencial exigiu, em média, um total de 5 horas e meia, enquanto que OpenMPI e MPI exigiram cerca de 44 minutos, e CUDA 32 minutos. A tendência é que estes valores sejam muito mais discrepantes para valores ainda maiores de n e k. Os *speedups* podem ser visualizados na tabela 2.

4 Conclusões

Com os experimentos realizados sobre o problema estudado, foi possível observar que a computação paralela é crucial para garantir resultados em espaços mais curtos de tempo. Ademais, cada uma das bibliotecas analisadas possui suas particularidades, estejam estas

n	**Sequencial**		**CUDA**		**OpenMP**		**MPI**	
	t	S	t	S	t	S	t	S
2048	311,26	1,0	109,17	2,85	41,65	7,47	41,10	7,57
4096	1243,41	1,0	234,52	5,30	166,72	7,46	163,84	7,59
8192	4980,87	1,0	591,33	8,42	665,75	7,48	654,47	7,61
16384	19888,64	1,0	1911,30	10,41	2659,86	7,48	2616,34	7,60

Tabela 2: Tempo t em segundos e *speedup* S obtido por cada implementação para $k = 65536$.

relacionadas ao desempenho ou à facilidade de programação. No caso de CUDA, a tendência é que para problemas suficientemente grandes, o desempenho compense o esforço de programação, que costuma ser relativamente mais árduo que em OpenMP e MPI. É importante destacar que estas duas últimas bibliotecas forneceram *speedup* quase linear para todos os tamanhos de entrada, o que é um resultado excelente do ponto de vista de escalabilidade.