Algoritmos de Alto Desempenho

Relatório do Projeto

Luis Sousa: nºmec 108583
Simão Almeida: nºmec 113085

luisbfsousa@ua.pt
spsa@ua.pt

Link Repositório GitHub

Universidade de Aveiro
Departamento de Electrónica, Telecomunicações e Informática
2025

Contents

1 Introdução

O objetivo deste projeto é a implementação eficiente de algoritmos de mineração para as moedas digitais 'DETI Coins'. Este trabalho consiste na pesquisa exaustiva de ficheiros de 55 bytes cujo conteúdo inicie com a string "DETI coin 2 " e termine com o caráter \n. O critério de validação exige que a hash SHA1 resultante comece, em hexadecimal, por "aad20250" , sendo o valor da moeda proporcional ao número de bits zero subsequentes.

Com o intuito de maximizar a eficiência do processo de mineração, foram exploradas diversas arquiteturas de computação e estratégias de paralelismo. O trabalho foca-se no desenvolvimento de soluções de alto desempenho visando a maximização do *throughput*.

O relatório documenta a implementação destas técnicas e fornece uma análise de desempenho comparativa. São também apresentados dados estatísticos sob a forma de histogramas, caracterizando o tempo de execução e a frequência de moedas encontradas em cada execução.

2 Procura com o CPU sem instruções SIMD

2.1 Objetivo

O objetivo principal desta implementação foi estabelecer uma baseline de desempenho robusta e otimizada, removendo ineficiências comuns na manipulação de strings e geração de números aleatórios dentro do ciclo de procura.

2.2 Implementação

Inicialização Alocação e Limpeza de Memória É instanciada uma estrutura de união (union) que sobrepõe um vetor de 64 bytes (u08_t c[64]) a um vetor de 16 inteiros de 32 bits (u32_t i[16]). Antes de qualquer operação, todo o bloco de memória é inicializado a zero. Isto é crucial não só para garantir um estado determinístico, mas também porque o algoritmo SHA1 exige que o espaço entre o bit de padding e o comprimento da mensagem (nos últimos 64 bits do bloco) seja preenchido com zeros.

Otimização do espaço de procura A eficiência do algoritmo reside na minimização das operações realizadas dentro do ciclo while. A construção da moeda foi dividida em duas fases: estática e dinâmica.

Antes de iniciar a procura, o programa cria um template da moeda que permanece inalterado durante a execução.

- Os bytes iniciais "DETI coin 2 " e o sufixo obrigatório "\n" são fixos.
- Os bytes intermédios (índices 12 a 40) são preenchidos com caracteres ASCII gerados aleatoriamente via rand(). Esta operação ocorre apenas uma vez. Isto garante que cada execução do programa explora uma região diferente do espaço de procura global, sem incorrer no custo computacional de chamar rand() a cada tentativa de hash.

Dentro do ciclo principal, apenas os bytes correspondentes ao nonce (índices 40 a 53) são alterados. Em vez de usar funções de formatação lentas como sprintf, implementou-se uma conversão manual de inteiro para hexadecimal:

```
coin.c[pos ^ 3] = (u08_t)hex_map[temp_n & 0xF];
temp_n >>= 4;
```

Esta técnica utiliza operações bitwise:

- & 0xF: Isola os 4 bits menos significativos (um dígito hexadecimal).
- hex_map[...]: Mapeia instantaneamente o valor (0-15) para o caráter ASCII correspondente usando uma lookup table.
- >> 4: Desloca os bits para processar o próximo dígito

Esta abordagem é muito mais rápida do que a aritmética de divisão e módulo (% e /).

Validação A cada iteração, a função sha1 calcula o hash do bloco. A verificação da validade é feita comparando a primeira palavra do hash com a assinatura alvo 0xAAD20250.

3 Procura com AVX e AVX2

3.1 Objetivo

Para superar as limitações da execução escalar, desenvolveu-se uma implementação paralela utilizando instruções vetoriais AVX. Esta abordagem permite explorar o paralelismo de dados (SIMD - Single Instruction, Multiple Data), processando simultaneamente 4 ou 8 moedas numa única instrução de CPU.

3.2 Implementação

Estrutura de Memória Intercalada Ao contrário da implementação escalar, onde os dados de uma moeda são armazenados contiguamente na memória, a implementação vetorial exige uma reorganização dos dados para maximizar a eficiência das instruções de carga e armazenamento vetoriais.

Adotou-se uma organização de memória intercalada, definida como:

```
u32_t interleaved_data[14][N_LANES]
```

Nesta estrutura, N_LANES é definido como 4 para AVX e 8 para AVX2. O elemento interleaved_data[i][*lane*] armazena a i-ésima palavra de 32 bits da mensagem correspondente à moeda processada pela *lane* (faixa) especificada. Isto significa que:

- interleaved_data[0] contém as primeiras 4 palavras de 32 bits das 4 moedas sendo processadas em paralelo.
- Esta disposição permite carregar um vetor __m128i completo com uma única instrução, alimentando o kernel SHA1 vetorial de forma eficiente.

Otimização do Espaço de Procura (Base-95) A geração dos candidatos a moeda segue uma lógica aritmética para garantir a cobertura sistemática do espaço de procura e evitar a repetição de candidatos entre diferentes execuções ou threads.

1. No início, um base_nonce aleatório de 64 bits é gerado e convertido para a representação base-95.
2. As *lanes* são inicializadas sequencialmente: *Lane* 0 recebe base, *Lane* 1 recebe base + 1, etc.
3. A cada iteração do ciclo principal, a base_nonce é incrementada pelo numero de *lanes*, garantindo que cada vetor processa um conjunto único de candidatos.

Construção da moeda A moeda é composta por:

1. Cabeçalho e sufixo: As palavras correspondentes a "DETI coin 2 ", ao carácter de nova linha (\ n) e ao padding (0x80).
2. Corpo estático: Uma secção intermédia da moeda é preenchida com valores aleatórios (que permanecem constantes durante a execução do programa).
3. Atualização dinâmica: Dentro do ciclo de procura, apenas as palavras da estrutura interleaved_data correspondentes aos dígitos do nonce que sofreram alteração são atualizadas, minimizando o número de escritas.

Validação dos resultados Após a execução, os resultados (hashes) ficam disponíveis na matriz interleaved_hash. A validação é realizada sequencialmente pelo CPU:

- Itera-se sobre cada uma das 4 *lanes*.
- Verifica-se se a primeira palavra do hash corresponde à assinatura alvo 0xAAD20250.
- Em caso de sucesso, a moeda correspondente é reconstruída a partir da estrutura intercalada e guardada.

3.3 Discussão

A arquitetura AVX2 apresenta, teoricamente, uma vantagem significativa sobre o conjunto de instruções AVX base, primariamente devido ao alargamento dos registos vetoriais de 128 bits para 256 bits. Isto permite duplicar o paralelismo de dados, processando 8 moedas (*lanes*) por ciclo de relógio, em contraste com as 4 permitidas pelo AVX.

Adicionalmente, o AVX2 introduz suporte nativo e eficiente para operações de deslocamento de bits e manipulação de inteiros em registos de 256 bits, operações fundamentais para o algoritmo SHA1. No entanto, o ganho de desempenho real raramente é perfeitamente linear (2×) devido a vários fatores limitantes:

- **Overhead de Interleaving:** O custo computacional de organizar e transpor os dados para o formato de memória intercalada (*Structure of Arrays*) cresce com o número de *lanes*, podendo criar um gargalo na preparação dos dados antes do cálculo do hash.
- **Thermal Throttling (AVX Offset):** Os processadores modernos frequentemente reduzem a sua frequência de funcionamento (*clock speed*) quando detetam o uso intensivo de instruções AVX2 para manter a dissipação térmica (TDP) dentro dos limites de segurança. Isto significa que, embora se processem mais dados por ciclo, o número de ciclos por segundo diminui.
- **Largura de Banda:** O aumento do débito de processamento coloca maior pressão sobre a hierarquia de memória e caches (L1/L2), podendo saturar a largura de banda disponível se os dados não forem gerados inteiramente nos registos.

4 Procura com instruções SIMD e OpenMP

4.1 Objetivo

O principal objetivo desta implementação é maximizar o *throughput* através da exploração simultânea de dois níveis de paralelismo:

1. **Paralelismo de dados (SIMD)**: Utilização das unidades vetoriais (AVX/AVX2) para processar multiplos candidatos a moeda numa única instrução de CPU.
2. **Paralelismo de tarefas (Multithreading)**: Recurso á biblioteca OpenMP para distribuir a carga de trabalho por todos os núcleos físicos e lógicos disponiveis no CPU.

Esta abordagem visa saturar os recursos computacionais da máquina, transformando a procura sequêncial num processo massivamente paralelo.

4.2 Implementação

Separação do espaço de procura Para evitar que *threads* diferentes processem os mesmos candidatos, o espaço de procura é particionado da seguinte maneira:

- **Seed por Thread**: Cada *thread* inicializa o seu base_nonce aleatória utilizando o seu identificador único.
- **Stride (Passo)**: O incremento do *nonce* a cada iteração vetorial é igual ao produto entre o número de *lanes* e o de *threads*

```
stride = N_LANES * nth
```

Isto garante que as *threads* cobrem uma vasta área sem haver sobreposições.

Processamento em batches Para reduzir os custos de gestão e permitir uma vetorização mais eficiente, o processamento é realizado em lotes (*batches*). Definiu-se um BATCH_SIZE de 256 *hashes*. A estrutura de dados foi expandida para acomodar estes lotes:

```
u32_t interleaved_data[BATCH_SIZE][14][N_LANES]
```

O ciclo de procura tem três fases diferentes para cada lote:

1. **Preparação**: O CPU preenche o *buffer* interleaved_data com os dados de BATCH_SIZE * N_LANES moedas.
2. **Cálculo do Hash**: O algoritmo SHA1 é executado sobre todo o lote. Dependendo da arquitetura detetada na compilação, utiliza-se sha1_avx512f, sha1_avx2 ou sha1_avx.
3. **Verificação**: Os resultados são comparados com a assinatura criptográfica (`aad20250`) e validados.

Sincronização e registo de moedas : Quando uma moeda válida é encontrada, o acesso ao mecanismo de armazenamento deve ser protegido para evitar *race conditions*, dado que múltiplas *threads* podem tentar escrever simultaneamente. Utilizou-se a diretiva `#pragma omp critical` para garantir a exclusão mútua durante a gravação.

5 Procura com CUDA

5.1 Objetivo

Esta implementação consistiu em explorar o paralelismo oferecido pelas GPU's. Utilizando a arquitetura CUDA, desenvolveu-se uma solução capaz de executar milhares de *threads* em simultâneo, permitindo testar vastos conjuntos de candidatos a DETI coins em paralelo.

5.2 Implementação

Arquitetura do Kernel de Procura Ao contrário das abordagens utilizando o CPU abordadas anteriormente onde o numero de threads é limitado, na GPU é possível executar milhares de threads. A estratégia adotada foi:

- **Grid Unidimensional**: O kernel é lançado com um grid linear de threads.
- **Cálculo do Índice Global**: Cada thread calcula o seu identificador global único com base no índice do bloco e da thread:

```
unsigned long long idx = blockIdx.x * blockDim.x + threadIdx.x;
```

Esse índice é somado a uma *base_nonce* global para determinar a moeda especifica que será processada por essa thread.

Minimização de transferências entre o CPU e o GPU : Para evitar a transferência de dados excessiva implementou-se uma estratégia em que, ao invés de gerar as moedas no CPU e copiá-las para a GPU, o CPU envia apenas o base_nonce e o *padding* estático. Depois cada thread constrói a sua moeda na memória local, copiando o cabeçalho e gerando a parte dinâmica usando o seu id e a informação recebida pelo CPU.

Gestão de Resultados : Como as DETI coins são raras, não é eficiente reservar espaço de memória para todos os resultados de todas as threads. Em vez disso, utilizou-se um *buffer* de resultados compacto e um contador global em memória partilhada. Quando uma *thread* encontra uma moeda válida, utiliza uma instrução atómica para reservar uma posição no *buffer* de saída:

```
int pos = atomicAdd(found_count, 1);
if(pos < max_found) {
    // Copiar moeda para o buffer global de saída
}
```

Isto garante que múltiplas threads podem escrever resultados simultaneamente sem que ocorram *race conditions*, mantendo a coerência dos dados.

6 Procura com OpenCL

6.1 Objetivo

A utilização do OpenCL tem como objetivo criar uma solução de mineração que funcione em qualquer lugar. Ao contrário do CUDA que só corre nas placas NVIDIA, o OpenCL é aberto. Isto significa que pode correr em CPU's, em GPU's (NVIDIA, AMD, Intel) e noutros aceleradores, utilizando um modelo paralelo.

6.2 Implementação

6.2.1 Kernel de procura

O núcleo desta solução é o *kernel* `search_coins_kernel`. A estratégia de paralelização baseia-se num espaço n-dimensional, onde cada instância de execução processa um candidato a moeda independente.

Identificação e Nonce Cada instância de execução obtem o seu id através da função

```
ulong idx = get_global_id(0);
```

Este id é somado ao base_nonce fornecido pelo *host* para gerar a sequência numérica única que será testada por essa instância.

Memória privada e registos Para maximizar a performance e reduzir a latência de acesso à memória global, a construção da moeda é feita em memória privada. O *kernel* copia o *template* estático da memória constante para um vetor privado e, em seguida, insere os dígitos do *nonce* calculados localmente.

6.2.2 Lógica do Host

O código do *host* é responsavel pela orquestração de todo o processo. As suas principais funções incluem:

- **Descoberta e Configuração**: Utiliza as API's `clGetPlatformIDs` e `clGetDeviceIDs` para identificar os aceleradores disponiveis no sistema. O código dá prioridade à GPU, recorrendo ao CPU como *fallback*.
- **Compilação**: O código fonte do *kernel* é lido e compilado em tempo de execução. Isto garante que o binário gerado é optimizado para a arquitetura específica onde o programa está a correr.
- **Gestão de Memória**: São alocados *buffers* de memória para armazenar o *template* estático da moeda e para recolher os resultados.

6.2.3 Sincronização de resultados

Semelhante à implementação CUDA, a escrita de resultados é feita através de operações atómicas para evitar *race conditions*. Quando um *hash* válido é encontrado, a função `atomic_add` incrementa o contador global de forma segura, reservando uma posição no *buffer* de saída para a gravação dos dados da moeda.

7 Servidor e Cliente

7.1 Objetivo

O objetivo desta implementação é escalar a capacidade de mineração alem dos limites de uma única máquina fisica, permitindo a colaboração de multiplos dispositivos na resolução do mesmo problema, gerindo a distribuição das tarefas e a recolha de resultados de forma centralizada.

7.2 Implementação

7.2.1 Servidor

O servidor é responsavel por manter o estado global da procura e gerir as conexões com os clientes.

Gestão do estado global O servidor mantém uma estrutura centralizada (`server_state_t`) protegida por exclusão mútua, que armazena:

- O próximo intervalo de *nonces* para atribuição
- Estatísticas globais como o total de moedas encontradas e o total de *hashes*
- Lista de clientes ativos

Protocolo de Comunicação A comunicação é feita atraves de sockets TCP, garantindo a entrega fiável das mensagens. O servidor utiliza uma *thread* dedicada oara cada cliente conectado, permitindo o processamento concorrente de pedidos. O ciclo de vida de uma conexão inclui:

1. **Handshake**: O cliente identifica-se (`MSG_CLIENT_HELLO`) e o servidor responde (`MSG_SERVER_HELLO`).
2. **Atribuição de trabalho**: Quando um cliente solicita trabalho (`MSG_REQUEST_WORK`), o servidor reserva um bloco de *nonces* e envia os limites ao cliente.
3. **Receção dos resultados**: O servidor processa mensagens de moedas encontradas (`MSG_REPORT_COIN`), validando e guardando os dados, e atualiza as estatisticas quando um cliente reporta a conclusão do trabalho (`MSG_WORK_COMPLETE`).

7.2.2 Cliente

A função do cliente é solicitar tarefas ao servidor, executar a mineração e reportar os resultados.

Abstração do hardware O código do cliente é compilado condicionalmente para tirar partido da melhor tecnologia disponivel na máquina onde corre (AVX2, AVX, NEON ou Escalar): Isto é gerido através de macros de pré-processamento que definem `N_LANES` e a função de *hash* apropriada.

Ciclo de Mineração Após receber um trabalho do servidor, o cliente utiliza OpenMP para paralelizar a execução localmente:

- O intervalo recebido é dividido em lotes menores
- Multiplas *threads* processam os lotes em paralelo
- Quando uma moeda é encontrada, o cliente envia uma mensagem ao servidor

Esta arquitetura permite que o cliente se foque apenas no processamento das *hashes*, enquanto o servidor gere a distribuição e agregação de dados.

8 WebAssembly

8.1 Objetivo

Executar a procura de DETI *coins* diretamente no navegador recorrendo a WebAssembly (WASM), com versão escalar e versão com SIMD WASM, permitindo comparação entre browsers/arquiteturas.

8.2 Implementação

Existem duas versoes de código:
`wasm_search.c` (escalares) e `wasm_simd_search.c` (SIMD WASM). Ambas reutilizam o macro SHA1 de `aad_sha1.h`, geram as mensagens *on the fly* e reportam tentativas por segundo.

Os *bindings* e a página estão em `WebAssembly/`: ficheiro HTML de benchmark e ficheiros JS gerados pelo Emscripten (por exemplo, `wasm_search_scalar.js`, `wasm_search_simd.js`).

8.3 Página de Benchmark

Na página web:

- **Load Modules**: Inicializa os módulos WASM.
- **Difficulty**: Número de hex nibbles que devem coincidir com `0xAAD20250`.
- **Iterations**: Nãoces por execução (por omissão: 300 000).
- **Run Scalar / Run SIMD**: Executar cada variante e medir *throughput*.
- **Compare**: Corre ambas e mostra a diferenca.
- **Until Coin (SIMD)**: Repetir até encontrar coin (limitado a 10x do esperado).
- **Download Vault**: Exporta todas as moedas encontradas para um ficheiro de text (108583_113085_webassembly_vault.txt).

8.4 Notas de Compatibilidade

O suporte a SIMD WASM varia entre browsers/versões. Em equipamentos sem suporte a `msimd128`, usar a versão escalar. Resultados podem divergir por políticas de energia, JIT e limites de *timers* no browser.

8.5 Sistema de Dificuldade

Controla quantos *nibbles* hexadecimais iniciais de `hash[0]` devem coincidir com `0xAAD20250`:

- **Dificuldade 1**: `0x???????0`
- **Dificuldade 5**: `0xAAD2?250`
- **Dificuldade 8**: `0xAAD20250` (match completo a 32 bits)

Menor dificuldade ↓ implica moedas mais fáceis para testes.

8.6 Desempenho Esperado (indicativo)

Em CPUs modernos observam-se tipicamente: Escalar 5–10 MH/s; SIMD WASM 20–60 MH/s (2×–6× speedup). Valores reais dependem do browser e do sistema.

8.7 Resolução de Problemas

- **Sem coins**: reduzir dificuldade (5 ou 4), aumentar *iterations* ou usar *Until Coin*.
- **Coins não aparecem**: *hard refresh* (Ctrl+Shift+R) ou reconstruir com o script.

9 Procurar DETI coins com uma *string* personalizada

Uma das funcionalidades implementadas durante este projeto é a capacidade de procurar DETI coins com uma *string* personalizada. Isto permite, por exemplo, procurar moedas que contenham o nome do utilizador ou uma mensagem específica, mantendo os requisitos de validade criptográfica.

9.1 Implementação

Para ativar esta funcionalidade, foi adicionado um argumento de linha de comandos opcional (`-s`) a todos os programas desenvolvidos. O utilizador pode invocar o programa da seguinte forma:

```
./cpu_search -s "A minha string"
```

A lógica de implementação segue os seguintes passos:

1. **Parsing de Argumentos**: No início da execução, o programa verifica a presença da *flag* `-s`. Se encontrada, a *string* fornecida é capturada.
2. **Injeção da String (Sobrescrita)**: Se o utilizador forneceu uma *string* personalizada, o programa copia os caracteres dessa *string* para o início do corpo estático, sobrescrevendo os valores aleatórios nessas posições.
3. **Preservação do Restante**: Caso a *string* seja mais curta que o espaço disponível, os bytes subsequentes não são alterados, mantendo os valores aleatórios gerados no passo anterior. Desta forma, o bloco de dados mantém sempre o tamanho fixo e a entropia necessária.

10 Histogramas

10.1 Tempo para correr o Kernel

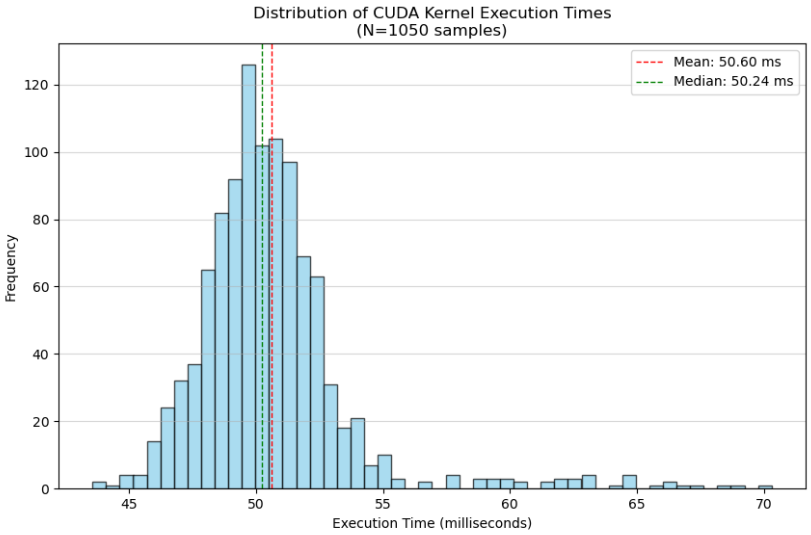

Figure 1: Tempo de execução do Kernel

Neste histograma podemos observar que a grande maioria das execuções demora entre 48 ms e 53 ms. Alem disso, como a média e a mediana estão muito próximas, podemos concluir que o desempenho do *kernel* é bastante consistente.

10.2 Coins encontradas por cada execução do Kernel

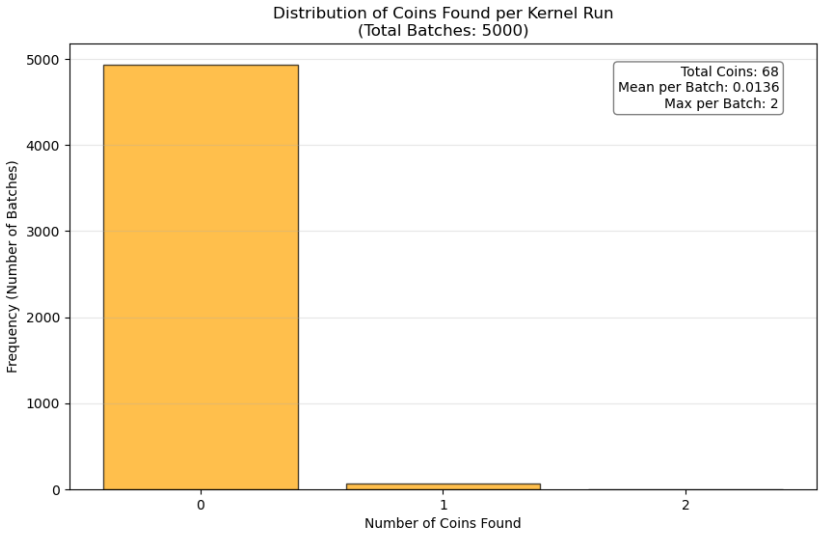

Figure 2: Moedas por execução do kernel

Foram testadas 5000 *kernel runs*, resultando numa média global de 0.01 moedas por *run*. A grande maioria dos testes teve resultado nulo; o aparecimento de uma moeda é

esporádico e a ocorrência de duas moedas na mesma *run* é extremamente improvável.

11 Avaliação do desempenho

11.1 Resultados obtidos

Este capítulo apresenta uma análise detalhada do desempenho das diferentes soluções desenvolvidas. Os resultados foram obtidos através de testes com duração de 60 segundos para cada implementação, permitindo medir o débito médio de processamento e garantir a estabilidade das métricas.

Os testes foram conduzidos em dois ambientes de hardware distintos: um portátil com processador Ryzen 7 5700U e gráficos integrados, e um portatil com processador Ryzen 7 5800H equipada com uma GPU dedicada RTX 3060.

11.1.1 AMD Ryzen 7 5700U com Radeon Graphics

Implementação	Hashes/min (M)	Hashes/s (M)
CPU	830.99	13.85
AVX	1784.75	29.74
AVX2	3311.77	55.19
SIMD + OpenMP	15876.74	264.62
CUDA	N/A	N/A
OpenCL	3673.62	61.23

11.1.2 AMD Ryzen 7 5800H com RTX 3060

Implementação	Hashes/min (M)	Hashes/s (M)
CPU	859.90	14.33
AVX	1876.75	31.28
AVX2	3426.62	57.96
SIMD + OpenMP	26013.76	433.57
CUDA	210676.92	4784.37
OpenCL	235948.72	3932.52

11.2 Análise dos resultados

AVX vs Escalar Teóricamente, a utilização do AVX deveria permitir um fator de aceleração de 4x em comparação à escalar, uma vez que processa 4 inteiros de 32 bits em paralelo. No entanto, os resultados mostram um ganho real de cerca de 2.1x. Esta diferença deve-se, principalmente:

- **Overhead de Gestão de Memória**: A versão vetorial exige que os dados estejam organizados num formato intercalado. O custo computacional de preparar os dados neste formato e de extrair os resultados consome ciclos de processamento que, na versão escalar, sáo dedicados ao cáclulo do *hash*.

AVX vs AVX2 Ao passar de AVX para AVX2, a capacidade teórica de processamento duplica (8 *lanes* em vez de 4). Os resultados experimentais mostram um ganho entre 1.8x e 1.9x, o que é muito próximo do valor teórico. A pequena perda de eficiência pode dever-se à gestão termica dos dispositivos. As instruções AVX2 necessitam de mais energia, o que faz o processador aquecer mais, que pode levar a uma redução da frequência do CPU.

Impacto do Multithreading A implementação híbrida, que combina o paralelismo de dados do AVX2 com o multithreading do OpenMP, demonstrou ser a solução mais eficiente baseada em CPU. No dispositivo com Ryzen 7 5800H (8 núcleos, 16 threads), esta abordagem atingiu cerca de 433 MH/s, contra apenas 58 MH/s da versão AVX2 *single-thread*.

Isto representa um *speedup* de aproximadamente **7.5x**, um valor próximo do número de núcleos físicos disponíveis. Este resultado de escalabilidade quase linear comprova que a estratégia de divisão do espaço de procura (baseada no *stride*) foi eficaz, evitando a sobreposição de trabalho e minimizando o *overhead* de gestão das *threads*, permitindo saturar todos os núcleos do processador.

CPU vs GPU A implementação em GPU (CUDA) representou o salto qualitativo mais significativo deste projeto, demonstrando o poder do paralelismo massivo em tarefas de *hashing*.

Ao tirar partido dos milhares de nucleos de processamento, o sistema atingiu débitos na ordem dos 4800 milhões de hashes por segundo, superando a implementação escalar em cerca de 334 vezes. Alem disso, o poder das GPUs para a resolução deste tipo de problemas tambem é visivel na implementação com o OpenCL, onde no dispositivo com GPU dedicada alcançou um fator de aceleração de 64x face ao dispositivo limitado ao CPU.

12 Conclusões

O trabalho desenvolvido permitiu explorar e implementar com sucesso múltiplas abordagens de computação de alto desempenho para a mineração de DETI coins. A análise comparativa demonstrou que a exploração do paralelismo massivo é fundamental para este tipo de problema criptográfico.

A arquitetura CUDA destacou-se significativamente, atingindo o maior throughput (cerca de 4800 MH/s), seguida pelas implementações vetoriais (SIMD) em CPU combinadas com multithreading (OpenMP).

Ficou evidente que a maximização do desempenho depende não apenas do hardware, mas também de estratégias de software eficazes como a minimização do *overhead*, a gestão de memória eficiente e o paralelismo adaptado.

Os resultados quantitativos detalhados, incluindo histogramas de tempos de execução e análise de escalabilidade, corroboram estas conclusões e validam as otimizações implementadas.