Paralelização do Dedup para Sistemas Multi-core com GPUs

Charles Michael Stein[1], **Dinei Rockenbach**[1,2], **Dalvan Griebler**[1,2]

[1]Laboratório de Pesquisas Avançadas para Computação em Nuvem (LARCC),
Faculdade Três de Maio (SETREM), Três de Maio – RS – Brasil

[2]Escola Politécnica, Pontifícia Universidade Católica do Rio Grande do Sul (PUCRS),
Porto Alegre – RS – Brasil

{charlesmst,dineiar}@gmail.com,dalvan.griebler@acad.pucrs.br

Resumo. *O maior volume de dados gerado, trafegado e processado aumenta a demanda por mais poder de processamento e por algoritmos de compressão eficientes. Este trabalho tem como objetivo explorar o paralelismo de stream para arquiteturas multi-core com GPUs na aplicação Dedup, usando SPar com CUDA e OpenCL. Apesar do desempenho não ser o esperado, o artigo contribui com uma análise detalhada dos resultados e sugestões futuras de melhorias.*

1. Introdução

O paradigma da programação paralela tem persistentemente desafiado os programadores, incentivando o surgimento de abordagens como *algorithmic skeletons* e *parallel design patterns* com o objetivo de facilitar o desenvolvimento de algoritmos paralelos [McCool et al. 2012]. Essas abordagens levaram ao desenvolvimento de ferramentas e linguagens voltadas a facilitar a programação paralela. A linguagem SPar (*Stream Parallelism*) [Griebler et al. 2017] foi criada para que desenvolvedores de aplicações de *stream* possam explorar o paralelismo presente em CPUs (*Central Processing Units*) *multi-core* sem se preocupar com detalhes da arquitetura. A SPar é uma linguagem específica de domínio (DSL) baseada nos atributos introduzidos no padrão C++11. O compilador da SPar interpreta esses atributos e realiza transformações no código-fonte, introduzindo código C++ paralelo através de chamadas à biblioteca FastFlow.

Embora a programação paralela remonte à década de 60, apenas com a introdução das CPUs *multi-core* nos computadores *desktop* em 2005 foi que o impacto real dessa mudança chegou à indústria de software. Da mesma forma, a exploração do paralelismo massivo das GPUs (*Graphical Processing Units*) iniciou em 2001 através do uso de APIs (*Application Programming Interfaces*) como DirectX e OpenGL, no entanto, somente com o surgimento de CUDA em novembro de 2006 que a programação em GPGPU (*General Purpose GPU*) começou a se popularizar. Neste trabalho, o desafio é explorar o paralelismo de stream de forma eficiente em uma arquitetura *multi-core* com GPUs na aplicação Dedup. Ela é baseada na deduplicação de dados e é largamente utilizada em segmentos como *storages* e *backups* para economizar espaço em disco através da redução da redundância dos blocos. Ela também faz parte do PARSEC Benchmark Suite [Bienia et al. 2008], que é utilizado para avaliar o desempenho de sistemas paralelos e cujo destaque está em prover cargas de dados baseados em sistemas reais.

Dentre os trabalhos relacionados a este, [Suttisirikul and Uthayopas 2012] criaram um sistema de *backup* em nuvem baseado em deduplicação de dados utilizando *hashes* SHA-256 calculados na GPU para identificar dados duplicados e transferir apenas os dados necessários para o servidor em nuvem. O *speedup* alcançado em relação à

versão sequencial foi de 53x. O presente trabalho se assemelha ao utilizar a GPU para a computação de *hashes*, e se diferencia ao utilizar a compressão de blocos deduplicados em GPU. O trabalho de [Bhatotia and Rodrigues 2012] também implementou paralelismo em uma aplicação baseada em deduplicação para sistemas de *storage*. Semelhante a este trabalho, a aplicação foi estruturada em um *pipeline*. Contudo, o conteúdo completo dos blocos foi transferido para a GPU para processar o *rabin fingerprint* e o SHA-1, obtendo um *speedup* de 5x. Além disso, não foi aplicado o suporte a multi-GPU e o trabalho focou na otimização do particionamento de blocos e transferência de memória. Nenhum dos trabalhos implementou o paralelismo em GPU para o algoritmo Dedup do PARSEC Benchmark.

Uma vez que a programabilidade e o desempenho da aplicação Dedup utilizando a SPar já foi estudada em [Griebler et al. 2018], o presente artigo evolui a partir dessa implementação para explorar o paralelismo e avaliar o desempenho do Dedup em GPUs. Portanto, as contribuições do artigo são: (a) implementação do suporte à múltiplas GPUs com CUDA e OpenCL na aplicação Dedup, partindo da implementação para CPUs *multi-core* com a SPar de [Griebler et al. 2018]; e (b) identificação de gargalos que degradam o desempenho no paralelismo massivo do algoritmo Dedup em GPUs devido ao uso do algoritmo *rabin fingerprint*. Desta forma, a Seção 2 apresenta como foi implementado e explorado o paralelismo. Depois, uma avaliação do desempenho e das implementações é discutida na Seção 3. Por fim, a Seção 4 conclui este trabalho.

2. Desenvolvimento

A implementação do Dedup paralelo com a SPar de [Griebler et al. 2018] partiu do código fonte sequencial do PARSEC Benchmark Suite [Bienia et al. 2008], que foi dividido em 3 estágios: 1) aplicação do algoritmo *rabin fingerprint* para fragmentar a entrada em blocos a serem comprimidos; 2) cálculo do *hash* SHA-1 dos blocos, verificação de duplicidade e compressão dos dados; 3) ordenação dos dados e escrita na saída. No caso, o estágio 2 é replicado. Na implementação do presente trabalho, a fragmentação é executada na CPU e os índices gerados pelo *rabin fingerprint* são transferidos para a GPU, que realiza o cálculo do SHA-1. Outra alteração foi utilizar LZSS como algoritmo de compressão ao invés do Bzip2 [Griebler et al. 2018], uma vez que o primeiro já foi implementado para GPU em [Stein et al. 2019].

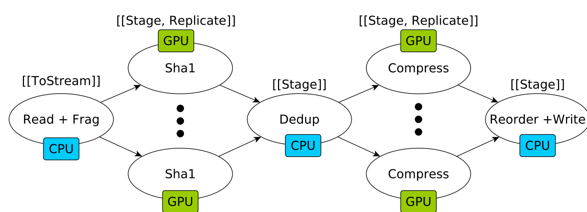


Figura 1. Grafo de atividade.

Ao anotar o código-fonte com os atributos da DSL SPar, o fluxo da aplicação foi dividido em 5 estágios anotados com a SPar, conforme ilustra a Figura 1: 1) o arquivo de entrada é lido pela CPU em blocos de 10MB e executa o *rabin fingerprint*, passando os sub-blocos gerados adiante para o próximo estágio; 2) os sub-blocos são transferidos para a(s) GPU(s), que fazem a geração do *hash* SHA-1. A estratégia adotada foi que cada *thread* da GPU gera o *hash* de um sub-bloco do *rabin fingerprint* e salva em um vetor que é enviado para o próximo estágio. Este estágio é replicado conforme o número de GPUs disponíveis; 3) verifica a deduplicidade dos sub-blocos; 4) comprime os sub-blocos não deduplicados, utilizando o algoritmo LZSS sobre os dados dos sub-blocos ainda presentes na GPU (já transferidos na etapa b); 5) reordena os sub-blocos e os blocos para escrita na saída.

Antes de chegar na versão paralela representada na Figura 1 foi implementada uma versão onde apenas o estágio SHA1 era paralelo. Neste caso, a compressão era o gargalo, pois o tempo de compressão é muito alto. Outro desafio na paralelização do Dedup foi a refatoração de código, pois muitas operações precisaram ser reimplementadas em CUDA e OpenCL a fim de serem chamadas dentro do *kernel*. Além disso, as operações da *stream* precisam ser modeladas em *batch* para fazer um uso efetivo do poder computacional da GPU. Diante disso, implementar paralelismo de *stream* com o uso de GPU é um grande desafio para os programadores. Desta forma, nota-se claramente a necessidade de abstrações de mais alto nível para simplificar a programação paralela em arquiteturas *multi-core* com GPUs.

3. Avaliação

O desempenho da aplicação foi avaliado com versões sequenciais, paralelismo na CPU através da SPar e paralelismo em uma e em duas GPUs utilizando CUDA e OpenCL. Testes foram executados em um ambiente com duas GPUs Titan Xp (arquitetura Pascal, *compute capability* 6.1, 12GB de memória GDDR5X), um processador Intel(R) Core(TM) I9-7900X CPU (10 núcleos físicos e 20 *threads*), 32GB de RAM e 2TB de disco. O sistema operacional utilizado foi o Ubuntu Server 18.04 (Kernel 4.15.0-38-generic). Foram utilizados os compiladores G++ 7.3.0 e `nvcc` 10 com a opção `-O3`. Foram utilizados três *datasets*: (a) Input large (185MB), presente no PARSEC Benchmark; (b) Linux Source (816MB), que é composto de um arquivo tar do código fonte do Linux; (c) Silesia (203MB), composto de vários *datasets* agregados como códigos-fonte, XMLs, dentre outros dados de casos de uso reais. Foram realizados testes para definir o melhor tamanho para os elementos do *stream*, que ficou em 10MB. Na Figura 2 está apresentado o desempenho de todas as versões da aplicação com todos os *datasets*. Como pode ser visto, apenas em alguns *datasets* a versão em GPU obteve um desempenho melhor do que a versão sequencial, e os *speedups* foram pequenos. Em todos os casos as versões paralelas utilizando apenas a CPU tiveram desempenho melhor do que as que utilizaram a GPU. Estes comportamentos não eram esperados, uma vez que o uso do LZSS em GPU representa um grande *speedup* [Stein et al. 2019].

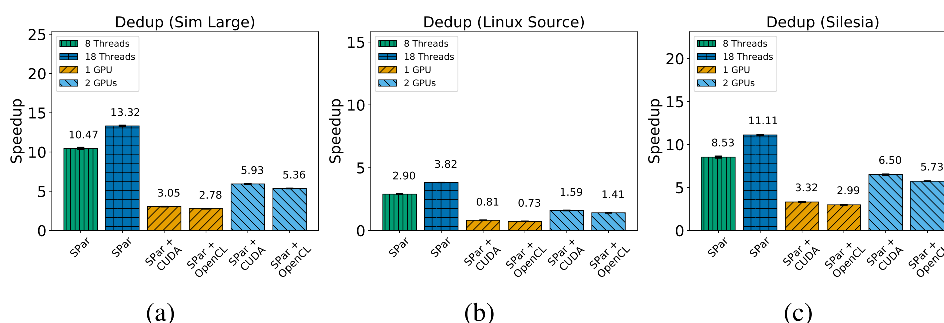
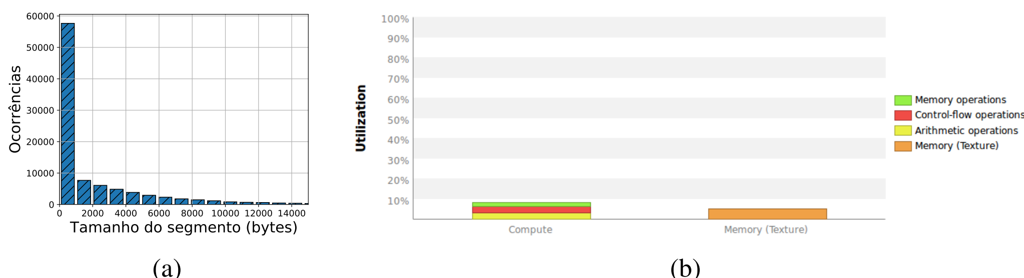


Figura 2. Speedup do Dedup.

Para investigar a causa dessa lentidão na GPU foi utilizado o *profiler* `nvvp`, através do qual foi possível notar que o gargalo estava no grande número de chamadas ao *kernel* do LZSS. Identificou-se que a segmentação feita pelo algoritmo *rabin fingerprint* estava gerando sub-blocos muito pequenos para a compressão na GPU, com a maioria dos blocos menores que 1KB. A Figura 3(a) apresenta o histograma dos tamanhos dos blocos gerados pelo algoritmo. Blocos tão pequenos não são capazes de utilizar a GPU de

forma otimizada e a razão entre o tempo em que a GPU está efetivamente computando e o tempo gasto na coordenação do trabalho entre CPU e GPU fica pequena. A Figura 3(b) demonstra o baixo percentual de uso da GPU, sendo que a maior parte das operações são de controle de fluxo. Desta forma, uma solução para o baixo desempenho seria otimizar o algoritmo de compressão para que ele seja executado em *batches*, a fim de melhor utilizar os recursos da GPU.



(a) (b)

Figura 3. Análise do Dedup.

4. Conclusões

Neste trabalho a aplicação Dedup foi paralelizada para arquiteturas multi-core com GPUs, usando SPar com CUDA e OpenCL. Como pôde ser visto, não foi possível aproveitar adequadamente o paralelismo massivo das GPUs devido ao tamanho dos blocos gerados pelo algoritmo de segmentação, gerando pouca carga para a compressão pelo LZSS. Como trabalho futuro, a otimização do LZSS pode ser feita para que esta segmentação não precise ser feita, executando apenas um *kernel* por bloco para múltiplas compressões. Além disso, pode-se implementar a geração automática de código CUDA e OpenCL a partir dos atributos da linguagem SPar, solucionando problemas comuns e permitindo uma maior abstração ao programador de aplicações de *stream* que possuam paralelismo de dados.

Referências

Bhatotia, P. and Rodrigues, R. (2012). Shredder: GPU-Accelerated Incremental Storage and Computation. In *USENIX Conference of File and Storage technologies (FAST)*.

Bienia, C., Kumar, S., Singh, J. P., and Li, K. (2008). The PARSEC Benchmark Suite: Characterization and Architectural Implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*.

Griebler, D., Danelutto, M., Torquati, M., and Fernandes, L. G. (2017). SPar: A DSL for High-Level and Productive Stream Parallelism. *Parallel Processing Letters*, 27(01):1740005.

Griebler, D., Hoffmann, R. B., Danelutto, M., and Fernandes, L. G. (2018). High-Level and Productive Stream Parallelism for Dedup, Ferret, and Bzip2. *International Journal of Parallel Programming*, pages 1–19.

McCool, M., Robison, A. D., and Reinders, J. (2012). *Structured Parallel Programming: Patterns for Efficient Computation*. Morgan Kaufmann.

Stein, C. M., Griebler, D., Danelutto, M., and Fernandes, L. G. (2019). Stream Parallelism on the LZSS Data Compression Application for Multi-Cores with GPUs. In *27th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*, Pavia, Italy. IEEE.

Suttisirikul, K. and Uthayopas, P. (2012). Accelerating the Cloud Backup Using GPU Based Data Deduplication. In *18th International Conference on Parallel and Distributed Systems*, pages 766–769, Singapore. IEEE.