Análise Comparativa da Fatoração LU: Estudo de Caso OpenCL CPU x GPU

Estevan Braz Brandt Costa[1], Fabio Takeshi Matsunaga[1], Jacques Duílio Brancher[1]

Universidade Estadual de Londrina (UEL) – Departamento de Computação
Caixa Postal 6001 – 86051-990 – Londrina – PR – Brasil l

Resumo Com o advento da GPU (Graphical Processing Unit), e de sua utilização para auxiliar em processos matemáticos através do surgimento da GPGPU (General Purpose for GPU), diversas plataformas surgiram para que os desenvolvedores pudessem usar esta arquitura em seu favor. Apesar do desenvolvimento de algoritmos para GPU ter ficado mais simples e rápido, ainda há muito o que se fazer e pensar ao se desenvolver um algoritmo que faça uso de tal tecnologia. Este trabalho vem mostrar o estudo das principais características que devem ser levado em conta quando desenvolve-se um algoritmo para ser executado na GPU, como a transferência de dados entre CPU e GPU. Um estudo de caso foi feito e analisado através da implementação do algoritmo de fatoração LU, e resultados mostraram um ganho médio de 93% no desempenho com todas as otimizações consideradas. Os principais fatores que contribuiram para a melhora de desempenho foram o gerenciamento da memória e os tipos de processos e dados que são executados e transferidos nos kernels.

Palavras-Chave: Fatoração LU, Computação de Alto Desempenho, Álgebra Linear

Abstract With the GPU (Graphical Processing Unit) advent, and its use in mathematical processes and applications, by emergence of GPGPU (General Purpose for GPU), many platforms have arisen to allow developers to use this architecture in their favor. Despite the development of algorithms based on GPU programming have been simpler and faster, there is still many things to do and think to develop an algorithm which make use of such technology. The aim of this work is to show the study of the main characteristics that must be taken into account when developing an algorithm to run on the GPU device. A case study was analyzed through the LU factorization algorithm and results showed a gain of approximately 93% in performance, considering all optimizations implemented. The main factors that contributed to the performance improvement were the memory management and types of processes and data that are executed and transferred in kernels.

Keywords: LU Factorization, High Performance Computing, Linear Algebra

1 Introdução

A resolução de sistemas de equações lineares $Ax = b$ está presente em grande parte dos problemas científicos como um processo fundamental, tornando-se um fator determinante no desempenho dos mesmos [1]. Existem diversos métodos numéricos que resolvem sistemas lineares como a eliminação de Gauss, fatoração LU, de Cholesky e QR, para trabalhar com diferentes tipos de matrizes. Uma das principais fontes de pesquisa dos últimos anos é justamente o desenvolvimento de códigos computacionais eficientes desses métodos, que utilizem as capacidades de recursos computacionais de alto desempenho, como os aceleradores e as GPUs (*Graphical Processing Unit*).

Graças a popularização das GPUs, que a princípio eram voltadas paa renderização gráfica de modo eficiente, e por alcançar a excelência em conseguir trabalhar com grandes quantidades de dados, surgiu o GPGPU (*General Purpose for* GPU). Com isto, foi possível utilizar as placas gráficas para realizar operações de cálculo numérico em grande quantidade de dados, que vão além dos processamentos gráficos oferecidos pela GPU. Com o advento da GPGPU, o poder computacional de se trabalhar com simulações numéricas e métodos de álgebra matricial tem aumentado exponencialmente, fazendo com que estes consigam atingir novos limites [2].

Porém, tem ocorrido a necessidade de retrabalho nos algoritmos que já foram implementados [3]. Isto porque nos últimos anos está acontecendo uma mudança de paradigma de muitas aplicações da área computacional [4], além de que as arquiteturas de computadores estão se tornando paralelos e heterogêneos, contendo CPU e GPU para processamento de diversas aplicações. Considerando-se isto, o objetivo deste trabalho é mostrar um método de implementação da fatoracão LU utilizando a plataforma OpenCL, e levantar quais são os principais aspectos na programação que devem ser levados em conta ao se programar utilizando a abordagem GPGPU, com o intuito de avaliar o desempenho final de diversos métodos de implementação.

Assim, esse artigo está organizado da seguinte forma: na seção 2 serão apresentados alguns trabalhos relacionados que motivaram a consecução deste. A seção 3 apresenta uma visão geral do algoritmo de fatoração LU tradicional, cujas melhoras e otimizações em ambiente CPU/GPU na plataforma OpenCL são mostrados na seção 4. Os resultados dos testes e execuções dos algoritmos otimizados são apresentados na seção 5 e finalmente em seguida, as conclusões e algumas perspectivas de trabalhos futuros.

2 Trabalhos Relacionados

Diversos estudos sobre a implementação da fatoração LU eficiente em ambiente GPU tem sido desenvolvidos nos últimos anos. Ino [3] por exemplo chegou à conclusão de que essa abordagem não era viável para resolver equações do tipo $Ax = b$. Isto porque os pesquisadores se depararam com o problema de gerenciamento de memória, que diminuia o desempenho do algoritmo para matrizes

acima de 512 x 512. Porém, o autor aponta que este resultado logo será modificado pela rápida evolução que a GPU passa.

Já Galoppo [5] conseguiu equiparar o resultado de execução do algoritmo de fatoracão LU de seu algoritmo e de uma instância da ATLAS devidamente otimizada. Isto porque houve um grande trabalho para se minimizar os custos de transferência de dados entre CPU e GPU, que é considerada uma operação de alto custo computacional. Nesta mesma abordagem, Barrachina [6] foi mais além, pois demonstrou que a utilizacão de técnicas como *padding*, recursão e utilizacão de ambientes híbridos podem ajudar a obter melhores resultados.

Após ter sido verificado que o uso da GPU pode auxiliar no desempenho de execução de uma aplicacão, estudos como realizados por Cupertino [7] buscaram analisar de forma pontual aspectos como o gerenciamento de memória da matriz em uma GPU. Nesta pesquisa, ficou claro que, após implementar a fatoracão LU acessando os elementos de formas diversas, o acesso aos elementos por linha se mostrou mais eficiente que os demais.

Com a constante melhora do *hardware*, e uma maior faicilidade para se usá-los através de plataformas como OpenCL e CUDA, pesquisadores conseguiram alcançar outras áreas da álgebra matricial, criando algoritmos diversos que também usufruem do poder computacional oferecido pela GPU [8,9]. Com o aumento do número de algoritmos sendo estudados e produzidos, surgiram bibliotecas voltadas para aproveitar deste novo ambiente computacional. Como exemplo de bibliotecas deste tipo pode-se citar CULA [10], PLALAPACK [11] e DPLASMA [12].

Apesar de todos os esforços generalizados mencionados para usufruir-se da GPU, ainda há pesquisas na área que demonstram bons resutlados utilizando apenas CPU. Michailidis e Margaritis [13] demonstraram que através da utilização do OpenMP, é possível solucionar problemas pontuais da fatoracão LU de forma eficiente e eficaz. Além disso, pode-se ver um esforço para conseguir melhorar o desempenho do método em sua base matemática com uma variante mais estável do critério de Markowitz, tornando o resultado mais consistente [14].

Atualmente, pesquisas realizadas por Alonso [15] analisaram o impacto da implementação destes algoritmos na questão de consumo de energia pelo computador. O trabalho de Agullo [16] mostra uma solução híbrida, ou seja, que utiliza tanto GPU quanto CPU para melhorar o desempenho da fatoração LU. Com isso, diante das referências bibliográficas citadas anteriormente, o presente trabalho irá elaborar uma nova abordagem de implementação do método em questão que através de uma solução que use tanto o ambiente CPU como GPU e que explore diversos mecanismos supracitados que podem interferir no desempenho do algoritmo.

3 Fatoração LU

O algoritmo da fatoração LU consiste em em resolver o sistema de equações $Ax = b$ através da utilização das propriedades encontradas na eliminação de Gauss.

Seu algoritmo possui 2 etapas bem distintas. A primeira etapa, denominada decomposição LU, consiste em decompor a matriz A em duas matrizes: uma triangular superior, denominada L e outra triangular inferior denominada U. Este passo consome a maior parte do tempo de execução do algoritmo devido ao grande número de atualizacões e cálculos realizados, logo esta etapa é a mais que mais necessita de melhorias e otimizações.

Porém, devido sua natureza de execução ser do tipo iterativa, conseguir buscar os benefícios oferecidos pela paralelização do codigo não é algo trivial. Sua execução consiste em primeiro calcular os fatores de uma única coluna e, logo em seguida, atualizar a matriz utilizando estes fatores. Somente depois da matriz atualizada, que novos fatores poderão ser calculados, criando assim uma interdependência entre os processos. Ao final de cada ciclo da decomposição, uma coluna é gerada em L e uma linha é gerada em U.

A primeira etapa é formada, basicamente, por 2 cálculos numéricos. Para cada elemento de L, onde i representa a i-ésima, e n é a n-ésima linha em que $n > i$, tem-se:

$$l_{n,i} = a_{n,i}/a_{i,i}$$

Já para os elementos de U, há a necessidade de efetuar apenas uma atribuição simples. Considerando que i é a i-ésima iteração, e m representa a m-ésima coluna, onde $m > i$, têm-se:

$$u_{i,m} = a_{i,m}$$

Já para efetuar a atualização da matriz A, passo necessário para iniciar a próxima iteracão, há a necessidade de se utilizar tanto os elementos contidos em L, quanto os contidos em U. Considerando que i é a i-ésima iteração, n é a n-ésima linha e m é a m-ésima coluna, onde $n > i$, e $m > i$, têm-se:

$$a_{n,m} = a_{n,m} - (l_{n,i} * u_{i,m})$$

Após a decomposição LU, é realizado o passo para calcular os elementos do vetor solução x. Primeiramente, é feito a resolução do sistema $Ly = b$, cuja solução y é utilizado para calcular o sistema $Ux = y$, encontrando os valores de x. Como L e U são matrizes triangulares, os cálculos de x e y podem ser realizados eficientementes utilizando a eliminação de Gauss sem a necessidade de permutação de linhas, isto é, através do processo da retrosubstituição.

4 Comparação entre o algoritmo tradicional e otimizado

Antes de analisar o algoritmo em OpenCL, faz-se necessário o estudo do algoritmo tradicional. Isto porque o OpenCL já possui uma estrutura de comandos voltadas para o paralelismo. Logo, encontrar as partes do código que podem ser paralelizadas se torna algo crucial, porém não trivial de ser realizado. De acordo com o Algoritmo 1, pode-se verificar diversos pontos passíveis de paralelização, procurando operações que não dependem de execuções passadas.

Algoritmo 1 Algoritmo de Fatoracão LU Tradicional

```
1:  fator ← 0
2:  for i ← 0 to m do
3:      pivotamentoParcial(A, i)
4:      for col ← i to m do
5:          U[i][col] ← A[i][col]
6:      end for
7:      for k ← i + 1 to m do
8:          fator ← A[k][i]/A[i][i]
9:          for j ← i to m do
10:             A[k][j] ← A[k], [j] − (fator ∗ A[i][j])
11:         end for
12:         A[k][j] ← fator
13:     end for
14:     L[i][i] ← 1
15: end for
16: for i ← 0 to m do
17:     for j ← 0 to i do
18:         L[i][j] ← A[i][j]
19:     end for
20: end for
```

Tais pontos de paralelização ocorrem entre as linhas 7 e 11, onde os fatores são calculados e a linha daquele fator é atualizada. Como uma linha não interfere na próxima, não há necessidade de que a segunda linha comece após a primeira linha, assim como a terceira não depende da segunda, e assim sucessivamente. Apesar de haver outros pontos paralelizáveis, como ocorre entre as linhas 4 e 5, e 16 e 20, estes não foram cogitados para serem paralelizados. Isto porque estas operações exigem transferência de dados entre vetores A contida da GPU, para os vetores L e U, contidas na CPU, e em OpenCL o custo de se carregar os objetos da memória do host para o dispositivo é muito caro, tornando sua paralelização iniviável.

O OpenCL trabalha com funções chamadas *kernels* previamente criadas, que são carregadas em um dispositivo, juntamente com os dados para efetuar as operações em tempo de execução da função da aplicação principal (*host*) na CPU. Foram criadas duas *kernels* para a representação do codigo descrito nas linhas 7 a 11 do Algoritmo 1. Isso porque além de cada linha estar desvinculada uma da outra, as colunas que são atualizadas na estrutura de repetição interna também estão desvinvuladas entre si. Logo foi criado um *kernel* para calcular os fatores, e outra para atualizar a matriz A.

Para que não ocorrer uma grande quantidade de transferência e reenvio de código de *kernel* para o dispositivo desejado (CPU/GPU), foi feito um algoritmo para que todos os fatores fossem calculados primeiro para depois atualizar todas as linhas da matriz A. Assim, o código é carregado uma única vez, atualizando apenas os dados, até ser finalizado e dar lugar ao próximo. Considerando que i represente o número da iteracão que esteja ocorrendo, o primeiro *kernel*, deno-

minado 'kernelCalculaFatores', realiza os mesmos calculos encontrados na linha 8 do Algoritmo 1.

Além do cálculo dos fatores, este kernel atualizaz também a matriz U. Como não é possivel atualizar elementos que possuem índices menores que i, há a necessidade de se criar desvios condicionais dentro do kernel, para evitar atualizações incorretas e desnecessárias. Uma vez que o dispositivo finalizar a executação do *kernel* que lhe foi imposto, este retorna os resultados para o *host* em forma de objetos de memória. Este retorno é tão oneroso quanto o envio. Neste caso, as 3 matrizes (A, L e U) estão sendo carregadas e lidas do dispositivo. Além destas, outras variáveis mais simples são enviadas, representando o tamanho do lado da matriz e a iteração atual.

O segundo kernel, denominado 'kernelAtualizaMatriz' é responsável pela atualização do restante da matriz. Este, que recebe também as matrizes A, L e U, realiza o código representado na linha 10 do Algoritmo 1. Há também a necessidade de um controle sobre quais elementos da matriz A atualizar, pois os elementos que possuem índices menores que i, não devem ser atualizados. As matrizes L e U se fazem presentes por fornecerem os dados necessários para efetuar o cálculo. A ordem de execução, bem como os procedimentos complementares a execucão do *kernel* estão representando no Algoritmo 2.

Pode-se perceber que no Algoritmo 2 ainda existe uma relação de interdependência entre os *kernels*, representados na estrutura de repetição. As chamadas do método executarKernel(), realizadas nas linhas 4 e 8, recebem os objetos de memóia (representando as matrizes e outras variáveis necessárias) necessários para execução do próprio *kernel* de fatoação LU. Quanto à execução paralela, o próprio OpenCL se encarrega de fazê-lo no dispositivo escolhido pelo *host* (CPU/GPU).

Após o algoritmo transposto para OpenCL, e a paralelização dos processos alcançados, foi estudado técnicas para melhorar o desempenho de um algoritmo dentro deste ambiente (modificações ilustradas no Algoritmo 3). O primeiro tópico levantado foi analisar se as instruções contidas nos *kernels* são compatíveis com a classe de computação paralela SIMD (*Single Instruction, Multiple Data*), ou seja, há pouca ou nenhuma diferença em tempo de execução entre os diferentes processos. Estas diferenças se dão através de comandos de controles como o *if* ou *switch*. Logo foi feito um trabalho para eliminá-los dos *kernels*, uma vez que estes são prejudiciais para o desempenho de execução paralela de um algoritmo.

Tanto em 'kernelCalculaFatores' como em 'kernelAtualizaMatriz', foram encontrados instruções de controle, que não permitiam realizar cálculos nas posições que já foram calculadas em iterações anteriores. Para resolver este problema, foram criados variáveis auxiliares que pegam os valores atuais das matrizes originais. Dentro do *kernel* todos os cálculos são feitos, até mesmo aqueles que já foram calculados, para priorizar a familiariedade da execução das operações com a classe SIMD. Essas variáveis auxiliares são retornadas dos *kernels*, porém ao passar o valor calculado de volta para a matriz original, seus valores são selecionados e mantidos para não modificar valores que são frutos de iterações anteriores.

Algoritmo 2 Algoritmo de Fatoracão LU Tradicional em OpenCL

▷ %Considere que a matriz A é uma matriz cheia quadrada de ordem 'm', preenchida com valores aleatórios. %

1: **for** $i = 0$ to m **do**

▷ % Primeira Kernel irá calcular os fatores de A e guardar em L%

2: $memObjects[] = preparar Objetos Memoria(L, U, A, m, i)$

3: $preparar Kernel(kernel Calcula Fatores, memObjects)$

4: $executar Kernel(kernel Calcula Fatores)$

5: $recuperar Valores(kernel Calcula Fatores, L, U, A)$

▷ % Segunda Kernel irá atualizar a matriz A e U%

6: $memObjects[] = preparar Objetos Memoria(L, U, A, m, i)$

7: $preparar Kernel(kernel Atualiza Matriz, memObjects)$

8: $executar Kernel(kernel Atualiza Matriz)$

9: $recuperar Valores(kernel Atualiza Matriz, A)$

10: **end for**

Algoritmo 3 Algoritmo de Fatoracão LU Otimizado em OpenCL

▷ %Considere que a matriz A é uma matriz cheia quadrada de ordem 'm', preenchida com valores aleatórios. %

1: **for** $i = 0$ to m **do**

▷ % Primeira Kernel irá calcular os fatores de A e guardar em A%

2: $memObjects[] = preparar Objetos Memoria(A, m, i)$

3: $preparar Kernel(kernel Calcula Fatores Otimizado, memObjects)$

4: $executar Kernel(kernel Calcula Fatores Otimizado)$

5: $recuperar Valores(kernel Calcula Fatores Otimizado, Aaux)$

6: $atualizar Matriz A(A, Aaux, i)$

▷ % Segunda Kernel irá atualizar a matriz A. Neste caso, é utilizado um vetor auxiliar de tamanho 'm', para auxiliar no momento de calcular os valores dentro da kernel%

7: $memObjects[] = preparar Objetos Memoria(A, vet Aux, m, i)$

8: $preparar Kernel(kernel Atualiza Matriz Otimizado, memObjects)$

9: $executar Kernel(kernel Atualiza Matriz Otimizado)$

10: $recuperar Valores(kernel Atualiza Matriz Otimizado, Aaux)$

11: $atualizar Matriz A(A, Aaux, i)$

12: **end for**

Outro aspecto que deve ser levado em conta é a quantidade de informações e objetos de memória que são passados e retornados dos *kernels*. Esta operacão, por ser muito custosa, influencia diretamente no desempenho final da aplicação e da resolução do sistema linear. Foi visto que para o 'kernelCalculaFatores', não há necessidade de se passar a matriz A inteira (tamanho nxn). Como seus fatores são calculados em uma única coluna, foi criado um vetor do tamanho de uma coluna de A (tamanho n), que contém os valores retirados da mesma. Assim apenas o vetor é passado e retornado.

Para o 'kernelAtualizaMatriz', ao invés de passar 3 matrizes inteiras (L, U e A de tamanhos nxn), foi passado 1 matriz e 2 vetores, representando os valores de A e os valores de L e U respectivamente, que serão utilizados para o cálculo da fatoração. Como L e U são gerenciados fora dos *kernels*, não há necessidade de que se retorne estes vetores, deixando apenas a matriz atualizada para o retorno. É possivel vislumbrar no Algoritmo 3 que as diferenças com relação ao Algoritmo 2 são poucas.

5 Resultados obtidos

Ambos os programas (fatoração LU tradicional e otimizado em OpenCL) foram executados 10 vezes, com tamanhos de variáveis diferentes, porém com valores randômicos entre 0 e 1. As especificações do computador em que foram realizados os testes de desempenho de execução dos algoritmos se encontra na Tabela 1.

Tabela 1. Especificações da máquina em que foram realizados os testes

Tipo	**Nome**	**Propriedades**
CPU	Core i5 760 2.8GHz	80GFLOPS PD
Memória RAM	4GB DDR3 1333MHz	21GB/s
Sistema Operacional	Windows 7 Pro	x86 64 bits
Barramento gráfico	PCIe 2.0	8 GB/s
GPU	Barts Pro 820MHz	1488GFLOPS PD
VRAM	1GB GDDR5 1050MHz	128GB/s

A medição do desempenho ocorre no momento em que as matrizes já estão prontas, e sua decomposição é totalmente realizada. Os resultados do desempenho da execução de ambos os algoritmos estão mostrados na Tabela 2 (CPU) e na Tabela 3 (CPU/GPU).

Na tabela 2 pode-se ver que o ganho de desempenho foi de 91,79%. Além disso, a distância entre um resultado e outro aumentava juntamente com o tamanho da matriz. Isto pode ser constatado, pois nas matrizes de tamanho 16 x 16, a diminuição do tempo de execução foi de 73,86%, enquanto que o ganho para as matrizes de tamanaho 4096 X 4096 foi de 98,01 %.

Os resultados mostrados na tabela 3 mostram o mesmo padrão de ganho de desempenho supracitado nos resultados para apenas CPU. A melhoria de

Tabela 2. Resultados dos desempenho dos algoritmos em CPU, em segundos.

Ordem do sistema	Não Otimizado (seg)	Otimizado (seg)	Ganho relativo
16	0,0013	0,0003	76,92 %
32	0,0045	0,0006	86,67 %
64	0,0224	0,0016	92,86 %
128	0,1432	0,0068	95,25 %
256	1,0372	0,0461	95,56 %
512	8,5192	0,2892	96,61 %
1024	126,2631	2,6914	97,87 %
2048	1070,9700	21,5841	97,98 %
4096	10373,3900	207,4660	98,00 %

desempenho em médio foi de 84,80%, sendo que as matrizes de menor tamanho tiveram o menor ganho de desempenho. As matrizes de tamanho 16 x 16 tiveram um ganho de 60,07% e as 4096x4096 tiveram um ganho de 96,80%.

Tabela 3. Resultados dos desempenhos dos algoritmos em GPU/CPU, em segundos.

Ordem do sistema	Não Otimizado (seg)	Otimizado (seg)	Ganho relativo
16	0,0128	0,0051	60,16%
32	0,0219	0,0065	70,32%
64	0,0441	0,0101	77,10%
128	0,2173	0,0240	88,96%
256	1,1353	0,0619	94,55%
512	5,6870	0,2967	94,78%
1024	38,6920	1,4710	96,20%
2048	298,3024	9,9661	96,66%
4096	2530,406	80,973	96,80 %

Também é possivel observar tanto na Figura 1 quanto na Figura 2 que a escalabilidade do algoritmo otimizado é muito superior ao algoritmo normal em ambas as arquiteturas.

Apesar do código-fonte otimizado (Algoritmo 3) ter sido pouco alterado com relação ao tradicional em OpenCL (Algoritmo 2), os resultados mostraram um ganho excepcional de desempenho, principalmente para matrizes de ordens muito grandes, a partir de 1000 (Figura 1 e Figura 2). Logo, utilizar o OpenCL para criar códigos que resolvam problemas da álgebra matricial pode se tornar ineficiente, caso não se pense em detalhes importantes, como a questão de transferência de dados entre a CPU e a GPU.

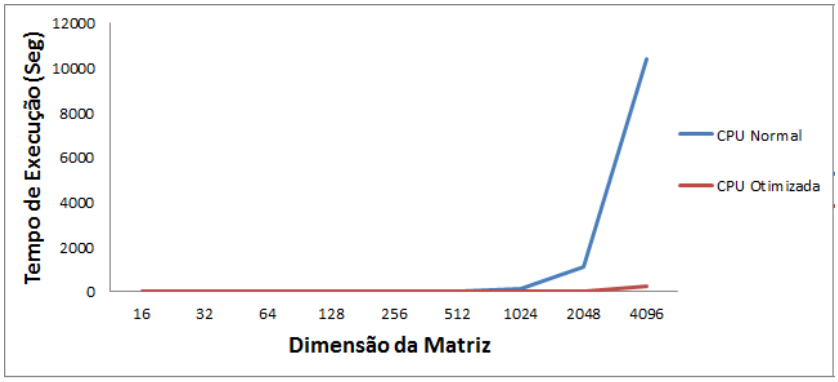

Figura 1. Comparação de resultados entre os algoritmos otimizado e não-otimizado, variando o tamanho da matriz A entre 16 e 4096 em um ambiente CPU.

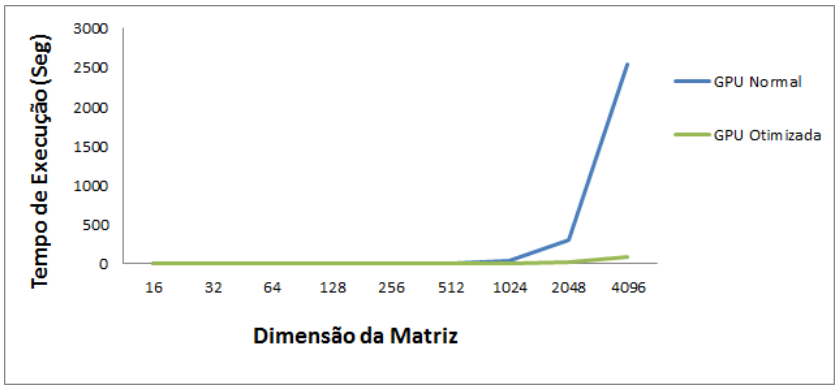

Figura 2. Comparação de resultados entre os algoritmos otimizado e não-otimizado, variando o tamanho da matriz A entre 16 e 4096 em um ambiente GPU.

6 Conclusões e perspectivas

Após avaliação dos resultados obtidos, pode-se verificar diversas características importantes relativas ao desempenho do algoritmo. Primeiro que ao utilizar a GPU para a realização dos cálculos, esta obteve não somente resultados melhores, mas também consegue manter o desempenho em um nível que o algoritmo consiga ser utilizável para matrizes de ordem acima de 1000. Apesar da GPU fornecer um ganho apenas por inserí-la no contexto de execução, pode-se notar um grande ganho de desempenho quando se considera certos pontos no momento do desenvolvimento.

Dentre estes pontos, o gerenciamento da memória e o tipo de processo executado nos *kernels* se mostraram bastante eficazes e fáceis de serem aplicados. Este ganho se dá primeiramente pela forma como os objetos de memória utilizado são passados para os dispositivos, e também pela forma como os dispositivos

trabalham. Estes, por trabalharem com instruções simples, porém com grande quantidades de dados, instruções do tipo SIMD são ideais para serem executadas neles.

Com o ganho de aproximadamente 96% no desempenho do algoritmo, se torna claro que estes pontos supracitados são de extrema relevância para se criar aplicacões de alto desempenho utilizando OpenCL. O próximo passo para elevar o desempenho da fatoracão LU consiste em estudar como otimizar o codigo para usufruir ambientes com múltiplos CPUs e GPUs organizados em *clusters*.

Referências

1. Rodriguez-Alvarez, M.J., Sanchez, F., Soriano, A., Iborra, A.: Sparse Givens resolution of large system of linear equations: Applications to image reconstruction . Mathematical and Computer Modelling **52**(7-8) (2010) 1258–1264
2. Du, P., Luszczek, P., Tomov, S., Dongarra, J.: Soft error resilient QR factorization for hybrid system with GPGPU . Journal of Computational Science (0) (2013) –
3. Ino, F., Matsui, M., Goda, K., Hagihara, K.: Performance Study of LU Decomposition on the Programmable GPU. 12th IEEE Intl Conf. High Performance Computing (HiPC05) (16016254) (2005)
4. Hu, L., Che, X., Xie, Z.: GPGPU cloud: A paradigm for general purpose computing. Tsinghua Science and Technology **18**(1) (2013)
5. Galoppo, N.: LU-GPU : Efficient Algorithms for Solving Dense Linear Systems on Graphics. Architecture (c) (2005)
6. Barrachina, S., Castillo, M., Igual, F.D., Mayo, R., Quintana-Ortí, E.S.: Solving dense linear systems on graphics processors. Euro-Par 08: Proceedings of the 14th international Euro-Par conference on Parallel Processing (2008)
7. Cupertino, L.F., Singulani, A.P., Silva, C.P., Aur, M., Pacheco, C., Janeiro, R.D., Farias, R.: LU Decomposition on GPUs : The Impact of Memory Access. 22nd International Symposium on Computer Architecture and High Performance Computing Workshops (2010)
8. Matsumoto, K., Nakasato, N., Sakai, T., Yahagi, H., Sedukhin, S.G.: Multi-level Optimization of Matrix Multiplication for GPU-equipped Systems. Procedia Computer Science **4** (January 2011) 342–351
9. Nakasato, N.: Implementation of a parallel tree method on a GPU. Journal of Computational Science **3**(3) (2012) 132 – 141
10. Humprey, J.R., Price, D.K., Spagnoli, K.E., Polini, A.L., Kelmelis, E.J.: CULA: hybrid GPU accelerated linear algebra routines. Society of Photo-Optical Instrumentation Engineers (SPIE) Conference Series (April 2010)
11. Fogue, M., Igual, F.D., Quintana-ortí, E.S., Geijn, R.V.D.: Retargeting PLAPACK to clusters with hardware accelerators flame working note 42. (2010)
12. Bosilca, G., Bouteiller, A., Herault, T., Lemarinier, P., Saengpatsa, N., Tomov, S., Dongarra, J.: A unified HPC environment for hybrid manycore/GPU distributed systems. LAPACK Working Note, Tech. Rep. 234 (October 2010)
13. Michailidis, P.D., Margaritis, K.G.: Parallel direct methods for solving the system of linear equations with pipelining on a multicore using OpenMP. Journal of Computational and Applied Mathematics **236**(3) (September 2011) 326–341
14. Dobes, J., Cerny, D., Biolek, D.: Efficient procedure for solving circuit algebraic-differential equations with modified sparse LU factorization improving fill-in suppression. 2011 20th European Conference on Circuit Theory and Design (ECCTD) (2) (August 2011) 689–692

15. Alonso, P., Dolz, M.F., Igual, F.D., Mayo, R., Quintana-Orti, E.S.: Saving Energy in the LU Factorization with Partial Pivoting on Multi-core Processors. 2012 20th Euromicro International Conference on Parallel, Distributed and Network-based Processing (February 2012) 353–358
16. Agullo, E., Augonnet, C., Dongarra, J., Faverge, M., Langou, J., Ltaief, H., Tomov, S.: LU factorization for accelerator-based systems. 2011 9th IEEE/ACS International Conference on Computer Systems and Applications (AICCSA) (December 2011) 217–224