Impacto da Organização dos Dados em Operações com Matrizes Esparsas na GPU

Paula Prata[1,2], Gilberto Melfe[2], Ricardo Pesqueira[2], João Muranho[1,3]

[1] Instituto de Telecomunicações,
[2] Departamento de Informática,
Universidade da Beira Interior, 6201-001 Covilhã, Portugal
[3] IMAR – Instituto do Mar, Departamento de Zoologia, FCTUC,
Universidade de Coimbra, 3004-517 Coimbra
e-mail: pprata@di.ubi.pt, a23049@ubi.pt, ricardopesqueira705@hotmail.com, muranho@mail.telepac.pt,

Resumo. A utilização de placas gráficas (GPU) para operações sobre matrizes, nomeadamente sobre matrizes esparsas, tem sido alvo de intensa investigação. Pretende-se obter o máximo desempenho de uma arquitectura com centenas de cores e um modelo de paralelismo de dados com execução simultânea de milhares de *threads*. Numerosas aplicações científicas e de engenharia manipulam matrizes esparsas, sendo o produto matriz-vector a operação base para vários algoritmos iterativos de resolução de sistemas de equações esparsas. Neste trabalho avaliamos o desempenho da operação matriz-vector para dois dos formatos mais importantes de armazenamento de matrizes esparsas: CSR e ELL. Mostramos como a ordenação das linhas pode aumentar significativamente o desempenho da operação estudada, no caso do formato ELL, e estudamos o comportamento da solução proposta na resolução de sistemas de equações esparsas correspondente a um problema real.

Palavras-Chave: placa gráfica (GPU), CUDA, paralelismo de dados, matrizes esparsas, sistemas de equações lineares, formatos de armazenamento para matrizes esparsas.

1 Introdução

As actuais placas gráficas (Graphics Processing Units – GPU's) permitem obter um desempenho no processamento massivo de dados em vírgula flutuante comparável ao desempenho obtido, até agora, apenas com supercomputadores. O recente aparecimento de interfaces de programação para a GPU como o "Compute Unified Device Architecture" (CUDA) da NVIDIA [1], o Brook+ da AMD/ATI [2] ou ainda o OpenCL [3], inicialmente desenvolvido pela Apple e posteriormente generalizado para outras arquitecturas, tornou possível programar a placa gráfica usando linguagens de alto nível como o C/C++, ou mesmo o Java. A maior facilidade de programação fez com que a GPU começasse a ser usada para sistemas de computação em larga escala. Surgiu assim um aumento significativo da investigação sobre como paralelizar algoritmos já existentes de forma a optimizar a utilização da GPU.

As GPU's são dispositivos com grande capacidade de cálculo mas que não possuem caches que permitam optimizar os acessos à memória. Possuindo uma elevada latência no acesso à memória global, torna-se necessária a execução de milhares de threads de muito baixa granularidade para conseguir tirar partido da GPU. É também necessário distribuir de forma adequada a carga de trabalho pelos conjuntos de threads que constituem as unidades de escalonamento (warps em linguagem CUDA). Cada warp só é dado por concluído quando todas as suas threads (actualmente 32) tiverem terminado.

Nem todos os algoritmos são pois adequados para paralelizar em GPU. A GPU é especialmente útil para aplicações com grande intensidade de cálculo numérico e com paralelismo de dados, onde cálculos similares são executados em grandes quantidades de dados organizados de forma regular (por exemplo, vectores e matrizes).

Vários estudos mostram que, em problemas que manipulam matrizes densas, a GPU permite obter elevados desempenhos [4], [5]. No entanto, enquanto uma matriz densa tem uma estrutura regular, as matrizes esparsas podem ser representadas em formatos bastante irregulares que poderão condicionar o desempenho.

Neste trabalho analisamos a multiplicação matriz-vector em que a matriz é esparsa. Esta operação, pelo número de vezes que é executada, é a operação dominante em vários algoritmos iterativos para resolução de sistemas de equações lineares e em problemas de cálculo de valores próprios. Um estudo recente sobre a implementação do produto matiz esparsa / vector é apresentado em [6] e [7]. Nestes estudos, Bell e Garland mostram que o melhor desempenho para a operação estudada é obtido com o formato de armazenamento ELL (ELLPACK/ITPACK) em matrizes em que o número de elementos não zero por linha varia pouco. Este formato permite que os valores da matriz a processar por um *warp* estejam em posições contínuas de memória, optimizando assim os tempos de acesso. Se as linhas manipuladas por cada *warp* tiverem aproximadamente o mesmo tamanho (isto é, o mesmo número de elementos não zero) então todas as threads estarão ocupadas em simultâneo sem desperdício de capacidade de cálculo. Como, em problemas reais, o número de elementos não zero por linha é variável, propomos a ordenação das linhas de forma a uniformizar o trabalho a realizar por cada *warp* mesmo em matrizes onde os comprimentos das linhas variam substancialmente. Analisamos o impacto da ordenação das linhas para o formato ELL, concluindo que permite obter melhorias de desempenho que podem atingir os 30%. Aplicámos o mesmo mecanismo a um dos formatos de representação de matrizes esparsas mais comum, o CSR (Compressed Sparse Row). Também aqui é obtido algum ganho mas menos significativo, no melhor caso obteve-se um ganho de 13%.

Finalmente seguimos a mesma abordagem para um problema que envolve a resolução de sistemas de equações esparsas através do método do gradiente conjugado. Trata-se de um método iterativo, que executa a operação matriz-vector em cada uma suas iterações. Os sistemas de equações usados foram gerados por um simulador de redes de distribuição de água, e correspondem a problemas que representam situações reais ou casos de estudo. Concluímos que para estes sistemas de equações, onde as matrizes de coeficientes apresentam um número muito reduzido de valores não zero (abaixo dos 0.5%) e uma certa uniformidade no número de elementos não nulos por linha, o impacto da ordenação das linhas não é substantivo.

Neste artigo apresentamos, na secção 2, o modelo de programação CUDA da NVIDIA e o seu mapeamento na arquitectura da GPU. Na secção 3, descrevemos os formatos de armazenamento de matrizes esparsas usados e as operações implementados. Na secção 4 são apresentados os resultados obtidos e finalmente na secção 5 apresentam-se as conclusões e algumas propostas de trabalho futuro.

2 O Modelo de Programação CUDA e a Arquitectura da GPU

O CUDA veio proporcionar aos utilizadores de linguagens de alto nível, como o C/C++, a possibilidade de tirarem partido do poder de processamento paralelo da GPU [1]. Nesta secção vamos descrever brevemente o modelo de programação CUDA e a arquitectura da GPU usada para este trabalho, a GeForce GTX 295.

2.1 O Modelo de Programação CUDA

No modelo de programação paralela CUDA, uma aplicação consiste na execução de um programa sequencial que corre no CPU (*host*) capaz de lançar na GPU (*device*) funções que serão executadas em paralelo. Cada função que vai ser executada na GPU, designada por *kernel* é executado sequencialmente, mas em simultâneo, por um elevado número de threads. Cada thread tem um ID único podendo ser-lhe atribuída uma tarefa em particular. Há que ter em conta que, em operações que necessitem de pontos de sincronização entre threads, por exemplo quando várias threads manipulam dados em comum, irá haver uma degradação do desempenho. As threads são organizadas em blocos e o conjunto dos blocos de threads irá constituir a grelha de execução (Grid). Cada bloco da grelha de execução irá ser mapeado num multiprocessador da GPU, cuja estrutura descreveremos adiante. Ao ser invocado um *Kerne*l, são passados dois parâmetros especiais que definem o número de threads por bloco e o número de blocos por grelha. Assim sendo, o número de threads que vão executar um *kernel* é o resultado da multiplicação do número de threads de um bloco pelo número de blocos que constitui a grelha de execução.

2.2 A Arquitectura da GPU da NVIDIA

A GPU da NVIDIA é construída como um array de multi-processadores em que cada multi-processador possui 8 núcleos/processadores, um conjunto de registos e uma área de memória partilhada [1]. As operações com valores inteiros e valores em vírgula flutuante de precisão simples são executadas pelos núcleos, enquanto as operações em vírgula flutuante de precisão dupla são executadas por uma unidade partilhada pelos 8 núcleos de um mesmo multiprocessador (apenas para placas com capacidade de computação de 1.3 ou superior).

Quando um *kernel* é lançado, os blocos contidos na grelha de execução são distribuídos e numerados de forma automática para os multiprocessadores com capacidade de os executar. Uma vez associado um bloco de threads a um multiprocessador, o mesmo fica responsável por distribuir as diferentes threads do

bloco pelos diferentes processadores que possui. As threads de um mesmo bloco podem comunicar entre si através da memória partilhada do multi-processador. Uma vez terminado um bloco, e se ainda se encontrarem blocos por executar, os mesmos são automaticamente lançados para um dos multiprocessadores que se encontre livre. Desta forma, pode dizer-se que uma thread está associada a um processador, um bloco a um multi-processador e para cada *kernel* é definida uma grelha (ver figura 1).

Devido ao facto de a GPU apenas tratar dados armazenados na sua memória, os mesmos têm de ser copiados para a memória global da GPU antes da execução do *kernel*.

A GPU usada neste trabalho, a GeForce GTX 295, possui 30 multiprocessadores, cada um com 8 cores, isto é um total de 240 cores (a 1.24GHz), suporta até 512x512x24 blocos com um máximo de 512 threads por bloco. Esta GPU tem 2GB de memória global e capacidade de computação 1.3. Foi programada usando a versão 2.3 do CUDA. A máquina hospedeira é um Intel Core 2 Quad Q9550 a 2.83 GHz com 4 GB de RAM, com sistema operativo Microsoft Windows XP Professional 64-bit.

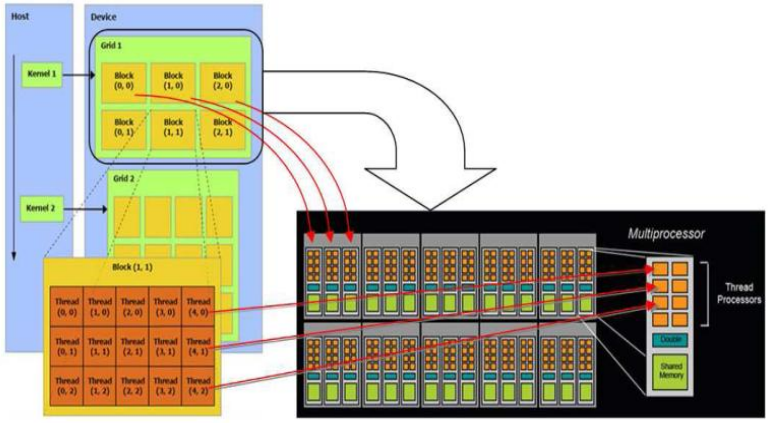

Fig. 1. Associação entre a estrutura do software e a arquitectura do hardware

3 Matrizes Esparsas

Uma matriz esparsa é por definição uma matriz em que a maioria dos seus elementos é igual a zero. Estas matrizes são geralmente armazenadas de forma compacta de modo a guardar apenas os valores diferentes de zero e sua localização. Obtém-se assim um ganho significativo em termos da memória necessária para o seu armazenamento. Vamos descrever os formatos COO, CSR e ELL e as operações estudadas. Uma representação visual destes formatos pode ser analisada em [6].

3.1 Formatos de Representação

O formato base de representação de matrizes esparsas, muitas vezes usado para posterior conversão para outros formatos, é o formato de coordenadas (COOrdinate

Format) designado por COO. O formato COO consiste em armazenar a matriz em três vectores todos de tamanho igual ao número de elementos não zero da matriz: o vector das linhas onde ficam armazenados os índices das linhas em que cada elemento se encontra, o vector das colunas onde ficam armazenados os índices das colunas em que cada elemento se encontra e o vector de dados onde são armazenados os valores dos elementos não zero. Neste trabalho, este formato foi usado para armazenar as matrizes geradas, sendo também o formato usado no repositório de matrizes Matrix Market [8] utilizado como fonte de matrizes de teste.

Formato de Compressão Segundo Linhas. O Formato de compressão segundo linhas ou CSR pode ser considerado como uma extensão do formato de coordenados. A diferença do formato de compressão CSR para o formato COO consiste na substituição do vector que contém os índices das linhas por um outro, geralmente mais curto, de apontadores para a posição no vector das colunas do índice do primeiro elemento não zero da linha correspondente à posição do primeiro vector (isto é, do vector de apontadores). O tamanho do novo vector é o número de linhas mais um, sendo o último elemento do vector o número de não zeros existentes na matriz. Através da subtracção do elemento da posição i+1 pelo elemento da posição i, obtém-se o número de elementos que contém a linha i. O vector de dados vai conter apenas os elementos não zero, ordenados por linhas.

Formato ELLPACK/ITPACK. O formato ELLPACK/ITPACK ou ELL utiliza dois vectores, um para os valores não zero e um outro para os índices das colunas. Supondo uma matriz com M linhas e em que K é o número máximo de elementos não zero por linha, os valores não zero são armazenados por colunas considerando que cada linha tem comprimento K. Para linhas com o número de não zeros menor que K, as posições finais serão preenchidas com um valor pré-definido, por exemplo com o valor zero. Assim, os primeiros M elementos do vector de dados serão o primeiro elemento não zero da primeira linha, o primeiro elemento não zero da segunda linha e assim sucessivamente até ao primeiro elemento não zero da linha M. O segundo vector, o vector dos índices das colunas, corresponde a uma matriz de dimensão M por K, armazenada por colunas, em que cada posição representa a coluna do valor na posição correspondente no vector de dados. Os algoritmos de manipulação deste formato, utilizados neste trabalho, foram optimizados através da utilização de um terceiro vector, consistindo do número de não zeros da linha correspondente.

3.2 Operações estudadas

A operação matriz-vector, como operação base do produto de matrizes, surge nos algoritmos de factorização de matrizes (factorização LU, Cholesky, QR, etc) e portanto, nos algoritmos de resolução de sistemas, algoritmos de cálculo de valores e vectores próprios, algoritmos de decomposição em valores singulares, entre outros. É pois uma operação que apesar de muito simples, sendo executada milhões de vezes, pode ser crítica no desempenho de numerosas aplicações.

Produto Matriz Esparsa/Vector em GPU. No trabalho apresentado em [6] são estudadas três implementações desta operação em GPU: atribuir uma thread a cada elemento não zero da matriz, atribuir uma thread a cada linha da matriz e finalmente atribuir um *warp* a cada linha da matriz. Os resultados mostram que os melhores desempenhos são obtidos para o formato ELL quando cada linha da matriz é processada por uma thread e para o formato CSR quando cada linha é processada por um *warp*. Neste trabalho usámos os formatos ELL e CSR como formatos base para o estudo do efeito da ordenação das linhas da matriz no desempenho da operação.

No formato ELL, como a matriz está armazenada por colunas, a atribuição de uma thread por linha, com a thread i a processar a linha i, acedendo na iteração j ao j-ésimo elemento da linha i, permite que as threads de um mesmo *warp* acedam em cada iteração a posições de memória contíguas, na iteração j, cada thread i acede ao j-ésimo elemento da sua linha. Na iteração 1, a thread 1 vai processar o primeiro elemento da linha 1, a thread 2 o primeiro elemento da linha 2, e assim por diante. Sendo a matriz armazenada por colunas, todos estes elementos estão em posições consecutivas de memória, e uma vez que as threads do mesmo *warp* executam em cada instante a mesma instrução, temos as threads a aceder a posições consecutivas da memória. Este padrão de acesso à memória global, conhecido em linguagem CUDA por "coalesced memory access", é o que permite o melhor desempenho [9].

Se as matrizes tiverem linhas com um número muito variável de elementos não zero, irá acontecer que num mesmo *warp* haverá threads com muito trabalho, isto é com linhas com muitos elementos e threads com pouco trabalho, isto é, com linhas com poucos elementos. Como em cada processador o modelo de execução é SIMD (Simple Instruction Multiple Data) o *warp* só terminará quando todas as suas threads terminarem. É assim de esperar que a ordenação das linhas pelo seu comprimento consiga uniformizar o trabalho de cada *warp* e melhorar o desempenho. Note-se que a utilização de um vector com os tamanhos de cada linha permite parar o processamento no momento adequado (quando a thread já não tem mais elementos para processar) sem alterar o padrão de acesso à memória (mesmo que uma thread ou mais threads não acedam à memória não é violado o esquema dos acessos ordenados).

No caso do formato CSR, o melhor desempenho é obtido colocando as 32 threads de um *warp* a processar a mesma linha. Para os multiprocessadores o modelo de execução é SPMD (Simple Program Multiple Data) e portanto quando um *warp* termina é lançado novo *warp*. Assim, se as threads do mesmo *warp* estão a processar elementos da mesma linha da matriz, a troca de linhas não poderá trazer qualquer ganho, donde neste formato optámos por estudar o impacto da ordenação das linhas só para a variante de uma thread por linha.

Resolução de Sistemas de Equações em GPU. Um problema em que a operação matriz-vector é utilizada frequentemente é a resolução de sistemas de equações, nomeadamente em algoritmos iterativos para sistemas de equações esparsas [10].

Neste trabalho usamos os *kernels* anteriores para matrizes esparsas, em formato CSR e ELL, para paralelizar o produto matriz-vector em cada iteração do método do gradiente conjugado na resolução do sistema de equações esparsas associadas a um programa de simulação de sistemas de abastecimento de água sob pressão, o EPANET [11]. Para cada um dos formatos, estudámos o impacto da ordenação das linhas no desempenho do algoritmo.

4 Resultados

Para avaliarmos o desempenho da operação matriz-vector, gerámos matrizes esparsas quadradas de ordem 4096, 8192 e 16384 contendo valores aleatórios. Para cada linha foi gerado um valor aleatório entre 1 e 20% do número de colunas da matriz. Este será o número de valores não zero da linha, a gerar. A posição de cada valor não zero foi também gerada aleatoriamente. Foram geradas matrizes com valores em precisão simples (*float*) e precisão dupla (*double*). No final, as matrizes geradas tinham 10,01% de valores não zero.

Na figura 2 mostra-se o gráfico do número de valores não zero por linha para a matriz de ordem 4096. Como se pode observar existe variabilidade no comprimento das linhas. Entenda-se por comprimento da linha o número de elementos não zero. Pela forma como foram geradas, as outras matrizes têm gráficos de distribuição semelhantes.

A execução da operação matriz-vector em CPU para os formatos CSR e ELL mostrou que o formato ELL é bastante mais lento que o formato CSR, o que se deve ao armazenamento da matriz por colunas, para um algoritmo que em CPU percorre a matriz por linhas. Os tempos médios, obtidos com 4 execuções, em milissegundos são mostrados nas tabelas 1 e 2 respectivamente para formato CSR e formato ELL, na linha correspondente à ordem da matriz e à frente do tipo de valores. Por razões de falta de espaço não pôde ser incluída uma tabela independente. Estes valores servirão para comparação com os valores obtidos em GPU. Em todos os resultados os tempos de GPU representam o tempo de execução do *kernel* não incluindo a cópia dos dados para a memória da GPU nem a cópia dos resultados para o *host*. Na obtenção dos resultados com ordenação das linhas o tempo gasto na ordenação (feita em CPU) não é considerado. Repare-se que geralmente os problemas reais envolvem um grande número de iterações, sendo a ordenação das linhas realizada uma única vez.

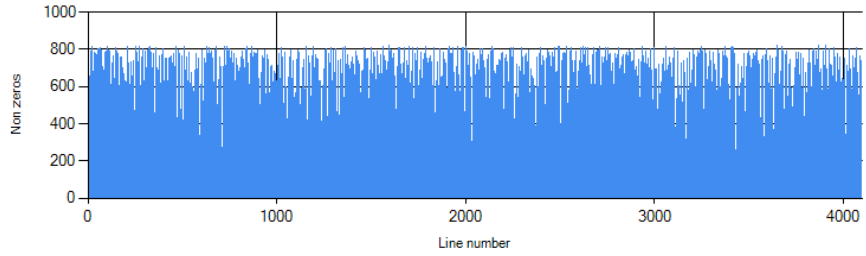

Fig. 2. Gráfico com número de não zeros (non zeros) por linha (line number).

4.1 Produto Matrix-Vector em GPU sem Ordenação de Linhas

Executámos para as mesmas matrizes o produto matriz-vector em GPU formatos CSR e ELL fazendo variar o tamanho do bloco de threads. Considerámos blocos com 32, 64 e 128 threads. O tamanho da grelha, isto é, o número de blocos é obtido dividindo a ordem da matriz pelo tamanho do bloco.

A tabela 1 mostra, além dos tempos em CPU que já referimos, os tempos de execução obtidos com o formato CSR em GPU por matriz e tamanho do bloco. Mostra ainda a

percentagem do tempo de execução em relação ao tempo de CPU, isto é, se houve ou não ganho com a execução em GPU. A tabela 2 mostra os resultados correspondentes para ELL.

Analisando os resultados, podemos ver que o formato CSR em GPU, com uma thread por linha não tem qualquer ganho em relação ao CPU. Para as matrizes mais pequenas parece haver algum ganho mas temos de ter em conta que não contabilizamos o tempo de cópia dos dados. Em precisão dupla, nunca há qualquer ganho o que se explica pelo facto de a unidade de dupla precisão ser partilhada pelos 8 cores de um mesmo multiprocessador. Para o formato ELL em GPU os resultados são completamente diferentes. O tempo gasto em GPU é sempre menos de 5% do tempo gasto em CPU. Observamos que quanto maior a matriz, maior o ganho em relação ao CPU e que quanto maior o tamanho do bloco, isto é quanto mais threads por bloco, melhor desempenho. Concluímos que o formato ELL permite uma utilização efectiva da capacidade de cálculo da GPU.

Tabela 1. Produto matriz-vector em GPU, formato CSR, sem ordenação de linhas

Bloco	**Grelha**	**Tempo (ms)**	**GPU/CPU*100**	**Tempo (ms)**	**GPU/CPU*100**
Ordem: 4096		Float *(CPU: 5.613)*		Double *(CPU: 3.688)*	
32x1	128x1	4.237	75.49	4.734	128.48
64x1	64x1	4.270	76.07	4.794	129.98
128x1	32x1	4.288	76.39	4,855	131.62
Ordem: 8192		Float *(CPU: 22.489)*		Double *(CPU: 15.849)*	
32x1	256x1	20.692	92.01	22.780	143.73
64x1	128x1	20.286	90.20	22.324	140.85
128x1	64x1	20.221	89.91	22.452	142.66
Ordem: 16384		Float *(CPU: 90.212)*		Double *(CPU: 63.105)*	
32x1	512x1	94.238	104.46	101.237	160.42
64x1	256x1	94.019	104.22	101.228	160.41
128x1	128x1	93.371	103.50	100.663	159.52

Tabela 2. Produto matriz-vector em GPU, formato ELL, sem ordenação de linhas

Bloco	**Grelha**	**Tempo (ms)**	**GPU/CPU*100**	**Tempo (ms)**	**GPU/CPU*100**
Ordem: 4096		Float *(CPU: 29.985)*		Double *(CPU: 30.347)*	
32x1	128x1	1.404	4.68	1.440	4.74
64x1	64x1	1.408	4.69	1.463	4.82
128x1	32x1	1.402	4.67	1.486	4.89
Ordem: 8192		Float *(CPU: 197.542)*		Double *(CPU: 147.589)*	
32x1	256x1	5.874	2.97	6.342	4.29
64x1	128x1	4.643	2.35	5.033	3.41
128x1	64x1	4.715	2.38	5.155	3.49
Ordem: 16384		Float *(CPU: 958.678)*		Double *(CPU: 929.724)*	
32x1	512x1	19.656	2.05	21.375	2.31
64x1	256x1	19.635	2.05	21.277	2.29
128x1	128x1	17.555	1.83	19.381	2.09

4.2 Produto Matrix-Vector em GPU com Ordenação de Linhas

Para avaliar o impacto da ordenação das linhas fez-se um pré-processamento dos dados de modo a que as linhas ficassem organizadas em memória por ordem crescente do seu tamanho. Para garantir a consistência dos resultados criou-se um vector adicional contendo a posição inicial das linhas.

A tabela 3 mostra os tempos de execução em GPU obtidos para o formato CSR por matriz e tamanho do bloco de threads, com ordenação por tamanho das linhas. Mostra tal como nas tabelas anteriores o ganho em relação à execução em CPU (*GPUor/CPU *100*) e apresenta também a percentagem de tempo de execução em relação ao tempo de GPU sem ordenação, (*GPUor/GPU*100*). *GPUor* representa o tempo em GPU da versão com ordenação. A tabela 4 mostra os resultados correspondentes para ELL.

Tabela 3. Produto matriz-vector em GPU, formato CSR, com ordenação de linhas

Bloco	**Grelha**	**Tempo (ms)**	**GPUor/CPU *100**	**GPUor/GPU *100**	**Tempo (ms)**	**GPUor/CPU *100**	**GPUor/GPU *100**
Ordem: 4096		Float			Double		
32x1	128x1	3.728	66.41	87.97	4.429	120.08	93.47
64x1	64x1	3.731	66.47	87.38	4.440	120.38	92.61
128x1	32x1	3.723	66.32	86.82	4.433	120.20	91.32
Ordem: 8192		Float			Double		
32x1	128x1	18.810	83.64	90.90	21.023	132.65	92.29
64x1	64x1	18.799	83.59	92.67	21.001	132.51	94.07
128x1	32x1	18.852	83.83	93.23	21.028	132.68	93.66
Ordem: 16384		Float			Double		
32x1	128x1	88.587	98.20	94.00	95.437	151.24	94.27
64x1	64x1	88.151	97.72	93.76	94.704	150.07	93.55
128x1	32x1	88.149	97.71	94.41	94.489	149.75	93.88

Tabela 4. Produto matriz-vector em GPU, formato ELL, com ordenação de linhas

Bloco	**Grelha**	**Tempo (ms)**	**GPU/CPU *100**	**GPUor/GPU *100**	**Tempo (ms)**	**GPU/CPU *100**	**GPUor/GPU *100**
Ordem: 4096		Float			Double		
32x1	128x1	1.231	4.11	87.68	1.315	4.33	91.32
64x1	64x1	1.227	4.09	87.16	1.335	4.39	91.23
128x1	32x1	1.228	4.09	87.57	1.363	4.49	91.74
Ordem: 8192		Float			Double		
32x1	128x1	3.948	2.00	67.21	4.365	2.95	68.84
64x1	64x1	3.890	1.97	83.79	4.332	2.94	86.07
128x1	32x1	3.944	2.00	83.65	4.440	3.01	86.13
Ordem: 16384		Float			Double		
32x1	128x1	14.110	1.47	71.78	15.556	1.68	72.78
64x1	64x1	14.931	1.56	76.04	15.837	1.71	74.43
128x1	32x1	14.975	1.56	85.30	16.08	1.74	83.01

Analisando os resultados podemos ver que para o formato CSR há um pequeno ganho quer em relação à execução em CPU quer em relação à execução em GPU sem ordenação. Esse ganho é na generalidade inferior a 10%. Apenas para a matriz de menor ordem existe um ganho de 13% em relação à execução em GPU sem ordenação com um bloco de 128 threads.

Para o formato ELL os resultados são bem melhores. Em precisão simples o ganho em GPU com a ordenação varia entre os 12.32% (para a matriz de ordem 4092, com 32 threads por bloco) e os 33.8% (para a matriz de ordem 8192 com 32 threads por bloco) Em precisão dupla atingem-se também ganhos com a ordenação, à volta dos 30% para as matrizes de maior ordem. Na generalidade dos casos quanto maior a dimensão maior o ganho em GPU com a ordenação por tamanhos das linhas.

4.3 Resolução de Sistemas de Equações

Finalmente estudamos o impacto da ordenação das linhas quando a operação matriz-vector é utilizada na resolução de sistemas de equações no âmbito de uma ferramenta de modelação de sistemas de abastecimento de água sob pressão. As matrizes produzidas apresentam uma percentagem de elementos não zero muito baixa. Para cada problema simulado, produziram-se 3 matrizes correspondendo a diferentes passos da simulação. Os resultados apresentados nas tabelas que se seguem correspondem, para cada tipo de matriz, à média obtida com a execução desse conjunto de matrizes. A tabela 5 mostra, para as matrizes estudadas, o tempo médio de execução do produto matriz-vector em CPU para os formatos CSR e ELL. Apenas é estudado o caso da precisão dupla (double) porque o algoritmo não converge em precisão simples. Como se pode observar, o tempo de execução em ELL é superior ao do formato CSR, tal como anteriormente.

Tabela 5. Produto matriz-vector em CPU para as matrizes dos sistemas de equações.

Matriz	Ordem	Nº de não zeros	% de não zeros	Tempo médio de execução (ms)	
				CSR - double	ELL - double
Richmond	865	3589	0.48	0.018	0.028
Wolf	1782	8244	0.26	0.038	0.054
Exnet	1891	10025	0.28	0.043	0.070
BWSN	12523	62463	0.04	0.360	0.670
FinMar	14991	71001	0.03	0.343	0,680

Na figura 3 podemos observar o gráfico do número de valores não zero por linha para a matriz de maior ordem (*FinMar*). Como se pode observar a estrutura desta matriz é completamente diferente das estudadas anteriormente. Existem algumas linhas com muitos não zeros e depois muitas linhas com 3, 4 ou menos elementos. As outras matrizes da tabela 5 têm estruturas semelhantes.

A tabela 7 apresenta os resultados da resolução do sistema Ax=b em CPU para os formatos CSR e ELL. Para cada tipo de matriz mostra-se o número médio de iterações realizado, o tempo médio de execução e o tempo médio por iteração.

A tabela 8 apresenta os resultados para a resolução em GPU, formato CSR com e sem ordenação. Em GPU foi usado um bloco de 32 threads para ambos os formatos.

Observando os resultados podemos concluir que o formato CSR em GPU tem algum ganho em relação à execução em CPU para as matrizes maiores (BWSN e FinMar) mas que para estas matrizes o impacto da ordenação é praticamente nulo. A ordenação apenas tem um pequeno impacto para as matrizes de menor dimensão mas para estas matrizes a resolução em CPU é mais rápida.

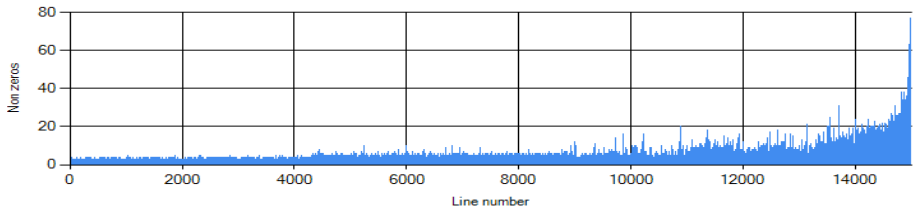

Fig. 3. Gráfico com número de não zeros (*non zeros*) por linha (*line number*) da matriz FinMar

Tabela 7. Resolução em CPU do sistema Ax=b, formatos CSR e ELL.

Matriz A	Nº médio de iterações	Tempo médio (ms) CSR		Tempo médio (ms) ELL	
		Resolução	Por iteração	Resolução	Por iteração
Richmond	901316	16227.926	0.019	20217.685	0.023
Wolf	50311	1953.901	0.059	2357.974	0.071
Exnet	108958	5290.277	0.049	6608.656	0.062
BWSN	844635	290462.537	0.437	357666.410	0.538
FinMar	2362729	903249.271	0.378	1092347.806	0.457

Tabela 8. Resolução em GPU do sistema Ax=b, formato CSR, com e sem ordenação.

Matriz A	Nº médio de iterações	Tempo médio de execução no formato CSR (ms)			
		Sem ordenação		Com ordenação	
		Resolução	Por iteração	Resolução	Por iteração
Richmond	895098	138485.976	0.155	136084.721	0.152
Wolf	50004	8197.074	0.164	8011.004	0.160
Exnet	109346	17922.707	0.164	17134.051	0.157
BWSN	839809	212168.191	0.253	212145.219	0.253
FinMar	2359948	598281.605	0.254	607690.063	0.258

Tabela 9. Resolução em GPU do sistema Ax=b, formato ELL, com e sem ordenação.

Matriz A	Nº médio de iterações	Tempo médio de execução no formato ELL (ms)			
		Sem ordenação		Com ordenação	
		Resolução	Por iteração	Resolução	Por iteração
Richmond	895098	139566.167	0.156	138967.563	0.155
Wolf	50004	7958.645	0.159	7972.861	0.159
Exnet	109346	17401.206	0.159	17217.717	0.157
BWSN	839809	187300.657	0.223	190242.438	0.227
FinMar	2359948	533866.135	0.226	536296.084	0.227

Finalmente a tabela 9 mostra os resultados para a execução do sistema em GPU com as matrizes em formato ELL com e sem ordenação. Podemos concluir que para as matrizes de maior dimensão o formato ELL tem melhor desempenho que o formato CSR mas o impacto da ordenação é nulo. O facto de a ordenação não ter impacto resulta da estrutura da matriz. Como vimos pelo gráfico da figura 3 a variabilidade do comprimento das linhas da matriz (dentro de cada *warp*) é muito pequena e portanto a ordenação das linhas implica alterações sem significado na estrutura das matrizes.

5 Conclusões e Trabalho Futuro

Neste trabalho estudamos o desempenho da operação matriz-vector em GPU para dois formatos de representação de matrizes CSR e ELL. Considerando a resolução em GPU em que uma thread é atribuída a cada linha da matriz, estudámos o impacto da ordenação das linhas pelo seu comprimento, isto é, pelo número de valores não zero. Concluímos que com o algoritmo estudado para GPU o formato CSR tem pouca ou nenhuma vantagem em relação ao CPU, enquanto o formato ELL apresenta ganhos entre os 95 a 98%. O impacto da ordenação para matrizes com cerca de 10% de não zeros e variabilidade no tamanho das linhas é menos de 10% para o formato CSR mas para o formato ELL pode significar ganhos na ordem dos 30%. Estudámos ainda a mesma operação, para os mesmos formatos com e sem ordenação, quando utilizada num algoritmo iterativo de resolução de um sistema de equações. Concluímos que para matrizes com um número de não zeros muito reduzido e pouca variabilidade no comprimento das linhas o impacto da ordenação é nulo.

Como trabalho futuro pretendemos explorar formas de distribuição dos dados com utilização da memória partilhada, mais pequena que a memória global mas de acesso muito mais rápido e estudar o produto matriz-vector para problemas reais que envolvam matrizes com diferentes estruturas.

References

1. NVIDIA Corporation, "NVIDIA CUDA Programming guide", version 2.3.2 (2009).
2. ATI, "Stream Computing – Technical Overview", http://developer.amd.com/gpu/atistreamsdk/pages/default.aspx, acedido em Junho de (2010)
3. Khronos group, OpenCl "Parallel Computing for Heterogeneous Devices", 54 páginas, http://www.khronos.org/opencl/, acedido em Junho de (2010).
4. Volkov, V. and Demmel, J. W., "Benchmarking, GPUs to tune dense linear algebra" in *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing* (Austin, Texas, November 15 - 21, 2008). Conference on High Performance Networking and Computing. IEEE Press, Piscataway, NJ, 1-11.
5. Barrachina, S., Castillo, M. Igual, F.D., Mayo, R. and Qintana-Ortí, "Solving dense linear systems on graphics processors" in Proc. 14th Int'l Euro-Par Conference, volume 5168 of Lecture Notes in Computer Science, pages 739-748, Springer, Aug.2008.
6. Bell, Nathan and Garland, Michael. "Implementing Sparse Matrix-Vector Multiplication on Throughput-Oriented Processors" in Proc. Supercomputing '09, November 2009.
7. Bell, Nathan and Garland, Michael, "Efficient Sparse Matrix-Vector Multiplication on CUDA". Technical Report NVR-2008-004, Dec. 2008 NVIDIA Corporation.
8. The Matrix Market, http://math.nist.gov/MatrixMarket/, acedido em Maio de 2010.
9. Kirk, David B. and Hwu, Wen-mei W., "Programming Massively Parallel Processors", 258 páginas. Morgan Kaufmann, Elsevier, 2010.
10. Saad, Y. Iterative Methods for Sparse Linear Systems. 2nd Edition. Society for Industrial and Applied Mathematics. 2003.
11. Rossman, L.A., "EPANET Users Manual" Risk Management Research Lab., U.S. Environmental Protection Agency, Cincinnati, Ohio, 2000.