Ensino de Organizações de Memória em Arquiteturas Paralelas usando Placas Gráficas Aceleradoras

Ricardo Ferreira, Geraldo Fontes
Departamento de Informática
Universidade Federal de Viçosa
Viçosa, 365700, Brazil
Email: ricardo@ufv.br

***Abstract*—This work proposes simple assignments by using Graphics Processing Units (GPU) to teach parallel architectures. NVIDIA GPU has become very popular in less than one decade, since the CUDA framework has appeared in 2007. A GPU is an interesting didactic resource as it is a parallel and programmable architecture, and it has several memory organizations to be explored as: main memory, shared memory in distributed banks, switch on/off and resize cache L1, specialized memories (textures and constant).**

***Keywords*-Memória; GPU; Placas Gráficas; Ensino de Arquitetura de Computadores**

I. Introdução

O tempo de acesso aos dados na memória e o consumo de energia são dois grandes desafios da última década no projeto de arquitetura de computadores. Este artigo aborda apenas o ensino dos sistemas de memória. A maioria das arquiteturas atuais são paralelas e heterogêneas, seja um sistema pequeno como um telefone celular ou um complexo supercomputador. Os aceleradores gráficos ou as GPUs (*Graphics Processing Units*) vem sendo incorporados em todos os sistemas. Já aparecem em livros clássicos de ensino de arquitetura de computadores como o livro de David Patterson e John Hennessy na sua 5 Edição [1]. Este artigo propõe usar as GPUs como um instrumento prático de ensino das diversas organizações paralelas de memória.

Existem muitos motivos para se usar uma GPU como instrumento de ensino. Primeiro, o tema é recente, fascina e desperta a curiosidade dos alunos. Segundo, os experimentos de medida são realizados com a execução dos códigos ao invés de usar simuladores ou emuladores. Além disso, a curva de aprendizado é rápida. Terceiro, existe a disponibilidade de ferramentas de análise (*profile*) com interface amigável para visualizar as medidas. Quarto, as GPUs possuem diversas configurações de memória e podem ser acessadas remotamente para experimentos. Quinto, a popularização das GPUs é uma realidade com criação dos padrões CUDA e OPENCL. A Tabela I mostra a evolução das GPU da empresa NVIDIA entre os anos de 2008 e 2013 (dados extraídos de [2]).

Table I
Popularização das GPU de 2008 à 2013

	ano	
	2008	2013
GPU com CUDA no Mercado	100 Milhões	430 Milhões
Download de CUDA	150 Mil	1 Milhão
Supercomputadores com GPU	1	50
Universidades com Curso de GPU	60	640
Artigos publicados	4.000	37.000

Inicialmente, as arquiteturas das três últimas gerações de GPU da NVIDIA serão apresentadas. Este trabalho tem foco nas placas da NVIDIA por ser o padrão mais popular no mercado. Posteriormente, as organizações de memória serão abordadas. Ao longo da apresentação das gerações e das organizações de memória, experimentos serão sugeridos. Eles podem ser usados para reduzir a distância entre o código em linguagem de alto nível e o impacto no desempenho da implementação. Sempre que possível serão recomendados livros didáticos com exemplos. Porém, como o assunto é recente, poucos livros abordam o tema [3], [4], [5], [6]. Em português, as sugestões são a tradução do livro [3] e a JAI [7].

II. Gerações de GPU da NVIDIA

As GPGPUs foram o primeiro passo para a popularização das GPUs ao propor uma arquitetura para processamento gráfico de finalidade geral. Motivados pelo poder de processamento e a vazão da memória das placas gráficas, problemas vetoriais não gráficos começaram a ser modelados/implementados e mostraram um ganho de mais de uma ordem de grandeza [8]. Entretanto, um problema não gráfico tinha que ser modelado como um problema gráfico.

O grande salto foi o lançamento da plataforma CUDA (*Compute Unified Device Architecture*) no final de 2006. O programador passou a ter acesso a uma placa para computação paralela vetorial fazendo uso de milhares de *threads* [9]. Do ponto de vista lógico, as *threads* são organizadas em uma estrutura hierárquica com grades de blocos. As grades e os blocos podem ter indexação uni, bi ou tri-dimensional o que facilita a modelagem de problemas físicos como dinâmica de corpos. Em uma GPU Kepler, os blocos podem ter até 1024 *threads* e cada grade pode ter até 4 Giga blocos. Ou seja, milhões ou mesmo bilhões de *threads* podem ser disparados no nível lógico. Entretanto, no nível físico em execução ao mesmo tempo, dependerá da disponibilidade de unidades de processamento na GPU em uso. CUDA permite a portabilidade do código. Ou seja, a medida que novas placas são disponibilizadas com mais processadores e recursos, o número de *threads* em execução concorrentemente pode ser maior, aumentando o desempenho.

Para entender melhor o paralelismo disponível na GPU é importante conhecer a sua arquitetura. A disponibilidade de mais 400 milhões de placas (Tabela I), faz da GPU uma arquitetura paralela de mercado diferente de muitas outras onde a disponibilidade era muito restrita como os protótipos de computadores paralelos ou/e os supercomputadores.

Do ponto de vista físico, as unidades de processamento são organizadas em SM (*Stream Multiprocessors*). Cada SM é formado por um conjunto de unidades de processamento SP (*Stream Processors*). Em termos de organização de memória, cada SM tem recursos que podem ser compartilhados pelos SPs e existe também, a memória global acessível a todos os processadores. A seguir, os detalhes das arquiteturas serão apresentados.

A. Série 8

Em 2008, a série 8 como a GPU GTX280 foi introduzida com uma largura de banda de 141.7 GB/s para memória global e 240 unidades de processamento em 30 SM com 8 SPs cada. Cada SM tem 16K de memória compartilhada. Não existe cache, porém a memória compartilhada pode funcionar como uma ”cache”. O uso destas placas permite aos estudantes explorarem uma arquitetura sem cache. Um experimento simples é a multiplicação de matrizes como ilustrado em [3]. Outro experimento é o uso da memória lógica local. Como não existe cache, se o conjunto de variáveis locais for maior que o número de registradores, a memória global será usada para as variáveis locais podendo gerar uma grande perda de desempenho. As memórias de textura e de constante também estão disponíveis na série 8.

B. Fermi

Em 2010, a arquitetura Fermi introduziu uma série de mudanças. Mas a largura de banda de memória não modificou, por exemplo, a GTX 480 tem 133.9 GB/s. A principal modificação foi a introdução da memória cache L1 em cada SM e uma cache L2 unificada de 768 KB. Cada SM passa a ter 32 SPs (4x mais que série 8), o que permite que mais SP dentro do mesmo SM se comuniquem através da memória compartilhada. Além disso, o tamanho da memória compartilhada e da cache L1 podem ser alterados através de paramêtros de compilação, podendo ser configurada para 48K de cache e 16K de memória compartilhada ou vice-versa. Este recurso permite experimentos com diferentes tamanhos de cache na mesma placa. Outro experimento é executar o mesmo código na série 8 e na Fermi para visualizar a diferença da presença da cache, uma vez que a largura de banda não foi modificada.

C. Kepler

A arquitetura Kepler foi lançada em 2012 com 7 bilhões de transistores, a maior arquitetura paralela de processamento integrada em um único circuito. As gerações Fermi e série 8 tinham 3 bilhões e 1 bilhão, respectivamente. Além do aumento da capacidade de integração, a eficiência em Watts por operação é três vezes melhor na Kepler em comparação com a Fermi. A Kepler mostrou que a NVIDIA investiu muito na tecnologia. A largura de banda da GTX680 é de 192.2 GB/s. O SM passa a ser chamado SMX e agrupa 192 SPs (6x mais que o SM da Fermi). A memória compartilhada e a cache podem ser configuradas em três tamanhos. A Kepler oferece suporte para vários *kernels* com número diferentes de *threads*, além de permitir que um *kernel* seja mais complexo pois suporta 4 vezes mais registros por *thread*, 255 em comparação com os 64 registros da Fermi. Outro recurso é a troca de dados com padrões na memória compartilhada que é útil em processamento de sinal, ordenação, etc.

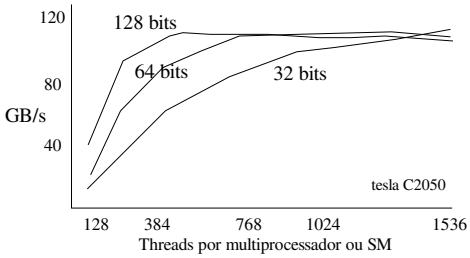

Figure 1. Vazão em função do número de Threads.

Com foco em memória, a NVIDIA pretende ainda lançar a arquitetura Maxwell em 2014 com um sistema unificado de memória virtual e espera que em 2016 já seja possível integrar a memória global no mesmo circuito da GPU usando a tecnologia de memória em 3D (camadas empilhadas) [2].

III. Organizações

Nesta seção iremos apresentar os vários tipos de memória presentes nas GPUs da NVIDIA e como usá-las para ensino.

A. Memória Global

Os dados são transferidos da memória principal da CPU para GPU e armazenados na memória global da GPU. A motivação é a vazão de até 200 GB/s da GPU em comparação com uma vazão de pico de 50GB/s em uma CPU. Do ponto de visto do ensino, uma questão importante: como a GPU consegue atingir estas taxas ?

Para responder à esta questão podemos ilustrar vários conceitos. Primeiro, vazão e latência. A memória global transmite 512 bits por vez ou 128 bytes ou 32 palavras de 32 bits, que pode gerar uma vazão entre 140-200 GB/s. Porém, a latência é de 400-800 ciclos. Portanto, para atingir a vazão máxima, muitas *threads* devem estar ativas solicitando dados. Um experimento simples é incrementar os elementos de um vetor e variar o número de *threads* como ilustra a Figura 1. Cada *thread* irá ler e atualizar o valor. Quando aumentamos a quantidade de *threads* ou o trabalho das *threads* (caso a palavra seja de 64 ou 128 bits) a vazão máxima é atingida mais rapidamente.

Como são 32 palavras por vez, outro conceito importante é entender que, durante a execução, as *threads* são agrupadas em *warps* de 32 *threads* cada. Cada *thread* tem um identificador único (ID). Os *threads* são agrupados

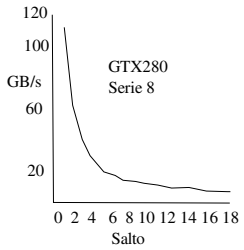

Figure 2. Vazão em função do salto na cópia de um vetor

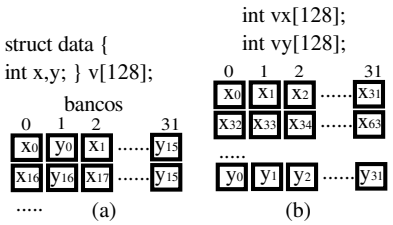

Figure 3. (a) Arranjo de estrutura (b) Estrutura de arranjos.

sequencialmente, isto é, os IDs 0-31 formam o *warp* 0, depois os IDs 32-63 no warp 1, e assim sucessivamente. Se os dados que as *threads* requesitam não estão agrupados, ao invés de receber todos os dados em uma única transferência, podem ser necessárias várias transferências e no pior caso 32. Um experimento simples são leituras com um salto em um vetor como ilustra o código abaixo:

```
__global__copia(float *out,float *in,
                int SALTO){
  int id= (blockIdx.x*blockDim.x
           +threadId.x)*SALTO;
  out[id] = in[id]; } // copia
```

A Figura 2 mostra a redução da vazão em função do salto. Além de agrupados, os dados devem estar alinhados em múltiplos de 128 bytes. Mesmo agrupados, se o dados não estiverem alinhados, um acesso a um pacote de 32 palavras pode gerar duas transferências reduzindo a vazão.

B. Memória Compartilhada em Bancos Distribuídos

A memória compartilhada é organizada em 32 bancos. A primeira palavra no banco 0, a segunda palavra é armazenada no banco 1 e a trigéssima segunda palavra no último banco. A palavra seguinte é no banco 0, e assim sucessivamente.

Na série 8 que não possui cache, a memória compartilhada tem um papel importante, como por exemplo o acesso com saltos ilustrado na seção anterior. Uma simples multiplicação de matrizes é exemplo clássico onde os saltos são necessários para o acesso por coluna. Dois experimentos podem ser realizados: matrizes[3] e histograma [4]. Outro experimento é o acesso a arranjos de estruturas e estrutura de arranjos como ilustrado na Figura 3. O acesso à v.x[i] para os 32 *threads* terá conflitos. Por exemplo, x0 e x16 estão no mesmo banco 0 quando usamos um arranjo de estrutura. Se os campos x e y são armazenados em vetores diferentes, não ocorrerão conflitos. Outros experimentos podem ser propostos para usar uma memória em bancos. A palavra chave *shared* determina que a alocação do vetor será na memória compartilhada.

C. Caches

A cache L1 pode ser desligada usando um paramêtro de compilação, basta compilar com a diretiva *-Xptxas-dlcm=cg*. Além disso, nas arquiteturas Fermi e Kepler pode-se usar diferentes tamanhos de cache também configurados pela compilação.

As ferramentas gráficas de análise, como o Nvidia Visual Profile mostram dados e o uso das caches e das outras memórias como ilustrados da Figura 4 para a versao 2.2 do Nivia Visual Profile.

D. Registros

Outro ponto importante são os registros que armazenam as variáveis locais. A memória de registros é a mais rápida da GPU, podendo ter a vazão de 1 TB/s. Faremos uso do exemplo extraído de [10]. Primeiro, considere a arquitetura Fermi. O termo local é usado pois as variáveis são privadas e cada *thread* possui sua própria cópia. Para obter desempenho, todos os *threads* vivos ocupam os registros do SM no qual eles estão alocados em execução. Cada *thread* pode ter alocado até 64 registros e o total de *threads* (blocos) ativos não pode ultrapassar 32K registros (limite da Fermi). Suponha o exemplo do código de diferenças finitas no domínio do tempo para uma sistema de equações de 3D [10]. O código foi compilado com os paramêtros abaixo fixando em 32 registros por *thread*:

```
$ nvcc fwd_o8.cu -maxrregcount=32 ...
....Used 32 registers, 44+0 bytes lmem
```

Além dos 32 registros, o compilador mostra que foram alocados 44 bytes de memória local (lmem) por *thread*. Ao executar o código com o profile foram extraídos os dados da Tabela II. A taxa de falhas para as leituras em L1 é igual a 86,05%. Devido as falhas em L1, a cache L2 recebe 2*4*564.332 acessos. O fator 2 se deve ao fato de ser uma leitura e uma escrita por falha. O fator 4 é devido a contagem ser por transação de 32 bytes para as quais se precisa de 4 para ter um pacote de 128 bytes. Portanto, as falhas na leitura de L1 geram 4.514.656 acessos a L2 por SM. Como a Fermi tem 16 SM, isto irá gerar 72.234.496 acessos a L2. Ou seja, metade dos acessos a L2 é devido a gravação temporária dos valores das variáveis locais na memória por não ter um número suficiente de registros alocados por *thread* para mantê-los.

Outro valor interessante que pode ser calculado é o número de instruções extras. Para copiar um valor temporariamente de um registro para a memória, o compilador deve gerar pelo menos uma instrução de leitura e outra de escrita. Somando todas as instruções de leitura e escrita da Tabela II temos 938.944 instruções, que representa apenas 4,6% das 20 milhões de instruções executadas.

Ao recompilar o código sem limite de registros, são alocados 46 registros por *thread* e não há necessidade de usar a memória local para cópias temporárias (*register*

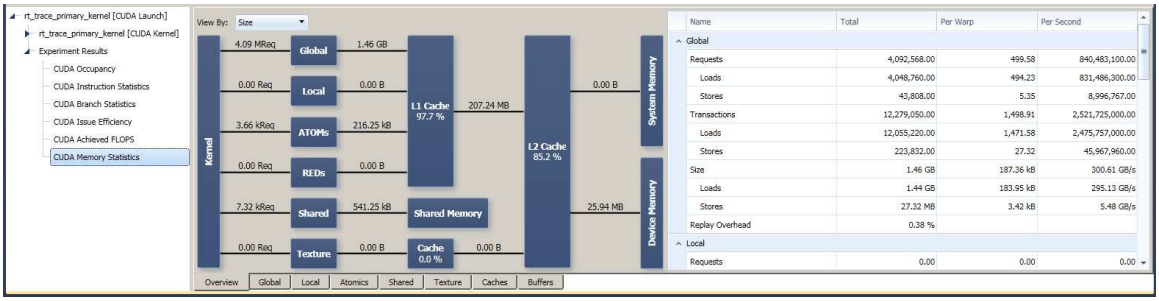

Figure 4. Uso das memórias com o Visual Profile 2.2

Table II
CONTADORES DE ACESSO AS CACHES

Contador por SM	Valores
Falha na Leitura Local L1	564.332
Acerto na Leitura Local L1	91.520
Falha na Escrita Local L1	269.215
Acerto na Escrita Local L1	13.477
Instruções executadas	20.412.215
Cache L2	
Leituras	99.435.608
Escritas	33.385.908

spilling). O desempenho do tempo de execução melhora em 22%, ou seja, a aplicação passa a executar 1,22 vezes mais rápida.

Outra opção é recompilar mudando o tamanho da cache de 16K para 48K e mantendo apenas 32 registros, que gera o uso de 44 bytes de memória local. Com a cache maior, a taxa de falhas nas leituras em L1 agora é de apenas 1,68%, que gera apenas um pequeno aumento de 1,63% no uso da L2. O desempenho da aplicação melhora em 1,45 vezes. Neste caso, a opção de aumentar a cache gerou um desempenho melhor.

E. Memórias Específicas

Além das memórias supracitadas, algumas memórias foram herdadas das gerações precedentes de GPGPUs.

1) Textura: A memória de textura possui recursos interessantes como o apoio do hardware para a interpolação no acesso a um vetor com indexação normalizada entre 0,0 e 1,0, ou seja, acesso com um número real como índice. Acesso fora da borda ou acesso modular. Por exemplo, suponha uma matriz 4 por 4. O acesso ao elemento m[5.5][1.5] será convertido em m[6 mod 4][2] = m[2][1].

O acesso com padrões 1D,2D e 3D são utéis em várias aplicações gráficas e/ou não gráficas com a propagação de uma onda de calor ilustrado como exemplo em [4].

2) Constante: A memória de constante possui o tamanho de 64Kb e só é eficiente se for acessada com o padrão *broadcast*.

3) Cache de Leitura: Como existem muitos casos de constantes que podem ser acessadas sem um padrão de *broadcast*, a geração Kepler introduziu uma cache apenas de leitura.

IV. CONSIDERAÇÕES FINAIS

Vários experimentos podem ser propostos com acesso local ou remoto à algumas placas de GPU e o conhecimento básico da sua arquitetura. Indicamos a referência [4] para introdução com exemplos práticos e simples de memória. Os conceitos de latência e vazão, agrupamento de dados para acesso paralelo podem ser explorados com o uso da memória global. Diferentes tamanhos de cache podem ser configurados com paramêtros de compilação. Outro ponto a ser explorado é a importância de se manter as variáveis locais nos registros. Experimentos com bancos de memória distribuída podem usar a memória compartilhada. A disponibilidade de ferramentas de análise do código executado, configurar a arquitetura com o compilador, os contadores em hardware, dentre outros recursos, permitem explorar vários aspectos. O grande desafio de acelerar o acesso aos dados permanence. Entender as soluções atuais com detalhes é importante para buscar novas alternativas.

REFERENCES

[1] J. L. Hennessy and D. A. Patterson, *Computer architecture: a quantitative approach*, 5th ed. Elsevier, 2012.

[2] J.-H. Huang, “What’s next in gpu technology?” in *GTC GPU Technology Conference*, 2013.

[3] D. B. Kirk and W. H. Wen-mei, *Programming massively parallel processors*. Morgan Kaufmaan, 2010.

[4] J. Sanders and E. Kandrot, *CUDA by example*. Addison-Wesley Professional, 2010.

[5] S. Cook, *CUDA Programming: A Developer’s Guide to Parallel Computing with GPUs*. Newnes, 2012.

[6] N. Wilt, *The CUDA Handbook: A Comprehensive Guide to GPU Programming*. Addison-Wesley Professional, 2013.

[7] F. Q. Pereira, *Tecnicas de otimizacao de codigo para placas graficas*. Jornadas de Atualizacao em Informatica, 2011.

[8] D. Luebke, M. Harris, N. Govindaraju, A. Lefohn, M. Houston, J. Owens, M. Segal, M. Papakipos, and I. Buck, “Gpgpu: general-purpose computation on graphics hardware,” in *ACM/IEEE Conf. on Supercomputing*, 2006.

[9] C. Nvidia, “Nvidia cuda programming guide,” 2011.

[10] P. Micikevicius, “Local memory and register spilling,” in *GTC GPU Technology Conference*, 2011.