Análise de desempenho das técnicas de vetorização, predicação e *loads* não temporais em processadores *Skylake* *

Mateus Felipe de Cássio Ferreira[1], Francis Birck Moreira[1], Marco Antonio Zanata Alves[1], Arthur Mittmann Krause[2], Paulo Cesar Santos[2]

[1]Departamento de Informática - Universidade Federal do Paraná (UFPR)

[2]Instituto de Informática - Universidade Federal do Rio Grande do Sul (UFRGS)

[1]{mfcf17, francis, mazalves}@inf.ufpr.br

[2]{amkrause, pcssjunior}@inf.ufrgs.br

Resumo. *Este trabalho avalia o desempenho, em termos de tempo de execução, de três técnicas de otimização de código. Embora as técnicas de vetorização de instruções e predicação demonstrem uma redução nesse tempo em cada benchmark proposto, a técnica de load não temporal, ao contrário do esperado, teve um desempenho inferior quando comparado com o modelo base proposto.*

1. Introdução

O uso de técnicas de otimização de código surge da necessidade de aproveitar melhor os recursos computacionais que estão disponíveis ou alocados para uma determinada aplicação, principalmente quando se trata de sistemas de *High-Performance Computing* (HPC). Assim, um código otimizado pode assumir diferentes benefícios, como menor tempo de execução, menor consumo de energia, ou até mesmo menor preenchimento do espaço disponível na memória [Cooper and Torczon 2011].

Existem diversas dessas técnicas descritas na literatura. Este trabalho avalia especificamente o desempenho da vetorização de instruções, predicação, e a utilização de instruções de *load* não temporais. Essas três técnicas escolhidas, embora possam ser utilizadas em conjunto, exploram diferentes recursos de *hardware* disponíveis. Sendo assim, o objetivo deste trabalho é criar um cenário simples para analisar os ganhos, em termos de tempo de execução, apresentados por cada uma dessas técnicas.

2. Fundamentação Teórica

Nesta seção serão descritas as três técnicas que foram avaliadas neste trabalho.

Vetorização: A técnica de vetorização consiste em utilizar instruções vetoriais no lugar de múltiplas instruções escalares [Khartchenko 2018]. Assim, o processador aplicará uma única operação lógica/aritmética sob múltiplos dados simultaneamente por meio do uso de registradores vetoriais, aumentando o paralelismo de operações dentro do processador.

Predicação: A predicação é uma técnica que permite que um processador superescalar remova um *branch* por executar os dois caminhos a serem seguidos, mas, ao final, desconsiderar um dos caminhos de acordo com o predicado [Šilc et al. 1999]. Para isso, a

*Trabalho suportado pela CAPES, CNPq e Instituto Serrapilheira (grant Serra-1709-16621).

Instruction Set Architecture (*ISA*) de um processador dispõe de instruções predicadas, que alteram o estado do processador (escrevem em registradores ou na memória) apenas se a condição do *branch* original for respeitada. Essa técnica é mais efetiva quando controles de dependência podem ser eliminados ou para saltos de difícil predição, pois troca-se o custo de predições erradas por instruções extras a serem executadas.

***Loads* não temporais:** A memória *cache* explora dois conceitos importantes no acesso a dados: localidade espacial e localidade temporal [Crawford et al. 1998]. Ao observar os dados trazidos para dentro da memória *cache*, aqueles que são acessados mais de uma vez são classificados como temporais, enquanto os que são utilizados apenas uma única vez são chamados de não temporais [Crawford et al. 1998].

Os processadores modernos possuem em sua *ISA* operações de *load* e *store* não temporais. Essas requisições ainda acessam o primeiro nível de *cache* apenas para a tradução de endereço no *Translation Lookaside Buffer*, mas, após a tradução, são tratadas diretamente pela memória principal. Dessa maneira, um programador poderá reduzir a latência da memória *cache* durante o acesso a dados não temporais e evitar a poluição dessa memória, ao não adicionar dados que serão utilizados apenas uma única vez.

Essas operações, no entanto, operam apenas em uma memória mapeada como *write-combining*, a qual permite que escritas na memória sejam temporariamente armazenadas em um *buffer* antes de serem tratadas pelo sistema de memória [Intel® 1998].

3. Metodologia

Foram desenvolvidos três *benchmarks*, implementados na linguagem C, com a biblioteca *Intrisics* da Intel. Utilizou-se o compilador GCC versão 7.5.0, no sistema operacional Linux 4.15.0-121, distribuição Ubuntu 18.04.4. As opções do GCC utilizadas para os testes foram: *O2*, *static*, *ggdb3*, *march=native*, e as opções de vetorização desejadas (msse2, mavx, ou mavx512f). Todas as execuções foram feitas com prioridade e no mesmo *core*. O processador utilizado nos testes foi um Intel Xeon Silver 4114 (*Skylake*), formado de 12 núcleos e 24 *threads* ativas com a *cache* de último nível de 16.5 MB.

Para avaliar a vetorização e *load* não temporal foi implementado o algoritmo de soma vetorial, em que dois vetores são preenchidos com n números inteiros aleatórios e somados. Entretanto, a avaliação da técnica de *loads* não temporais só foi possível a partir da criação de um dispositivo virtual em uma região de memória mapeada como *write-combining*. Por outro lado, para avaliar a técnica de predicação foi implementada uma operação de *select* similar à consulta em bancos de dados. Um vetor é preenchido com n valores inteiros aleatórios entre 0 e 1023 e todas as posições desse vetor que contém um número menor ou igual a 128 são somadas. Para cada tamanho de n, em todos os testes propostos, a execução foi repetida 100 vezes.

4. Resultados e Discussão

Na Figura 1 são ilustrados os testes de uso da vetorização e *load* não temporal no *benchmark* de soma vetorial usando diferentes conjuntos de instruções, em que o sufixo *NT* faz menção ao teste de *load* não temporal. As tecnologias tiveram melhoras de *performance* médias em comparação à arquitetura base de: 11% (máxima 32%) para SSE128, 11% (máxima 27%) para AVX256, 7% (máxima 30%) para AVX512. Já o uso de de *loads*

não temporais implicou em aumento no tempo médio de 142% para SSE128, 107% para AVX256, e 73% para AVX512. Observa-se que o conjunto de instruções AVX256 é o mais eficiente conforme o tamanho dos vetores aumenta. Essa inversão entre os desempenhos esperados do AVX256 e AVX512 se deve ao fato de que muitas arquiteturas ainda implementam o AVX512 utilizando dois registradores AVX256, o que torna a *performance* similar. Embora uma unidade funcional AVX512 tenha o dobro de largura de uma unidade funcional AVX256, ela possui uma latência mais alta, e, portanto, executar duas unidades AVX256 em paralelo pode levar a resultados melhores. Já o uso de *loads* não temporais prejudicou o desempenho do *benchmark* em todos os conjuntos de instruções. Ao estudar o motivo, foi descoberto que o *prefetcher* da arquitetura elimina as latências da memória ao prever e trazer os dados para a memória *cache*.

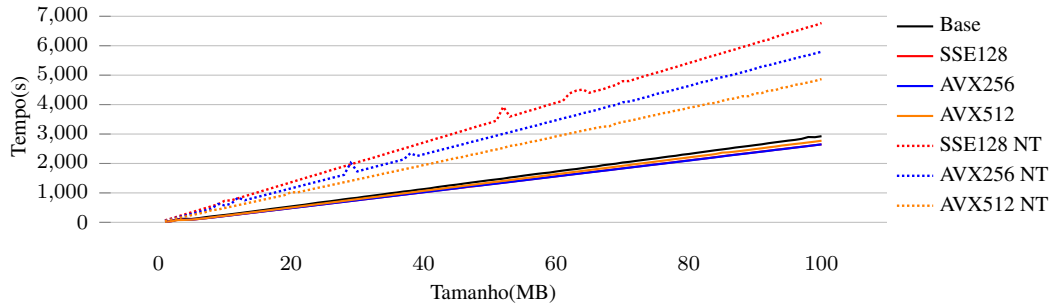

Figura 1. Tempo de execução com vetorização (AVX) e vetorização + *loads* não temporais.

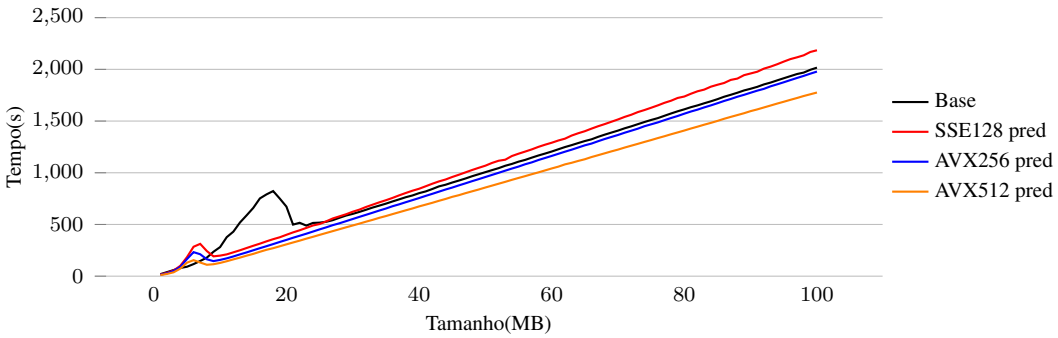

Figura 2. Tempo de execução com predicação.

Na Figura 2 são ilustrados os testes de uso da predicação no *benchmark* do operador de seleção em consultas de bancos de dados usando os diferentes conjuntos de instruções. As tecnologias tiveram melhoras de *performance* médias em comparação à arquitetura base de: -2,7% (máxima de 68%) para SSE128, 9,2% (máxima de 64%) para AVX256, 20,6% (máxima 68,5%) para AVX512. O único conjunto de instruções que implementa uma predicação efetiva para o teste é o AVX512, o que fica ainda mais evidente para as consultas maiores. Nota-se, também, duas curvas distintas. Na primeira, todos os testes (com exceção ao base) sofrem perda de desempenho conforme o tamanho do teste ultrapassa a capacidade de armazenamento da *cache* de segundo nível. Na segunda, apenas o teste base perde desempenho quando o tamanho do teste ultrapassa a capacidade de armazenamento do último nível de *cache*. Essas anomalias acontecem devido à perturbações relacionadas à política de substituição de *cache*, a qual é afetada para um n menor com a predicação devido à vetores adicionais para valores de predicado.

5. Trabalhos Correlatos

Os trabalhos correlatos avaliados serviram de motivação para a escolha das técnicas que foram avaliadas neste artigo. O primeiro trabalho [Gepner et al. 2011] avalia dois sistemas com configurações distintas de CPU: um com AVX habilitado, e outro com AVX desabilitado. Os autores utilizaram *benchmarks* padrão na área de arquitetura, como o LINPACK, STREAM, HPCC, e NPB. Os testes mostraram melhoria na *performance* com o uso de AVX, principalmente em códigos de computação intensiva. Neste trabalho, foi decidido implementar um *benchmark* sintético para avaliar o uso do AVX com maior grau de controle. Por outro lado, o trabalho de [Barredo et al. 2020] propôs uma nova abordagem que aumenta a eficiência de instruções SIMD (*Single Instruction, Multiple Data*) predicadas. Assim, a técnica consegue aumentar a eficiência em até 25% e reduzir o consumo de energia dinâmica em 43% nos *benchmarks N-body* e RNG.

6. Considerações Finais e Trabalhos Futuros

Durante a implementação de aplicações, um programador poderá escolher diversas técnicas de otimização. Cada técnica possui suas peculiaridades e seu modo de funcionamento ideal. Entre as três técnicas de otimização avaliadas neste trabalho, a única que não apresentou melhoria no desempenho foi a de *load* não temporal. Assim, este trabalho demonstra que, apesar de ser uma técnica consolidada a alguns anos na área de arquitetura de computadores, especificamente para tratar acessos à placa gráfica, ainda existem gargalos a serem superados para que ela consiga ganhar desempenho em uma aplicação real de maneira simples. Como trabalho futuro, pretende-se verificar se a vetorização pressiona outros subsistemas do processador, além de considerar a verificação do uso dessas técnicas pelo compilador de maneira automática usando diferentes *flags*.

Referências

Barredo, A., Cebrian, J. M., Moretó, M., Casas, M., and Valero, M. (2020). Improving predication efficiency through compaction/restoration of simd instructions. In *Int. Symp. on High Performance Computer Architecture*.

Cooper, K. D. and Torczon, L. (2011). *Engineering a Compiler*, page 405. Elsevier.

Crawford, J., Doshi, G., Sailer, S. E., Fu, J. W. C., and Mathews, G. S. (1998). Method and apparatus for managing temporal and non-temporal data in a single cache structure. `https://patents.google.com/patent/US6542966B1`.

Gepner, P., Gamayunov, V., and Fraser, D. L. (2011). Early performance evaluation of avx for hpc. In *Int. Conf. on Computational Science*.

Intel® (1998). Write combining memory implementation guidelines. `https://download.intel.com/design/PentiumII/applnots/24442201.pdf`.

Khartchenko, E. (2018). Vectorization: A key tool to improve performance on modern cpus. `https://software.intel.com/content/dam/develop/external/us/en/documents/vectorization-performance-quantifi-755040.pdf`.

Šilc, J., Robič, B., and Ungerer, T. (1999). *Processor Architecture: From Dataflow to Superscalar and Beyond*, pages 146–148. Springer-Verlag Berlin Heidelberg.