Otimização de desempenho de software para análise filogenética (ExaML) utilizando a arquitetura CUDA

Marcelo Gomes Martins[1], **Timoteo Alberto Peters Lange**[1]

[1]Instituto Federal de Educação, Ciência e Tecnologia do Rio Grande do Sul
Rua Santos Dumont, 2127 – Bairro Albatroz – 95.520-000 – Osório – RS – Brasil

marcelogm94@hotmail.com, timoteo.lange@osorio.ifrs.edu.br

Resumo. *O aumento das bases genéticas de dados e da complexidade das análises filogenéticas requer maior poder computacional: este artigo apresenta o processo de reimplementação das principais funções da aplicação ExaML para a arquitetura CUDA, demonstrando aspectos relevantes no processo de otimização de uma aplicação de alto desempenho para GPU e apresentando bons resultados na manipulação de grandes entradas de dados.*

1. Introdução

O processo de análise filogenética baseia-se na construção de árvores filogenéticas que, por definição, são diagramas utilizados para ilustrar de forma elegante quais organismos possuem uma estrutura genética semelhante [Vandamme 2009]. Essa estrutura é utilizada para representar um possível histórico evolutivo dos organismos envolvidos em uma análise.

Em 2004, Alexandros Stamatakis apresentou a implementação de um algoritmo para pesquisa de árvores filogenéticas utilizando o método estatístico de máxima verossimilhança: *Randomized Axecelerated Maximum Likelihood* (RAxML) [Stamatakis 2004], programa desenvolvido em linguagem C para operar em ambiente paralelo e distribuído. RAxML teve seu desenvolvimento em um período anterior à popularização[1] do uso das *Graphics Processing Units* (GPUs) como unidades de processamento auxiliares e do surgimento das plataformas CUDA e OpenCL.

Posteriormente, foram apresentados trabalhos relacionados ao RAxML, como a *Phylogenetic Likelihood Library* [Izquierdo-Carrasco et al. 2013] e o ExaML [Kozlov et al. 2015], algoritmo desenvolvido para operar em supercomputadores de forma paralela e distribuída. O objetivo deste artigo é apresentar os principais aspectos envolvidos no processo de paralelização da aplicação ExaML para a arquitetura CUDA, acompanhados dos resultados obtidos.

2. Configuração de ambiente

Para compor o ambiente de desenvolvimento foi utilizado o processador Intel Core I7-7700K (4,5 GHz), 8 GB DDR4, placa gráfica Nvidia GeForce GTX 1060 (1.280 CUDA *cores*), sistema operacional Ubuntu 16.04.3 LTS e CUDA *Toolkit* 9.0.176.

Como teste de validação do modelo, foi utilizado o arquivo **49**[2] (*patterns* de DNA de 48 *taxa* variados), possuindo 628 *patterns*. Para os testes de desempenho da aplicação,

[1]Ainda que em 2005 já existam trabalhos utilizando BrookGPU na paralelização de funções da aplicação [Charalambous et al. 2005], não há uma versão oficial para processamento em GPU.

[2]Disponível no repositório do ExaML no Github: `https://github.com/stamatak/ExaML`.

utilizou-se a ferramenta INDELible [Fletcher and Yang 2009] para gerar um arquivo de alinhamento de DNA. O arquivo foi dividido em *taxa* distintos com a ajuda de um *script* desenvolvido em *Python* 3.

3. Desenvolvimento

Utilizando *Fixed-Position Logging Profiling* [Hegner 1999], comandos para medição de tempo foram distribuídos no inicio e no final das principais funções da aplicação. Após a análise das medições, conclui-se que a função **newview** representa cerca de 58% do tempo total de execução; sendo assim, essa função foi escolhida como escopo inicial do trabalho.

3.1. Implementação inicial

A função **newview** consiste em um *switch statement* com três caminhos lógicos (TIP_TIP, TIP_INNER e INNER_INNER), obteve-se então cinco *kernels*, dois dedicados a pré-computação dos dados para TIP_TIP e TIP_INNER e três para computação de cada caso. Originalmente os *kernels* foram implementada utilizando *buffers* na memória global da GPU, o que representa que, para a cada execução de uma função *kernel*, é necessário transferir matrizes entre o *host* e o *device*. Isto obviamente torna a aplicação mais lenta, uma vez que o desempenho da GPU fica condicionado a velocidade de transferências de dados realizada pelo barramento PCI-E [Cook 2012]; todavia, essa implementação inicial foi suficiente para validar, logo no início, os tempos de execução dos *kernels* (considerando a eliminação dessa transferência de memória).

3.2. Otimizações

A maneira usual de diminuir a movimentação de memória entre *device* e *host* é mantendo as variáveis que serão acessadas por determinado *kernel* na memória global do próprio *device* [Nvidia 2012]. Subentende-se que todos os cálculos que serão realizados sobre determinadas variáveis devem ser realizados na GPU, ocasionando a necessidade de implementação outras sub-rotinas da aplicação.

O transporte dos vetores para a memória da GPU foi possível em praticamente todos os casos, com exceção dos vetores recalculados a cada execução de **newview**. Dessa forma, amplia-se o número de *kernels* indispensáveis para tornar a utilização da arquitetura CUDA viável na aplicação, sendo agora necessária a reimplementarão das funções **core**, **evaluate** e **sum**, refletindo no desenvolvimento de seis novos *kernels*.

No desenvolvimento das funções, foram utilizados comandos de *loop unrolling* (permitindo a eliminação instruções de controle dos laços de repetição e suas penalidades naturais[3] [Davidson and Jinturkar 1996]) e redução paralela [Harris 2007] na soma de valores dos vetores.

3.3. Novo layout

Após o desenvolvimento inicial, um novo *layout* de *kernel* foi concebido a partir da ideia de redução paralela e da utilização da memória compartilhada: o objetivo do novo *layout* é

[3]As penalidades de desvios condicionais ocorrem em arquiteturas que utilizam *pileline*. Em uma *pipeline* com *prefetching* sequencial, o processador descarta instruções antecipadas quando deparado com um desvio condicional, esse descarte reduz consideravelmente o desempenho do processamento [Lalja 1988].

transferir a responsabilidade de indexação dos vetores para uma estrutura de *thread blocks* que facilite seu acesso e permita que múltiplas *threads* trabalhem de forma cooperativa utilizando a memória compartilhada. Nesse novo *layout* cada *thread* trabalha com uma pequena fração do cálculo, refletindo em um aumento considerável de utilização da placa gráfica.

4. Resultados

Em um primeiro momento, foram executados testes empregando o arquivo de entrada **49** (Seção 2), com o objetivo de comprovar a coerência das arvores filogenéticas geradas pelas novas implementações a partir de dados biológicos empíricos. Após validarmos os resultados, as aplicações foram submetidas para coleta dos tempos de execução.

Tabela 1. Tempos de execução (em segundos) para até 10.000 padrões.

	Número de padrões							
Versão da aplicação	**1.000**	**2.000**	**3.000**	**4.000**	**5.000**	**6.000**	**8.000**	**10.000**
SSE3	6,38	11,01	23,94	-	-	-	-	-
AVX	2,07	3,56	7,61	12,11	8,98	16,81	11,70	24,14
OMP-AVX	1,15	1,81	3,74	5,83	4,29	7,90	6,49	11,43
OLD-CUDA	5,15	5,38	8,98	11,77	8,25	14,64	9,27	17,76
NEW-CUDA	5,00	4,83	7,65	10,04	7,54	13,04	7,92	15,45

Em uma primeira análise da Tabela 1, os resultados parecem desanimadores, uma vez que os tempos de execução das aplicações utilizando CUDA só conseguem ser melhores que os tempos da versão SSE3; todavia, ao analisamos considerando os tempos de execução para conjuntos de dados maiores, é possível perceber duas tendências: (1) a redução da diferença entre a versão mais rápida (OMP-AVX) e as implementações em CUDA e (2) o distanciamento dos tempos de execução dos dois *layouts* desenvolvidos. Também é possível perceber que as implementações em CUDA, a partir de 4.000 *patterns*, já superam os tempos de execução da versão AVX.

Tabela 2. Tempos de execução (em segundos) para até 100.000 padrões.

	Número de padrões					
Versão da aplicação	**10.000**	**20.000**	**40.000**	**60.000**	**80.000**	**100.000**
SSE3	74,90	140,33	356,21	616,57	-	-
AVX	24,14	48,39	129,57	231,26	339,49	482,00
OMP-AVX	11,43	25,12	80,37	154,02	222,32	322,38
OLD-CUDA	17,76	34,87	79,30	130,92	179,77	250,17
NEW-CUDA	15,45	25,89	56,74	92,35	125,15	171,27

Quando submetemos conjuntos de dados maiores, entre 10.000 e 100.000 *patterns* (Tabela 2), as duas implementações desenvolvidas nesse trabalho ganham destaque: com 40.000 padrões, superam o tempo de execução da versão OMP-AVX. Com 100.000 padrões, CUDA-NEW obtém um ganho de desempenho de 181% sobre a versão AVX. Em nenhum dos testes NEW-CUDA obteve desempenho inferior à OLD-CUDA.

5. Conclusão

As implementações desenvolvidas em CUDA apresentam desempenho inferior quando submetidas a sequências de entrada menores, porém apresentam um ganho de desempenho considerável quando submetidas a um número maior de *patterns*, atingindo tempos

de processamento que podem chegar a duas ou três vezes o tempo de execução em CPU. Em análises usuais, o número de *patterns* utilizados por partição não costuma passar de 2.000, entretanto existem casos específicos onde são utilizadas partições com um número maior de padrões. É possível concluir que, no processamento com GPU, trabalhar com uma grande quantidade de dados não é somente desejável, mas necessário. Atualmente o processamento por inferência apresenta melhores resultados em processadores x86[4], entretanto há uma potencial obtenção de desempenho a partir do desenvolvimento de uma aplicação heterogênea, capaz de aproveitar o poder de processamento das GPUs atuais. Os códigos e utilizados neste artigo estão disponíveis no repositório do Github[5].

Referências

Charalambous, M., Trancoso, P., and Stamatakis, A. (2005). Initial experiences porting a bioinformatics application to a graphics processor. *Advances in Informatics*, pages 415–425.

Cook, S. (2012). *CUDA programming: a developer's guide to parallel computing with GPUs*. Newnes.

Davidson, J. W. and Jinturkar, S. (1996). Aggressive loop unrolling in a retargetable, optimizing compiler. In *International Conference on Compiler Construction*, pages 59–73. Springer.

Fletcher, W. and Yang, Z. (2009). Indelible: a flexible simulator of biological sequence evolution. *Molecular biology and evolution*, 26(8):1879–1888.

Harris, M. (2007). Optimizing cuda. *SC07: High Performance Computing With CUDA*.

Hegner, S. J. (1999). Analysis of algorithm fall 1999: Profiling of algorithms. Datavetenskap - Umeå universitet. Acesso em: 13 de outubro de 2017.

Izquierdo-Carrasco, F., Alachiotis, N., Berger, S., Flouri, T., Pissis, S. P., and Stamatakis, A. (2013). A generic vectorization scheme and a gpu kernel for the phylogenetic likelihood library. In *Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2013 IEEE 27th International*, pages 530–538. IEEE.

Kozlov, A. (2017). amkozlov/raxml-ng: Raxml-ng v0.2.0 beta.

Kozlov, A. M., Aberer, A. J., and Stamatakis, A. (2015). Examl version 3: a tool for phylogenomic analyses on supercomputers. *Bioinformatics*, page btv184.

Lalja, D. J. (1988). Reducing the branch penalty in pipelined processors. *Computer*, 21(7):47–55.

Nvidia (2012). Cuda programming guide.

Stamatakis, A. (2004). *Distributed and parallel algorithms and systems for inference of huge phylogenetic trees based on the maximum likelihood method.* PhD thesis, Technical University Munich.

Vandamme, A. (2009). *The phylogenetic handbook*. Cambridge, UK: Cambridge University Press.

[4]A aplicação RAxML *Next Generation* [Kozlov 2017] representa o estado da arte em relação a inferência por máxima verossimilhança em processadores x86.

[5]Repositório no Github: `https://github.com/marcelogm/CUDAExaML`