Revisando a Programação Paralela com CUDA nos Benchmarks EP e FT

Gabriell A. de Araujo[1], **Dalvan Griebler**[1,2], **Luiz G. Fernandes**[1]

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

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

{gabriell.araujo,dalvan.griebler}@acad.pucrs.br,luiz.fernandes@pucrs.br

Resumo. *Este trabalho visa estender os estudos sobre o NAS Parallel Benchmarks (NPB), os quais possuem lacunas relevantes no contexto de GPUs. Os principais trabalhos da literatura consistem em implementações antigas, abrindo margens para possíveis questionamentos. Nessa direção, foram realizados novos estudos de paralelização para GPUs das aplicações EP e FT. Os resultados foram similares ou melhores que o estado-da-arte.*

1. Introdução

NAS Parallel Benchmarks (NPB) foi desenvolvido pela NASA e se trata de um conjunto de *benchmarks* baseados em dinâmica de fluídos [Bailey et al. 1994]. NPB têm se destacado em pesquisas sobre GPUs, contudo, ainda existem lacunas nesse contexto, uma vez que os principais trabalhos da área são de longa data, questiona-se a possibilidade de serem obtidos melhores resultados na aceleração das aplicações. Este trabalho inicia uma série de estudos que visam responder se é possível obter melhor desempenho do NPB através de implementações mais atuais, apresenta uma implementação CUDA de EP e FT, compara os resultados com os principais trabalhos da literatura e visa estender [Griebler et al. 2018], o qual apresentou a implementação C++ do NPB para sistemas multi-core. O artigo está organizado da seguinte forma. Seção 2 discorre sobre os trabalhos relacionados. Seção 3 descreve as implementações. Seção 4 discute os resultados obtidos. Seção 5 apresenta as conclusões.

2. Trabalhos Relacionados

EP e FT são paralelizados com CUDA em [Pilla 2009]. [Gong et al. 2010] implementa EP com CUDA. [Malik et al. 2012] efetua um estudo comparativo com CUDA, OpenCL e OpenACC utilizando CG, EP, FT e MG. [Jin et al. 2012] e [Stone and Elton 2015] paralelizam o algoritmo SP com OpenACC. [Dümmler and Rünger 2013] apresenta versões CUDA de BT, LU e SP. BT, CG, EP, FT, IS, LU, MG e SP são paralelizados com OpenCL em [Seo et al. 2011] e OpenACC em [Xu et al. 2015]. [Pilla 2009], [Dümmler and Rünger 2013], [Seo et al. 2011] e [Xu et al. 2015] são os únicos que disponibilizam as implementações. [Seo et al. 2011] e [Xu et al. 2015] são os trabalhos mais importantes da literatura, uma vez que implementaram e disponibilizaram o NPB completo para GPUs, uma parcela significativa da literatura passou a utiliza-los como base de suas pesquisas. Na seleção dos artigos foram descartados trabalhos que utilizam ferramentas que geram código automático para GPUs ou que focam em programação Multi-GPU.

3. Implementação

EP é caracterizado por um laço cujas iterações geram números aleatórios baseados em desvios gaussianos [Bailey et al. 1994]. A implementação paralela consiste em criar uma quantidade de *threads* correspondente ao número de iterações e cada *thread* então executa as computações de uma única iteração. Os resultados das computações das *threads* devem ser combinados em um vetor nq, uma variável sx e uma variável sy. Apenas uma cópia de nq, sx e sy são mantidas na memória global. Cada *thread* calcula localmente os valores de nq, sx e sy, estando sx e sy localizadas na memória compartilhada. No final da função cada *thread* escreve os valores de nq na memória global utilizando operações atômicas. Após, o bloco é sincronizado e em paralelo, a *thread* de id 0 combina os valores sx calculados pelas *threads* do bloco e a *thread* de id 1 combina os valores de sy. Por fim, com operações atômicas, a *thread* 0 escreve o valor combinado de sx na memória global e a *thread* 1 escreve o valor combinado de sy.

FT calcula transformada rápida de *Fourier* em matrizes tridimensionais de números complexos [Bailey et al. 1994]. As iterações do laço principal podem ser executadas de maneira independente. Também é possível implementar paralelismo de dados dentro de cada iteração. Cada iteração é dividida em fases síncronas, respectivamente. *Evolve*, *Fourier* na dimensão x (*fft1*), *Fourier* na dimensão y (*fft2*), *Fourier* na dimensão z (*fft3*) e combinação dos resultados (*checksum*). Executada antes do fluxo de iterações, a zona denotada como *setup* calcula as condições iniciais da aplicação e também é paralelizável. Nesta região são computados os valores das matrizes globais ex, $indexmap$, u e $u1$. Para esta região são criadas quatro *threads* na CPU, cada *thread* é responsável por calcular uma dessas matrizes. A *thread* responsável por ex computa sequencialmente os valores da matriz e após copia os dados para a GPU. Os 3 laços aninhados que computam $indexmap$ são combinados e executados na GPU através de um *kernel* que é lançado pela *thread* responsável por $indexmap$. A *thread* responsável por $u1$ calcula sequencialmente as sementes para esta, e após as copia para a GPU e lança um *kernel* cujo realiza uma série de computações relativas às sementes e então atribui os valores à matriz $u1$. Apenas o laço mais externo dessa computação é paralelizado devido a dependências inerentes. Por fim, a *thread* responsável por u calcula sequencialmente os valores da matriz e após copia os dados para a GPU. Os 3 laços da função *Evolve* são combinados em um kernel. Cada *thread* calcula uma posição das matrizes recebidas como argumentos da função. As funções *Fourier fft1*, *fft2* e *fft3* possuem comportamento parecido. Copiam valores da matriz de entrada para uma matriz auxiliar *y*, calculam *Swarztrauber* e *Stockham* na matriz *y* e por fim, escrevem os valores de *y* na matriz de saída. *Fourier* consiste em 3 kernels. Combinação dos 4 laços que copiam os valores da matriz de entrada para a matriz *y*. Combinação dos laços que computam *Swarztrauber* e *Stockham*. E por fim, combinação dos 4 laços que copiam os valores da matriz *y* para a matriz de saída. Os 3 *kernels* são executados de forma síncrona. Uma vez que *fft1*, *fft2* e *fft3* possuem diferentes padrões de acesso às matrizes, a memória se comporta de forma diferente em cada função. Durante o estudo foi notado que um fator definitivo para o desempenho dessas funções é o padrão de acesso escolhido para as matrizes. Foi alcançado melhor desempenho utilizando as seguintes formas. Na função *fft1* mantém-se o acesso à matriz y com o padrão $[z][x][y]$. Na função *fft2* mantém-se o acesso à matriz y com o padrão $[z][y][x]$. Na função *fft3* mantém-se o acesso à matriz y de forma vertical. Na função *checksum* o laço mais externo é paralelizado e cada *thread* executa uma iteração deste. O grau de paralelismo

da função é limitado pois possui número fixo de 1024 iterações. Ao final da função os valores computados pelas *threads* são combinados por bloco e após são escritos de forma atômica na memória global.

4. Resultados

Os experimentos das implementações deste trabalho e dos trabalhos relacionados foram realizados em uma máquina com dois processadores Intel Xeon E5-2620 e uma GPU Nvidia Titan X de 3072 CUDA Cores. Para coleta dos resultados, utilizou-se média aritmética simples de 40 execuções. O desvio padrão permaneceu próximo a 0 e o *speedup* foi calculado em relação ao código sequencial executado na CPU. Na Figura 1, apresenta-se a aceleração obtida pela abordagem deste trabalho e pelas implementações da literatura.

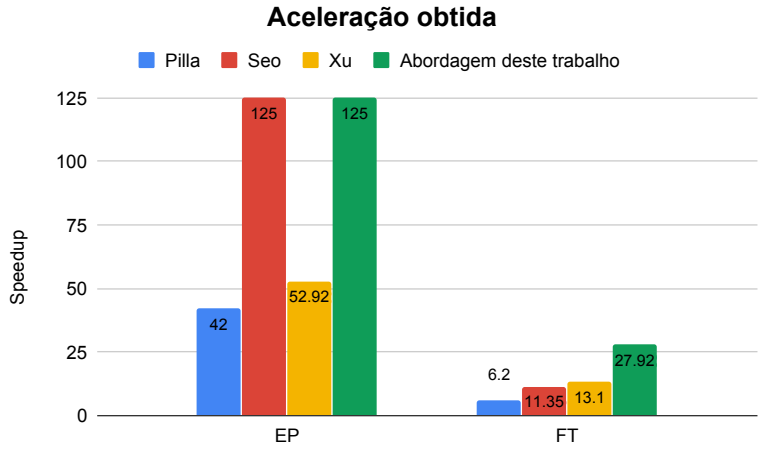

Figura 1. Resultados obtidos.

Em EP não houve diferença em relação ao trabalho com melhor desempenho [Seo et al. 2011], ou seja, mesmo com uma implementação mais recente não foi possível aumentar a aceleração do algoritmo. Em FT obteve-se mais que o dobro de desempenho em relação ao trabalho com melhor aceleração [Xu et al. 2015], o que revela que para ao menos uma aplicação do NPB uma implementação mais atual pode impactar significativamente o seu desempenho. Uma vez que a aplicação EP é mais simples, as otimizações deste trabalho não impactaram no desempenho, porém como FT é uma aplicação mais complexa e com várias regiões paralelas, o impacto tornou-se substancial. De forma geral a diferença de desempenho em FT se deve pelo padrão de acesso à memória nas funções *Fourier*, eliminação de divergências de *branch*, diminuição da granularidade e aumento do paralelismo bem como a própria estratégia de paralelização. Na região *setup*, as quatro matrizes globais são computadas de maneira concorrente por *threads* na CPU e *kernels* na GPU, algo não explorado nos trabalhos relacionados, além disso essas computações foram melhor atribuídas para a CPU e GPU. Em *Evolve* manteve-se a estratégia da literatura. Nas funções *Fourier* enquanto foi escolhido separar cada função em três *kernels* onde dois *kernels* efetuam apenas a leitura ou escrita de dados e um *kernel* efetua os cálculos, no trabalho com melhor desempenho [Xu et al. 2015], é criado apenas um *kernel*, aumenta-se a granularidade, diminui-se o paralelismo e uma série de leituras e escritas são realizadas por laços em cada uma das *threads* aumentando as divergências

de *branch*. Na função *checksum* a abordagem adotada apresentou vantagem por utilizar memória memória compartilhada e explorar maior paralelismo na redução dos valores.

5. Conclusões

Neste trabalho são apresentadas implementações atuais de EP e FT do NPB e os resultados são comparados com o estado-da-arte. Foi demonstrado que novas implementações podem até mais que dobrar o desempenho de pelo menos uma das aplicações do NPB em relação à literatura. Como trabalhos futuros, pretende-se implementar os demais algoritmos do NPB.

Referências

[Bailey et al. 1994] Bailey, D. H., Barszcz, E., Barton, J. T., Browning, D. S., Carter, R. L., Fatoohi, R. A., Frederickson, P. O., Lasinski, T. A., Simon, H. D., Venkatakrishnan, V., and Weeratunga, S. K. (1994). The nas parallel benchmarks rnr-94-007. Technical report, NASA Advanced Supercomputing Division.

[Dümmler and Rünger 2013] Dümmler, J. and Rünger, G. (2013). Execution schemes for the npb-mz benchmarks on hybrid architectures: A comparative study. In *Parallel Computing: Accelerating Computational Science and Engineering (CSE)*, volume 25.

[Gong et al. 2010] Gong, C., Liu, J., Qin, J., Hu, Q., and Gong, Z. (2010). Efficient embarrassingly parallel on graphics processor unit. In *2010 2nd International Conference on Education Technology and Computer*, volume 4, pages V4–400–V4–404.

[Griebler et al. 2018] Griebler, D., Loff, J., Mencagli, G., Danelutto, M., and Fernandes, L. G. (2018). Efficient NAS Benchmark Kernels with C++ Parallel Programming. In *26th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*, PDP'18, pages 733–740, Cambridge, UK. IEEE.

[Jin et al. 2012] Jin, H., Kellogg, M., and Mehrotra, P. (2012). Using compiler directives for accelerating cfd applications on gpus. In *OpenMP in a Heterogeneous World*, volume 7312, pages 154–168.

[Malik et al. 2012] Malik, M., Li, T., Sharif, U., Shahid, R., El-Ghazawi, T., and Newby, G. (2012). Productivity of gpus under different programming paradigms. *Concurr. Comput. : Pract. Exper.*, 24(2):179–191.

[Pilla 2009] Pilla, L. L. (2009). *Análise de Desempenho da Arquitetura CUDA Utilizando os NAS Parallel Benchmarks*. Universidade Federal do Rio Grande do Sul, UFRGS.

[Seo et al. 2011] Seo, S., Jo, G., and Lee, J. (2011). Performance characterization of the nas parallel benchmarks in opencl. In *2011 IEEE International Symposium on Workload Characterization (IISWC)*, pages 137–148.

[Stone and Elton 2015] Stone, C. P. and Elton, B. H. (2015). Accelerating the multi-zone scalar pentadiagonal cfd algorithm with openacc. In *Proceedings of the Second Workshop on Accelerator Programming Using Directives*, WACCPD '15, pages 2:1–2:7, New York, NY, USA. ACM.

[Xu et al. 2015] Xu, R., Tian, X., Chandrasekaran, S., Yan, Y., and Chapman, B. (2015). Nas parallel benchmarks for gpgpus using a directive-based programming model. In Brodman, J. and Tu, P., editors, *Languages and Compilers for Parallel Computing*, pages 67–81, Cham. Springer International Publishing.