Implementação Paralela de Blockchain

Fernanda Ribeiro Passos Cirino,
Henrique Mendonça Castelar Campos,
Ludmila Bruna Santos Nascimento,
Marcos Ani Cury Vinagre Silva

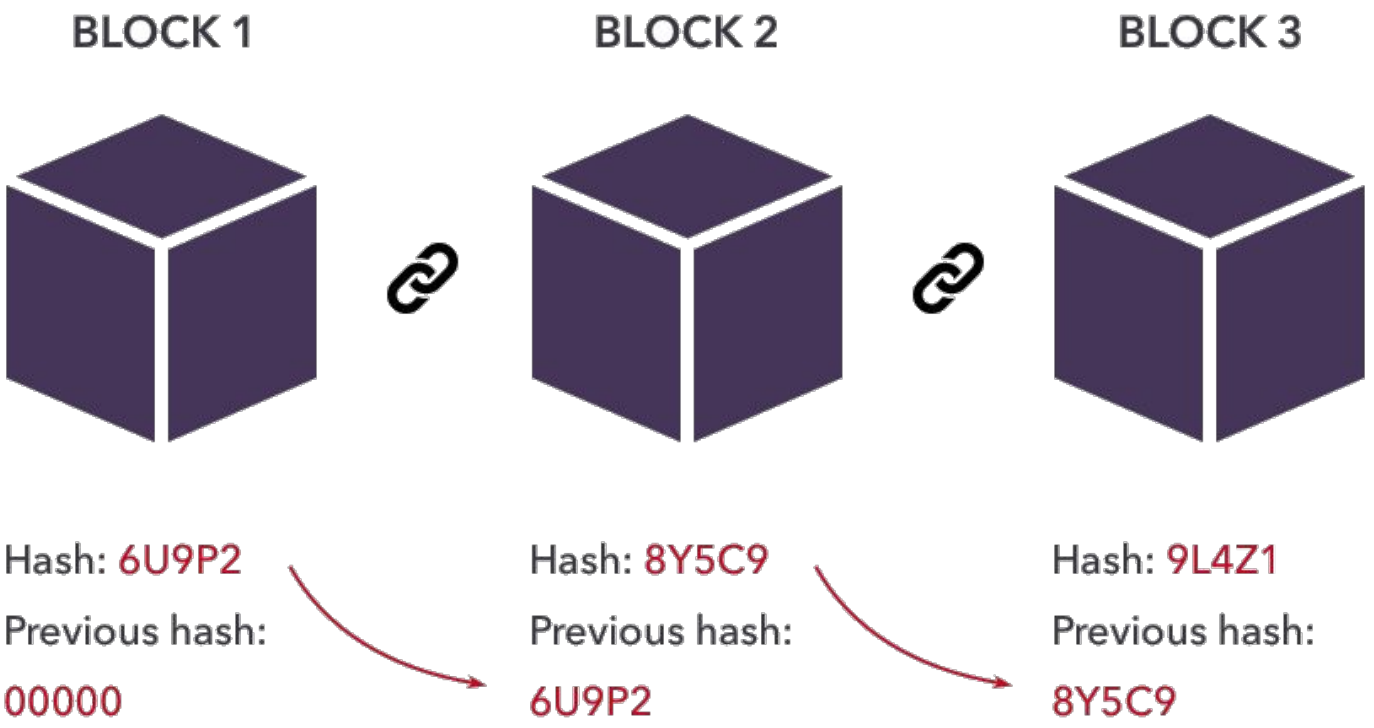
Sumário

- Introdução a Blockchain
  - O que é Blockchain
  - Como funciona a Blockchain
  - Aplicações da Blockchain
  - Algoritmo SHA 256
- Implementação da Blockchain
  - Código-fonte
  - Análise das Versões
  - Versão Sequencial
  - Versão Open MP Multicore
  - Versão Open MP GPU
  - Versão Nvidia CUDA para GPU

O que é Blockchain

- Livro de registros
- Descentralizado
- Compartilhável
- Imutável

Como funciona a Blockchain



Aplicações da Blockchain

HYPERLEDGER
FABRIC

SHA 256

Função Hash criptográfica

Usada para criar identificadores únicos e seguros

Hash de Transações

Merkle Trees

Prova de Trabalho

Endereços de Carteira

Código-Fonte: Sequencial

Função utilizada para paralelização.

```
void SHA256::transform(const unsigned char *message, unsigned int block_nb)
{
    uint32 w[64];
    uint32 wv[8];
    uint32 t1, t2;
    const unsigned char *sub_block;
    int i;
    int j;
    for (i = 0; i < (int) block_nb; i++) {
        sub_block = message + (i << 6);
        for (j = 0; j < 16; j++) {
            SHA2_PACK32(&sub_block[j << 2], &w[j]);
        }
        for (j = 16; j < 64; j++) {
            w[j] =  SHA256_F4(w[j -  2]) + w[j -  7] + SHA256_F3(w[j - 15]) + w[j - 16];
        }
        for (j = 0; j < 8; j++) {
            wv[j] = m_h[j];
        }
        for (j = 0; j < 64; j++) {
            t1 = wv[7] + SHA256_F2(wv[4]) + SHA2_CH(wv[4], wv[5], wv[6])
                 + sha256_k[j] + w[j];
            t2 = SHA256_F1(wv[0]) + SHA2_MAJ(wv[0], wv[1], wv[2]);
            wv[7] = wv[6];
            wv[6] = wv[5];
            wv[5] = wv[4];
            wv[4] = wv[3] + t1;
            wv[3] = wv[2];
            wv[2] = wv[1];
            wv[1] = wv[0];
            wv[0] = t1 + t2;
        }
        for (j = 0; j < 8; j++) {
            m_h[j] += wv[j];
        }
    }
}
```

Código-Fonte: Open MP Multicore

Função paralelizada

Tentativas de ganho de desempenho com abordagem SIMD

```
void SHA256::transform(const unsigned char *message, unsigned int block_nb) {
    int i;
    int j;
    omp_set_nested(1);

#pragma omp parallel for default(none) schedule(auto) shared(message, block_nb, w, wv) private(i, sub_block, j, t1, t2)
    for (i = 0; i < (int) block_nb; i++) {
        sub_block = message + (i << 6);

//#pragma omp parallel for simd schedule(auto) default(none) shared(sub_block, w)
        for (j = 0; j < 16; j++) {
            SHA2_PACK32(&sub_block[j << 2], &w[j]);
        }

        for (j = 16; j < 64; j++) {
            w[j] = SHA256_F4(w[j - 2]) + w[j - 7] + SHA256_F3(w[j - 15]) + w[j - 16];
        }

//#pragma omp parallel for simd schedule(auto) default(none) shared(wv)
        for (j = 0; j < 8; j++) {
            wv[j] = m_h[j];
        }

        for (j = 0; j < 64; j++) {
            t1 = wv[7] + SHA256_F2(wv[4]) + SHA2_CH(wv[4], wv[5], wv[6])
                 + sha256_k[j] + w[j];
            t2 = SHA256_F1(wv[0]) + SHA2_MAJ(wv[0], wv[1], wv[2]);
            wv[7] = wv[6];
            wv[6] = wv[5];
            wv[5] = wv[4];
            wv[4] = wv[3] + t1;
            wv[3] = wv[2];
            wv[2] = wv[1];
            wv[1] = wv[0];
            wv[0] = t1 + t2;
        }

//#pragma omp parallel for simd schedule(auto) default(none) shared(wv)
            for (j = 0; j < 8; j++) {
#pragma omp atomic
                m_h[j] += wv[j];
            }
    }
```

Código-Fonte: Open MP GPU

Função paralelizada.

Utiliza de uma abordagem target para direcionar processamento a GPU.

Utiliza de teams para distribuir loops entre threads da equipe.

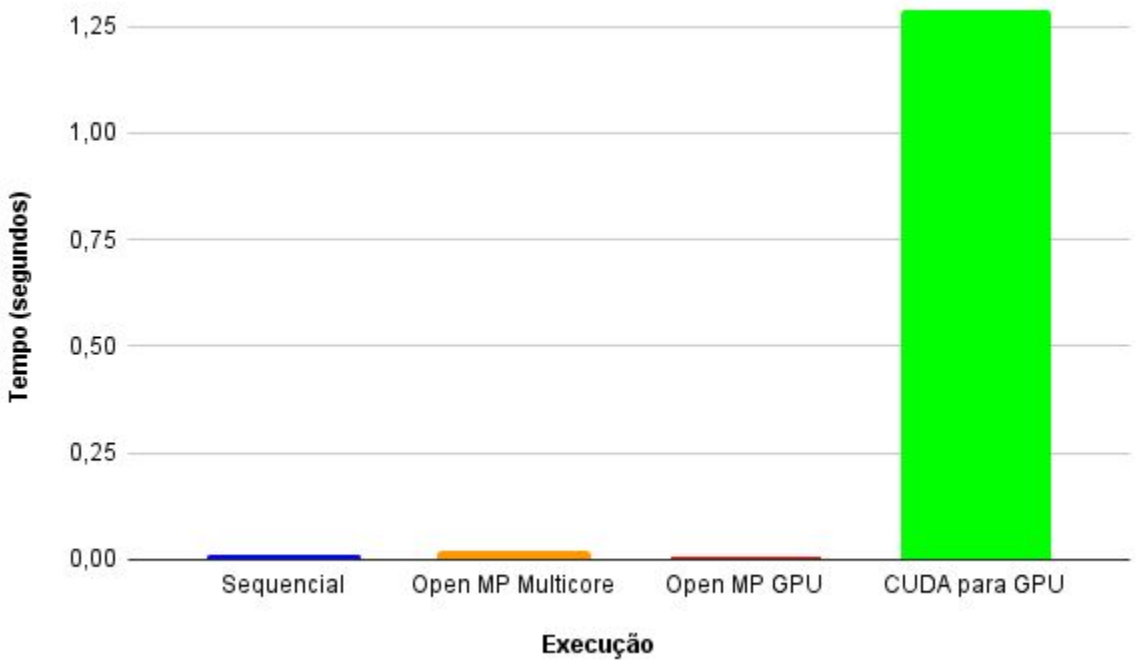
```
void SHA256::transform(const unsigned char *message, unsigned int block_nb)
{
    uint32 w[64];
    uint32 wv[8];
    uint32 t1, t2;
    const unsigned char *sub_block;
    int i;
    int j;

#pragma omp target map(to:message[0:block_nb*SHA224_256_BLOCK_SIZE], block_nb, sha256_k) map(tofrom: m_h[0:8])
#pragma omp teams distribute parallel for default(none) shared(message, block_nb, w, wv) private(i, sub_block, j, t1, t2)
    for (i = 0; i < (int) block_nb; i++) {
        sub_block = message + (i << 6);
        for (j = 0; j < 16; j++) {
            SHA2_PACK32(&sub_block[j << 2], &w[j]);
        }
        for (j = 16; j < 64; j++) {
            w[j] =  SHA256_F4(w[j -  2]) + w[j -  7] + SHA256_F3(w[j - 15]) + w[j - 16];
        }
        for (j = 0; j < 8; j++) {
            wv[j] = m_h[j];
        }
        for (j = 0; j < 64; j++) {
            t1 = wv[7] + SHA256_F2(wv[4]) + SHA2_CH(wv[4], wv[5], wv[6])
                 + sha256_k[j] + w[j];
            t2 = SHA256_F1(wv[0]) + SHA2_MAJ(wv[0], wv[1], wv[2]);
            wv[7] = wv[6];
            wv[6] = wv[5];
            wv[5] = wv[4];
            wv[4] = wv[3] + t1;
            wv[3] = wv[2];
            wv[2] = wv[1];
            wv[1] = wv[0];
            wv[0] = t1 + t2;
        }
        for (j = 0; j < 8; j++) {
            m_h[j] += wv[j];
        }
    }
}
```

Código-Fonte: Nvidia CUDA para GPU

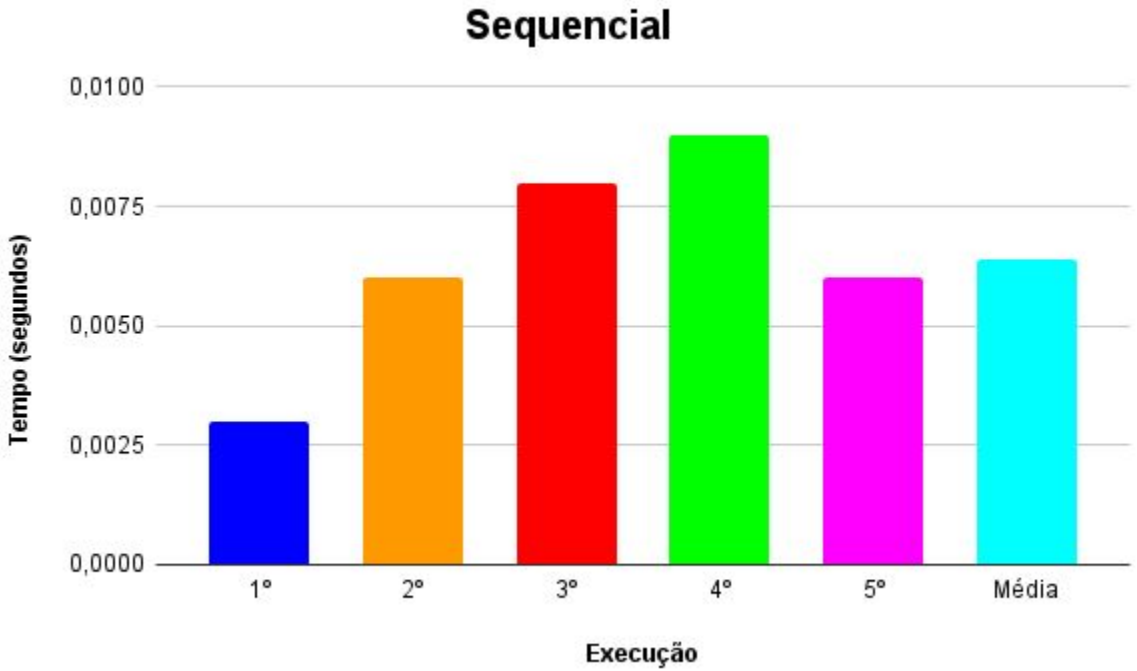
```
void SHA256::transform(const unsigned char *message, unsigned int block_nb) {

    // Variáveis
    unsigned char* message_d;
    uint32* m_h_d;

    // Tamanho das variáveis
    size_t size_m_h = 8*sizeof(uint32);
    size_t size_message = 64 * block_nb * sizeof(unsigned char);
    size_t size_sha256_k = 64 * sizeof(uint32);

    // Alocando memória
    cudaMalloc((void**)&message_d, size_message);
    cudaMalloc((void**)&m_h_d, size_m_h);

    // Copiando variáveis e constantes
    cudaMemcpy(m_h_d, m_h, size_m_h, cudaMemcpyHostToDevice);
    cudaMemcpy(message_d, message, size_message, cudaMemcpyHostToDevice);
    cudaMemcpyToSymbol(sha256_k_d, sha256_k, size_sha256_k);

    // Executando o kernel
    int block_size = 32;
    dim3 dimGrid((block_nb-1)/block_size + 1, 1, 1);
    dim3 dimBlock(block_size,1,1);
    sha256_transform<<<dimGrid, dimBlock>>>(message_d, block_nb, m_h_d);

    // Copiando variável
    cudaMemcpy((void**)m_h, (void**)m_h_d, size_m_h, cudaMemcpyDeviceToHost);

    // Liberando memória
    cudaFree((void*)m_h_d);
    cudaFree((void*)message_d);
}
```

```
__global__ void sha256_transform(const unsigned char *message, unsigned int block_nb, uint32* m_h_d) {

    __shared__ uint32 w[64];
    __shared__ uint32 wv[8];
    int tid = blockIdx.x * blockDim.x + threadIdx.x;

    if (tid < block_nb) {

        const unsigned char *sub_block = message + (tid << 6);
        for (int j = 0; j < 16; j++) {
            SHA2_PACK32(&sub_block[j << 2], &w[j]);
        }

        for (int j = 16; j < 64; j++) {
            w[j] =  SHA256_F4(w[j -  2]) + w[j -  7] + SHA256_F3(w[j - 15]) + w[j - 16];
        }

        for (int j = 0; j < 8; j++) {
            wv[j] = m_h_d[j];
        }

        for (int j = 0; j < 64; j++) {
            uint32 t1 = wv[7] + SHA256_F2(wv[4]) + SHA2_CH(wv[4], wv[5], wv[6])
                 + sha256_k_d[j] + w[j];
            uint32 t2 = SHA256_F1(wv[0]) + SHA2_MAJ(wv[0], wv[1], wv[2]);
            wv[7] = wv[6];
            wv[6] = wv[5];
            wv[5] = wv[4];
            wv[4] = wv[3] + t1;
            wv[3] = wv[2];
            wv[2] = wv[1];
            wv[1] = wv[0];
            wv[0] = t1 + t2;
        }

        for (int j = 0; j < 8; j++) {
            atomicAdd(&m_h_d[j], wv[j]);
        }
    }
}
```

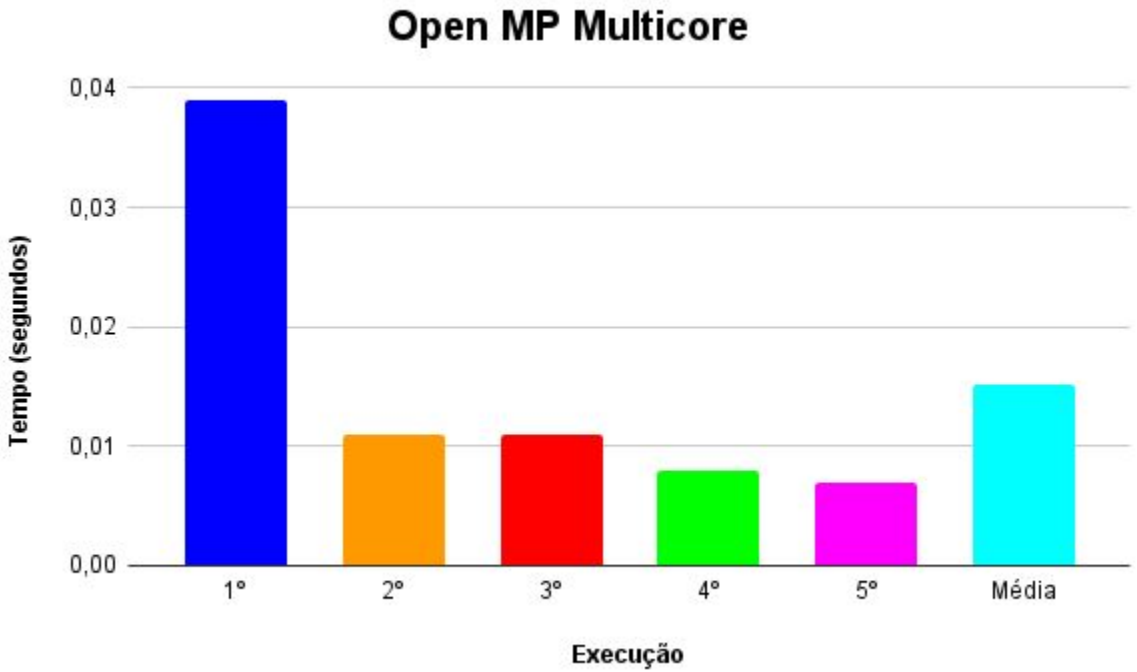
Análise das Quatro Versões



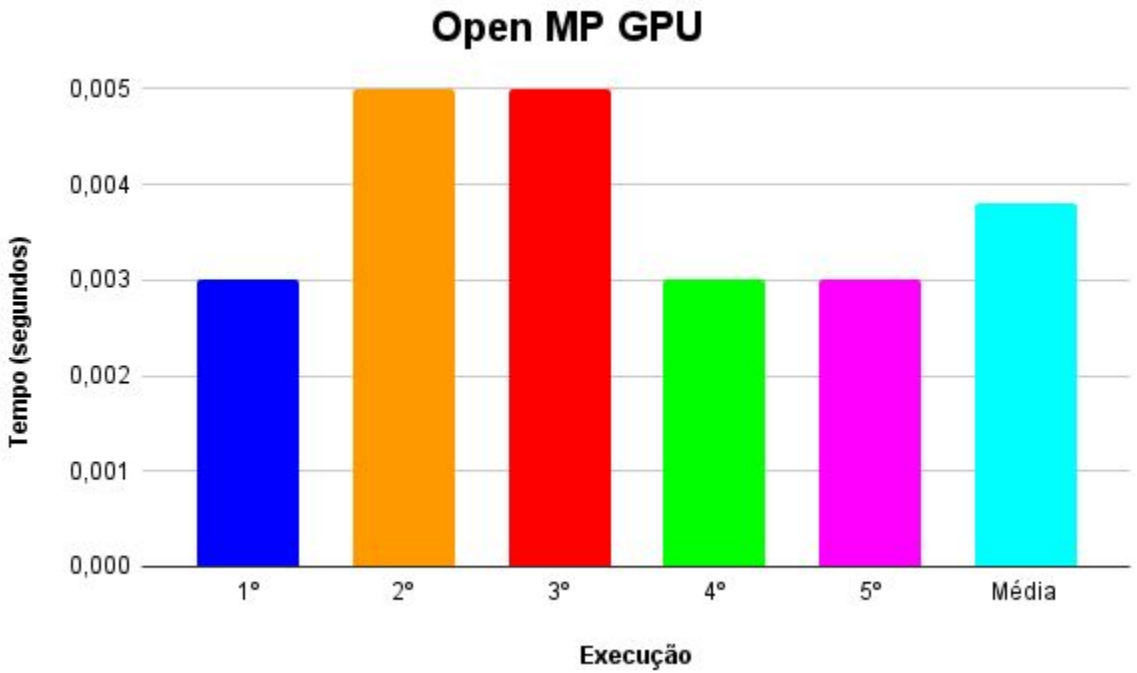
Versão Sequencial



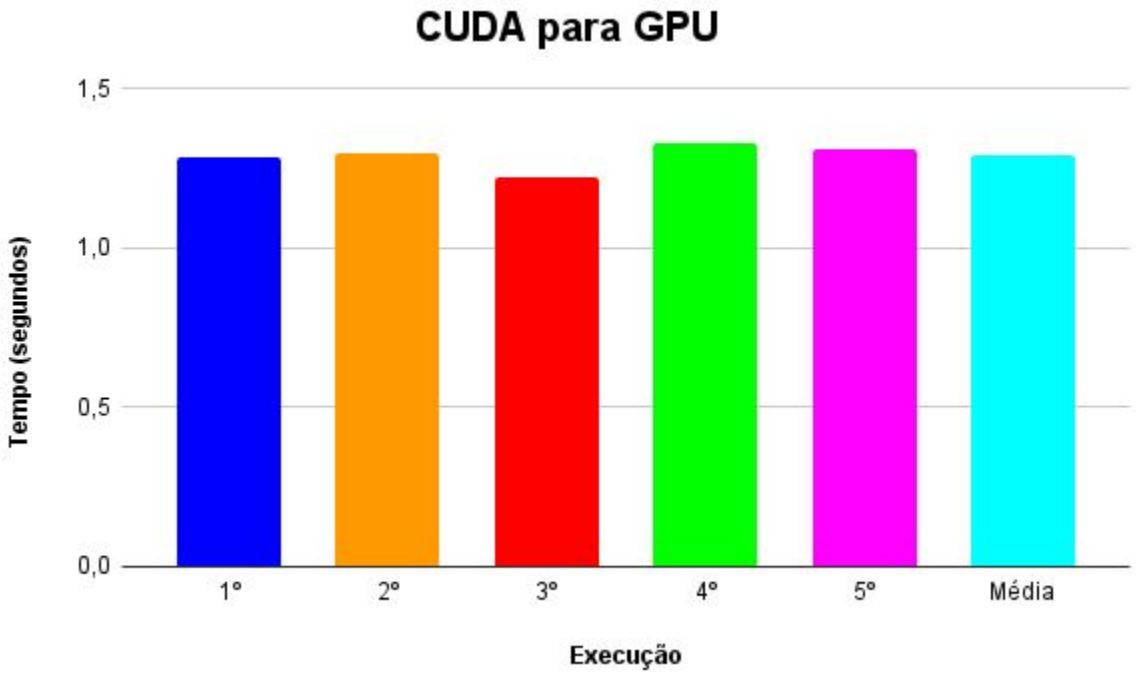
Versão Open MP Multicore



Versão Open MP GPU



CUDA para GPU



Referências

https://davenash.com/2017/10/build-a-blockchain-with-c/

https://www.oracle.com/br/blockchain/what-is-blockchain/

https://www.ibm.com/br-pt/topics/blockchain

https://pt.wikipedia.org/wiki/Blockchain