1.03.04 – Sistemas de Computação

OTIMIZAÇÃO DA TRANSPOSIÇÃO DE MATRIZES NA FUNÇÃO DE RESUMO SHA-256

Victória Serra de Lima Moraes[1]*, Ana Karina Dourado Salina de Oliveira[2]
1. Mestranda em Ciência da Computação da Universidade Estadual de Campinas (IC-UNICAMP)
2. Professora da FACOM-UFMS - Orientadora

Resumo

Funções *hash* garantem a integridade de dados e fornecem autenticação de mensagens, sendo muito úteis em criptografia por permitir fácil verificação de dados ao comparar o valor do *hash* fornecido. Elas são utilizadas em softwares para consulta de dados, visto que aceleram consultas a bancos de dados por meio da detecção de duplicatas. São utilizadas, ainda, em protocolos de segurança como *blockchain* e aplicações como sistemas de assinatura digital.

Este trabalho tem o objetivo de melhorar o desempenho da da utilização da transposição de matrizes na chamada da função de resumo SHA-256. São usados três diferentes métodos – dois com instruções AVX e um com criação de *threads* através do OpenMP – no processador Haswell para acelerar a função de resumo subjacente, o que se resulta em um melhor desempenho nos protocolos que utilizam esta função.

A performance é medida em um processador Haswell com Intel Core i7-4770K, as implementações são comparadas e os resultados, tabulados.

Palavras-chave: criptografia; paralelização; otimização.

Introdução

Antigamente, a criptografia era prática exclusivamente militar e de agências de inteligência. No entanto, na atualidade, algoritmos criptográficos são utilizados por praticamente todos os sistemas computacionais. A criptografia está, cada vez mais, deixando de ser uma forma de arte utilizada apenas para esconder segredos militares e se tornando uma ciência que ampara indivíduos diariamente.

As funções *hash* (funções de resumo) consistem em funções que podem ser utilizadas para garantir a integridade de dados e fornecer autenticação de mensagens. Essas funções fazem o mapeamento de dados de tamanho arbitrário para tamanho fixo. Uma família dessas funções amplamente utilizada atualmente é a **SHA** (*Secure Hash Algorithms*), publicada pelo Instituto Nacional de Padrões e Tecnologia [1] como um padrão federal de processamento de informações dos EUA.

Esquemas de Assinatura baseados em funções *hash* executam muitas aplicações dessas funções para assinar e verificar documentos. Como demonstrado em [2], utilizar recursos específicos das máquinas, como o conjunto de instruções vetoriais AVX2 (*Advanced Vector Extensions*), melhora o desempenho destes algoritmos. Essas máquinas possuem um conjunto de instruções SIMD (*Single Instruction Multiple Data*) que permite executar uma única instrução sobre um conjunto de dados, explorando o paralelismo de dados. Como as funções *hash* realizam muitos embaralhamentos de dados, elas podem se beneficiar do uso de vetores de instruções para executar operações em paralelo com SIMD. Assim, a implementação dos esquemas de assinatura baseados em funções de resumo também é favorecida utilizando o conjunto de instruções vetoriais AVX2.

Para utilizar o conjunto de instruções vetoriais AVX2, os dados a serem executados precisam ser transpostos nos registradores. Essa implementação conta extensamente com transposições de matrizes. Como esse algoritmo normalmente possui um comportamento consideravelmente lento[3], com complexidade $\Theta(n^2)$, surge a necessidade de executá-lo com um fluxo de execução alternativo para diminuir o custo em questão.

Esse trabalho explora dois possíveis métodos de paralelismo aplicados à transposição de matrizes, apresentado os resultados e comparando-os com a implementação sequencial.

Metodologia

Nesta seção serão apresentadas as quatro diferentes implementações que foram analisadas e realizadas neste projeto com o objetivo de obter melhor desempenho na transposição de matrizes.

A primeira implementação para a transposição de matrizes foi feita com o algoritmo sequencial, onde cada linha de uma matriz **n×n** é convertida em uma coluna de forma que $\mathbf{x_{ij} = x_{ji}}$. Esse tipo de implementação, embora simples, pode exibir um desempenho ruim devido à má utilização da linha de cache, especialmente quando **n** é uma potência de dois, devido a conflitos de linha de cache em um cache de CPU com associação limitada. A razão para isto é que, como **j** é incrementado no *loop* interno, o endereço de memória correspondente a $\mathbf{x_{ij}}$ ou $\mathbf{x_{ji}}$ salta de forma descontínua por **n** na memória, ou seja, o algoritmo não explora a localidade de referência.

A segunda implementação, por sua vez, propõe um aprimoramento ao algoritmo sequencial com o uso da API OpenMP. Assim, implementou-se um algoritmo no qual atribui-se uma matriz a cada *thread*, com o

objetivo de que as operações sejam realizadas em paralelo e, assim, o tempo de execução seja reduzido. Logo, cada *thread* fica com a tarefa de fazer a transposição de uma matriz.

Conforme proposto em [2], uma forma direta de usar as instruções vetoriais para otimizar a função é processar mais de um valor de resumo por vez. Assim, a terceira implementação consiste em mapear as mensagens processadas nos registradores de 256 bits, particionando-se cada mensagem em um vetor com 8 palavras de 32 bits. Para que os valores possam ser executados ao mesmo tempo para as mensagens, todas as posições de cada mensagem deverão estar no mesmo registrador. A implementação do algoritmo paralelo de transposição de matrizes por mudança de posição (TMMP) ocorreu motivada pela observação das instruções de carregamento dos registradores. O algoritmo proposto consiste em trocar as posições da matriz nos registradores AVX. Assim, é necessária a utilização de uma única operação a fim de definir os elementos nos registradores como os contidos na matriz original.

A quarta implementação utiliza a proposta do algoritmo de Ahmed Sherif Zekri [4], que realiza a transposição de matrizes de tamanho 4×4, reduzindo as demais a matrizes menores para realizar as transposições. Supondo que os registradores **x** armazenam a matriz a ser transposta; os registradores **e**, a matriz identidade de forma pré-determinada (*hard coded*); e os registradores **t**, a matriz de saída, que é inicializada com elementos nulos. O algoritmo é dividido em duas partes: as iterações, que são realizadas $\frac{n^2}{4}$ vezes, tal que **n** é a ordem da matriz; e a permutação intermediária, que será executada entre iterações. Nas iterações, são multiplicados os registradores **x** pelos registradores **e**, a nível de elemento. A seguir, esses resultados são acumulados em um vetor intermediário. Então, os vetores **x** são deslocados um elemento para a direita, de forma circular. A permutação intermediária consiste em renomear os registradores **t** de tal forma que **tj= t** $_{j\otimes 1}$, j = 0,1,...,n-1; com $j\otimes 1$ representando a adição de 1 em módulo 4.

Resultados e Discussão

Os resultados foram obtidos em um processador Haswell Core i7-4770K 3.4 Ghz, com registradores de 256 bits e com as tecnologias Intel Turbo Boost e Hyper-Threading desabilitadas. A implementação foi escrita em linguagem C e compilada utilizando o GNU C Compiler v6.3.0.

Foi definido o uso de, no máximo, 3 threads na segunda implementação do algoritmo com OpenMP, devido ao fato de que esse processador comporta até 4 threads reais, sendo que uma já é reservada para a execução do sistema operacional. Assim, apenas 3 *threads* reais são utilizadas para a execução do algoritmo, fazendo com que o uso de um número maior destas resulte em um comportamento não-previsível dos processadores. Os testes foram executados 4000 vezes para cada um dos diferentes algoritmos, a fim de se obter uma média confiável dos tempos obtidos.

Para mensurar o desempenho dos algoritmos de maneira adequada, foram utilizados dois modos de medição: *a priori*, os algoritmos foram executados de maneira individual a fim de realizar uma comparação específica e, a seguir, os mesmos foram inseridos na função de resumo para medir a performance de maneira contextualizada.

Individualmente, a quarta implementação, baseada em Zekri [4], mostra um resultado consideravelmente pior que o sequencial, com um número de ciclos 19,28 vezes maior e, consequentemente, consumindo 19,33 vezes mais tempo. Isso ocorre devido à latência da operação de multiplicação.

O algoritmo proposto (TMMP), no entanto, é o único algoritmo testado que possui tempo significativamente menor que a implementação sequencial, com apenas 5 ciclos de clock (7,2 vezes menos) e tempo de execução 6 vezes melhor, o que, em frente ao número de vezes que a transposição é executada no SHA2-256, é esperado que resulte em um desempenho melhor de forma notável.

A Tabela 1 apresenta uma comparação entre as diferentes implementações, isoladamente, no processador Haswell, incluindo uma porcentagem da melhoria em relação ao algoritmo sequencial.

Algoritmo	**Número de ciclos**	**Tempo**	**Melhoria**
Sequencial	36	0,000012 ms	0%
TMMP	5	0,000002 ms	86,11%
Zekri	694	0,000232 ms	-1827,78%
OpenMP 1 thread	66	0,000022 ms	-83,33%
OpenMP 2 threads	638	0,000213 ms	-1672,22%
OpenMP 3 threads	566	0,000189 ms	-1472,22%

Tabela 1: Performance dos diferentes algoritmos de forma individual

Pode-se, ainda, constatar que os algoritmos utilizando o OpenMP possuem desempenho muito inferior

ao sequencial, o que pode ser atribuído ao fato de que, individualmente, o custo de criação das *threads* pode não compensar quando comparado ao tempo de execução real do algoritmo. Portanto, para comprovar o desempenho em um âmbito prático, é necessário inserir esses algoritmos na função de resumo SHA2-256.

Foram executados testes com diferentes tamanhos de matrizes a fim de se entender para quais tamanhos o algoritmo com OpenMP fornecia ganhos quando comparado ao sequencial. Com essa comparação, os algoritmos começaram a mostrar tempos menores a partir de matrizes 16×16. Esses resultados não foram aqui expostos devido ao fato de esse trabalho ser focado na implementação do SHA2-256, que utiliza apenas matrizes 8×8 ou 8×16.

Para comprovar o desempenho em um âmbito prático, esses algoritmos de transposição de matrizes foram inserido na chamada da função de resumo SHA2-256. A performance e comparação dos algoritmos com o sequencial, quando inseridos na função de resumo, pode ser vista na Tabela 2.

Algoritmo	Número de ciclos	Tempo	Melhoria
Sequencial	1443	0,000482 ms	0%
TMMP	1374	0,000459 ms	4,78%
Zekri	2020	0,000675 ms	-39,99%
OpenMP 1 thread	1543	0,000516 ms	-6,93%
OpenMP 2 threads	1140	0,000381 ms	21%
OpenMP 3 threads	981	0,000328 ms	32,02%

Tabela 2: Performance dos algoritmos quando inseridos na chamada da função SHA-256

Como esperado, com a inserção no SHA2-256, o TMMP possui um desempenho nitidamente melhor que o algoritmo sequencial, com um tempo 4,78% menor, legitimando a utilização do algoritmo. A implementação baseada em Zekri [4], por sua vez, se mostra pior que a sequencial, com dados compatíveis com o que foi mostrado anteriormente. Os algoritmos com OpenMP, no entanto, apresentam tempo melhor que ambos os algoritmos sequencial e o TMMP, constatando que o custo de criação de *threads* é justificável em frente ao custo total da execução. A paralelização fazendo uso de três *threads*, em especial, exibe tempo 1,47 vezes menor que o sequencial, com um ganho significativo de 32,02%.

Conclusões

O SHA-2 é um algoritmos de *hash* seguro que é utilizado em algoritmos e protocolos criptográficos, para a proteção de informações confidenciais. Com a popularização do *blockchain* e o constante aumento de criptomoedas como o Bitcoin, o SHA-256 está sendo usado cada vez mais para verificar transações e calcular prova de trabalho ou prova de participação.

Nesse cenário, esse trabalho apresenta um estudo aprofundado de funções de resumo, seus fluxos de execução e tipos de implementações paralelas com o intuito de melhor aproveitar esses fluxos, obtendo um melhor desempenho na execução da função de resumo SHA2-256.

Com a utilização de instruções Intel AVX e da API OpenMP, foi realizada a paralelização do algoritmo de transposição de matrizes, utilizado na chamada da implementação do SHA2-256 gerando 8 *hashes* em paralelo. Os resultados foram testados utilizando uma máquina com a arquitetura Haswell e tabulados, a fim de serem comparados.

Com base nos resultados obtidos e nas análises realizadas, o uso das instruções vetoriais AVX nas implementações demonstrou aumentar a eficiência em uma das implementações dos algoritmos de transposição de matrizes. No entanto, a paralelização com a criação de *threads* apresentou os melhores resultados quando inserida na função SHA2-256, devido ao melhor aproveitamento da criação das *threads*.

Os algoritmos implementados usando o OpenMP mostraram resultados muito superiores às implementações com instruções SIMD, o que foi surpreendente quando levada em conta a Tabela 1 e o baixo nível das instruções intrínsecas.

O algoritmo proposto TMMP, por sua vez, mostrou resultados bastante satisfatórios, em especial quando executado de forma independente. Sua implementação é consideravelmente simples e apresenta ganhos notáveis, de forma que possa ser facilmente inserido em algoritmos maiores com ganhos garantidos. Durante a revisão bibliográfica realizada para esse trabalho, não foi encontrado nenhum algoritmo de transposição de matrizes com desempenho superior.

Em função da indisponibilidade de mais tempo para a criação de uma tecnologia de registro distribuído,

isso é proposto para trabalhos futuros. A paralelização otimizada da transposição de matrizes pode tornar possível a geração de valores *hash* para múltiplos blocos simultaneamente, o que, em teoria, melhoraria o tempo e a eficiência de todo o protocolo.

Ademais, visto que novas implementações de criptomoedas e aplicações utilizando esse protocolo são criadas com frequência, a criação de testes para as mesmas poderia resultar em resultados interessantes e de grande importância atual.

Finalmente, a implementação da transposição pode ser realizada utilizando a API CUDA e programação paralela para GPUs. O desempenho de tal implementação precisa ser testado e comparado às implementações aqui descritas.

Referências bibliográficas

[1] NIST, «Secure hash standard (shs)», FIPS 180-4, Gaithersburg, EUA, rel. téc., 2015. Endereço:https://csrc.nist.gov/csrc/media/publications/fips/180/4/final/documents/fips180-4-draft-aug2014.pdf#page=1&zoom=auto,-74,792.

[2] A. K. D. S. Oliveira, «Implementação Eficiente em Software do Esquema de Assinatura de Merkle e suas Variantes», Unpublished, 2017.

[3] M. S. M. Sanil Shanker KP, «Time complexity of matrix transpose algorithm using identity matrix as reference matrix», (IJCSIT) International Journal of Computer Science and Information Technologies, vol. 7, no5, pp.2347–2348, 2016, issn:0975-9646. Endereço:http://ijcsit.com/docs/Volume%5C%207/vol7issue5/ijcsit20160705043.pdf.

[4] A. S. Zekri, «Enhancing the matrix transpose operation using intel avx instruction», International Journal of Computer Science Information Technology (IJCSIT), vol.6, no3, pp. 67–78, 2014