Otimização de Desempenho em Código Interpretador de Programação Genética

Lucas Bicalho Oliveira[1], **Álvaro Luiz Fazenda**[1], **Vinícius Veloso de Melo**[1]

[1]Instituto de Ciência e Tecnologia – Universidade Federal de São Paulo (UNIFESP)
São José dos Campos, SP – Brasil

{lucas.bicalho,alvaro.fazenda,vinicius.melo}@unifesp.br

Abstract. *Genetic Programming is an evolutionary algorithm technique that often presents many instructions and steps that can be performed concurrently. Thus, parallel and high performance programming techniques can be employed to optimize the computational performance of the developed algorithm. This article demonstrates the performance improvement obtained for a code that interprets Genetic Programming algorithms, through the use of programming that adheres to OpenMP and OpenACC standards. As a result, it was possible to observe that the GPU executions showed a significant performance improvement when used with inputs considered large for the problem. It should also be noted that the version developed in OpenACC allows portability of the code, allowing efficient execution of the multithreaded form in CPU, as well as in GPU, representing an option with effective portability.*

Resumo. *A Programação Genética é uma técnica de algoritmo evolutivo que frequentemente apresenta muitas instruções e etapas que podem ser executadas concorrentemente. Assim, técnicas de programação paralela e de alto desempenho podem ser empregadas para otimização do desempenho computacional do algoritmo desenvolvido. Este artigo demonstra a melhoria de desempenho obtida para um código interpretador de algoritmos de Programação Genética, através do uso de programação aderente aos padrões OpenMP e OpenACC. Como resultado, foi possível observar que as execuções em GPU mostraram significativa melhoria de desempenho quando utilizadas com entradas consideradas grandes para o problema. Cabe destacar ainda que a versão desenvolvida em OpenACC permite a portabilidade do código entre diferentes plataformas, permitindo execução eficiente da forma multithread em CPU, tanto quanto em GPU, representando uma opção com portabilidade efetiva.*

1. Introdução

Após o surgimento de computadores pessoais com mais de um núcleo de processamento, oriundo da saturação do desempenho das máquinas *singlecore*, foi exigido do programador uma mudança de paradigma na programação destes dispositivos, de maneira a aproveitar eficientemente os recursos disponíveis. Assim, a programação concorrente/paralela tornou-se necessária para melhoria de desempenho de programas que apresentem grande demanda computacional. A concorrência em programação em nível de procedimentos, pode ser gerada utilizando-se de múltiplas *threads* ou múltiplos processos, dependendo da arquitetura do sistema a ser utilizado.

Muitas bibliotecas ou linguagens são projetadas e desenvolvidas para melhorar a produtividade de programação. Construções paralelas, como, por exemplo, laços paralelos do OpenMP [Chandra 2001], permitem identificar conjuntos de instruções potencialmente concorrentes, sendo comumente utilizada em aplicações numéricas.

A Programação Genética [Koza 1994] é uma técnica de algoritmo evolutivo que permite a melhoria de programas de computador para executar uma programação automática. Nesta técnica, representa-se indivíduos como expressões, as quais permitem avaliar funções envolvendo variáveis, constantes ou ainda outras funções, as quais são recursivamente calculadas para obter um melhor valor final para o indivíduo. Essa etapa geralmente é que mais demanda poder de computação do algoritmo. Por exemplo, para tarefas de classificação ou regressão, um programa genético (PG) precisa, normalmente, utilizar um algoritmo interpretador para calcular o resultado de cada indivíduo, para cada amostra no conjunto de dados utilizado.

Neste trabalho, procurou-se otimizar a versão de código original proposta por Melo et al [de Melo et al. 2020], por meio de programação aderente aos padrões OpenMP e OpenACC, e avaliar o desempenho, tanto em CPU quanto em GPU, da versão com múltiplas instruções do mesmo autor, mostrando os parâmetros mais sensíveis à implementação efetuada, suas limitações e possíveis melhorias futuras.

2. Referencial Teórico

Vários trabalhos na literatura tratam de investigar diferentes abordagens para otimizar o desempenho no processo de avaliação das expressões em PGs. Uma técnica em particular foi desenvolvida para uso em modernas CPUs, e busca explorar a vetorização de instruções únicas, normalmente disponível nas CPUs que permitem o uso de instruções SSE (*Streaming SIMD Extensions*) [Chitty 2012] ou AVXintrinsics (*Advanced Vector eXtensions*). Desta forma, simples instruções, tal como adições, podem ser aplicadas a conjuntos (*arrays*) de dados de forma concorrente, normalmente acelerando desempenho.

O artigo de Melo et al [de Melo et al. 2020] demonstrou uma evolução na técnica citada propondo o uso de múltiplas instruções no interpretador do PG, permitindo realizar até quatro instruções matemáticas diferentes simultaneamente. O trabalho desenvolvido avalia o desempenho em um única geração do procedimento usual de um PG (não incluindo as etapas de seleção, cruzamento e mutação), uma vez que o foco do trabalho reside na investigação de desempenho do interpretador desenvolvido com múltiplas instruções, o qual avalia indivíduos aleatoriamente gerados.

3. Resultados e Discussões

Nos experimentos, usou-se dados sintéticos para avaliação de desempenho paralelo do interpretador. Todos os indivíduos de cada *dataset* apresentam 10 variáveis aleatoriamente distribuídas no intervalo $[-1.0, 1.0]$. O código possui três parâmetros de entradas. O primeiro parâmetro (*size*) corresponde a quantidade de registros no *dataset*, sendo definidos os valores: 100, 5k, 10k, 100k, 1M; o segundo (*evals*) corresponde ao tamanho da população (quantidade de indivíduos), onde cada indivíduo representa uma transformação a ser aplicada ao *dataset*, sendo definidos: 100, 1k, 5k, 10k; e o terceiro (*depth*) corresponde à profundidade máxima da árvore utilizada, onde cada nó da árvore representa uma operação aritmética realizada por cada indivíduo. O valor de *depth* foi fixado em 20

níveis, por ser um valor com tempo razoável de execução em combinação com os valores das demais entradas.

Os equipamentos utilizados para os testes possuem a seguinte configuração: servidor dual Intel Xeon E5-2660v4@2.00GHz, com 14 núcleos de processamento por processador, totalizando 28 núcleos por nó, com 128GB de memória principal. A GPU utilizada possui as seguintes características: NVIDIA TITAN Xp com 3840 CUDA Cores a 1.58 GHz, com 12GB de memória.

Foram desenvolvidas duas versões paralelas. A primeira versão segue o padrão OpenMP para execuções na CPU citada no parágrafo acima, com múltiplas *threads*. A segunda versão é aderente ao padrão OpenACC, a qual foi testada, sem modificações, a duas plataformas distintas, no mesmo servidor para CPU com múltiplas *threads* e na GPU citada, permitindo avaliar a portabilidade efetiva. Em ambas as versões, os alvos de otimizações foram laços que poderiam operar concorrentemente.

A versão do código com OpenACC executada em GPU obteve um tempo de execução inferior em relação ao tempo de execução das versões que foram executadas em CPU com 28 *threads* apenas para a combinação de entrada de maior tamanho. A Figura 1 mostra uma comparação entre os valores de *speedup* obtidos para cada combinação de valores de entrada, utilizando a quantidade máxima de núcleos existentes no servidor. A versão de referência para o cálculo dos *speedups* foi a versão serial original.

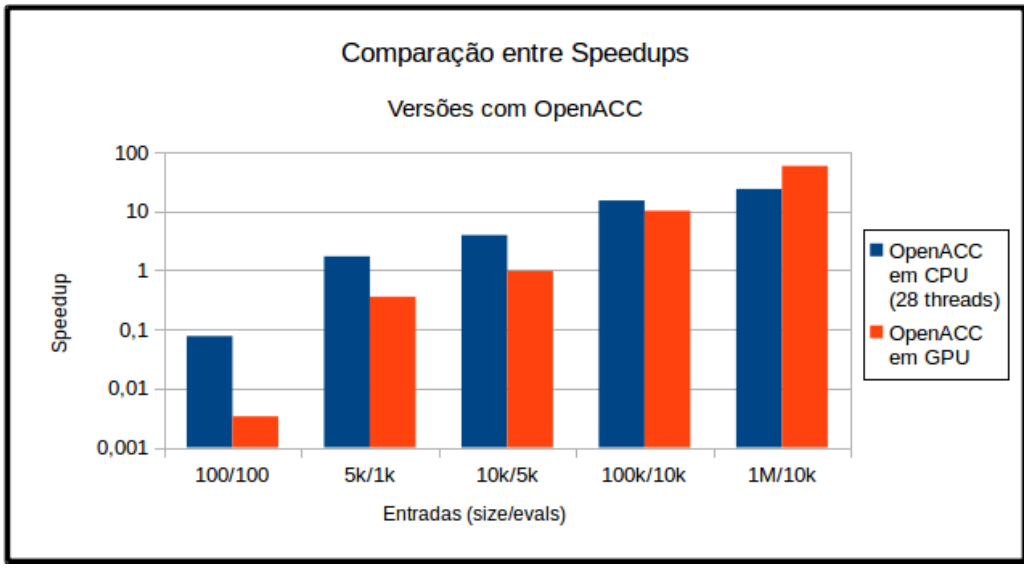

Figura 1. Comparação entre Speedups para as Versões com OpenACC.

Com o objetivo de se comparar o desempenho das versões com o dado de entrada de maior tamanho, as duas versões foram executadas em CPUs *multithread* e a versão com OpenACC também foi executada em GPU. Os dados de entrada foram fixados em *size* com o valor de 1000000 (1M) e *evals* com o valor de 10k. O resultado pode ser observado na Figura 2, em que o eixo relativo ao tempo de execução está em escala logarítmica para melhor visualização. Para o tempo de execução em GPU, não é considerada nenhuma variação na quantidade de *threads*, utilizando-se a totalidade dos recursos do dispositivo.

É possível observar que a versão com OpenACC em GPU obteve um desempenho

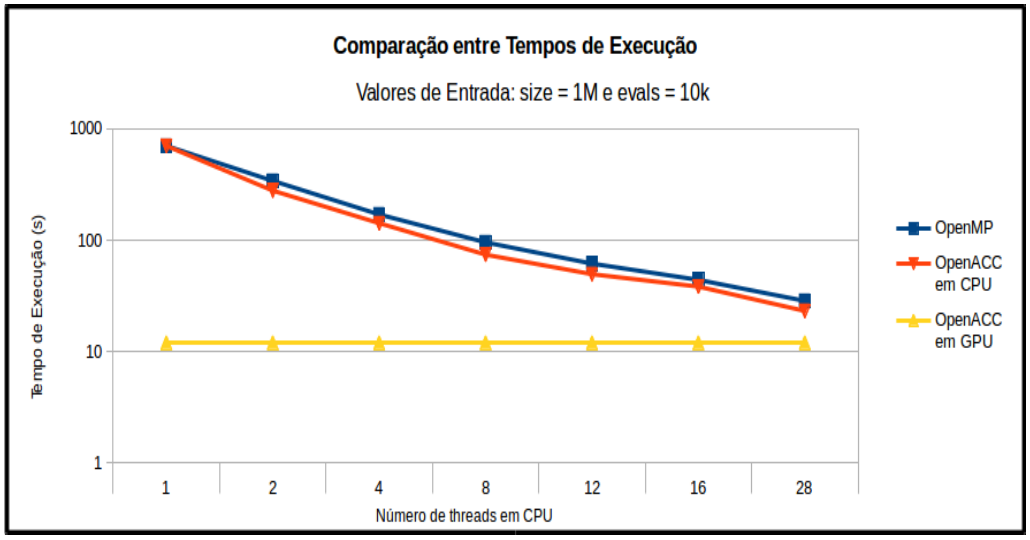

Figura 2. Comparação entre os tempos de execução de todas as versões testadas, para os valores de entrada 1M (*size*) e 10k (*evals*).

superior em relação às outras versões, alcançando um tempo mais de 50 vezes menor que o tempo de execução da versão serial. Novamente, as versões OpenMP e OpenACC executadas em CPU obtiveram desempenho semelhante, com pouca superioridade da versão OpenACC. Essa combinação de dados de entrada foi a única em que a versão com OpenACC em GPU superou as demais em CPU para o número máximo de *threads*.

4. Conclusão

Tendo em vista os comportamentos discutidos acima para o código, pode-se afirmar que as versões paralelizadas, de modo geral, são eficientes apenas para combinações de valores de entrada suficientemente grandes, devido à ação da Lei de Amdahl. Dentre as duas versões testadas, foi observado que seus comportamentos foram muito similares para a maioria dos casos, ao serem executadas em CPU, com a versão OpenACC em GPU apresentando o melhor desempenho.

Como resultado, a versão do código paralelizada com OpenACC possui mais vantagens do que a versão com OpenMP, pois, além de ter gerado resultados melhores para entradas de tamanhos grandes, também é plausível de ser executada em CPU e em GPU.

Referências

Chandra, R. (2001). *Parallel Programming in OpenMP*. Morgan Kaufmann Publishers, San Francisco, CA.

Chitty, D. M. (2012). Fast parallel genetic programming: multi-core CPU versus many-core GPU. *Soft Computing*, 16(10):1795–1814.

de Melo, V. V., Fazenda, Á. L., Sotto, L. F. D. P., and Iacca, G. (2020). A mimd interpreter for genetic programming. In Castillo, P. A., Jiménez Laredo, J. L., and Fernández de Vega, F., editors, *Applications of Evolutionary Computation*, pages 645–658. Springer International Publishing, Cham.

Koza, J. R. (1994). Genetic programming as a means for programming computers by natural selection. *Statistics and Computing*, 4(2):87–112.