GPUramfs: Um Sistema de Arquivos em Espaço de Usuário para Armazenamento Volátil em GPU

Jorge Pires Correia
jpcorreia@inf.ufpr.br
Universidade Federal do Paraná
Curitiba, Paraná, Brasil

Wagner M. Nunan Zola
wagner@inf.ufpr.br
Universidade Federal do Paraná
Curitiba, Paraná, Brasil

ABSTRACT

GPUs are increasingly present in personal computers. Most of the time they are idle. Using the memory resources of these devices can free up space in the host's RAM, allowing the operating system to use it efficiently. This work presents the initial implementation of GPUramfs, a volatile file system that uses the GPU as a storage device. The system exhibited good performance even in comparison to EXT4 using NVMe high throughput disks.

KEYWORDS

GPU, File System, RAM, Idle Resources

1 INTRODUÇÃO

A utilização de GPUs para diversos tipos de processamento vem crescendo nos últimos anos. O auxílio proporcionado por elas através do grande número de núcleos de processamento (*cores*) permite a implementação de algoritmos paralelos escaláveis. Contudo, uma GPU não é composta somente de núcleos de processamento. Modelos de GPU denominados integrados utilizam a memória RAM da CPU, no entanto as GPUs dedicadas possuem memória RAM própria e mecanismos de DMA, os quais possuem grande potencial para auxiliar determinadas tarefas do sistema. A capacidade que as GPUs têm de auxiliar o processamento de grandes volumes de dados é inegável. Mas tais dispositivos podem também auxiliar os sistemas computacionais de outras formas. Nos computadores pessoais, a GPU é utilizada em situações específicas onde existe uma necessidade bem definida, de forma que em grande parte do tempo, seus componentes ficam ociosos. A utilização da memória dedicada destes dispositivos para armazenamento de arquivos temporários pode evitar a utilização da memória RAM do *host*, permitindo que sistema operacional utilize este espaço em prol do desempenho do sistema, na forma de caches de páginas de disco e diminuição de swap de páginas por exemplo.

Um dos principais problemas enfrentados quando as GPUs são utilizadas é a vazão proporcionada pelo barramento *PCI Express*, o qual conecta a GPU à CPU e a outros dispositivos. A vazão deste barramento é ainda pequena quando comparada à vazão dos barramentos que conectam memória(s) ao(s) processador(es). Contudo, tipicamente, a vazão deste barramento dobra a cada nova versão.

Este trabalho apresenta a primeira implementação do GPUramfs, um sistema de arquivos implementado em *userspace* através do framework FUSE [1], que utiliza parte da memória RAM dedicada da GPU como dispositivo de armazenamento volátil, liberando espaço de memória do *host* e contribuindo para melhor utilização do sistema computacional.

2 TRABALHOS RELACIONADOS

BAG (Buffer cAche on GPU) [2] é um sistema que utiliza a memória RAM dedicada da GPU como *buffer cache* no sistema operacional, de forma que a GPU é vista como um dispositivo de bloco virtual. BAG é um dos poucos trabalhos existentes que tomam a memória da GPU como principal foco. Sua implementação consiste em três principais componentes: *indirector*, *relay* e *deamon*. O *indirector* é um código que executa em *kernel space*, o qual intercepta todas as requisições de *block I/O* que são enviadas dos sistemas de arquivos para os dispositivos de disco. Para cada interceptação, o *indirector* verifica se a operação é de escrita ou de leitura e se o bloco está presente no cache. Se a requisição se enquadra em algum desses dois casos, então ela é repassada para o *relay*, caso contrário, a requisição é repassada para o dispositivo de armazenamento. O *relay* é um código que executa em *kernel space* que recebe as requisições de *block I/O* selecionadas pelo *indirector* e mapeia as áreas de memórias envolvidas na requisição para o *deamon*. Este, por sua vez, é um código que executa em *userspace* e que realiza a real interação com a GPU através da interface CUDA, assim como chamadas de códigos que rodam na GPU para realizar procedimentos de criptografia e *garbage collector*. Alguns trabalhos utilizam GPUs como componentes que aceleram determinados procedimentos do sistema de arquivos. GPUstore [3] é um sistema que se comporta como como uma interface genérica em *kernel space* para sistemas de arquivos, de forma que estes possam utilizar a GPU de maneira transparente e otimizada.

3 GPURAMFS

No presente trabalho construímos o GPUramfs como um sistema de arquivos implementado em *userspace* através da biblioteca FUSE. O GPUramfs possibilita a utilização da memória da GPU como dispositivo de armazenamento. Nesta implementação inicial, somente duas estruturas definem as entidades do sistema: *inode* e *dentry*. Um *inode*, apresentado na Figura 1a, é uma estrutura de 256 bytes que armazena metadados dos arquivos que são repassados para o FUSE quando requisitados. Uma *dentry* é uma estrutura de 256 bytes que define uma entrada em um diretório, armazenando o nome de um arquivo nesse diretório e o seu número de *inode* associado.

Ainda que o GPUramfs encare a GPU como um dispositivo de armazenamento, o espaço disponível é muito menor que nos dispositivos de armazenamento comuns. Dois valores definidos pelo usuário são usados na inicialização do GPUramfs: o número de páginas de memória reservadas na GPU para armazenar dados e metadados dos arquivos, e o número de páginas de memória reservadas na CPU a serem utilizadas como cache. Diante disso, o tamanho e a quantidade dos arquivos presentes no sistema de arquivos deve ser condizente com o espaço disponível. O que delimita a quantidade

máxima de arquivos é a quantidade de *inodes*. O GPUramfs dedica uma porcentagem das páginas reservadas na GPU para armazenar essas estruturas, de forma que cada página dedicada ao armazenamento de *inodes* comporta 16 estruturas. O tamanho máximo de um arquivo é definido pela quantidade de páginas de dados que ele tem. Cada *inode* possui um vetor de 40 posições, armazenando em cada posição um número de página. Essas páginas referenciadas pelo *inode*, aqui referidas como *buckets*, são responsáveis por armazenar os números das páginas de dados, formando um nível de ponteiros indiretos. Desse modo, um *inode* possui 40 *buckets*, cada *bucket* é capaz de referenciar 1024 páginas de dados, limitando o tamanho máximo dos arquivos em 160MB.

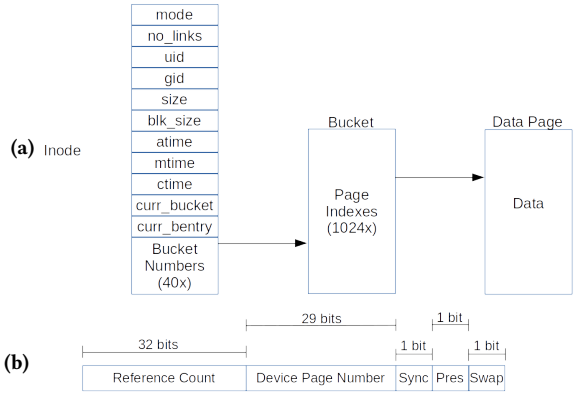

Figura 1: (a) Estrutura do inode e (b) mapeamento de uma página de cache

A arquitetura do GPUramfs pode ser dividida em quatro partes: interface FUSE, interface CUDA, *helpers* e gerenciador de memória. A interface FUSE contém a implementação das funções que constituem os pontos de entrada para o GPUramfs. Essas funções são definidas através da estrutura *fuse_operations*. A interface CUDA implementa funções que utilizam a API CUDA, para que os demais componentes do sistema possam acessá-las mais facilmente. Os *helpers* formam um conjunto de funções que são utilizadas muitas vezes durante a implementação, como a aquisição do endereço do *inode* através do seu número, expansão e redução da quantidade de *buckets* utilizados para determinado arquivo e aquisição do número da página que contém o próximo byte a ser escrito no arquivo. O gerenciador de memória é responsável por manter os mapas de *inodes* livres, páginas livres da GPU e gerenciar a cache na memória da CPU. Os *inodes* livres, páginas livres da GPU e páginas livres da cache foram mapeados através de pilhas, proporcionando acessos eficientes a estes recursos. O mapeamento das páginas de cache foi realizado através de um vetor de inteiros de 64 bits que seguem a estrutura apresentada na Figura 1b, um inteiro para cada página da cache. O bit menos significativo é denominado *swappable*. O seu valor indica se página pode ser movida para a memória da GPU ou se será sempre mantida em cache. Essa funcionalidade foi implementada para testes futuros, pensando em manter informações muito acessadas sempre na memória da CPU. O segundo bit menos significativo, denominado *present*, determina se a página está presente em cache ou se a posição está livre. O terceiro bit menos significativo é denominado *sync_write*. O valor um indica que uma escrita àquela página deve resultar em uma escrita imediata à GPU; caso zero, a escrita para a GPU será executada somente quando essa página precisar ser movida para GPU. Os próximos 29 bits armazenam o número da página da GPU que aquela página de cache está referenciando. Os 32 bits de alta ordem armazenam o contador de referências à página, de forma que ela não pode ser movida para a memória da GPU se o contador for diferente de zero, pois isso indica que a página está em uso. A troca de páginas entre a cache e a memória da GPU ainda é realizada de forma ingênua: um *token* indica qual página deve ser movida para a cache. Caso a página não esteja disponível para mudança, o *token* passa para a próxima página em sequência de forma circular. Quando uma página é movida para a GPU, o *token* é incrementado.

4 RESULTADOS INICIAIS

O ambiente de testes utilizado neste trabalho consiste de um processador Intel i7-10700, com barramento *PCI Express* 3.0, disco NVMe de alta vazão, sistema operacional Linux kernel 5.4 x86_64, GPU NVIDIA GTX 1070 e biblioteca CUDA na versão 11.4.

O GPUramfs armazena dados na memória da GPU, que tem alta vazão se comparada à vazão de acesso à memória da CPU. No entanto, a transferência de dados da CPU para a GPU ocorre via barramento *PCI Express* que tem vazão bastante inferior. Nesse caso, o barramento *PCI Express* representa o gargalo para a vazão de armazenamento de dados no GPUramfs. Assim, o desempenho do GPUramfs não é diretamente comparável ao desempenho de um sistema de arquivos em memória da CPU. Com o objetivo de realizar uma análise mais justa do desempenho, nesse trabalho foram feitas comparações de desempenho entre um sistema de arquivos em disco NVMe (*Non-Volatile Memory express*) de alta vazão. Esses dispositivos possibilitam armazenamento em memória *NAND flash*, também ligados à CPU via barramento *PCI Express*. A vazão de pico indicada pelo fabricante do dispositivo NVMe utilizado é de 1800 MB/s para leituras e 1200 MB/s para escritas

Para comparar o desempenho do GPUramfs, foi implementado um outro sistema de arquivos com a biblioteca FUSE que simplesmente redireciona as chamadas ao sistema de arquivos para o sistema nativo EXT4 em disco NVMe de alta vazão. Assim, a comparação inclui o overhead inerente ao sistema FUSE, evitando a comparação direta com o EXT4. O ambiente de benchmarks Filebench [4] foi utilizado para realizar as medições. A carga (*workload*) padrão denominada *fileserver* foi utilizada, mas com algumas alterações, visto que as características de um servidor de arquivos e de um sistema de arquivos em RAM são um pouco diferentes. Em cada interação da *workload*, a quantidade e o tamanho máximo de arquivos foram diminuídos. Em um caso de teste, a diretiva dsync foi utilizada na abertura dos arquivos, para que fosse possível a verificação do desempenho desconsiderando a *page cache* do sistema operacional. Outra carga de trabalho *fileserver-reduzida* foi criada, a qual realiza somente operações de escrita e leitura, sem operar com metadados. Dessa forma é possível verificar o impacto destas operações no sistema de arquivos.

5 CONCLUSÃO

Mesmo com uma implementação inicial simplificada de organização de diretórios, mapeamento e troca de páginas do cache, o

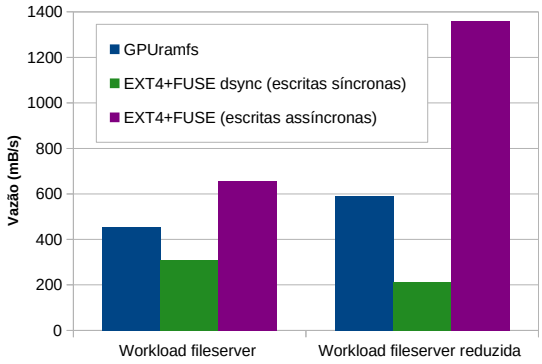

Figura 2: Comparação entre o GPUramfs e o EXT4 em disco NVMe de alta vazão, com o overhead inerente ao FUSE

GPUramfs apresentou resultados melhores que com o EXT4 em escritas síncronas (descontado o *overhead* do FUSE). Os resultados também demonstraram boa eficiência para a carga de trabalho que não realiza operações com metadados. Futuros aperfeiçoamentos nesses mecanismos poderão melhorar o desempenho geral do sistema. O tamanho da *page cache* e os algoritmos que a gerenciam são características de alta importância para o desempenho do sistema de arquivos. É possível, também, implementar códigos que executem na GPU, incluindo algoritmos de gerenciamento de cache com o objetivo de diminuir a quantidade de cópias de memória e aproveitar os mecanismos de DMA.

REFERÊNCIAS

[1] Bharath Kumar Reddy Vangoor, Vasily Tarasov, and Erez Zadok. To FUSE or not to FUSE: Performance of User-Space file systems. In *15th USENIX Conference on File and Storage Technologies (FAST 17)*, pages 59–72, Santa Clara, CA, February 2017. USENIX. ISBN 978-1-931971-36-2.

[2] Hao Chen, Jianhua Sun, Ligang He, Kenli Li, and Huailiang Tan. BAG: Managing GPU as buffer cache in operating systems. *IEEE Transactions on Parallel and Distributed Systems*, 25(6):1393–1402, 2013.

[3] Weibin Sun, Robert Ricci, and Matthew L. Curry. GPUstore: Harnessing GPU computing for storage systems in the os kernel. In *Proceedings of the 5th Annual International Systems and Storage Conference*, SYSTOR '12, New York, NY, USA, 2012. Association for Computing Machinery. ISBN 9781450314480.

[4] Vasily Tarasov, Erez Zadok, and Spencer Shepler. Filebench: A flexible framework for file system benchmarking. *USENIX; login*, 41(1):6–12, 2016.