

Arquiteturas de Alto Desempenho

First assignment report

Bruno Gomes, Diogo Silva



Universidade de Aveiro

Arquiteturas de Alto Desempenho

DEPARTAMENTO DE ELETRÓNICA
TELECOMUNICAÇÕES E INFORMÁTICA

First assignment report

Bruno Gomes, Diogo Silva
(103320) brunofgomes@ua.pt, (104341) diogobranco.as@ua.pt

06/12/2024

Abstract

The main focus of this project is on the computational challenge of mining *DETI* coins, which are specialized 52-byte files whose *MD5* message digests ends with at least 32 bits with value zero. This assignment involves using advanced computational techniques to maximize efficiency in discovering *DETI* coins, such as the use of Single Instruction Multiple Data (SIMD) instructions (*AVX*, *AVX2*, *AVX512*), parallel processing with *OpenMP* and *MPI*, GPU-based computation using *CUDA*, and more experimental methods like WebAssembly.

Performance evaluation was a key aspect, with metrics based on the number of *MD5* computations per hour achieved using different computational techniques. The obtained results provided us with a clear idea of what computational methods were better in speeding up the process of finding the desired coins.

Contents

1	AVX	1
1.1	AVX Implementation	1
1.2	Results and Conclusions	4
2	AVX2	6
2.1	AVX2 Implementation	6
2.2	Results and Conclusions	8
3	AVX512	9
3.1	AVX-512 Implementation	9
3.2	Results and Conclusions	11
4	Special Search	12
4.1	Special Search Implementation	12
4.2	Results and Conclusions	13
5	OpenMP	15
5.1	OpenMP implementation	15
5.2	Results and Conclusions	17
6	Open MPI	18
6.1	Open MPI implementation	18
6.2	Results and Conclusions	19
7	CUDA	21
7.1	CUDA Implementation	21
7.2	Results and Conclusions	23
8	WebAssembly	24
8.1	WebAssembly implementation	24
8.2	Results and Conclusions	25

List of Figures

1.1	CPU search results	5
1.2	AVX search results	5
2.1	AVX2 search results	8
2.2	Hashing times for different instruction sets	8
4.1	Special search results	14
5.1	OpenMP search results	17
6.1	MPI search results	20
7.1	CUDA search results	23
8.1	WebAssembly Results	25

Chapter 1

AVX

1.1 AVX Implementation

Advanced Vector Extensions (AVX) instructions allows us to process 4 *DETI* coins simultaneously, in our implementation we are using 128 bit vectors to interleave the data of the coins and their respective hashes.

```
1 typedef int v4si __attribute__((vector_size (16)));
```

The code above represents the new data type that we defined to store both the coins and the hashes.

Additionally, to create some diffusion in the *MD5* computation, we used two *GCC*-specific intrinsics that provide access to low-level SIMD operations for shifting the elements of a 128-bit vector.

```
1 # define ROTATE(x,n) ( (__builtin_ia32_pslldi128(x,n) |  
    __builtin_ia32_psrlldi128(x,32 - (n)))
```

The defined macro **ROTATE** uses the intrinsics functions to shift a 128-bit vector *n* bits to the left or to the right.

To summarize, since the AVX contains 16 registers of 128 bits, we can confidently say that according to our calculations we can process 4 *DETI* coins at the same time. Given the fact that 128×16 gives us 2048 bits, dividing this value by the size of one coin in bits which is 416 bits ($52 \text{ bytes} \times 8$), this gives us roughly 4 *DETI* coins.

Analyzing the algorithm that searches for the coins we can separate the code into two essential parts:

1. DETI coin initialization

In this project, we decided to initialize the coin byte per byte because it was an easy way to write characters into it. Moreover, we decided to introduce a random number that occupied 4 bytes of the coin, in order to make each run have its own id, therefore, introducing more variability for each search done. The code below, shows how we approached the coin initialization.

```

1  //
2  // mandatory for a DETI coin
3  //
4  bytes[0u] = 'D';
5  bytes[1u] = 'E';
6  bytes[2u] = 'T';
7  bytes[3u] = 'I';
8  bytes[4u] = ' ';
9  bytes[5u] = 'c';
10 bytes[6u] = 'o';
11 bytes[7u] = 'i';
12 bytes[8u] = 'n';
13 bytes[9u] = ' ';
14 //
15 // arbitrary, but printable utf-8 data terminated with a
16 // '\n' is highly desirable
17 //
18 u32_t random_num = rand() % 10000;
19
20 char random_str[5];
21 snprintf(random_str, sizeof(random_str), "%04u",
22 random_num);
23
24 bytes[10u] = ' ';
25 bytes[11u] = ' ';
26 bytes[12u] = ' ';
27 bytes[13u] = ' ';
28 bytes[14u] = ' ';
29 bytes[15u] = ' ';
30
31 bytes[16u] = 's';
32 bytes[17u] = 'e';
33 bytes[18u] = 'a';
34 bytes[19u] = 'r';
35 bytes[20u] = 'c';
36 bytes[21u] = 'h';
37 bytes[22u] = ' ';
38 bytes[23u] = 'i';
39 bytes[24u] = 'd';
40 bytes[25u] = '=';
41
42 bytes[26u] = '[';
43 bytes[27u] = random_str[0];
44 bytes[28u] = random_str[1];
45 bytes[29u] = random_str[2];
46 bytes[30u] = random_str[3];
47 bytes[31u] = ']';
48
49 for (idx = 32u; idx < 13u * 4u - 1u; idx++)
50     bytes[idx] = ' ';
51 //
52 // mandatory termination
53 //
54 bytes[13u * 4u - 1u] = '\n';

```

2. Finding the DETI coins

Firstly, we need to interleave the coins to place them in a proper structure to calculate their respective coin hashes.

```

1  //
2  //  interleave data for AVX processing
3  //
4  for (idx = 0; idx < 4; idx++){
5      for (n = 0; n < 12; n++){
6          interleaved_coins[4*n+idx] = coin[n];
7      }
8      interleaved_coins[4*n+idx] = coin[n] + idx; // change
9      last block of the coin to introduce variability
10 }

```

The code above begins by iterating through each lane where the coins will be placed, which in this specific case will be 4 lanes. The coin that was initialized in the beginning will be copied for each lane, with the exception of the last block of the coin, which will be modified according to the lane where the respective coin was placed initially, in this way we avoid computing the same coin repeatedly, thus introducing even more variability.

The next step is to calculate the hash of each coin, since we have 4 coins we will also have 4 hashes, for that we use the "md5_cpu_avx()" function, which will fill up an array that will store the hash of each coin.

After that, we iterate through the array that contains the hashes, and for each lane we extract the respective hash, then we byte-reverse the hash and lastly we count the number of trailing zeros to check if we got a *DETI* coin, we will have a coin if the number of zeros is equal or more than 32. Additionally, when storing the coin in the buffer, it is necessary to increment (add the lane of the coin whose hash was validated) the last block of the original coin, to match the change that was made initially when storing the 4 coins in the interleaved array, after storing the coin in the buffer, we subtract the added value to the original coin, in order to preserve the original structure of it, this aspect will be essential when finding coins in other lanes.

The code below shows how the implementation of the algorithm was done.

```

1  md5_cpu_avx((v4si *)interleaved_coins, (v4si *)
2  interleaved_hash);
3      for (idx = 0u; idx < 4u; idx++){
4          for (n = 0u; n < 4u; n++){
5              hash[n] = interleaved_hash[4u*n + idx];
6          }
7
8          //
9          //  byte-reverse each word (that's how the MD5
10 message digest is printed...)
11          //
12          hash_byte_reverse(hash);
13          //

```



```

12         // count the number of trailing zeros of the MD5
13     hash
14         //
15         n = deti_coin_power(hash);
16         // if the number of trailing zeros is >= 32 we
17     have a DETI coin
18         //
19         if (n >= 32u)
20         {
21             coin[12] += idx;
22             save_deti_coin(coin);
23             coin[12] -= idx;
24             n_coins++;
25         }
26     }
27

```

Furthermore, before starting another attempt to find another coin, we introduce some more variability to the original coin, with the purpose of producing a coin whose hash is valid. Each byte of the original coin, starting from the 10th byte, is incremented by one, if a byte of the coin reaches the value of 126 (which corresponds to the ASCII character '~'), it gets replaced by the character ' '. Basically, this algorithm rotates the bytes of the coin in the 0x20 to 0x7E range.

```

1     for (idx = 10u; idx < 13u * 4u - 1u && bytes[idx] == (
2         u08_t)126; idx++) // (u08_t)126 = '~' character
3         bytes[idx] = ' ';
4     if (idx < 13u * 4u - 1u)
5         bytes[idx]++;

```

To conclude, after all attempts to find *DETI* coins are done, all found coins will be written in a text file and a statement will appear in the terminal showing the number of attempts done to find the coins, the number of coins found, the expected number of coins, and the elapsed time.

1.2 Results and Conclusions

To start with, we tested the cpu search without SIMD instructions to have a reference point to compare with our AVX solution. This were the results obtained for a 1-hour search using the standard cpu search.

```
aad31@banana:~/Bruno/AAD-Project01$ ./deti_coins_intel -s0 3600
searching for 3600 seconds using deti_coins_cpu_search()
deti_coins_cpu_search: 10 DETI coins found in 41308674860 attempts (expected 9.62 coins)
```

Figure 1.1: CPU search results

As shown in the image above, we successfully found 10 *DETI* coins after making 41×10^9 attempts. In comparison, our AVX implementation achieved twice as many coins with significantly fewer attempts. The results obtained for a 1-hour AVX search were the following:

```
aad31@banana:~/Bruno/AAD-Project01$ ./deti_coins_intel -s1 3600
searching for 3600 seconds using deti_coins_cpu_avx_search()
deti_coins_cpu_avx_search: 23 DETI coins found in 22757705541 attempts (expected 5.30 coins)
```

Figure 1.2: AVX search results

We are able to find more *DETI* coins using the AVX implementation, because since we process 4 coins at the same time, the probability of finding a coin is higher, and since SIMD instructions increase the throughput of the *MD5* computations, the number of attempts to find a *DETI* coin will also be less.

Chapter 2

AVX2

2.1 AVX2 Implementation

There are only minor differences between the AVX and AVX2 implementations. Rather than going through the entire code, we will focus on highlighting the key changes introduced in the AVX2 implementation.

To begin with, instead of using 128-bit vectors to store the coins and their respective hashes, we defined a new data type called "v8si" that allowed us to work with 256-bit vectors.

```
1 typedef int v8si __attribute__((vector_size(32)));
```

In this way, we are able to process double the coins and hashes when compared to the AVX implementation, this means that we will have higher chances of finding coins and also have higher throughput to compute *MD5* hashes, therefore increasing the number of coins found while needing less attempts to find the desired coins.

Moreover, since we increased the size of the vectors that we are working with, we also had to change the function responsible for performing the rotation of the bits, instead of rotating bits in a 128-bit vector, this function is able to do the same with 256-bit vectors, making it possible to achieve some diffusion when calculating the *MD5* hashes.

```
1 #define ROTATE(x,n) ( __builtin_ia32_pslldi256(x,n) |  
    __builtin_ia32_psrlldi256(x,32 - (n)) )
```

Regarding the algorithm that searches for the coins, the only the differences when compared to the AVX search are the following:

1. **Vector size:** As referenced before, in this implementation we are using 256-bit vectors instead of 128-bit vectors to store the interleaved coins and hashes.

```

1      static u32_t interleaved_coins[13u * 8u] __attribute__((aligned(32))); // 8 DETI coins
2      static u32_t interleaved_hash[4u * 8u] __attribute__((aligned(32))); // 8 MD5 hashes
3

```

2. **Coin Interleaving:** To process 8 coins in this search, we need to ensure that a partial copy of the initialized coin is placed in all 8 lanes during interleaving. The for-loop structure in the AVX2 implementation is similar to the one used in AVX. However, there are two key differences: the "idx" variable now iterates up to 8 instead of 4, and the "interleaved_coins" array is populated with strides of 8 instead of 4.

```

1      //
2      // interleave data for AVX2 processing
3      //
4      for (idx = 0; idx < 8; idx++) {
5          for (n = 0; n < 12; n++) {
6              interleaved_coins[8*n+idx] = coin[n];
7          }
8          interleaved_coins[8*n+idx] = coin[n] + idx; // change
9          last block of the coin to introduce variability
10     }

```

3. **Interleaved hashes iteration:** In this part the difference is in the way we extract the desired hash from the "interleaved_hash" array, here we also need to make sure we iterate through the array with strides of 8, due to the fact of having 8 hashes in it. In the outer for-loop the "idx" goes up to 8, with the purpose of being able to access the lanes where the hashes are placed.

```

1      //
2      // compute MD5 hash using AVX2
3      //
4      md5_cpu_avx2((v8si *)interleaved_coins, (v8si *)
5      interleaved_hash);
6      for (idx = 0u; idx < 8u; idx++) {
7          for (n = 0u; n < 4u; n++) {
8              hash[n] = interleaved_hash[8u * n + idx];
9          }
10         hash_byte_reverse(hash);
11
12         n = deti_coin_power(hash);
13
14         if (n >= 32u) {
15             coin[12] += idx;
16             save_deti_coin(coin);
17             coin[12] -= idx;
18             n_coins++;
19         }
20     }
21

```

2.2 Results and Conclusions

The results obtained for the AVX2 search were the following:

```
aad31@banana:~/Bruno/AAD-Project01$ ./deti_coins_intel -s2 3600
searching for 3600 seconds using deti_coins_cpu_avx2_search()
deti_coins_cpu_avx2_search: 35 DETI coins found in 19289619311 attempts (expected 4.49 coins)
```

Figure 2.1: AVX2 search results

As shown in the image above, we can clearly see that by using AVX2 instructions, we managed to almost double the number of coins found with a few less attempts required, when comparing to the AVX implementation. Moreover, this is achievable because the *MD5* algorithm calculates a hash twice as fast with AVX2 instructions compared to AVX instructions. Since the hashing takes less time, it means that the algorithm can compute more hashes per second, therefore increasing the chances of finding coins. This happens because the throughput of data is even higher when using AVX2 instructions.

```
aad31@banana:~/Bruno/AAD-Project01$ ./deti_coins_intel -t
time per md5 hash ( cpu):  71.647ns  71.655ns
time per md5 hash ( avx):  30.025ns  30.028ns
time per md5 hash ( avx2): 14.961ns  14.963ns
```

Figure 2.2: Hashing times for different instruction sets

Chapter 3

AVX512

3.1 AVX-512 Implementation

The AVX-512 implementation builds upon the AVX2 version, introducing several key differences that allow it to process a significantly larger amount of data simultaneously. Below, we highlight the primary changes introduced in the AVX-512 implementation compared to AVX2.

To start, instead of using 256-bit vectors to store the coins and their respective hashes, we defined a new data type called "v16si" that enabled us to work with 512-bit vectors. This allows us to process 16 coins and their respective hashes simultaneously, doubling the throughput compared to the AVX2 implementation.

```
1 typedef int v16si __attribute__((vector_size(64)));
```

With this, the implementation can process twice the number of coins and hashes as AVX2, leading to higher throughput for computing MD5 hashes. Consequently, this increases the likelihood of finding coins while requiring fewer attempts.

Additionally, since the size of the vectors used in the computation has doubled, we also needed to adapt the function responsible for performing bit rotations. The updated function handles 512-bit vectors instead of 256-bit vectors, allowing for efficient MD5 computation with proper diffusion.

```
1 #define ROTATE(x, n) (_mm512_rol_epi32((__m512i)(x), (n)))
```

Regarding the coin search algorithm, the main differences compared to the AVX2 implementation are as follows:

1. Vector size: The AVX-512 implementation uses 512-bit vectors instead of 256-bit vectors, which doubles the number of interleaved coins and hashes processed.

```

1  static u32_t interleaved_coins[13u * 16u] __attribute__((
    aligned(64))); // 16 DETI coins
2  static u32_t interleaved_hash[4u * 16u] __attribute__((
    aligned(64))); // 16 MD5 hashes
3

```

2. Coin interleaving: To process 16 coins simultaneously we ensured that a partial copy of the initialized coin was placed in all 16 lanes during interleaving. The for-loop structure is similar to the AVX2 implementation, but the "idx" variable now iterates up to 16 instead of 8 and the "interleaved_coins" array is populated with strides of 16 instead of 8

```

1  for (idx = 0; idx < 16; idx++) {
2  for (n = 0; n < 12; n++) {
3      interleaved_coins[16 * n + idx] = coin[n];
4  }
5  interleaved_coins[16 * n + idx] = coin[n] + idx; // change
6  last block for variability
7  }

```

3. Interleaved hashes iteration: Similar to the coin interleaving step, the interleaved hashes must be processed using strides of 16, given that there are 16 hashes in the interleaved array. The "idx" variable now iterates up to 16 to extract the desired hash values from the array.

```

1  md5_cpu_avx512((v16si *)interleaved_coins, (v16si *)
    interleaved_hash);
2  for (idx = 0; idx < 16; idx++) {
3      for (n = 0; n < 4; n++) {
4          hash[n] = interleaved_hash[16 * n + idx];
5      }
6
7      hash_byte_reverse(hash);
8      n = deti_coin_power(hash);
9
10     if (n >= 32) {
11         coin[12] += idx;
12         save_deti_coin(coin);
13         coin[12] -= idx;
14         n_coins++;
15     }
16 }
17

```

3.2 Results and Conclusions

Due to hardware limitations, we were unable to run the AVX-512 implementation.

However, based on the expected performance improvements, we anticipate the AVX-512 version to double the throughput compared to AVX2, yielding results similar to those observed in the transition from AVX to AVX2.

Chapter 4

Special Search

4.1 Special Search Implementation

To implement this solution, we used our AVX2 implementation and modified the initial structure of the coin to give it a unique appearance. We added the initials of the group members' first names, followed by the string "special edition", and finally, a random number. The idea of generating a random number each time a search was initiated originated from this implementation. We found this approach beneficial for introducing randomness to the searches and decided to apply it across all other searches in our project. This effectively creates an identification number for each search, allowing us to differentiate them.

The code snippet below demonstrates how we initialized the special coin.

```
1  //
2  // mandatory for a DETI coin
3  //
4  bytes[0u] = 'D';
5  bytes[1u] = 'E';
6  bytes[2u] = 'T';
7  bytes[3u] = 'I';
8  bytes[4u] = ' ';
9  bytes[5u] = 'c';
10 bytes[6u] = 'o';
11 bytes[7u] = 'i';
12 bytes[8u] = 'n';
13 bytes[9u] = ' ';
14 //
15 // arbitrary, but printable utf-8 data terminated with a '\n'
16 // is highly desirable
17 //
18 u32_t random_num = rand() % 10000;
19 char random_str[5];
20 snprintf(random_str, sizeof(random_str), "%04u", random_num);
21
22 bytes[10u] = ' ';
23 bytes[11u] = ' ';
```

```

24     bytes[12u] = ' ';
25     bytes[13u] = ' ';
26     bytes[14u] = ' ';
27     bytes[15u] = ' ';
28     bytes[16u] = 'B';
29     bytes[17u] = '&';
30     bytes[18u] = 'D';
31     bytes[19u] = ' ';
32     bytes[20u] = 's';
33     bytes[21u] = 'p';
34     bytes[22u] = 'e';
35     bytes[23u] = 'c';
36     bytes[24u] = 'i';
37     bytes[25u] = 'a';
38     bytes[26u] = 'l';
39     bytes[27u] = ' ';
40     bytes[28u] = 'e';
41     bytes[29u] = 'd';
42     bytes[30u] = 'i';
43     bytes[31u] = 't';
44     bytes[32u] = 'i';
45     bytes[33u] = 'o';
46     bytes[34u] = 'n';
47     bytes[35u] = ' ';
48
49     bytes[36u] = '[';
50     bytes[37u] = random_str[0];
51     bytes[38u] = random_str[1];
52     bytes[39u] = random_str[2];
53     bytes[40u] = random_str[3];
54     bytes[41u] = ']';
55
56     for (idx = 42u; idx < 13u * 4u - 1u; idx++)
57         bytes[idx] = ' ';
58
59     // mandatory termination
60
61     bytes[13u * 4u - 1u] = '\n';

```

4.2 Results and Conclusions

The results did not differ much from the AVX2 implementation because in the special search we are using 256-bit vectors to store both the hashes and the coins and we also used the function "md5_cpu_avx2" to compute the *MD5* hashes. Basically, this is a slightly modified AVX2 search that does not introduce any performance enhancing features.

The image below shows the similarity of results between this implementation and the AVX2 implementation.

```
aad31@banana:~/Bruno/AAD-Project01$ ./deti_coins_intel -s9 3600
searching for 3600 seconds using deti_coins_cpu_special_search()
deti_coins_cpu_avx2_search: 39 DETI coins found in 19278388599 attempts (expected 4.49 coins)
```

Figure 4.1: Special search results

To conclude this section, we successfully obtained 39 *DETI* coins, a result very close to the 35 coins achieved with the AVX2 implementation. As mentioned earlier, this outcome was expected due to the absence of optimization improvements in this implementation.

Chapter 5

OpenMP

5.1 OpenMP implementation

For this solution, we decide to use our AVX2 implementation as the foundation for our *OpenMP* implementation, with the main purpose of trying to speed up the process of finding the coins. *OpenMP* will be used to distribute the workload of the search algorithm across multiple threads, further enhancing parallelism and performance.

All in all, the general idea of this solution is to combine and leverage both data-level parallelism (via AVX2) and task-level parallelism (via *OpenMP*)

Getting into the code, few changes were made in order to make this implementation possible, below we will go through every single aspect that makes this code run even more efficiently:

1. **Main pragma and local variables:** In the initial section of the code, we define the number of threads that will execute a specific portion of the code and initialize the variables "n_attempts" and "n_coins" to 0. Following this, we define a pragma directive responsible for parallelizing the main part of the code, which handles the coin search. Within this pragma, a reduction operation is applied to the "n_attempts" and "n_coins" variables. This ensures that each thread, which operates with a private copy of these variables, contributes to the global totals once its execution is complete.

It is important to note that any variable declared inside the pragma will be local to that specific thread. Additionally, we had to modify the declaration of the "interleaved_coins" and "interleaved_hash" arrays. Initially, these arrays were declared as static, which caused them to be shared among all threads, resulting in numerous hashing errors. By removing the static declaration, we ensured that each thread had its own private copy of these arrays, preventing conflicts.

```

1 #define NUM_THREADS 4
2
3 static void deti_coins_cpu_avx2_omp_search(void)
4 {
5
6     u64_t n_attempts = 0 , n_coins = 0;
7 #pragma omp parallel num_threads(NUM_THREADS) reduction(+:
8     n_attempts, n_coins)
9 {
10     u32_t n, idx, coin[13u], hash[4u];
11     u08_t *bytes;
12     u32_t interleaved_coins[13u * 8u] __attribute__((aligned
13     (32))); // 8 DETI coins
14     u32_t interleaved_hash[4u * 8u] __attribute__((aligned(32)
15     )); // 8 MD5 hashes
16
17     bytes = (u08_t *)&coin[0]; // accesses coin information
18     byte per byte
19     ...
20

```

2. **Coin variations across different threads:** To ensure that each thread generated a unique set of coins, we modified the last block of the coin for each thread. Specifically, we added the thread number to the last block of the coin, ensuring that each thread operated on a distinct set of coins.

```

1     coin[12] += (omp_get_thread_num()); //different set of
2     coins for different threads

```

3. **Critical sections:** When parallelizing code we had to make sure that accesses to shared memory were made in an exclusive manner, meaning that only one thread at the time could access a shared memory block, this is important because we want to avoid race conditions that could potentially cause corruption of coin data. We did so, by adding a new pragma responsible for managing accesses to the buffer where the coins were saved, and the function that is responsible for writing the coins in a text file was colocated outside the main pragma, in this way the coins that are stored in buffer will be only written to the disk after the parallelized section of the code is done running.

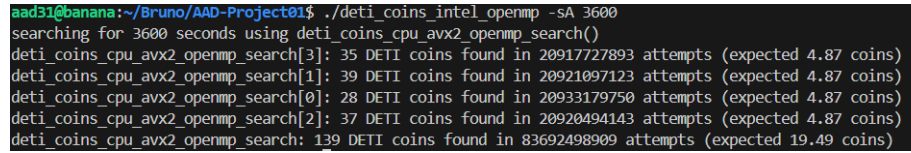
```

1     if (n >= 32u) {
2         coin[12] += idx;
3 #pragma omp critical
4 {
5         save_deti_coin(coin); // write coins in a buffer
6     }
7     coin[12] -= idx;
8     n_coins++;
9 }
10

```

5.2 Results and Conclusions

The results obtained for this implementation can vary a lot, because it will always depend on the number of threads used for running the parallel section of the program. For this specific test, we ran the code for 1 hour and allocated 4 threads to do the task of finding the coins. There was a significant improvement in the number of the coins found when comparing it directly with the standard AVX2 implementation.

A terminal window showing the execution of a program. The prompt is 'aad31@banana:~/Bruno/AAD-Project01\$'. The command is './deti_coins_intel_omp -SA 3600'. The output shows the program searching for 3600 seconds using 'deti_coins_cpu_avx2_omp_search()'. It then displays results for four threads: thread 3 found 35 DETI coins, thread 1 found 39 DETI coins, thread 0 found 28 DETI coins, and thread 2 found 37 DETI coins. Finally, it shows the aggregated result: 139 DETI coins found in 83692498909 attempts, with an expected 19.49 coins.

```
aad31@banana:~/Bruno/AAD-Project01$ ./deti_coins_intel_omp -SA 3600
searching for 3600 seconds using deti_coins_cpu_avx2_omp_search()
deti_coins_cpu_avx2_omp_search[3]: 35 DETI coins found in 20917727893 attempts (expected 4.87 coins)
deti_coins_cpu_avx2_omp_search[1]: 39 DETI coins found in 20921097123 attempts (expected 4.87 coins)
deti_coins_cpu_avx2_omp_search[0]: 28 DETI coins found in 20933179750 attempts (expected 4.87 coins)
deti_coins_cpu_avx2_omp_search[2]: 37 DETI coins found in 20920494143 attempts (expected 4.87 coins)
deti_coins_cpu_avx2_omp_search: 139 DETI coins found in 83692498909 attempts (expected 19.49 coins)
```

Figure 5.1: OpenMP search results

As shown in the image above, we significantly increased the number of coins found, going from 35 *DETI* coins to 139. In the final section of the parallel code, we included a print statement to display the results computed by each thread, such as the number of coins found and attempts made by each thread. Additionally, we added a print statement outside the parallel section to display the aggregated statistics from all threads combined.

To sum up, by combining AVX2 with *OpenMP*, this implementation can utilize multiple CPU cores, further enhancing performance and leading to a higher throughput because each thread can independently process multiple *DETI* coins using AVX2 instructions.

Chapter 6

Open MPI

6.1 Open MPI implementation

For this solution, we utilized *OpenMPI*, an implementation of the Message Passing Interface (MPI) specification designed for developing distributed computing applications. The foundation of this program is the basic "cpu_search" implementation, as we encountered several challenges when attempting to integrate our AVX2 implementation with this library.

By adapting the serial code to a distributed system, we leveraged the parallel processing capabilities of multiple processors across nodes. Below are the most important aspects of this implementation:

1. **Initialization of MPI environment:** The MPI environment was initialized using "MPI_Init", and each process determined its unique rank within the communicator using "MPI_Comm_rank", the total number of processes of the specified communicator is given by "MPI_Comm_size". In this implementation the "MPI_COMM_WORLD" is the default communicator that includes all the processes in the MPI program.

```
1  int rank, size;  
2  MPI_Init(NULL, NULL);  
3  MPI_Comm_rank(MPI_COMM_WORLD, &rank);  
4  MPI_Comm_size(MPI_COMM_WORLD, &size);  
5
```

To clarify the meaning of the variables "rank" and "size": "rank" serves as a unique identifier for each process, while "size" represents the total number of processes participating in the communicator.

2. **Distinct set of coins:** Each process was assigned a unique starting point by adding its rank to the last block of the coin. This ensured that all processes generated distinct set of coins, avoiding overlap and redundancy.

```
1  coin[12] += rank;  
2
```

3. **Coin Search and Hashing:** Each process independently computes the *MD5* hashes for its assigned coins using the "md5_cpu()" function, similar to the "cpu_search" implementation. The key operations, such as byte-reversal and counting trailing zeros, remained unchanged.

```
1 md5_cpu(coin , hash);  
2 hash_byte_reverse(hash);  
3 n = deti_coin_power(hash);  
4
```

4. **Aggregation of Results:** After all processes complete their searches, "MPI_Reduce" is used to aggregate the results, summing up the total number of coins found and the total attempts made across all processes. This step is performed on the root process (rank 0).

```
1 MPI_Reduce(&n_coins, &total_coins, 1, MPI_UNSIGNED_LONG,  
MPI_SUM, 0, MPI_COMM_WORLD);  
2 MPI_Reduce(&n_attempts, &total_attempts, 1, MPI_UNSIGNED_LONG,  
MPI_SUM, 0, MPI_COMM_WORLD);  
3
```

5. **Output and finalization:** The root process is responsible for printing the aggregated results, while all other processes ensure that the coins are stored before the MPI environment is finalized.

```
1 if(rank == 0){  
2     printf("total_coins: %lu\n", total_coins);  
3     printf("total_attempts: %lu\n", total_attempts);  
4 }  
5  
6 STORE_DETI_COINS();  
7  
8 MPI_Finalize();  
9
```

6.2 Results and Conclusions

The results obtained for this implementation were the following:


```

aad31@banana:~/Bruno/AAD-Project01$ mpirun -np 8 ./deti_coins_intel_mpi -s8 3600
searching for 3600 seconds using deti_coins_cpu_mpi_search()
searching for 3600 seconds using deti_coins_cpu_mpi_search()
searching for 3600 seconds using deti_coins_cpu_mpi_search()
searching for 3600 seconds using deti_coins_cpu_mpi_search()
searching for 3600 seconds using deti_coins_cpu_mpi_search()
searching for 3600 seconds using deti_coins_cpu_mpi_search()
searching for 3600 seconds using deti_coins_cpu_mpi_search()
n_coins: 10
total_attempts: 39813295906
n_coins: 9
total_attempts: 40059368221
n_coins: 8
total_attempts: 39848926706
n_coins: 9
total_attempts: 39918783318
n_coins: 12
total_attempts: 40025847243
n_coins: 9
total_attempts: 39898142830
n_coins: 14
total_attempts: 39825030657
n_coins: 14
total_attempts: 39912067472
total_coins: 85
total_attempts: 319301462353

```

Figure 6.1: MPI search results

We left the program running for 1 hour with 8 active processes, and we got a total of 85 coins, which is a great improvement when compared to the basic "cpu_search", in which we only obtained 10 coins. In the terminal we can see the number of coins that each process found and the number of attempts, the last two prints represent the global statistics of the program, which is the sum of all the coins and of all the attempts per process.

The MPI solution dramatically improves performance by leveraging parallelism. While the core logic of hashing and coin validation remains consistent, the distributed nature of MPI significantly accelerates the search process by spreading the computational load.

Chapter 7

CUDA

7.1 CUDA Implementation

The CUDA implementation of the DETI coin search algorithm leverages NVIDIA GPUs to parallelize the MD5 hash computation, significantly accelerating the process. Each GPU thread computes the hash of a single coin, enabling a massive number of attempts in parallel.

1. DETI Coin Initialization:

- The first bytes define the string "DETI coin"
- Two 32-bit random words (v1 and v2) are appended to create variability across runs
- Each thread also modifies certain bits of the coin to avoid repeated computations
- The coin is ended with "\n"

```

1      n = (u32_t)threadIdx.x + (u32_t)blockDim.x * (u32_t)
      blockIdx.x;
2      coin[ 0] = 0x49544544u; // "DETI"
3      coin[ 1] = 0x696f6320u; // "coi "
4      coin[ 2] = 0x6e20206eu; // "n  "
5      coin[ 3] = 0x20202020u;
6      coin[ 4] = 0x20202020u;
7      coin[ 5] = 0x20202020u;
8      coin[ 6] = 0x20202020u;
9      coin[ 7] = 0x20202020u;
10     coin[ 8] = 0x20202020u;
11     coin[ 9] = 0x20202020u;
12     coin[10] = v1;
13     coin[11] = v2;
14     coin[12] = 0x0a202020u; // "\n  "
15     coin[ 4] += (n % 64) << 0; n/=64;
16     coin[ 4] += (n % 64) << 8; n/=64;
17     coin[ 4] += (n % 64) << 16; n/=64;
18     coin[ 4] += (n % 64) << 24;
19     for(n = 0; n < 64; n++){ // Thread-specific
      modifications
20

```

2. Parallel Hash Calculation: The kernel, `deti_coins_cuda_kernel_search`, executes in parallel, with each thread:

- Computing the MD5 hash for using a custom MD5 message-digest
- Checking the resulting hash has at least 32 zeros in binary representation
- If this condition is met, storing the coin in a shared storage array using an atomic operation to prevent race conditions.

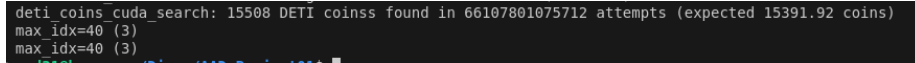
```

1      if (hash[3] == 0) { // Hash condition for a valid DETI
      coin
2      u32_t n = atomicAdd(storage_area, 13);
3      if (n + 13 <= 1024) {
4          for (int i = 0; i < 13; i++) {
5              storage_area[n + i] = coin[i];
6          }
7      }
8  }
9      coin[12] += 1 << 16; // Increment for the next attempt
10

```

3. Host-Device Communication: The host program manages the GPU:
 - (a) Transfers data between the host and device using `cuMemcpyHtoD` and `cuMemcpyDtoH`
 - (b) Invokes the CUDA kernel with the appropriate configuration, processing coins in parallel.
 - (c) Periodically retrieves results, storing valid coins and introducing variability for subsequent attempts.

7.2 Results and Conclusions



```
deti_coins_cuda_search: 15508 DETI coinss found in 66107801075712 attempts (expected 15391.92 coins)
max_idx=40 (3)
max_idx=40 (3)
```

Figure 7.1: CUDA search results

When tested on the banana server, in a one-hour run, 15508 coins were found in approximately 6.61×10^{13} attempts.

The CUDA implementation achieves a higher throughput than the various AVX searches because of a higher degree of parallelism, which we can see from the results of the picture above. The efficiency provided by atomic operations also ensured efficient data sharing and conflict resolution, which resulted in a scalable solution that can be adapted by more powerful GPU's

Chapter 8

WebAssembly

8.1 WebAssembly implementation

The WebAssembly implementation of the DETI coin search algorithm focuses on providing high portability and performance in browser environments. While WebAssembly (WASM) cannot leverage the parallel processing power of GPUs, its efficiency in utilizing CPU resources still allows significant performance improvements over interpreted JavaScript.

1. DETI Coin Initialization:

- The initialization process for coins is similar to the other implementations, ensuring compatibility and reproducibility of results.
- The string "DETI coin" forms the base of the coin, with two 32-bit random values (v1 and v2) introducing variability.
- Each WASM thread (executed sequentially within the JavaScript runtime) further modifies bits to avoid repeated computations.

8.2 Results and Conclusions

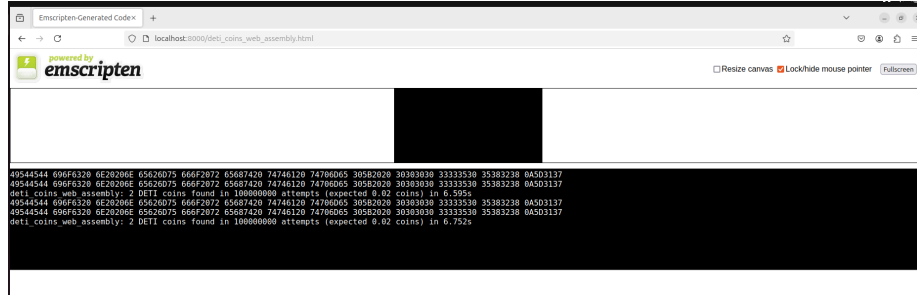


Figure 8.1: WebAssembly Results

During the testing phase, the WASM implementation ran for 6.596 seconds and found 2 coins, giving us the following result

1. Attempts per second:

$$\frac{\text{Total Attempts}}{\text{Execution Time}} = \frac{100,000,000}{6.596} \approx 15,160,710 \text{ attempts/s.}$$

Author's Contribution

Each member contributed equally to the development of this project, therefore, we decided to allocate 50% to each.

Acrónimos

SIMD Single Instruction Multiple Data

AVX Advanced Vector Extensions

WASM WebAssembly

MPI Message Passing Interface