

Implementação CUDA dos Kernels NPB

Gabriell Alves de Araujo¹, Dalvan Griebler^{1,2}, Luiz G. Fernandes¹

¹ Escola Politécnica, Grupo de Modelagem de Aplicações Paralelas (GMAP), Pontifícia Universidade Católica do Rio Grande do Sul (PUCRS), Porto Alegre, Brasil.

²Laboratório 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. *NAS Parallel Benchmarks (NPB) é um conjunto de benchmarks utilizado para avaliar hardware e software, que ao longo dos anos foi portado para diferentes frameworks. Concernente a GPUs, atualmente existem apenas versões OpenCL e OpenACC. Este trabalho contribui com a literatura provendo a primeira implementação CUDA completa dos kernels do NPB, realizando experimentos com carga de trabalho inédita e revelando novos fatos sobre o NPB.*

1. Introdução

Unidades de Processamento Gráfico (GPUs) têm sido utilizadas como alternativa para computação de alto desempenho, pois oferecem capacidade massiva de paralelismo a baixo custo. No entanto, o paralelismo das GPUs é um desafio para os desenvolvedores [Xu et al. 2015]. *Benchmarks* têm assumido papel relevante no desenvolvimento, contribuindo para a avaliação de arquiteturas e técnicas de programação [Seo et al. 2011]. NAS Parallel Benchmarks (NPB) [Bailey et al. 1994] é um conjunto de *benchmarks* baseado em dinâmica de fluidos que possui pertinência científica e têm se destacado em pesquisas de GPUs nos últimos anos.

No entanto, ao observar a literatura, é possível perceber algumas lacunas em aberto. Os principais trabalhos são relativamente antigos, não foram realizados experimentos com cargas de trabalho grandes, baixa expectativa de aceleração foi relatada, e também não existe disponível uma versão CUDA dos *kernels* do NPB. O que é relatado pela literatura como impedimento para o avanço de pesquisas [Tian et al. 2016].

As contribuições deste trabalho visam preencher estas lacunas identificadas. Assim, o trabalho apresenta a primeira versão CUDA completa dos *kernels* do NPB. Realiza experimentos com a carga de trabalho Classe C, não explorada nos trabalhos anteriores por limitações das abordagens e arquiteturas. Compara a versão CUDA deste trabalho com as versões OpenCL [Seo et al. 2011] e OpenACC [Xu et al. 2015] do estado-da-arte. Avalia desempenho, consumo de memória e quantidade de linhas. Novos fatos são revelados sobre o NPB, alcançando melhores resultados que os registros da literatura, atualizando as expectativas de aceleração através de nova implementação e experimentos.

2. Implementação

A implementação paralela foi realizada a partir da versão C++ do NPB [Griebler et al. 2018]. Foram adotados princípios de desenvolvimento tais como, implementar os *kernels* de GPU da forma mais simples possível, evitando fluxos complexos e divergências de *branches*, coalescer acessos na memória, utilizar a hierarquia de memória evitando acessos na memória global e implementar paralelismo de GPU apenas em trechos que oferecem boa *performance* na GPU, observando *overheads* e custos de transferência de memória.

CG realiza aproximações de valores e foi desenvolvido com o objetivo de avaliar computações irregulares [Bailey et al. 1994]. A computação mais intensiva de CG consiste na multiplicação de submatrizes não estruturadas, uma lista de tarefas onde cada tarefa acessa diferentes locais da memória e realiza quantidades diferentes de computações. A maneira mais adequada de abordar este problema é isolar as computações irregulares. Para isto, cria-se um grupo de *threads* para cada uma das tarefas. As computações de cada tarefa são divididas entre as *threads* do grupo, e a execução é então realizada de forma regular em cada grupo de *threads*.

EP visa estimar a capacidade de ponto-flutuante da arquitetura alvo e por isso possui poucas dependências [Bailey et al. 1994]. EP possui um laço global onde cada iteração gera um conjunto de números aleatórios e após calcula desvios Gaussianos. Para aplicar o paralelismo, cada iteração é atribuída a um grupo de *threads* e cada *thread* do grupo calcula um subconjunto dos números e desvios. As *threads* reutilizam a memória até calcularem todos os resultados. Isto é necessário pelo motivo de que os subconjuntos consomem muita memória da GPU.

FT resolve equações diferenciais de números complexos [Bailey et al. 1994]. As computações mais intensivas são as funções que computam Transformação Rápida de Fourier [Bailey et al. 1994] nas dimensões x , y e z dos vetores. Estas funções possuem muitos laços aninhados, dependências de dados, divergências de *branches* e baixo grau de paralelismo. Para implementar o *benchmark* de forma a extrair bom desempenho na GPU, dividi-se cada uma destas funções em três estágios, eliminando dependências de dados, divergências de *branches* e aumentando o grau de paralelismo. O primeiro estágio realiza cópia de dados, o segundo estágio realiza o processamento dos dados e o terceiro estágio escreve os dados na saída. Para o cálculo da função no eixo x , adicionalmente é necessário transformar o padrão de acesso para coalescer a memória.

IS estima a velocidade e comunicação de números inteiros através de computações de ordenação [Bailey et al. 1994]. Diferente dos outros *benchmarks*, IS é constituído por sequências simples de laços, e o mapeamento para as *threads* de GPU se resume em criar uma *thread* para cada iteração. No entanto, as computações são pouco intensivas, o grau de paralelismo é baixo, e são necessárias sincronizações entre as sequências de laços, características que impactam de forma negativa o desempenho das GPUs.

MG é uma implementação *multi-grid* simplificada [Bailey et al. 1994]. MG possui quatro funções de computação intensiva, `interp`, `rprj3`, `psinv` e `resid`. As funções `interp` e `rprj3`, possuem laços aninhados, onde a computação interna destes laços trata-se de uma sequência de laços. Para aplicar o paralelismo na GPU, os laços externos são combinados em único laço, onde cada iteração é associada a um grupo de *threads*. Junta-se a sequência interna de laços em um único laço, e as iterações deste são divididas entre as *threads* do grupo. As funções `psinv` e `resid` possuem organização similar a `interp` e `rprj3`, por esse motivo são paralelizados da mesma forma, porém, os laços internos não podem ser combinados em um único laço. Portanto, os grupos de *threads* precisam executar sequências de laços em vez de um único laço.

3. Experimentos

Os experimentos foram realizados em uma máquina equipada com um processador Intel Xeon E5-2620 2.0 GHz, 16 Gigabytes de RAM, e uma GPU NVIDIA Titan X Pascal com 3584 *CUDA Cores* e 12 Gigabytes de memória dedicada. O sistema operacional utilizado foi o *Ubuntu* 14.04 LTS. Os *softwares* utilizados foram *CUDA* 10, *GCC*-9,

OpenCL 1.1, e OpenACC 2.5. Utilizou-se a *flag* de compilação *-O3* em todos os *benchmarks*. Cada experimento foi repetido 10 vezes para a computação das métricas. Como carga de trabalho foi utilizada a Classe C do NPB. Os tempos de execução obtiveram desvio padrão irrelevante e são apresentados na Figura 1(a). O consumo de memória da GPU é apresentado na Figura 1(b). A quantidade de linhas de código necessária em cada implementação é apresentada na Figura 2 e na Tabela 1. A implementação CUDA deste trabalho é apresentada como NPB-CUDA, os trabalhos do estado-arte com OpenCL e OpenACC são apresentados respectivamente como NPB-OpenCL [Seo et al. 2011] e NPB-OpenACC [Xu et al. 2015]. NPB-OpenCL e NPB-OpenACC precisaram da *flag* de compilação *-mmodel=large* e do comando *ulimit -s unlimited* para permitir mais memória na *stack*, pois implementam consumo de memória de forma ineficiente, alocando grande vetores multi-dimensionais nesta região de memória.

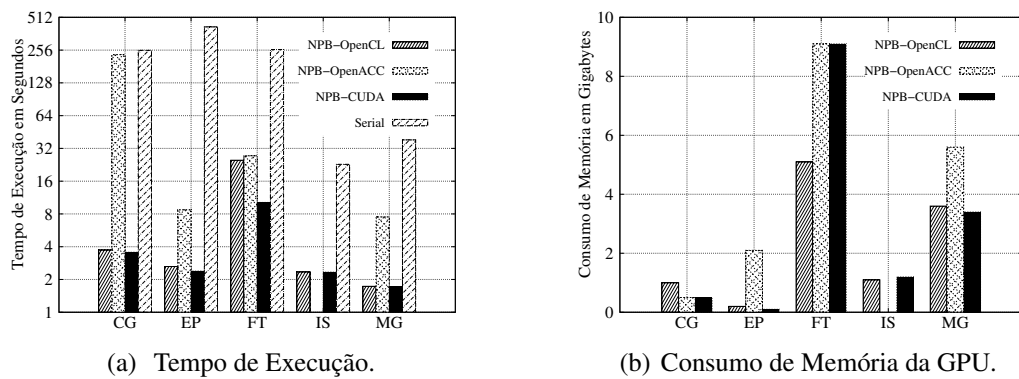


Figura 1. Resultados dos Experimentos com a Classe C.

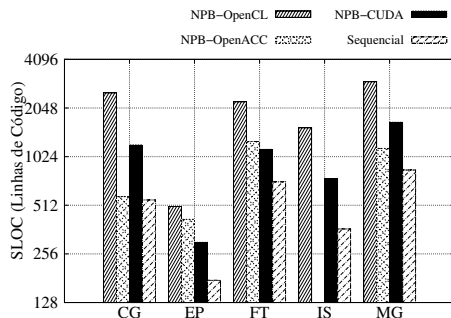


Figura 2. Total em linhas de código.

Tabela 1. Aumento de linhas de código em relação ao código sequencial.

Kernel	NPB-OpenCL	NPB-OpenACC	NPB-CUDA
CG	360,87%	4,89%	119,93%
EP	185,80%	138,07%	71,59%
FT	213,15%	77,62%	58,88%
IS	323,56%	–	105,21%
MG	252,13%	36,05%	97,75%

Em CG, NPB-CUDA obteve 71.5 de *speedup*, desempenho obtido através do isolamento das computações irregulares. O desempenho foi 4.48% melhor que NPB-OpenCL por explorar maior paralelismo através de *kernels* de GPU concorrentes. NPB-OpenACC apresentou baixo desempenho por não isolar as computações irregulares. Em EP, NPB-CUDA alcançou 171.5 de *speedup*, por meio do paralelismo de grão fino implementado. O resultado foi 9.98% melhor que NPB-OpenCL. NPB-OpenCL possui mais conflitos de memória e misses, recebendo impacto na aceleração. NPB-OpenACC teve menor desempenho por utilizar estratégia de grão grosso. Em FT, NPB-CUDA atingiu 25.3 de *speedup*, 143.26% melhor que NPB-OpenCL. Os trabalhos NPB-OpenCL e NPB-OpenACC não eliminaram dependências de dados e divergências de *branches*, e mantiveram baixo grau de paralelismo. Por isso, o desempenho foi significativamente inferior. Em IS, não foi

possível realizar experimentos com NPB-OpenACC, pois os autores não disponibilizaram a implementação. O *speedup* de NPB-CUDA foi 9.8, muito próximo a NPB-OpenCL. Em MG, NPB-CUDA também apresentou desempenho próximo a NPB-OpenCL, decorrência da similaridade entre as abordagens. NPB-OpenACC atingiu menor desempenho pelo motivo de que a implementação possui *overhead* de sincronizações entre GPU e CPU.

O desempenho dos *benchmarks* varia significativamente. EP roda exclusivamente na GPU. Os demais *benchmarks* precisam de sincronizações entre CPU e GPU. FT e MG possuem divergências de caminhos que não podem ser removidas e possuem diferentes padrões de acessos. IS possui baixo grau de paralelismo. O consumo de memória das implementações NPB-CUDA foi próximo aos melhores resultados entre NPB-OpenCL e NPB-OpenACC, com exceção de FT, onde o trabalho NPB-OpenCL implementa reutilização de memória. Quanto a quantidade necessária de linhas de código para aplicar o paralelismo, NPB-OpenCL apresenta número superior a NPB-CUDA. NPB-OpenACC possui mais linhas de código que NPB-CUDA apenas em EP e FT.

4. Conclusão

Este trabalho apresentou a primeira versão CUDA completa dos *kernels* do NPB e foi comparado aos trabalhos OpenCL e OpenACC do estado-da-arte. Os resultados demonstraram aceleração considerável nos *benchmarks*. Adicionalmente, a implementação deste trabalho obteve melhor desempenho em três aplicações. As investigações deste trabalho contribuem para o domínio de dinâmica de fluidos, avaliando o comportamento em GPUs. Como trabalhos futuros, pretende-se implementar versão multi-GPU dos *kernels* do NPB, paralelizar as pseudo-aplicações e também utilizar outros *frameworks* de GPUs.

Referências

- 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., Venkatakrisnan, V., and Weeratunga, S. K. (1994). The NAS Parallel Benchmarks RNR-94-007. Technical report, NASA Advanced Supercomputing Division.
- 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.
- 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.
- Tian, X., Xu, R., Yan, Y., Chandrasekaran, S., Eachempati, D., and Chapman, B. (2016). Compiler Transformation of Nested Loops for General Purpose GPUs. *Concurrency and Computation: Practice and Experience*, 28(2):537–556.
- 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.