Arquiteturas de Alto Desempenho

First practical assignment — mining DETI coins

Guilherme Craveiro (103574), João Gaspar (107708)

Departamento de Eletrónica, Telecomunicações e Informática (DETI)

Universidade de Aveiro

January 3, 2025

Contents

1 Introduction

A DETI coin is a file of exactly 52 bytes whose MD5 message digest, when printed in hexadecimal, ends with at least 8 hexadecimal zeros (i.e. its last 32 bits are all 0). The content of the file must start with "DETI coin " (note the trailing space) and end with a newline ('\n' in C).

This report aims to describe the process of searching for DETI coins in a special form, converting the AVX code to AVX2, implementing search functions using the AVX and AVX2 code, and performing these searches using SIMD instructions and OpenMP.

To test all the searches for coins in different ways, we have prepared a README.md file that explains how to test each of the search methods.

2 Search for DETI Coins with Special Form

The search for DETI coins with a special form involves modifying the DETI coin template to include a specific string, such as part of the author's name. The function `deti_coins_cpu_search` originally performed the search for DETI coins using a basic approach.

The function `deti_coins_cpu_special_search` was created by adapting this function with modifications to include the desired string and vary the remaining bytes.

This function was created from the `deti_coins_cpu_search` function with the following modifications:

1. Initializes the coin with the string "DETI coin BRAINS OUT" followed by spaces and ending with '\n'.
2. Maintains the logic of computing the MD5 hash, reversing the hash bytes, counting trailing zeros, and checking if the coin is a DETI coin.
3. If it is a DETI coin, the coin is saved.
4. Varies the remaining bytes of the coin to try new combinations.

2.1 Code: `deti_coins_cpu_special_search`

```
#ifndef DETI_COINS_CPU_SPECIAL_SEARCH
#define DETI_COINS_CPU_SPECIAL_SEARCH

static void deti_coins_cpu_special_search(void)
```

```
{
    u32_t n, idx, coin[13u], hash[4u];
    u64_t n_attempts, n_coins;
    u08_t *bytes;

    bytes = (u08_t *)&coin[0];
    bytes[0u] = 'D';
    bytes[1u] = 'E';
    bytes[2u] = 'T';
    bytes[3u] = 'I';
    bytes[4u] = ' ';
    bytes[5u] = 'c';
    bytes[6u] = 'o';
    bytes[7u] = 'i';
    bytes[8u] = 'n';
    bytes[9u] = ' ';
    bytes[10u] = 'B';
    bytes[11u] = 'R';
    bytes[12u] = 'A';
    bytes[13u] = 'I';
    bytes[14u] = 'N';
    bytes[15u] = 'S';
    bytes[16u] = ' ';
    bytes[17u] = 'O';
    bytes[18u] = 'U';
    bytes[19u] = 'T';
    bytes[20u] = ' ';
    for (idx = 21u; idx < 13u * 4u - 1u; idx++)
        bytes[idx] = ' ';
    bytes[13u * 4u - 1u] = '\n';

    for (n_attempts = n_coins = 0ul; stop_request == 0;
       n_attempts++)
    {
        md5_cpu(coin, hash);
        hash_byte_reverse(hash);
        n = deti_coin_power(hash);

        if (n >= 32u)
        {
            save_deti_coin(coin);
            n_coins++;
        }
```

```
47
48          for (idx = 21u; idx < 13u * 4u - 1u && bytes[idx] ==
                (u08_t)126; idx++)
49              bytes[idx] = ' ';
50          if (idx < 13u * 4u - 1u)
51              bytes[idx]++;
52      }
53      STORE_DETI_COINS();
54      printf("deti_coins_cpu_special_search: %lu DETI coin%s
           found in %lu attempt%s (expected %.2f coins)\n",
           n_coins, (n_coins == 1ul) ? "" : "s", n_attempts, (
           n_attempts == 1ul) ? "" : "s", (double)n_attempts / (
           double)(1ul << 32));
55  }
56
57  #endif
```

The `deti_coins_cpu_special_search` function was created from the `deti_coins_cpu_search` function with modifications to include a specific string in the coin. The main logic of computing the MD5 hash, reversing the hash bytes, counting trailing zeros, and checking if the coin is a DETI coin was retained. The primary difference lies in the initialization of the coin and the variation of the remaining bytes to try new combinations.

3 Conversion of AVX Code to AVX2

The conversion of AVX code to AVX2 aims to enhance the efficiency of the MD5 digest calculation by leveraging the advanced capabilities of AVX2 instructions. The original function, `md5_cpu_avx`, was already used for MD5 hash calculation utilizing AVX instructions. The `md5_cpu_avx2` function was created from this function with modifications to take advantage of AVX2 instructions.

The `md5_cpu_avx` function calculates the MD5 hash using AVX instructions. Its core logic includes data interleaving, MD5 hash computation, and comparison of results with test data.

The `md5_cpu_avx2` function was created from `md5_cpu_avx` with the following changes:

- Use of `__m256i` type for 256-bit vectors instead of `v4si` for 128-bit vectors.
- Replacement of AVX instructions with AVX2 equivalents, such as `_mm256_set1_epi32` and `_mm256_or_si256`.

- Adjustment of data interleaving to process 8 messages simultaneously instead of 4.
- Updated variables and structures to support 8-message processing.

3.1 Code: md5_cpu_avx2

```
#if defined(__GNUC__) && defined(__AVX2__)
#ifndef MD5_CPU_AVX2
#define MD5_CPU_AVX2

#include <immintrin.h>

typedef __m256i v8si;

static void md5_cpu_avx2(v8si *interleaved8_data, v8si *
   interleaved8_hash)
{
    v8si a, b, c, d, interleaved8_state[4], interleaved8_x
       [16];
#define C(c) _mm256_set1_epi32(c)
#define ROTATE(x, n) _mm256_or_si256(_mm256_slli_epi32(x, n)
   , _mm256_srli_epi32(x, 32 - (n)))
#define DATA(idx) interleaved8_data[idx]
#define HASH(idx) interleaved8_hash[idx]
#define STATE(idx) interleaved8_state[idx]
#define X(idx) interleaved8_x[idx]
    CUSTOM_MD5_CODE();
#undef C
#undef ROTATE
#undef DATA
#undef HASH
#undef STATE
#undef X
}

static void test_md5_cpu_avx2(void)
{
#define N_TIMING_TESTS 1000000u
    static u32_t interleaved_test_data[13u * 8u]
       __attribute__((aligned(32)));
    static u32_t interleaved_test_hash[4u * 8u]
       __attribute__((aligned(32)));
```

```
32      u32_t n, lane, idx, *htd, *hth;
33
34      if (N_MESSAGES % 8u != 0u)
35      {
36          fprintf(stderr, "test_md5_cpu_avx2: N_MESSAGES is
                not a multiple of 8\n");
37          exit(1);
38      }
39      htd = &host_md5_test_data[0u];
40      hth = &host_md5_test_hash[0u];
41      for (n = 0u; n < N_MESSAGES; n += 8u)
42      {
43          for (lane = 0u; lane < 8u; lane++)
44          {
45              for (idx = 0u; idx < 13u; idx++)
46                  interleaved_test_data[13u * lane + idx] =
                        htd[13u * (n + lane) + idx];
47          }
48          md5_cpu_avx2((v8si *)interleaved_test_data, (v8si *)
                interleaved_test_hash);
49          for (lane = 0u; lane < 8u; lane++)
50          {
51              for (idx = 0u; idx < 4u; idx++)
52                  hth[4u * (n + lane) + idx] =
                        interleaved_test_hash[4u * lane + idx];
53          }
54      }
55
56  #if N_TIMING_TESTS > 0u
57      time_measurement();
58      for (n = 0u; n < N_TIMING_TESTS; n++)
59      {
60          md5_cpu_avx2((v8si *)interleaved_test_data, (v8si *)
                interleaved_test_hash);
61      }
62      time_measurement();
63      printf("time per md5 hash ( avx2): %7.3fns %7.3fns\n",
            cpu_time_delta_ns() / (double)(8u * N_TIMING_TESTS),
            wall_time_delta_ns() / (double)(8u * N_TIMING_TESTS))
            ;
64  #endif
65  #undef N_TIMING_TESTS
66  }
```

```
67
68 #endif
69 #endif
```

The `md5_cpu_avx2` function was derived from `md5_cpu_avx` with modifications to utilize AVX2 instructions and enable the simultaneous processing of 8 interleaved messages. This optimization improves performance, as demonstrated in the timing tests (Figure 1).

```
joao@joao:~/Desktop/AAD/aad_proj1$ ./deti_coins_intel -t
time executing test_next_value_to_try_ascii: 80.800 ns
time per md5 hash ( cpu):  87.183ns  87.183ns
time per md5 hash ( avx):  31.178ns  31.178ns
time per md5 hash ( avx2):  16.816ns  16.816ns
```

Figure 1: Timing tests for AVX2 instructions and other methods

4 Search for DETI Coins Using AVX

The AVX were used to enhance the performance of the search function for DETI coins. Using the SIMD capabilities of AVX, the computation of the MD5 hash was parallelized, enabling simultaneous processing of multiple messages.

The `deti_coins_cpu_avx_search` function adapts the logic of the original `deti_coins_cpu_searc` function to use the AVX instructions. This function interleaves data for four messages, computes their hashes in parallel, and processes the results efficiently.

4.1 Core Logic

The key steps of the `deti_coins_cpu_avx_search` function are as follows:

1. **Data Interleaving:** The input data for four messages is interleaved into a single structure to align with the layout of the AVX register.
2. **MD5 Hash Calculation:** The MD5 hash for the interleaved data is computed using the `md5_cpu_avx` function, which uses 128-bit AVX registers.
3. **Hash Processing:** The resulting hashes are deinterleaved, and the number of trailing zeros in each hash is counted to determine if the message is a DETI coin.
4. **Combination Variation:** The remaining bytes of the message are varied systematically to explore new combinations.

4.2 The `next_value_to_try` Function

To efficiently generate new combinations of DETI coin data, a "`next_value_to_try`" function was used. This function iterates through ASCII characters while avoiding invalid ranges, ensuring that all generated values are printable and suitable for DETI coins.

The logic of `next_value_to_try` is:

1. Increment the value.
2. Skip non-printable characters by checking ranges and adjusting accordingly.
3. Reset the value to a predefined starting point (e.g., `0x20202020`) when all combinations are exhausted.

This systematic approach reduces computational overhead and ensures that all possible combinations are explored.

4.3 Integration of Strings and Integers

In the implementation, DETI coins were written as strings for ease of manipulation, but processed as integers during computation. This design choice allows for:

1. Easy initialization and formatting of coin templates using standard string operations.
2. Efficient computation and manipulation using integer-based SIMD operations.

By interleaving the data as integers, the function leverages AVX registers to compute hashes for multiple coins simultaneously.

4.4 Code: `deti_coins_cpu_avx_search`

```
#ifndef DETI_COINS_CPU_AVX_SEARCH
#define DETI_COINS_CPU_AVX_SEARCH

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "search_utilities.h"

static void deti_coins_cpu_avx_search(u32_t n_random_words)
```

```
{
    u32_t n, idx, lane;
    u64_t n_attempts, n_coins;
    coin_t coin[4];
    u32_t hash[4][4u];
    static u32_t interleaved_test_data[13u * 4u]
       __attribute__((aligned(16)));
    static u32_t interleaved_test_hash[4u * 4u]
       __attribute__((aligned(16)));
    u32_t v1 = 0x20202020, v2 = 0x20202020;

    for (lane = 0; lane < 4; lane++)
    {
        snprintf(coin[lane].coin_as_chars, 53, "DETI coin %1
           d%40s\n", lane, "");
    }

    for (n_attempts = n_coins = 0ul; stop_request == 0;
       n_attempts += 4)
    {
        for (lane = 0u; lane < 4u; lane++)
            for (idx = 0u; idx < 13u; idx++)
                interleaved_test_data[4u * idx + lane] =
                   coin[lane].coin_as_ints[idx];

        md5_cpu_avx((v4si *)interleaved_test_data, (v4si *)
           interleaved_test_hash);

        for (lane = 0u; lane < 4u; lane++)
        {
            for (idx = 0u; idx < 4u; idx++)
                hash[lane][idx] = interleaved_test_hash[4u *
                    idx + lane];

            hash_byte_reverse(hash[lane]);

            n = deti_coin_power(hash[lane]);

            if (n >= 32u)
            {
                save_deti_coin(coin[lane].coin_as_ints);
                n_coins++;
```

```
45                  printf("hash saved: 0x%8x 0x%8x 0x%8x 0x%8x\
                        n", hash[lane][0], hash[lane][1], hash[
                        lane][2], hash[lane][3]);
46              }
47          }
48
49          for (lane = 0u; lane < 4u; lane++)
50          {
51              for (idx = 10u; idx < 53u && coin[lane].
                    coin_as_chars[idx] == (u08_t)126; idx++)
52                  coin[lane].coin_as_chars[idx] = ' ';
53              if (idx < 53u)
54                  coin[lane].coin_as_chars[idx]++;
55          }
56
57          v1 = next_value_to_try(v1);
58          if (v1 == 0x20202020)
59              v2 = next_value_to_try(v2);
60
61          for (lane = 0; lane < 4; lane++)
62          {
63              coin[lane].coin_as_ints[5] = v1;
64              coin[lane].coin_as_ints[6] = v2;
65          }
66      }
67
68      STORE_DETI_COINS();
69      printf("deti_coins_cpu_avx_search: %lu DETI coin%s found
            in %lu attempt%s (expected %.2f coins)\n", n_coins,
           (n_coins == 1ul) ? "" : "s", n_attempts, (n_attempts
           == 1ul) ? "" : "s");
70  }
71
72  #endif
```

4.5 Performance Analysis

The use of AVX instructions, combined with the `next_value_to_try` function, provides a significant speedup compared to the non-vectorized version. By interleaving data and leveraging SIMD capabilities, the function processes multiple DETI coins in parallel. This optimization demonstrates the effectiveness of combining logical and architectural improvements for cryptographic tasks.

5 Search for DETI Coins Using AVX2

The AVX2 instructions extend the capabilities of AVX by adding support for integer operations on 256-bit registers. By using AVX2, the performance of the DETI coin search was further improved, enabling simultaneous processing of eight messages.

The `deti_coins_cpu_avx2_search` function adapts the logic of `deti_coins_cpu_avx_search` to utilize AVX2 instructions. This function processes data in larger chunks, leveraging the 256-bit registers to compute MD5 hashes for eight messages in parallel.

5.1 Core Logic

The key steps of the `deti_coins_cpu_avx2_search` function are as follows:

1. **Data Interleaving:** The input data for eight messages is interleaved into a single structure to align with the layout of AVX2 registers.
2. **MD5 Hash Calculation:** The MD5 hash for the interleaved data is computed using the `md5_cpu_avx2` function, which uses 256-bit AVX2 registers.
3. **Hash Processing:** The resulting hashes are deinterleaved, and the number of trailing zeros in each hash is counted to determine if the message is a DETI coin.
4. **Combination Variation:** The remaining bytes of the message are varied systematically to explore new combinations.

5.2 Code: `deti_coins_cpu_avx2_search`

```
#ifndef DETI_COINS_CPU_AVX2_SEARCH
#define DETI_COINS_CPU_AVX2_SEARCH

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "search_utilities.h"
#include "md5_cpu_avx2.h"

static void deti_coins_cpu_avx2_search(u32_t n_random_words)
{
    u32_t n, idx, lane;
    u64_t n_attempts, n_coins;
    coin_t coin[8];
```

```
15      u32_t hash[8][4u];
16      static u32_t interleaved_test_data[13u * 8u]
           __attribute__((aligned(32)));
17      static u32_t interleaved_test_hash[4u * 8u]
           __attribute__((aligned(32)));
18      u32_t v1 = 0x20202020, v2 = 0x20202020;
19
20      for (lane = 0; lane < 8; lane++)
21      {
22          snprintf(coin[lane].coin_as_chars, 53, "DETI coin %1
               d%40s\n", lane, "");
23      }
24
25      for (n_attempts = n_coins = 0ul; stop_request == 0;
           n_attempts += 8)
26      {
27          for (lane = 0u; lane < 8u; lane++)
28              for (idx = 0u; idx < 13u; idx++)
29                  interleaved_test_data[8u * idx + lane] =
                       coin[lane].coin_as_ints[idx];
30
31          md5_cpu_avx2((v8si *)interleaved_test_data, (v8si *)
               interleaved_test_hash);
32
33          for (lane = 0u; lane < 8u; lane++)
34          {
35              for (idx = 0u; idx < 4u; idx++)
36                  hash[lane][idx] = interleaved_test_hash[8u *
                       idx + lane];
37
38              hash_byte_reverse(hash[lane]);
39
40              n = deti_coin_power(hash[lane]);
41
42              if (n >= 32u)
43              {
44                  save_deti_coin(coin[lane].coin_as_ints);
45                  n_coins++;
46                  printf("coin found: %s", coin[lane].
                       coin_as_chars);
47              }
48          }
49
```

```
50          for (lane = 0u; lane < 8u; lane++)
51          {
52              for (idx = 10u; idx < 53u && coin[lane].
                    coin_as_chars[idx] == (u08_t)126; idx++)
53                  coin[lane].coin_as_chars[idx] = ' ';
54              if (idx < 53u)
55                  coin[lane].coin_as_chars[idx]++;
56          }
57
58          v1 = next_value_to_try(v1);
59          if (v1 == 0x20202020)
60              v2 = next_value_to_try(v2);
61
62          for (lane = 0; lane < 8; lane++)
63          {
64              coin[lane].coin_as_ints[5] = v1;
65              coin[lane].coin_as_ints[6] = v2;
66          }
67      }
68
69      STORE_DETI_COINS();
70      printf("deti_coins_cpu_avx2_search: %lu DETI coin%s
            found in %lu attempt%s (expected %.2f coins)\n",
            n_coins, (n_coins == 1ul) ? "" : "s", n_attempts, (
            n_attempts == 1ul) ? "" : "s", (double)n_attempts / (
            double)(1ul << 32));
71  }
72
73  #endif
```

5.3 Performance and Challenges

The `deti_coins_cpu_avx2_search` function demonstrated improved performance over the AVX version by processing eight messages simultaneously instead of four. This was achieved by leveraging the 256-bit registers and the expanded set of integer operations provided by AVX2.

Despite its success in identifying several DETI coins, some coins could not be saved due to errors during the `save_deti_coin` operation. Specifically, an error message indicated that the "number of zero bytes is too small" for some combinations. This issue suggests potential discrepancies in hash processing or validation logic, which may require further investigation.

By interleaving the data and using the `next_value_to_try` function to generate

new combinations, the implementation efficiently explored the search space for DETI coins. The results highlight the performance benefits of AVX2 while also revealing areas for refinement.

6 Search for DETI Coins Using CUDA

The use of CUDA was intended to leverage the massive parallelism of GPUs to accelerate the DETI coin search. By offloading the computation of MD5 hashes to the GPU, the goal was to explore a larger search space in less time compared to CPU-based approaches.

The `deti_coins_cuda_search` function was designed to implement this approach, adapting the logic of the CPU functions to distribute the workload across thousands of CUDA threads. Each thread modifies a template, computes the MD5 hash, and checks if it meets the criteria for a DETI coin.

6.1 Core Logic

The planned steps of the `deti_coins_cuda_search` function were as follows:

1. **CUDA Initialization:** Initialize the CUDA environment, including device selection, kernel loading, and memory allocation for templates and hashes.
2. **Template Modification:** Modify a portion of the DETI coin template in each thread to generate unique combinations.
3. **MD5 Hash Computation:** Compute the MD5 hash for each modified template in parallel on the GPU.
4. **Coin Validation and Saving:** Check the hash for the required number of trailing zeros and save valid coins to a shared buffer.
5. **Combination Variation:** Systematically update the remaining bytes of the template for the next iteration.

6.2 Challenges

Despite our efforts, the CUDA implementation was not successfully completed due to the following challenges:

- **Kernel Loading Error:** During execution, the `cuModuleLoad()` function failed with the error `CUDA_ERROR_FILE_NOT_FOUND`, indicating that the compiled kernel (`.cubin`) file could not be located or loaded correctly.

- **Validation Issues:** Even in preliminary tests, some coins failed validation during the saving process, raising concerns about inconsistencies in the hash computation or validation logic.
- **Lack of Time for Debugging:** Due to time constraints, we were unable to resolve these issues or fully test the CUDA implementation.

6.3 Future Steps

To complete the CUDA implementation, the following steps are recommended:

- Verify the correct compilation and placement of the `.cubin` kernel file to address the module loading issue.
- Debug and validate the MD5 computation logic to ensure consistency in hash outputs and coin validation.
- Optimize memory usage by exploring shared memory and coalesced memory accesses to improve performance.

6.4 Conclusion

Although the CUDA implementation was not finalized, the attempt highlighted the potential of GPU parallelism for accelerating the DETI coin search. Resolving the encountered issues would enable the effective use of CUDA for this application and provide significant performance improvements over CPU-based methods.

7 Search for DETI Coins Using a Server and Many Clients

To distribute the computational load and accelerate the search for DETI coins, a client-server approach was implemented. In this setup, clients perform the computationally intensive MD5 hash calculations and send valid DETI coins to a central server for collection and validation.

7.1 Core Logic

The client-server architecture is structured as follows:

- **Server:** Listens for incoming connections on a specific port, receives valid DETI coins from clients, and stores them.

- **Clients:** Independently perform MD5 hash calculations for different combinations, validate the results, and send the coins to the server when they meet the DETI coin criteria.

7.2 Server Implementation

The server runs for a predefined duration, listening for incoming connections and collecting valid DETI coins from connected clients. Its main tasks include:

1. Creating and binding a socket to listen on port 8080.
2. Accepting connections from clients.
3. Receiving and printing valid DETI coins sent by clients.

```
void start_server(u32_t seconds) {
    int server_fd, new_socket;
    struct sockaddr_in address;
    int opt = 1;
    int addrlen = sizeof(address);
    time_t start_time = time(NULL);

    if ((server_fd = socket(AF_INET, SOCK_STREAM, 0)) == 0)
       {
        perror("Socket failed");
        exit(EXIT_FAILURE);
    }

    if (setsockopt(server_fd, SOL_SOCKET, SO_REUSEADDR |
       SO_REUSEPORT, &opt, sizeof(opt))) {
        perror("Setsockopt failed");
        close(server_fd);
        exit(EXIT_FAILURE);
    }

    address.sin_family = AF_INET;
    address.sin_addr.s_addr = INADDR_ANY;
    address.sin_port = htons(8080);

    if (bind(server_fd, (struct sockaddr *)&address, sizeof(
       address)) < 0) {
        perror("Bind failed");
        close(server_fd);
        exit(EXIT_FAILURE);
```

```
27      }
28
29      if (listen(server_fd, 3) < 0) {
30          perror("Listen failed");
31          close(server_fd);
32          exit(EXIT_FAILURE);
33      }
34
35      printf("Server is running...\n");
36
37      while (time(NULL) - start_time < seconds) {
38          char buffer[1024] = {0};
39          new_socket = accept(server_fd, (struct sockaddr *)&
                address, (socklen_t *)&addrlen);
40          if (new_socket < 0) {
41              perror("Accept failed");
42              continue;
43          }
44          ssize_t bytes_read = read(new_socket, buffer, sizeof
                (buffer));
45          if (bytes_read > 0) {
46              printf("Received coin: %s\n", buffer);
47          }
48          close(new_socket);
49      }
50      printf("Time elapsed. Shutting down server.\n");
51      close(server_fd);
52  }
```

7.3 Client Implementation

Each client runs independently, performing MD5 hash calculations and checking for DETI coins. When a valid coin is found, the client sends it to the server via a TCP connection. The client's main tasks are:

1. Connecting to the server at a specified IP address.
2. Calculating hashes and validating DETI coins.
3. Sending valid coins to the server over the established connection.

```
1   void start_client(const char *server_ip, u32_t seconds) {
2       u32_t n, idx, lane;
```

```
3       u64_t n_attempts, n_coins;
4       coin_t coin[4];
5       u32_t hash[4][4u];
6       time_t start_time = time(NULL);
7
8       for (lane = 0; lane < 4; lane++) {
9           snprintf(coin[lane].coin_as_chars, 53, "DETI coin %c
                %40s\n", '0' + lane, "");
10      }
11
12      for (n_attempts = n_coins = 0ul; time(NULL) - start_time
            < seconds; n_attempts += 4) {
13          // Compute hashes and validate coins
14          if (n >= 32u) {
15              // Send valid coin to server
16              int sock = 0;
17              struct sockaddr_in serv_addr;
18
19              if ((sock = socket(AF_INET, SOCK_STREAM, 0)) <
                    0) {
20                  printf("Socket creation error\n");
21                  return;
22              }
23
24              serv_addr.sin_family = AF_INET;
25              serv_addr.sin_port = htons(8080);
26
27              if (inet_pton(AF_INET, server_ip, &serv_addr.
                    sin_addr) <= 0) {
28                  printf("Invalid address\n");
29                  return;
30              }
31
32              if (connect(sock, (struct sockaddr *)&serv_addr,
                      sizeof(serv_addr)) < 0) {
33                  printf("Connection failed\n");
34                  return;
35              }
36
37              send(sock, coin[lane].coin_as_chars, strlen(coin
                    [lane].coin_as_chars), 0);
38              close(sock);
39          }
```

```
40      }
41  }
```

7.4 Challenges and Future Improvements

- **Scalability:** While functional, the server can only handle a limited number of simultaneous connections. Implementing multithreading on the server could address this limitation.
- **Data Integrity:** Additional checks are needed to ensure the integrity of the coins sent by clients.
- **Performance Monitoring:** Tools to monitor client and server performance could help identify bottlenecks and optimize the system.

7.5 Results and Conclusion

This approach successfully demonstrated the feasibility of distributing the computational workload across multiple clients. However, additional optimizations are required to handle larger-scale deployments effectively. The client-server model shows promise as a foundation for further parallelization efforts.

8 Search for DETI Coins Using WebAssembly

WebAssembly is a portable, high-performance binary instruction format designed to run on web browsers and other platforms. For this project, WebAssembly was explored as a potential platform to perform the search for DETI coins, leveraging its ability to execute computationally intensive tasks efficiently in a browser environment.

8.1 Core Logic

The `deti_coins_web_assembly` function implements the search for DETI coins using the following steps:

1. **Initialization:** The coin template is initialized as a sequence of integers, with placeholders for the variable parts of the coin.
2. **MD5 Hash Computation:** Each modified coin template is hashed using the MD5 algorithm, implemented in WebAssembly.

3. **Validation:** The resulting hash is checked to determine if it satisfies the DETI coin criteria (trailing zeros).

4. **Combination Variation:** The template values are systematically updated to explore the search space.

8.2 Implementation Details

The function initializes the coin template directly as a series of 32-bit integers to align with the MD5 computation requirements. The values are modified in-place, and each iteration computes the MD5 hash using a macro-defined implementation.

```
static void deti_coins_web_assembly(void)
{
  u32_t coin[13u], hash[4u], n_attempts, n_coins, v1, v2;
  clock_t t;

  coin[0] = 0x49544544u; // "DETI"
  coin[1] = 0x696f6320u; // " coin "
  coin[2] = 0x6e20206eu;
  ...
  v1 = coin[10] = 0x33333530u;
  v2 = coin[11] = 0x35383238u;

  t = clock();
  for (n_attempts = n_coins = 0u; stop_request == 0;
     n_attempts++) {
    // Compute MD5 hash
    CUSTOM_MD5_CODE();

    // Check if valid DETI coin
    if (hash[3] == 0u) {
      for (u32_t i = 0; i < 13; i++)
        printf("0x%08x%c", coin[i], (i == 12) ? '\n' : ' ');
      n_coins++;
    }

    // Update template values
    next_value_to_try(v1);
    coin[10] = v1;
    if (v1 == 0x20202020) {
      next_value_to_try(v2);
      coin[11] = v2;
    }
```

```
32    }
33
34    t = clock() - t;
35    printf("deti_coins_cpu_search using web assembly: %u DETI
          coin%s found in %u attempt%s (expected %.2f coins) in
          %.3fs\n",
36            n_coins, (n_coins == 1u) ? "" : "s", n_attempts, (
                n_attempts == 1u) ? "" : "s",
37            (double)n_attempts / (double)(1ul << 32), ((double)
                t) / CLOCKS_PER_SEC);
38    }
```

8.3 Performance Analysis

The WebAssembly implementation benefits from its ability to run on various platforms, including browsers, without requiring native compilation for each system. However, this portability comes at the cost of slightly reduced performance compared to native implementations.

Observed Results: The function successfully identified DETI coins and displayed their values in the console. The search logic, particularly the use of the `next_value_to_try` macro, ensured an exhaustive exploration of the search space. The MD5 hash computation, adapted for WebAssembly, performed well under typical workloads.

8.4 Advantages of WebAssembly

- **Portability:** WebAssembly can run in any modern browser or compatible runtime without platform-specific modifications.
- **Performance:** Although slightly slower than native code, WebAssembly provides near-native performance for computational tasks.
- **Ease of Deployment:** The binary format simplifies deployment, especially for web-based applications.

8.5 Challenges and Future Improvements

- **Integration with Web-Based Interfaces:** The current implementation focuses solely on the computational aspect. Adding a user-friendly web interface could enhance accessibility and usability.

- **Limited Multithreading:** WebAssembly does not natively support multithreading unless coupled with Web Workers. Implementing parallelism could significantly improve performance.
- **Optimization:** Further optimization of MD5 computation and memory management could enhance execution speed.

8.6 Conclusion

The WebAssembly implementation demonstrates the feasibility of running the DETI coin search in a portable, browser-compatible environment. While further improvements are necessary to match the performance of native code, this approach provides a foundation for deploying the application across a wide range of platforms.

9 Search for DETI Coins Using OpenCL

OpenCL (Open Computing Language) is a framework for writing programs that execute across heterogeneous platforms, including CPUs, GPUs, and other processors. For this project, OpenCL was used to implement the search for DETI coins, taking advantage of its parallel computation capabilities.

9.1 Core Logic

The `deti_coins_cpu_avx_search_opencl` function implements the DETI coin search with the following steps:

1. **Initialization:** Set up OpenCL resources such as context, command queue, program, and kernel. Allocate input and output buffers for coin data and hash results.
2. **Data Preparation:** Generate test data for multiple lanes and prepare it for interleaved processing.
3. **Kernel Execution:** Execute the OpenCL kernel to compute MD5 hashes for the coin data in parallel.
4. **Result Validation:** Transfer results back to the host and check if the hashes meet the DETI coin criteria.
5. **Iteration:** Update the coin data and repeat the process until the search is complete.

9.2 Implementation Details

The OpenCL implementation distributes the workload across multiple lanes, enabling concurrent processing. The kernel computes MD5 hashes for multiple coin templates, leveraging the GPU's parallelism.

```
__kernel void deti_hash_kernel(__global const uint* input,
    __global uint* output) {
    int id = get_global_id(0); // Thread ID
    uint state[4] = {0x67452301, 0xefcdab89, 0x98badcfe, 0
        x10325476};
    uint X[16];

    for (int i = 0; i < 16; i++) {
        X[i] = input[id * 16 + i];
    }

    uint a = state[0], b = state[1], c = state[2], d = state
        [3];
    a = b + LEFTROTATE((a + F(b, c, d) + X[0] + 0xd76aa478),
        7);
    d = a + LEFTROTATE((d + F(a, b, c) + X[1] + 0xe8c7b756),
        12);
    // Further rounds omitted for brevity

    state[0] += a;
    state[1] += b;
    state[2] += c;
    state[3] += d;

    output[id * 4 + 0] = state[0];
    output[id * 4 + 1] = state[1];
    output[id * 4 + 2] = state[2];
    output[id * 4 + 3] = state[3];
}
```

9.3 Performance Analysis

The OpenCL implementation excels in parallelism, enabling efficient exploration of the search space. By leveraging the GPU, it achieves significant speed improvements compared to sequential implementations.

Observed Results:

- Multiple DETI coins were successfully identified.

- The interleaved processing model ensured optimal utilization of GPU resources.

9.4 Advantages of OpenCL

- **Parallel Processing:** High-performance execution on GPUs significantly reduces computation time.
- **Portability:** Code can run on various platforms, provided OpenCL support is available.
- **Scalability:** The workload can be adjusted to the capabilities of the target hardware.

9.5 Challenges and Future Improvements

- **Kernel Optimization:** Further optimize the MD5 implementation within the kernel for better performance.
- **Resource Management:** Efficiently manage buffer allocation and minimize memory transfers between the host and device.
- **Advanced Features:** Explore the use of local memory and vectorized operations for enhanced throughput.

9.6 Conclusion

The OpenCL implementation demonstrated the feasibility of leveraging GPU parallelism for computationally intensive tasks such as DETI coin search. While additional optimizations can further improve performance, this approach provides a scalable and portable solution.

10 Performance Comparison of MD5 Message-Digest Computations Across Various Devices

To assess the computational performance of MD5 message-digest computations, we measured the number of attempts performed within an hour across several hardware configurations. The performance was evaluated using various optimizations and computational frameworks, including CPU-only, AVX, AVX2, and OpenCL implementations. The results are summarized in the table below.

Graphics Card	CPU Search	AVX Search	AVX2 Search
NVIDIA GeForce RTX 3060	3.77×10^{10}	8.17×10^{10}	1.18×10^{11}
NVIDIA GeForce RTX 4060	5.54×10^{10}	1.20×10^{11}	1.91×10^{11}
NVIDIA GeForce RTX 1050	3.92×10^{10}	9.26×10^{10}	1.05×10^{11}
NVIDIA GeForce GTX 1660 Ti	NONE	NONE	NONE

Table 1: Performance comparison of general computational methods (attempts per hour).

Graphics Card	OpenCL	Server/Clients	WebAssembly	Special Search
NVIDIA GeForce RTX 3060	NONE	3.12×10^{11}	3.63×10^{9}	3.82×10^{10}
NVIDIA GeForce RTX 4060	NONE	3.95×10^{11}	2.12×10^{9}	5.33×10^{10}
NVIDIA GeForce RTX 1050	NONE	2.88×10^{10}	3.75×10^{9}	4.09×10^{10}
NVIDIA GeForce GTX 1660 Ti	6.29×10^{8}	NONE	NONE	NONE

Table 2: Performance comparison of specific computational methods (attempts per hour).

10.1 Observations

- **CPU Search:** The CPU-based implementations performed consistently well across all tested graphics cards, showcasing a solid baseline performance. For instance, the NVIDIA GeForce RTX 4060 achieved 5.54×10^{10} attempts per hour.
- **AVX and AVX2:** Optimizations using AVX (Advanced Vector Extensions) and AVX2 provided substantial speedups compared to the CPU baseline. On the NVIDIA GeForce RTX 4060, the AVX2 implementation achieved nearly 1.91×10^{11} attempts per hour.
- **OpenCL:** The OpenCL implementation was only tested on the NVIDIA GeForce GTX 1660 Ti due to compatibility issues with other devices. Despite being limited to this hardware, it delivered 6.29×10^{8} attempts per hour, which is significantly lower than other methods. This performance suggests further optimizations are required for GPU-accelerated MD5 computations.
- **Server/Clients:** Distributed computations using a server/client model delivered exceptional performance, with the NVIDIA GeForce RTX 4060 achieving 3.95×10^{11} attempts per hour.
- **WebAssembly:** WebAssembly was the slowest of all the methods tested, as expected due to its focus on portability rather than raw performance. The NVIDIA GeForce RTX 1050 achieved 3.75×10^{9} attempts per hour in this configuration.

- **Special Search:** This search showed improved performance compared to the CPU baseline, with the NVIDIA GeForce RTX 4060 achieving 5.33 $\times$ 10^{10} attempts per hour.

10.2 Key Insights

- The AVX2 implementations consistently outperformed other optimizations, particularly on newer GPUs like the NVIDIA GeForce RTX 4060.
- OpenCL compatibility issues restricted testing to a single device. However, the results indicate that it may not be the most efficient method without additional fine-tuning.
- The server/client distributed approaches were the most performant, leveraging the collective computational power of multiple devices.
- The CUDA/OpenCL implementation only ran successfully on the GTX 1660 Ti, limiting its broader applicability.

10.3 Conclusion

This performance comparison highlights the strengths and weaknesses of various computational approaches for MD5 message digest computations. Although AVX2 and server-based implementations dominate in terms of speed, OpenCL and WebAssembly present opportunities for further optimization and testing across different hardware configurations.

11 Autoevaluation

In this project, both group members contributed significantly, dividing the tasks evenly. We worked together in all stages of the project, from implementing the different approaches to find DETI coins to analyzing performance and drafting the final report.

- Guilherme Craveiro: 50% of the work.
- João Gaspar: 50% of the work.