Análise de Complexidade do Método de Eliminação Gaussiana em GPU Através da Ferramenta EMA

Anderson Zudio[1], **Raquel M. de Souza**[1], **Igor M. Coelho**[1], **Cristiane O. Faria**[1], **Fabiano S. Oliveira**[1]

[1]Universidade do Estado do Rio de Janeiro, Brazil

azudio@ime.uerj.br, raquelmarcolino25@gmail.com,

{igor.machado, cofaria, fabiano.oliveira}@ime.uerj.br

Abstract. *In this work, it is presented an empirical analysis using a novel tool called EMA (Empirical Analysis of Algorithms). The purpose of EMA is to provide empirical analysis regarding the computional resources (time and space) of a specific algorithm, by means of iterative executions over an implementation given as input. As an application, we consider the resolution of linear systems with the Gaussian Elimination algorithm, divided in two phases. In the first phase, we perform a comparison with the literature exploring the paralelism of GPUs (Graphics Processing Units), and in phase two, we propose a novel algorithm that explores the parallelism between threads of a same block, in order to obtain a reduction in the total number of steps. The complexity analysis of each phase of the method in its parallel and sequential versions are determined by the EMA tool, also the behavior analysis of the memory transfers between device and host.*
Keywords: *Empirical Analysis of Algorithms, Large Systems, Gaussian Elimination, GPU Computing.*

Resumo. *Neste trabalho é apresentada uma análise empírica utilizando uma nova ferramenta chamada EMA (Empirical Analysis of Algorithms). O objetivo da EMA é fornecer análises empíricas sobre o uso de recursos (tempo e espaço) de um determinado algoritmo, através de execuções iterativas sobre a implementação fornecida como entrada. Como aplicação, consideramos a resolução de sistemas lineares através do algoritmo de Eliminação Gaussiana, dividido em duas etapas. Para a primeira etapa, apresentamos uma comparação com a literatura utilizando o paralelismo das GPUs (Unidade de Processamento Gráficos), enquanto que na segunda etapa propomos um algoritmo inovador que explora o paralelismo entre threads de um mesmo bloco, de forma a obter reduções no número de passos total. A análise de complexidade de cada etapa do método em sua versão sequencial e paralela são determinadas com os recursos oferecidos pela ferramenta EMA, e também o comportamento das operações de transferência de memória entre dispositivo e hospedeiro.*
Palavras-Chave: *Análise Empírica de Algoritmos, Sistemas de Grande Porte, Eliminação Gaussiana, Computação GPU.*

1. Introdução

O avanço de novas tecnologias na fabricação de *hardware* permitiu o surgimento de arquiteturas massivamente paralelas como as Unidades de Processamento Gráfico, ou GPUs, e

o potencial de paralelismo que pode ser extraído nesses dispositivos ultrapassam de forma significativa aqueles encontrados nas CPUs convencionais [Kirk and Wen-mei 2012]. Uma diferença fundamental das GPUs em relação às CPUs é o seu massivo número de núcleos (ou *Stream Processors*) que podem processar milhares de threads executadas em *warps* e organizadas em blocos, que exploram os rápidos acessos de memória compartilhada nos *Stream Multiprocessors* (SM), aliado à grande quantidade de memória global disponível. Estes fatores, aliados a seu baixo custo de aquisição, tornam a GPU uma alternativa bastante atrativa. Assim, houve um aumento significativo no número de trabalhos que buscam explorar e disseminar o uso destes dispositivos [Gonçalves et al. 2015], muitos dos quais envolvem a comparação de diferentes ambientes de programação como CUDA[1] e OpenACC[2].

Problemas de grande porte ou sistemas lineares de grande porte são encontrados em várias aplicações reais, como por exemplo, na meteorologia, engenharia de reservatórios, sísmica, sistemas financeiros, dentre outras, demandando um grande custo computacional, tornando-os fortes candidatos a utilizarem GPUs. Dentre os métodos de resolução para sistemas lineares, os métodos diretos são os mais utilizados, pois são os mais robustos e eficientes e o primeiro a ser desenvolvido foi o método da Eliminação Gaussiana [Golub and van Loan 1989].

Desde a década de 80 existe um grande interesse no estudo de implementações da Eliminação Gaussiana em máquinas paralelas SIMD/MIMD, visto sua grande eficiência teórica de paralelização [Robert and Trystram 1989, Marrakchi and Robert 1989]. Coprocessadores FPGAs e GPUs são usados em [Che et al. 2008], focados nos problemas de Eliminação de Gauss, *Data Encryption Standard* (DES) e *Needleman-Wunsch* e os resultados indicam que a Eliminação Gaussiana tem melhor performance na GPU do que no FPGA, o oposto do resultado com o DES. Em [Buluç et al. 2008] é apresentada uma técnica inovadora para resolução de sistemas lineares inspirada na Eliminação Gaussiana, adaptada para uso em GPUs. Em testes computacionais, a proposta foi capaz de obter um speedup de 28x em relação à respectiva implementação CPU utilizando recursos de vetorização. [Gonçalves et al. 2015] apresentam uma implementação didática da Eliminação Gaussiana para CUDA e OpenACC.

Neste contexto, este trabalho apresenta a nova ferramenta chamada EMA (*Empirical Analysis of Algorithms*) que permite facilitar a análise empírica de algoritmos em ambientes heterogêneos conduzindo os experimentos e interpretando os resultados. Desenvolvida em 2015, ela busca estimar as complexidades de tempo e de espaço do algoritmo através de execuções iterativas do mesmo com diversos parâmetros variáveis, armazenando o tempo decorrido e a memória consumida de cada execução [Oliveira 2016, de Souza 2015]. Com a base de dados recolhida ao longo das execuções, portanto, é possível estimar uma equação representante da complexidade computacional que, ao mesmo tempo que busca estar em conformidade com os resultados das execuções realizadas, busca também prever seu comportamento para valores de parâmetros não executados. Consequentemente, exibimos nossa implementação em GPU do método de Eliminação Gaussiana com diversos resultados empíricos quanto ao comportamento do algoritmo paralelo através da ferramenta EMA.

[1]Página de CUDA no site da NVIDIA: $http://www.nvidia.com/object/cuda\_home\_new.html$
[2]Página do OpenACC: $http://www.openacc.org$

O restante deste trabalho é organizado da seguinte forma: a Seção 2 apresenta a ferramenta EMA em detalhes, bem como seu mecanismo interno de ajuste das melhores equações. Na Seção 3 o algoritmo clássico de Eliminação Gaussiana é descrito na sua forma sequencial e, a seguir, são apresentadas propostas de paralelização GPU para esta técnica. Os resultados computacionais obtidos através da ferramenta EMA são descritos e analisados na Seção 4. Finalmente, a Seção 5 conclui o trabalho com as principais observações dos resultados computacionais e apresenta perspectivas futuras.

2. A ferramenta EMA

A ferramenta EMA realiza medições através de dois arquivos de entrada: uma implementação do algoritmo cuja complexidade deverá ser determinada; e um *script* capaz de gerar entradas com tamanhos arbitrários para alimentar este programa. Tal *script*, por sua vez, recebe por entrada um valor que representa o tamanho da entrada a ser gerada para a implementação testada. Então, o EMA varia tal valor em várias execuções deste *script* com consequente execução do programa a ser avaliado de modo a medir seu desempenho. Para este trabalho, o parâmetro que representará o tamanho da entrada é n, o número de equações (e variáveis) do sistema linear. Assim, o *script* irá sempre gerar uma matriz quadrada com tamanho $n \times n$ para executar o algoritmo de Eliminação Gaussiana, de cujas execuções serão extraídos o tempo e o espaço em memória consumidos para posterior análise, feita também pela própria ferramenta.

O critério de parada da ferramenta se dá através de um limite de tempo imposto pelo próprio usuário da ferramenta ou através de valores de parâmetros fixos. Se a primeira maneira for escolhida, a ferramenta é capaz de realizar um processo chamado *calibração*, onde ela estipula valores de n que respeitam o limite de tempo estipulado.

Após a calibração, a ferramenta utiliza os valores estipulados de n para serem executados e recolhe os dados a respeito do consumo de tempo e memória. A base de dados forma então diversos pontos, onde, para cada entrada com valor n, é produzida uma saída em tempo T (e/ou consumo de memória M), o que caracteriza uma função. A ferramenta então passa ao objetivo de estimar as complexidades da função, o fazendo através de testes com variados coeficientes de ajuste da seguinte equação:

$$T(n) = a_0 n^{a_1} a_2^{n^{a_3}} \log_2 n^{a_4} (\log_2 n)^{a_5}, \text{ com } a_0, a_2 > 0.$$

Os coeficientes a_0, a_1, a_2, a_3, a_4 e a_5 da equação são ajustados baseados em processos de ajuste de Regressão não-linear, que aqui chamamos de *fitting*. O objetivo do *fitting* é determinar valores para os coeficientes que minimizem o erro de estimação dos pontos obtidos. Após o *fitting*, a ferramenta retorna várias equações, cujos resultados mais representam os dados. Estas equações podem ser divididas em cinco diferentes grupos:

- **Equação de erro mínimo:** esta é a equação estimada pela ferramenta com a menor margem de erro possível. O cálculo deste é feito através do cálculo do erro quadrático médio [Lehmann and Casella 1991].
- **Equações distinguíveis:** estas são equações que se aproximam muito da equação de erro mínimo, mas que podem ser distinguidas desta através de novas execuções com novos pontos computacionalmente viáveis de serem executados. Isto ocorre quando o erro em relação à equação de erro mínimo é de pelo menos 5% em um

ponto qualquer da função. Este ponto, sendo o menor de todos eles, é chamado de *tie breaker*, informado pela ferramenta.

- **Equações não-distinguíveis:** estas são equações que se aproximam muito da equação de erro mínimo, tais como as equações distinguíveis, mas que, ao contrário destas, não podem ser encontradas através da execução de novos pontos, pois seu *tie breaker* torna futuras execuções inviáveis para os limites estabelecidos. Logo, de forma prática, equações deste grupo podem ser consideradas equivalentes à equação de erro mínimo.
- **Equações equivalentes:** estas são equações cujo seu erro em relação à equação de erro mínimo é de menos de 5% para todos os pontos calculados, sendo possível ser ajustado pelo usuário. Uma equação assim segue muito proximamente aquela de erro mínimo. As equações deste grupo podem ou não ser distinguíveis.
- **Melhor equação:** esta é a equação equivalente com o menor número de coeficientes e, em caso de empate, a com o menor erro absoluto. Por ser uma equação equivalente à equação de erro mínimo, ela é capaz de explicar melhor o comportamento da complexidade de forma mais simplificada.

Após apresentar todas as equações estimadas em seus respectivos grupos, a ferramenta encerra sua operação, cabendo ao usuário analisar seus resultados. Complexidades de outros recursos, fora tempo e espaço, têm fácil adaptação para o cálculo através da ferramenta EMA. A versatilidade oferecida pela ferramenta EMA e seu eficiente *fitting* de equação, então, faz com que ela se torne uma ferramenta conveniente para a pesquisa e estudo de algoritmos, incluindo algoritmos paralelos como no presente trabalho.

3. Eliminação Gaussiana

O método de Eliminação Gaussiana é utilizado na resolução de sistemas lineares, $Ax = B$, onde A é a matriz inversível de coeficientes de ordem n, x o vetor coluna de incógnitas e B o vetor coluna dos termos constantes [Roger 1970, Golub and van Loan 1989]. O método trabalha com a matriz aumentada $M = [A|B]$ e pode ser sumarizado em duas etapas:

- **Etapa 1 – Escalonamento:** Consiste em obter uma matriz triangular superior M' através de operações elementares nas linhas de M. Assim, obtemos um sistema linear equivalente mais simples em forma escalonada (veja o Algoritmo 1). A complexidade do método é $\Theta(n^3)$.
- **Etapa 2 – Substituição para trás:** A segunda etapa (Algoritmo 2), obtém o valor numérico de cada incógnita utilizando o sistema linear escalonado. Isto é feito através da substituição passo a passo, de baixo para cima, das incógnitas já obtidas nas equações precedentes. A complexidade desta etapa é $\Theta(n^2)$.

3.1. Versão CUDA C

A versão paralela do método executa ambas as etapas na GPU utilizando a extensão CUDA da linguagem de programação C. A entrada consiste do vetor (*array*) sistema que deve conter a matriz estendida M, o inteiro n com o número de equações e um vetor para receber a solução numérica do sistema. A matriz estendida é passada para a memória global do dispositivo e nela reside até o final da computação. Como a função está interessada somente na resposta, a matriz escalonada nunca é transferida de volta para memória

Algoritmo 1 Eliminação Gaussiana - Escalonamento

1: **função** ESCALONA(Matriz: A, Vetor coluna: B)
2: $n \leftarrow$ ordem de A
3: $M \leftarrow [A|B]$ ▷ Matriz aumentada
4: **para** $k \leftarrow 1$ até $n-1$ **faça**
5: $Pivo \leftarrow M[k][k]$
6: **para** $i \leftarrow k+1$ até n **faça**
7: $Coef \leftarrow \dfrac{M[i][k]}{Pivo}$
8: **para** $j \leftarrow k$ até $n+1$ **faça**
9: $M[i][j] \leftarrow M[i][j] - Coef \cdot M[k][j]$
retornar M

Algoritmo 2 Eliminação Gaussiana - Substituição para trás

1: **função** SUBSTITUIÇÃO(Matriz estendida: M)
2: $n \leftarrow$ número de linhas de M
3: **para** $i \leftarrow n$ até 1 **faça**
4: $x[i] \leftarrow M[i][n+1]$ ▷ $M[i][n+1] = b_i$
5: **para** $j \leftarrow n$ até $i+1$ **faça**
6: $x[i] \leftarrow x[i] - M[i][j] \cdot x[j]$
7: $x[i] \leftarrow \dfrac{x[i]}{M[i][i]}$
retornar x

do hospedeiro. Observe que a matriz M é referenciada por um único ponteiro, onde o acesso matricial é feito através de contas triviais. A construção foi feita desta forma para a aplicação não ter que executar n chamadas de *cudaMemcpy* na alocação do sistema na memória global da GPU evitando um *overhead* desnecessário. Vale notar que, como descrito no Capítulo 6 de [Kirk and Wen-mei 2012], o acesso de memória em matrizes linearizadas na memória global da GPU tem um acesso mais favorável em relação a matrizes alocadas em sua forma de ponteiro duplo.

Nenhuma técnica de sobreposição de computação com a transferência de memória entre hospedeiro e dispositivo foi utilizada. A implementação foi projetada para obter o gasto de recurso computacional da chamada *cudaMemcpy* separadamente visando a passagem deste gasto para o EMA. O Código 3.1 mostra o trecho de código *kernel* que implementa a Etapa 1 do método omitindo o processamento dos retornos *cudaErro_t* das funções CUDA por simplicidade. Os parâmetros de entrada são respectivamente a matriz estendida, o número de equações n, o número da linha l em que se encontra o pivô e sua coluna k que estão na diagonal secundária diferindo da Etapa 1 da implementação sequencial. Na versão sequencial cada pivô está presente na diagonal principal da matriz, mas na versão paralela eles estão na diagonal secundária. Assim, o resultado final do escalonamento possibilita a implementação de um *kernel* simplificado para a próxima etapa. Note que a diferença não afeta a complexidade ou performance do algoritmo, pois consiste de uma variação equivalente do mesmo.

Cada *thread* afeta um coeficiente específico a_{ij} da matriz estendida. A memória

compartilhada é carregada com os valores que todas as *threads* do bloco vão acessar em comum com a linha do pivô. A sincronização da linha 7 obriga todas as *threads* do bloco a esperar por esse resultado. Assim, temos um acesso mais rápido no próximo passo em que cada *thread* deve computar o novo valor de seu coeficiente a_{ij}, mas antes verificando se sua coordenada é válida dentro da matriz e se o seu coeficiente precisa ser alterado.

Código 3.1: Kernel da Etapa 1

```
1  __global__ void k_escalona(float *sistema, int n, int l, int k){
2     int i = bId.y * bDim.y + tId.y, j = bId.x * bDim.x + tId.x;
3     int n1 = n+1, in1 = i*n1, ln1 = l*n1;
4     const float pivo = sistema[ln1 + k];
5     __shared__ float linha[DIMTHREAD];
6     if(tId.y == 0) linha[tId.x] = sistema[ln1+j];
7     __syncthreads();
8     if(!(i < n) && (j < n1 )) return;
9     if((i > l) && ((j <= k) || (j == n)))
10        sistema[in1+j] -= sistema[in1+k]/pivo*linha[tId.x];}
```

Para a Etapa 2, são apresentadas duas opções para o código *kernel* (veja Código 3.2 e Código 3.3). Os parâmetros de entrada de ambas seguem o mesmo modelo do *kernel* anterior, onde l é o número da linha do pivô e k sua coluna. O Código 3.2 é uma primeira abordagem simples que extrai pouco paralelismo da Etapa 2. A ideia inicial é executar as multiplicações entre os coeficientes e as incógnitas já computadas de forma paralela, porém cada *thread* no final de sua execução constrói o valor da incógnita através de operações de adição atômica. Assim, a execução deste último passo se torna praticamente sequencial. Note que após a execução deste código, é necessário executar a chamada de uma *thread* especial externa para efetuar a divisão do resultado pelo coeficiente a_{lk} associada à incógnita computada na iteração atual.

Código 3.2: Kernel da Etapa 2 – Versão 1

```
1  __global__ void k_subs_tras1(float *s, float *r, int n, int l, int k){
2     int n1 = n+1, j = bId.x*bDim.x + tId.x;
3     if(j > k) return;
4     else if(j == k) atomicAdd(r+k,s[l*n1+n]);
5     else atomicAdd(r+k,r[j]*(-s[l*n1+j]));}
```

Código 3.3: Kernel da Etapa 2 – Versão 2

```
1  __global__ void k_subs_tras2(float *s, float *r, int n, int l, int k){
2     int n1 = n+1, ln1 = l*n1, j = bId.x*bDim.x + tId.x, i;
3     __shared__ float linha[NTHREAD];
4     linha[tId.x] = (j < k) ? (-s[ln1+j])*r[j] : 0;
5     __syncthreads();
6     for(i = NTHREAD_DIV2; i > 0; i /= 2){
7        if(tId.x >= i) return;
8        linha[tId.x] += linha[tId.x+i];
9         __syncthreads();}
10    if(j == 0) atomicAdd(&r[k],(linha[0]+s[ln1+n])/s[ln1+k]);
11    else atomicAdd(&r[k], linha[0]/s[ln1+k]);}
```

O Código 3.3 requer um simples pré-processamento para obter o valor da primeira incógnita que consiste em solucionar a equação $n - 1$. A ideia principal é efetuar a chamada da *kernel* uma vez para cada equação em processo de substituição minimizando o

número de *threads* que vão trabalhar em coeficientes iguais a zero, ou seja, os coeficientes além da diagonal secundária. Primeiro, cada *thread* executa a multiplicação do coeficiente a_{ij} com a resposta x_j obtida em uma equação precedente, onde i é o número da equação atual e j a coluna da incógnita computada. Esse resultado é mantido na memória compartilhada, onde a sincronização da linha 5 obriga todas as *threads* do bloco a esperar por esse carregamento completo para o próximo passo. As *threads* que estão trabalhando em colunas além da diagonal secundária colocam zero na memória compartilhada. O último passo é somar os valores que estão na memória compartilhada de cada bloco e juntá-las para formar a resposta da iteração atual. Uma técnica de redução foi utilizada para obter este efeito de maneira eficiente: nas iterações do *loop* da linha 6, cada *thread* deve somar dois coeficientes de modo que a cada iteração metade das *threads* ativas no bloco são desligadas. Na última iteração deve restar somente uma *thread* ativa por bloco com o valor reduzido de seu bloco na primeira posição da memória compartilhada. Esta *thread* fica encarregada de montar a resposta da incógnita k através de uma operação de soma atômica. Note que a *thread* da primeira coluna é a que soma o valor do termo constante. Ao final, o vetor de incógnitas é transferido para a memória do hospedeiro e uma última chamada de sincronização entre dispositivo e hospedeiro é feita.

4. Experimentos Computacionais

Para a realização dos experimentos utilizando a ferramenta EMA na condução dos testes e interpretação dos resultados, foram gerados dois conjuntos de testes. Cada conjunto consiste de diversas matrizes de ordem n, sendo o primeiro composto por matrizes esparsas e o segundo por matrizes cheias. Foram estipulados valores de n tais que $2 \cdot 10^3 \leq n \leq 5 \cdot 10^3$. Nesta seção, os seguintes acrônimos serão utilizados para identificar a versão do *kernel* da Etapa 2 sendo avaliada: Kv1 e Kv2, para a versão 1 e 2 respectivamente (Veja código 3.2 e 3.3). A configuração utilizada no EMA estabelece que a ferramenta deve utilizar a matriz gerada previamente correspondente à ordem desejada para cada valor de n, executando o algoritmo de entrada com marcações de tempo, a fim de obter o tempo total de execução (englobando todo o processo), o tempo de processamento da Etapa 2 (usando Kv1 ou Kv2) e o tempo de transferência da matriz de entrada entre a memória do hospedeiro e a do dispositivo.

A Subseção 4.1 mostra as respostas fornecidas pela ferramenta EMA para o método de Eliminação Gaussiana sequencial, a fim de verificar se o resultado empírico da ferramenta é compatível ao resultado teórico esperado. Na Subseção 4.2 apresentamos os resultados que a ferramenta fornece para a implementação CUDA do algoritmo e comparamos as duas técnicas de paralelismo propostas para a Etapa 2. Na última subseção, uma comparação com o código exibido em [Gonçalves et al. 2015] é apresentada, com o intuito de validar os códigos que foram propostos neste trabalho na Seção 3. Para a execução de todos os testes foi utilizado um ambiente munido com as seguintes especificações:

- Intel® Core ™ i7-4820K Processor 3.7 GHz (somente um core foi utilizado)
- CPU com 16GB RAM
- GeForce GTX 780, contendo 2304 CUDA Cores: 12 SMs com 192 cores
- Sistema Operacional Ubuntu 14.04 LTS (x86_64)

A calibração dos parâmetros de lançamento dos *kernels* GPU foi feita manualmente através de testes com matrizes de ordem 2000. Os melhores resultados obtidos

para a Etapa 1 envolvem a utilização de blocos bidimensionais de 16×16 *threads*. Para ambas versões da Etapa 2, os resultados que minimizaram o tempo de execução foram blocos unidimensionais de tamanho 64.

4.1. Algoritmo Sequencial: Resultados de Complexidade

Após realizar a execução do algoritmo para as diversas entradas, a ferramenta EMA provê várias equações que descrevem o comportamento dos pontos correspondentes às médias obtidas do recurso em questão, destacando a melhor equação e a equação de erro minímo. A Tabela 1 apresenta as equações destacadas pela ferramenta para o tempo total de execução e o tempo de processamento da Etapa 2, apresentando também a complexidade determinada na melhor equação. Os resultados são compatíveis à complexidade teórica do método de Eliminação Gaussiana sendo estes $\Theta(n^3)$ para o algoritmo, e a Etapa 2 sendo $\Theta(n^2)$ [Roger 1970]. As Figuras 1 e 2 mostram as equações relatadas na Tabela 1 em conjunto com as marcações das médias obtidas pela ferramenta EMA durante as simulações, comprovando que estas se ajustam aos pontos.

Tabela 1. Resultados reportados pelo EMA para o algoritmo sequencial da Eliminação Gaussiana

	Melhor Equação	**Equação de Erro mínimo**	**Complexidade**
Tempo Total			
Esparsa	$1,079 \times 10^{-6}\ n^3$	$1,400 \times 10^{-6}\ n^{2,97}$	$\Theta(n^3)$
Cheia		$1,492 \times 10^{-6}\ n^{2,96}$	$\Theta(n^3)$
Etapa 2			
Esparsa	$1,374 \times 10^{-6}\ n^2$	$7,409 \times 10^{-7}\ n^{2,07}$	$\Theta(n^2)$
Cheia		$3,963 \times 10^{-7}\ n^2\ \log_2^{0,5} n$	$\Theta(n^2)$

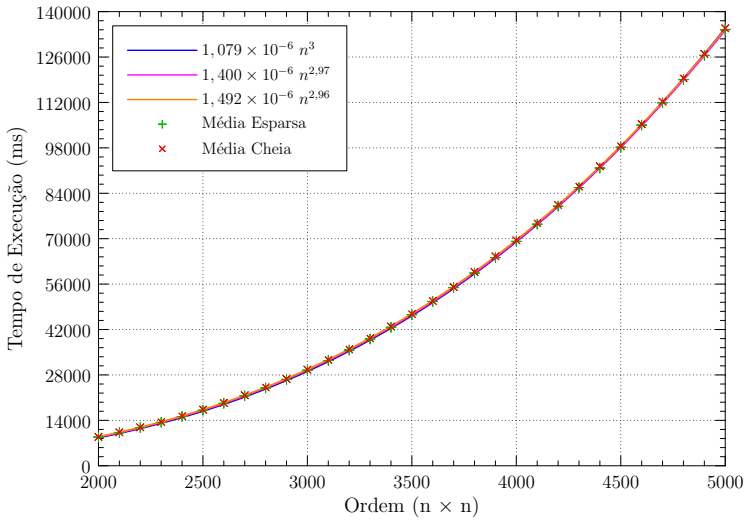

Figura 1. Equações reportadas pelo EMA referente ao tempo total de execução da implementação sequencial.

4.2. Algoritmo Paralelo/CUDA: Resultados de Complexidade

Os resultados coletados na Tabela 2 são referentes à execução do conjunto de testes com matrizes esparsas, pois respostas similares foram obtidas com matrizes cheias. As

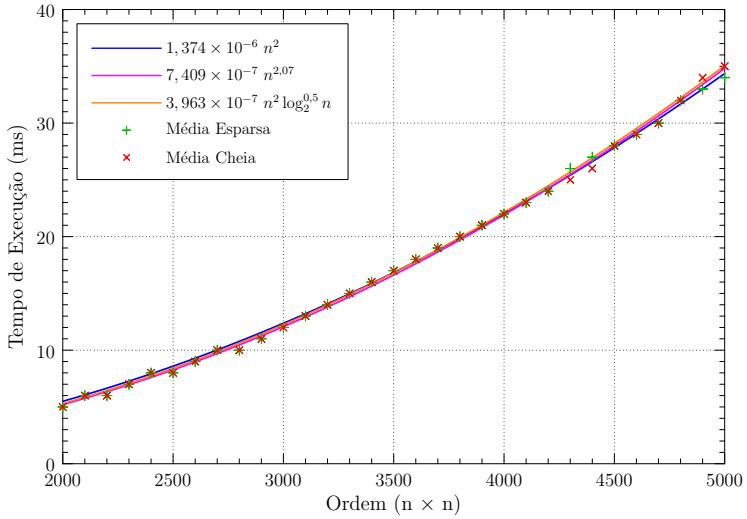

Figura 2. Equações reportadas pelo EMA referente ao tempo de execução da Etapa 2 do algoritmo sequencial.

Tabela 2. Equações reportadas pelo EMA para o algoritmo de Eliminação Gaussiana em GPU

	Equação reportada	Tipo	Complexidade
Tempo Total	$3,668 \times 10^{-4}\ n^2$ $1,375 \times 10^{-5}\ n^{1,5} \log_2^3 n$ $4,765 \times 10^{-5}\ n^{1,5} \log_2^{2,5} n$	Melhor Equação Erro Mínimo Não-distinguível	$\Theta(n^2)$
Kv1	$2,038 \times 10^{-3}\ n \log_2^{3,5}$ $3,320 \times 10^{-1}\ n^{1,43}$	Melhor Equação Erro Mínimo	$\Theta(n \log_2^{3,5} n)$
Kv2	$4,715 \times n$ $3,813 \times 10^{-1}\ n \log_2 n$	Melhor e Erro Mínimo Não-distinguível	$\Theta(n)$
Transferência CPU/GPU	$9,631 \times 10^{-4}\ n^2$ $6,223 \times 10^{-2}\ n^{1,5}$	Melhor equação Erro mínimo	$\Theta(n^2)$

equações destacadas pelo EMA como Melhor Equação e Erro Mínimo e as complexidades determinadas compatíveis com a literatura são apresentadas. Além disso, ela também mostra algumas das equações equivalentes que apresentam um erro pequeno em relação a equação de erro mínimo. Todos os resultados são sumarizados nas Figuras 3, 4 e 5 que, além das equações reportadas, mostram as marcações de tempo médio obtidas.

Na Figura 4 podemos observar que as técnicas utilizadas na Etapa 2 apresentam complexidades distintas. Apesar de Kv2 utilizar várias barreiras de sincronização, o fato dela utilizar uma operação atômica por bloco extrai uma quantidade maior de paralelismo. Também podemos notar que o comportamento assintótico de Kv2 é favorável em relação à Kv1, que realiza várias operações atômicas por bloco, sugerindo que Kv2 executa em menos tempo.

Ao paralelizar a Etapa 1 do método, a ferramenta indica que estamos alterando sua complexidade, portanto, se a Etapa 2 permanecer sequencial, a complexidade final do algoritmo pode ser alterada mesmo se tratando de uma porção que exige menor tempo de processamento. Ao paralelizar a Etapa 2 de forma eficiente podemos observar que ela não

deve alterar a complexidade final do algoritmo neste caso. Outra observação é que a ferramenta reporta uma equação distinguível quadrática para Kv1 como equivalente, exibindo um erro proporcionalmente alto em relação às outras, o que sugere uma complexidade equivalente ao sequencial para esta técnica quando se trata de matrizes de ordem superior.

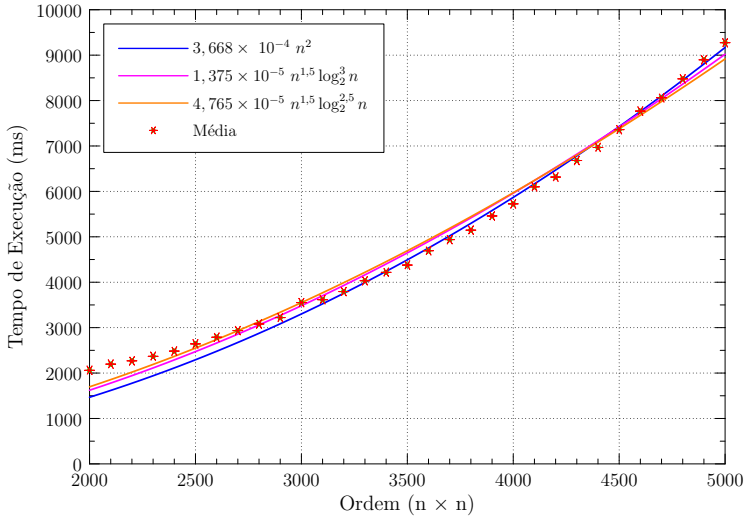

Figura 3. Equações reportadas pelo EMA referente ao tempo total de execução utilizando Kv2 na Etapa 2.

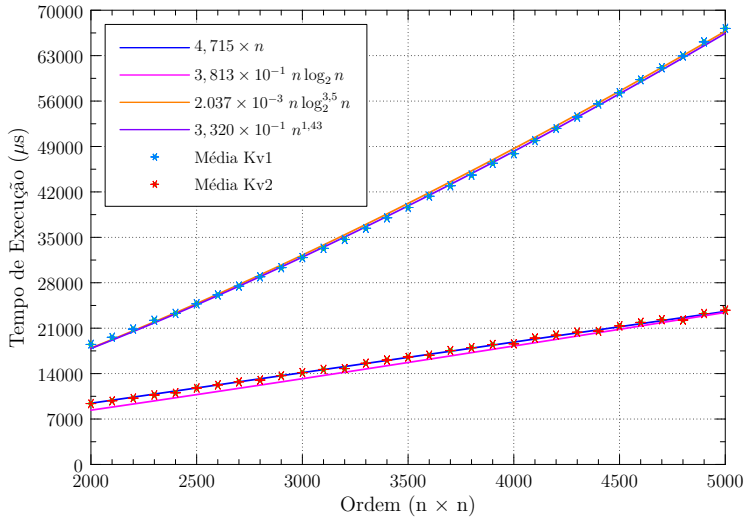

Figura 4. Comparação das equações reportadas pelo EMA para Kv1 e Kv2 da Etapa 2.

4.3. Comparação com a Literatura

A fim de validar a performance dos *kernels* propostos na Seção 3 deste trabalho, a Tabela 3 apresenta os resultados obtidos de um teste comparativo composto por matrizes cheias de ordem $2 \cdot 10^3 \leq n \leq 5 \cdot 10^3$. Os dados são referentes à dez execuções da versão sequencial de Eliminação Gaussiana, da versão CUDA apresentada neste trabalho utilizando Kv2 e do *kernel* proposto em [Gonçalves et al. 2015] onde só é explorado o paralelismo na Etapa 1 do método de Eliminação Gaussiana. Nota-se que, para todos as dimensões das

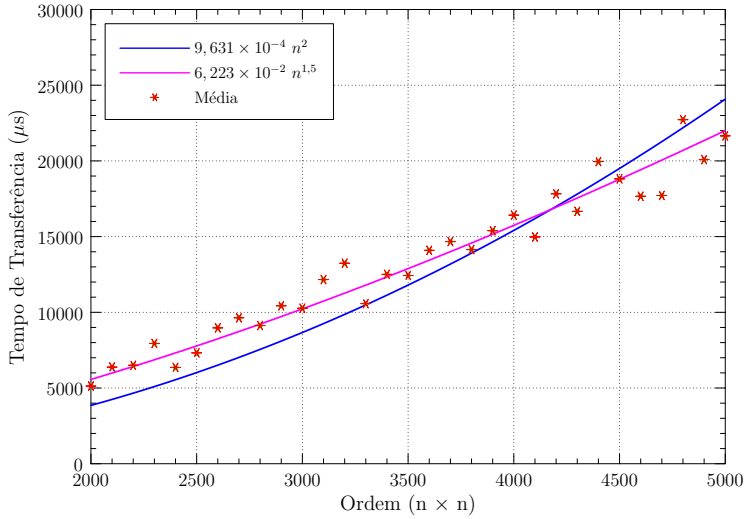

Figura 5. Equações reportadas pelo EMA referente a transferência da matriz entre hospedeiro e dispositivo.

matrizes o tempo médio de execução utilizando Kv2 é menor dos encontrados com o *kernel* proposto em [Gonçalves et al. 2015] bem como o speedup é maior em todos os valores de n.

Tabela 3. Tempo de execução da Eliminação Gaussiana

n	**Método**	**Média**(s)	**Desvio Padrão**	**Máximo**(s)	**Mínimo**(s)	**Speedup**
2000	Sequencial	8,392	0,003	8,396	8,389	1,00
	CUDA	0,396	0,002	0,398	0,394	21,19
	[Gonçalves et al. 2015]	1,602	0,026	1,664	1,573	5,24
3000	Sequencial	29,902	0,003	29,907	29,898	1,00
	CUDA	1,244	0,001	1,247	1,242	24,04
	[Gonçalves et al. 2015]	3,012	0,036	3,106	2,983	9,93
4000	Sequencial	70,786	0,985	70,796	70,762	1,00
	CUDA	2,888	0,002	2,893	2,885	24,51
	[Gonçalves et al. 2015]	5,720	0,045	5,785	5,673	12,38
5000	Sequencial	133,298	2,868	138,123	131,255	1,00
	CUDA	5,578	0,002	5,581	5,574	23,90
	[Gonçalves et al. 2015]	10,147	0,018	10,167	10,119	13,14

5. Conclusões e Trabalhos Futuros

Neste trabalho, utilizando a ferramenta EMA, foi possível determinar a complexidade computacional das duas etapas do algoritmo sequencial do método de Eliminação Gaussiana. Os resultados empíricos obtidos pela ferramenta são compatíveis aos resultados teóricos. Também foi proposto duas versões de paralelização para a Etapa 2 do método de Eliminação Gaussiana que foram comparadas através dos resultados obtidos pela ferramenta EMA. E assim como na implementação sequencial, a ferramenta EMA foi capaz de apresentar resultados empíricos de complexidade para o algoritmo paralelo em GPU, sendo tais resultados inéditos para o algoritmo, devido à dificuldade de determinação

teórica levando em conta os detalhes da arquitetura GPU, tais como: latência de acesso à memória global e compartilhada, escalonamento de blocos e *warps*. Também foi possível levantar dados inéditos sobre a complexidade da Etapa 2 do método em GPU, mostrando-se que mesmo se tratando de um algoritmo de baixo processamento, o fato de não paralelizarmos esta parte implica em uma possível alteração na complexidade final do algoritmo. A comparação mostra que é possível utilizar a ferramenta para obter o comportamento de diferentes métodos em ambientes heterogêneos para estudos comparativos.

Como extensão da ferramenta EMA para algoritmos de GPU, propõe-se a inclusão de informações específicas da arquitetura GPU como: uso da memória global, memória compartilhada e ocupação da placa. Com esses novos recursos, será possível automatizar a coleta de informações e dimensionamento das instâncias de teste no EMA.

Como trabalho futuro, objetiva-se explicar teoricamente os resultados empíricos obtidos para a versão paralela e desenvolve-los em outros ambientes heterogêneos. Também pretende-se realizar estudos com outros algoritmos paralelos para a resolução de sistemas lineares através da ferramenta EMA.

Referências

Buluç, A., Gilbert, J. R., and Budak, C. (2008). Gaussian elimination based algorithms on the gpu. Technical report, University of California.

Che, S., Li, J., Sheaffer, J. W., Skadron, K., and Lach, J. (2008). Accelerating compute-intensive applications with gpus and fpgas. In *Symposium on Application Specific Processors (SASP) 2008*, pages 101–107.

de Souza, R. M. (2015). *Análise Empírica do Algoritmo Shellsort*. TCC – Trabalho de Conclusão de Curso, Universidade do Estado do Rio de Janeiro.

Golub, G. H. and van Loan, C. F. (1989). *Matrix Computations*. Number 3 in Johns Hopkins Series in the Mathematical Sciences. The Johns Hopkins University Press.

Gonçalves, N., Costa, C., Araújo, J., Costa, J., and Panetta, J. (2015). Comparação e análise de desempenho de aceleradores gráficos no processamento de matrizes. In *Anais do XIV Workshop em Desempenho de Sistemas Computacionais e de Comunicação*, pages 1–13.

Kirk, D. B. and Wen-mei, W. H. (2012). *Programming massively parallel processors: a hands-on approach*. Newnes.

Lehmann, E. L. and Casella, G. (1991). *Theory of point estimation*. Wadsworth & Brooks/Cole Advanced Books & Software.

Marrakchi, M. and Robert, Y. (1989). Optimal algorithms for gaussian elimination on an mimd computer. *Parallel Computing*, 12(2):183 – 194.

Oliveira, F. S. (2016). EMA - WebPage. `http://fabianooliveira.ime.uerj.br/ema`. [Última vez acessado: 07 de Junho de 2016].

Robert, Y. and Trystram, D. (1989). Optimal scheduling algorithms for parallel gaussian elimination. *Theoretical Computer Science*, 64(2):159 – 173.

Roger, T. (1969-1970). Algèbre Linéaire, C3 Analyse Numérique. Technical report, University of California.