DETI COINS
CSAC Project

Henrique Freitas - 114990
João Pinto - 104384
Gabriel - 113682

27.12.2024

DETI COIN

FORMAT: DETI coin CsAC 2024 AAD14

HEXADECIMAL: (44 45 54 49 20 63 6F 69 6E 20 43 73 41 43 20 32 30 32 34 20 41 41 44 31 34)

- A DETI coin is a 52-byte file whose MD5 hash ends with at least 8 hexadecimal zeros (last 32 bits).
- The file must start with "DETI coin " and end with a newline (\n), with the remaining bytes encoding arbitrary (preferably UTF-8) text.
- The "power" of a DETI coin is defined by the number of trailing zero bits in its MD5 hash. Higher-power coins are rarer and more valuable.
- Search functions using CPU_SPECIAL, AVX/AVX2, OPENMP, MPI, CUDA and Web Assembly.

MD5 (Message digest algorithm 5)

Cryptographic hash function that produces a 128 bit hash value (32 char hexadecimal number)

“Henrique wanted a more secure alternative”

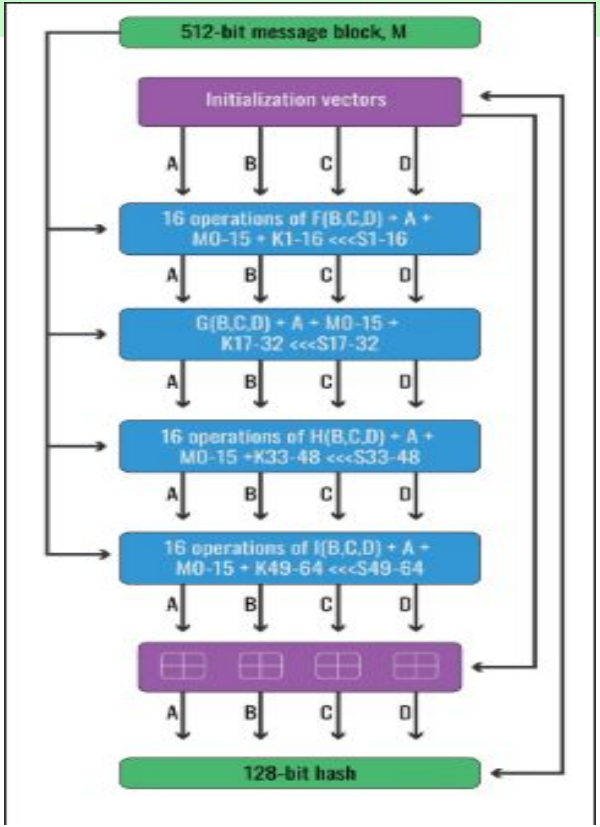
How does it work: Takes an input, processes it through several rounds, produce fixed size output.

Why do we use it: We use it to search “DETI coins”, generating specific MD5 hashes and looking for those patterns.

Example: “deti_coins_cpu_avx_search()”

```
coins[ 0 * N_LANES + lane] = 0x49544544; // ITED
coins[ 1 * N_LANES + lane] = 0x696f6320; // ioc_
coins[ 2 * N_LANES + lane] = 0x7343206e; // sC_n
coins[ 3 * N_LANES + lane] = 0x30324341; // 02CA
coins[ 4 * N_LANES + lane] = 0x41203432; // A_42
coins[ 5 * N_LANES + lane] = 0x34314441; // 41DA
coins[ 6 * N_LANES + lane] = 0x08200841 + (lane);
coins[ 7 * N_LANES + lane] = 0x08200820; //
coins[ 8 * N_LANES + lane] = 0x08200820; // 41DA
coins[ 9 * N_LANES + lane] = 0x08200820; // 41DA
coins[10 * N_LANES + lane] = 0x08200820; // 41DA
coins[11 * N_LANES + lane] = 0x08200820; // 41DA
coins[12 * N_LANES + lane] = 0x0A200820; // 41DA
```

```
md5_cpu_avx((v4si *)coins,(v4si *)hash);
//
//
//
for (u32_t lane= 0u; lane < N_LANES; lane++)
{
  if(hash[3 * N_LANES + lane] == 0)
  {
    for(u32_t idx = 0;idx < 13;idx++) coin[idx] = coins[idx * N_LANES + lane];
    save_deti_coin(coin);
    n_coins++;
    printf("FOUND ONE: %52.52s",(char *)coin);
    fflush(stdout);
  }
}
```


512-bit message block, M
Initialization vectors
A
B
C
D
16 operations of F(B,C,D) + A + M0-15 + K1-16 <<<S1-16
G(B,C,D) + A + M0-15 + K17-32 <<<S17-32
16 operations of H(B,C,D) + A + M0-15 +K33-48 <<<S33-48
16 operations of I(B,C,D) + A + M0-15 + K49-64 <<<S49-64
128-bit hash

AVX (Advanced Vector Extensions)

AVX (set of SIMD (Single Instruction Multiple Data) developed by Intel, that allows parallel processing of multiple data points with a single instruction. (Enables higher performance for mathematical computations)

Different vector sizes: (128, 256, 512 bits)

We use AVX to speed up the MD5 hashing process and parallelize computations.

We enable the parallel computation of hashes for multiple coin values at once. (Computing hashes for different combinations of coin values)

AVX

- Here we process 4 coins at once. Each AVX instruction computes 4 hashes.
- “v4si” -> vector of 4 integers (16 bytes) -> basic unit for AVX operations
- “interleaved4_data” -> contains the data of 4 coins (EACH LANE CONTAINS ONE COIN)
- “interleaved4_hash” -> stores de MD5 hash for the 4 coins

```
static void md5_cpu_avx(v4si *interleaved4_data,v4si *interleaved4_hash)
{ // four interleaved messages → four interleaved MD5 hashes
  v4si a,b,c,d,interleaved4_state[4],interleaved4_x[16];
# define C(c)         (v4si){ (int)(c),(int)(c),(int)(c),(int)(c) }
# define ROTATE(x,n)  (__builtin_ia32_pslldi128(x,n) | __builtin_ia32_psrldi128(x,32 - (n)))
# define DATA(idx)    interleaved4_data[idx]
# define HASH(idx)    interleaved4_hash[idx]
# define STATE(idx)   interleaved4_state[idx]
# define X(idx)       interleaved4_x[idx]
  CUSTOM_MD5_CODE();
```

AVX

- Explain the loops and lanes.

```
md5_cpu_avx((v4si *)coins,(v4si *)hash);
//
//
//
for (u32_t lane= 0u; lane < N_LANES; lane++)
{
  if(hash[3 * N_LANES + lane] == 0)
  {
    for(u32_t idx = 0;idx < 13;idx++) coin[idx] = coins[idx * N_LANES + lane];
    save_deti_coin(coin);
    n_coins++;
    printf("FOUND ONE: %52.52s",(char *)coin);
    fflush(stdout);
  }
}
```

```
#else
    u32_t idx = 6u;
    if((coins[idx*N_LANES + 0] & 0x00FF0000 ) != 0x007E0000)
       coins[idx*N_LANES + 0] += 0x00010000;
    else
    {
      coins[idx*N_LANES + 0] += 0xFFA20000; // UV7EWXYZ + FFA20000 = UV20WXYZ
      for(idx = 7;idx < 12;idx++)
        {
          if((coins[idx*N_LANES + 0] & 0x000000FF) != 0x0000007E)
          {
            coins[idx*N_LANES + 0] += 0x00000001;
            break;
          }
          if((coins[idx*N_LANES + 0] & 0x00FF0000) != 0x007E0000)
          {
            coins[idx*N_LANES + 0] += 0x0000FFA2;
            break;
          }
          coins[idx*N_LANES + 0] += 0xFFA1FFA2;
        }
    }
    for(;idx > 6;idx--)
      for (u32_t lane = 1; lane < N_LANES; lane++)
        coins[idx*N_LANES + lane] = coins[idx*N_LANES + 0];
    for (u32_t lane = 1; lane < N_LANES; lane++)
      coins[idx*N_LANES + lane] = coins[idx*N_LANES + 0] + lane;
#endif
  }
  STORE_DETI_COINS();
  printf("deti_coins_cpu_avx_search: %lu DETI coin%s found in %lu attempt%s (expected %.2f coins)\n"
# undef N_LANES
}
```

AVX2 vs AVX

AVX: Single precision floating point and double precision operations

AVX2: Advanced integer processing, gather operations, new vectorized operations, enhanced memory access patterns, more efficient handling of vectorized data

So it’s basically AVX but with some upgrades.

AXV2

In AVX2 we're able to work with type int32_t and int64_t meaning we can use 8 lanes, because, due to the vectorized integer operations in AVX2 we can work with 8 32-bit integers instead of 4 64-bit. (This to make the 256-bit wide vectors)

```
# define N_LANES 8u
```

AXV2 NOTES

It works with 8 32-bits but not 16 16-bits

Deti coin -> 52 bytes 11 bytes (letters) 40 bytes (arbitrary data) 1 byte newline

32 bits are zeros

MD5 works in 512 bits (64 bytes) AXV2 can process smaller blocks like 256 bits

8 32-bits can process multiple parts of the deti coin at the same time (8 lanes)

Since MD5 expect 32 bit words I'd have to combine two 16-bit values, these operations are usually done in 256 bit registers.

It wouldn't be optimal using 16 bits, but not impossible.

For example AVX2 supports 64 bit operations like vaddq_u64 so it can break down 64 bits into two 32-bits but doesn't support 16 bit operations.

Only handles 8 32-bit registers

AVX2

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

static void md5_cpu_avx2(v8si *interleaved4_data,v8si *interleaved4_hash)
{ // four interleaved messages -> four interleaved MD5 hashes
  v8si a,b,c,d,interleaved4_state[4],interleaved4_x[16];
# define C(c)         (v8si){ (int)(c),(int)(c),(int)(c),(int)(c),(int)(c),(int)(c),(int)(c),(int)(c) }
# define ROTATE(x,n)  (__builtin_ia32_pslldi256(x,n) | __builtin_ia32_psrldi256(x,32 - (n)))
# define DATA(idx)    interleaved4_data[idx]
# define HASH(idx)    interleaved4_hash[idx]
# define STATE(idx)   interleaved4_state[idx]
# define X(idx)       interleaved4_x[idx]
  CUSTOM_MD5_CODE();
# undef C
# undef ROTATE
# undef DATA
# undef HASH
# undef STATE
# undef X
```

CUDA (Compute Unified Device Architecture)

Parallel computing platform and programming model.
It allows programmers to use the GPU to perform computing tasks.
Key components: CUDA Toolkit; CUDA C/C++

OpenMP (Open Multi-Processing)

What is it: It's an API that supports multi-platform shared-memory parallel programming.

It allows you to run parts of your program in parallel across multiple CPU cores. Therefore it helps divide the workload so that different threads can execute simultaneously. (No need for Semaphores or anything)

We have "parallel regions" #pragma omp parallel

For example: "#pragma omp parallel for" will parallelize a for loop, meaning each iteration of the loop will be executed by a different thread.

OpenMP

- Explain:

```
#ifdef DETI_COINS_CPU_OpenMP_SEARCH
      case '7':
        printf("searching for %u seconds using deti_coins_cpu_OpenMP_search()\n",seconds);
        fflush(stdout);

        unsigned long total_coins = 0;
        unsigned long total_attempts = 0;

        // Parallel region with reduction to sum the totals across threads
        # pragma omp parallel num_threads(n_threads) reduction(+:total_coins, total_attempts)
          { // automatic variable are local to the thread
            int thread_number = omp_get_thread_num();
            unsigned long n_coins = 0;
            unsigned long n_attempts = 0;

            deti_coins_cpu_OpenMP_search(thread_number, &n_coins, &n_attempts);

            total_coins += n_coins;
            total_attempts += n_attempts;
          }

        // Final aggregated results
        double total_expected_coins = (double)total_attempts / (1ul << 32);
        printf("Total: %lu DETI coin%s found in %lu attempt%s (expected %.2f coins)\n",
              total_coins, (total_coins == 1ul) ? "" : "s", total_attempts,
              (total_attempts == 1ul) ? "" : "s", total_expected_coins);

        STORE_DETI_COINS();
        break;
#endif
```

```
static void deti_coins_cpu_OpenMP_search(int thread_number, unsigned long *out_n_coins, unsigned long *out_n_attempts)
```

MPI (Message passing interface)

- It focuses on distributed memory systems (each processor, has its own private memory).
- Used in parallel programming.
- It allows processes to communicate with each other by passing messages.

Each processor has its own local memory, and data is exchanged between processors using MPI functions.

MPI_Send, MPI_Recv, etc.

You need to specify how data will be transferred.

We preferred to do MPI over Client Server.

MPI and OpenMP

They're used together in hybrid parallel programming, combining each others benefits.

MPI handles the distribution of work across multiple nodes, and OpenMP handles parallelism within a single node, managing threads.

Feature	MPI	OpenMP
Memory Model	Distributed memory (message passing)	Shared memory (threads within a node)
Scope	Multi-node parallelism	Multi-core parallelism within a node
Granularity	Coarse-grained (process-level)	Fine-grained (thread-level)
Communication	Explicit communication (messages)	Implicit communication (shared memory)
Synchronization	Explicit (e.g., `MPI_Send`, `MPI_Recv`)	Implicit (e.g., `#pragma parallel for`)

MPI

MPI_Init initializes the MPI environment

MPI_Comm_size retrieves the total number of processes in the communicator

MPI_Comm_rank identifies the rank of the current process (ID)

MPI_Reduce aggregates data from all processes

MPI_Finalize to clean up and shutdown the environment

Phase	OpenMP Role	MPI Role
Initialization	Thread spawning	Process setup and rank determination
Computation	Thread-level parallelism	Process-level task division
Reduction	Local sum (reduction)	Global sum (MPI_Reduce)
Final Output	N/A	Root process displays results

```
// Global variables for total aggregation
unsigned long global_coins = 0;
unsigned long global_attempts = 0;

// Use MPI_Reduce to aggregate results from all processes to rank 0
MPI_Reduce(&local_coins, &global_coins, 1, MPI_UNSIGNED_LONG, MPI_SUM, 0, MPI_COMM_WORLD);
MPI_Reduce(&local_attempts, &global_attempts, 1, MPI_UNSIGNED_LONG, MPI_SUM, 0, MPI_COMM_WORLD);
```

```
//
// initialize the MPI environment, and get the number of processes and the MPI number of our process (the rank)
//
MPI_Init(&argc,&argv);
MPI_Comm_size(MPI_COMM_WORLD,&n_processes);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
```

Web Assembly

Binary instruction format, portable and efficient, for high level programming languages.

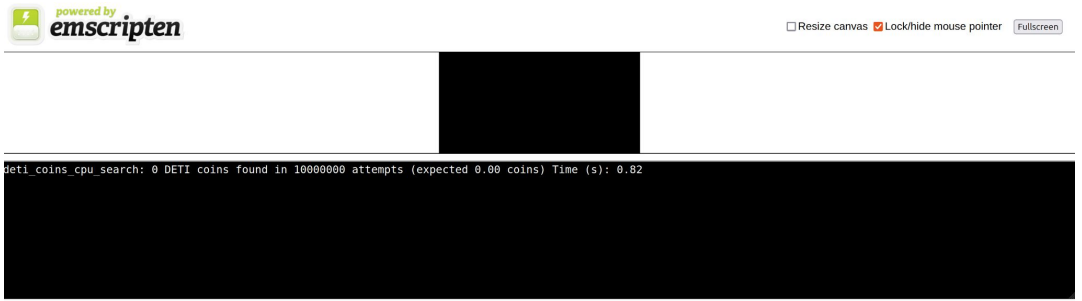
It enables running code in web browsers at near native speed.

It’s not limited to the browser.

(You just take the file and run it in the web)

python3 -m http.server 8000

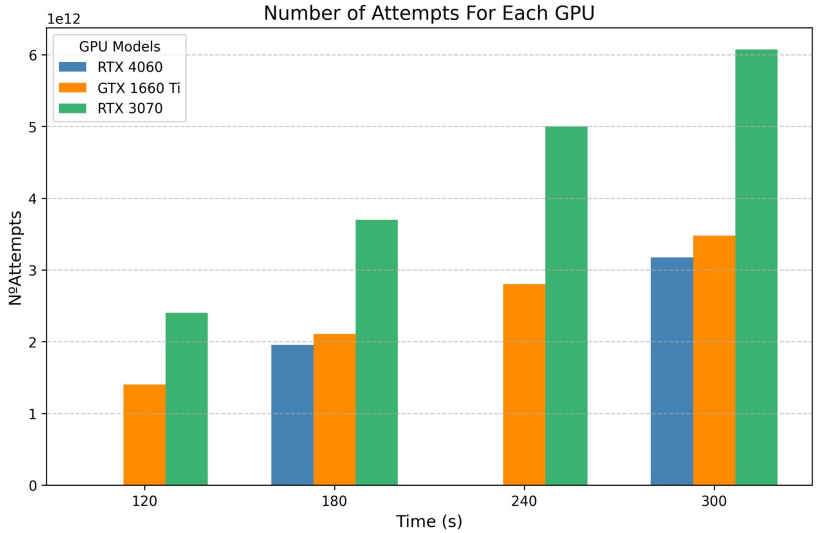
web_assembly.html

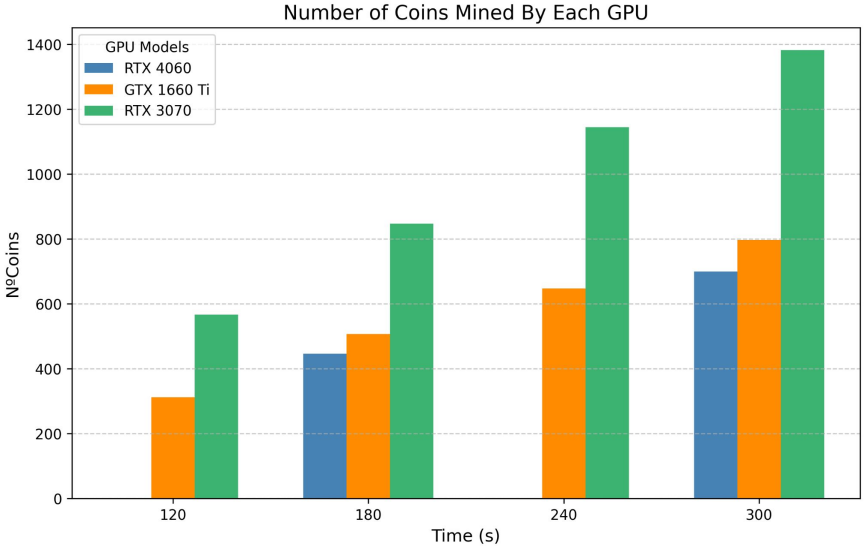


TESTS

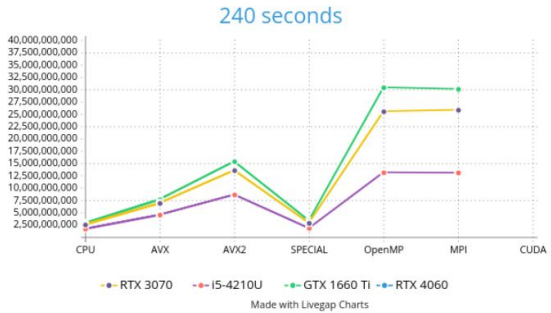
RTX 3070 - Ryzen 5600 (Capped at 220W)

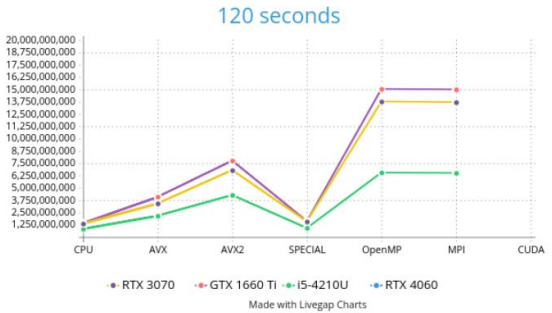
RTX 4060 - i9-13980HX

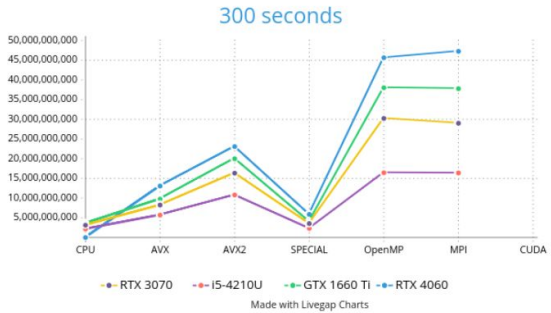


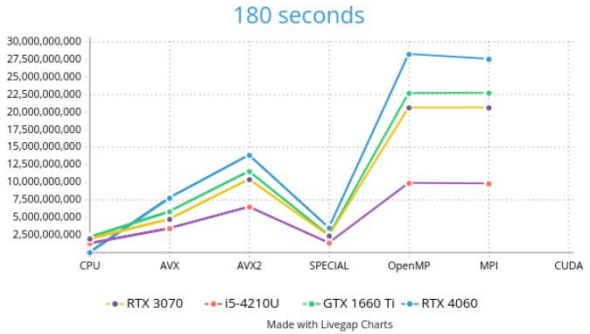


TESTS


240 seconds
40,000,000,000
37,500,000,000
35,000,000,000
32,500,000,000
30,000,000,000
27,500,000,000
25,000,000,000
22,500,000,000
20,000,000,000
17,500,000,000
15,000,000,000
12,500,000,000
10,000,000,000
7,500,000,000
5,000,000,000
2,500,000,000
CPU
AVX
AVX2
SPECIAL
OpenMP
MPI
CUDA
RTX 3070
i5-4210U
GTX 1660 Ti
RTX 4060
Made with Livegap Charts

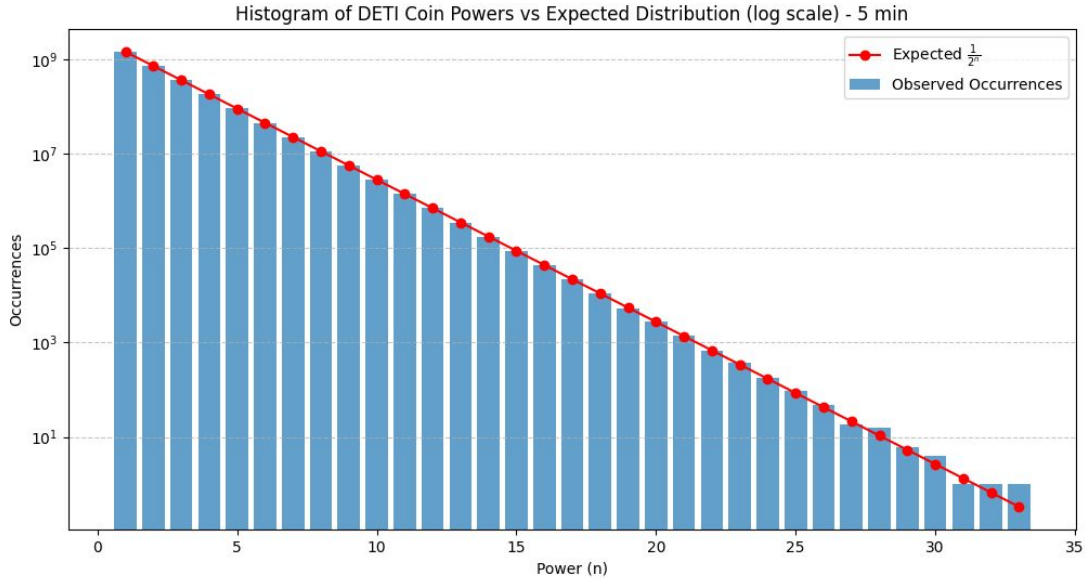

120 seconds
20,000,000,000
18,750,000,000
17,500,000,000
16,250,000,000
15,000,000,000
13,750,000,000
12,500,000,000
11,250,000,000
10,000,000,000
8,750,000,000
7,500,000,000
6,250,000,000
5,000,000,000
3,750,000,000
2,500,000,000
1,250,000,000
CPU
AVX
AVX2
SPECIAL
OpenMP
MPI
CUDA
RTX 3070
GTX 1660 Ti
i5-4210U
RTX 4060
Made with Livegap Charts


300 seconds
50,000,000,000
45,000,000,000
40,000,000,000
35,000,000,000
30,000,000,000
25,000,000,000
20,000,000,000
15,000,000,000
10,000,000,000
5,000,000,000
CPU
AVX
AVX2
SPECIAL
OpenMP
MPI
CUDA
RTX 3070
i5-4210U
GTX 1660 Ti
RTX 4060
Made with Livegap Charts


180 seconds
30,000,000,000
27,500,000,000
25,000,000,000
22,500,000,000
20,000,000,000
17,500,000,000
15,000,000,000
12,500,000,000
10,000,000,000
7,500,000,000
5,000,000,000
2,500,000,000
CPU
AVX
AVX2
SPECIAL
OpenMP
MPI
CUDA
RTX 3070
i5-4210U
GTX 1660 Ti
RTX 4060
Made with Livegap Charts

Extra Stuff - 11 a)

- **Histogram** of the values returned by *deti_coin_power()* during a CPU search.
- This function compute the DETI coin power (number of trailing zero bits of the byte-order reversed MD5 hash). Hash with *n* trailing zeros: P(power=n) = $1/2^n$.



n	Occurrences
1	1429595418
2	714797812
3	357423072
4	178720347
5	89356156
6	44670167
7	22333690
8	11174039
9	5586503
10	2794137
11	1399170
12	697614
13	348340
14	174245
15	87448
16	43691
17	21705
18	10944
19	5324
20	2736
21	1380
22	659
23	365
24	174
25	97
26	48
27	18
28	16
29	6
30	4
31	1
32	1
33	1