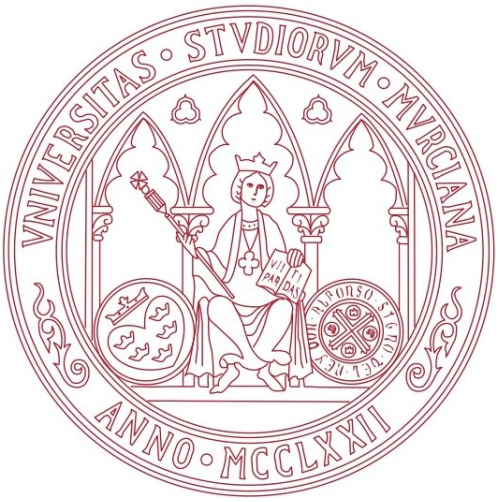

ARQUITECTURA Y ORGANIZACIÓN DE COMPUTADORES

Informe sobre el análisis del ancho de banda de caché

Alumno: Alejandro Carmona Martínez

Correo: alejandro.carmonam@um.es

Tutor: José Manuel García Carrasco

Curso Académico: 2022/23

Fecha: 26/06/2023

Índice

1. Introducción

Durante el desarrollo de la asignatura de Arquitectura y Organización de Computadores se estudian conceptos que ayudan a entender el funcionamiento real de un ordenador y obtener su máximo rendimiento, con el objetivo de que el alumno sea capaz de aplicar este conocimiento tanto si prosigue en la mención de Ingeniería de Computadores, como si realiza cualquier actividad en el campo de la Ingeniería Informática.

Un ejemplo de estos conceptos es el ancho de banda de memoria. Este concepto aparece explicado teóricamente en el Tema 1. Arquitectura de un multiprocesador-en-un-chip [1] y, de forma práctica, en la Práctica 1 - Introducción a la computación de alto rendimiento [2]. Según la Práctica 1, definimos el ancho de banda como *el número de bytes por segundo que es posible transferir entre la memoria y el procesador.* Gracias a la siguiente fórmula podemos obtener el ancho de banda de la memoria RAM de un ordenador:

$$AnchoBandaPico = controladoresMemoria \times canales/controlador \times bytes/transferencia \times frecuenciaMemoria$$

No obstante, tanto en la práctica como en la teoría, solo atacamos este concepto para la memoria principal, dejando de lado el ancho de banda de la memoria caché, este informa pretende cubrir ese espacio, de forma que también podamos estudiar el ancho de banda de la jerarquía de cachés.

Para ello vamos a utilizar herramientas externas, tales como el benchmark multipropósito CPU-X y herramientas naturales de la asignatura, en particular, se dará uso al programa peak-mem encontrado en la Práctica 1 facilitado por los profesores de la asignatura. CPU-X será nuestra referencia para obtener diferentes medidas de ancho de banda de la caché y modificaremos peak-mem con el objetivo de adaptarlo a las diversas pruebas que vamos a ejecutar.

2. CPU-X

2.1. ¿Qué es CPU-X?

CPU-X es un programa que proporciona al usuario conocimiento a cerca de la información básica sobre componentes y sistemas del ordenador tales como CPU, **memoria caché**, placa base, sistema operativo o subsistema gráfico.

Un proyecto de código abierto escrito en C y ensamblador entre otros lenguajes, CPU-X ofrece, en resumidas cuentas, una versión portátil en Linux del programa de Windows CPU-Z. Aunque en aplicaciones similares tales como MemTest86, es preciso realizar la instalación del programa en una memoria USB flash y ejecutarlo desde apagado, CPU-X es una aplicación de escritorio disponible con una interfaz gráfica simple, pero intuitiva, que aunque se puede instalar en un USB, permite ser instalada y ejecutada de forma sencilla en nuestro sistema Linux [4]:

Ubuntu:

```
sudo apt update
sudo apt install cpu-x
```

Para instalar CPU-X desde el código fuente en Linux:

```
sudo apt update
sudo apt install git cmake libgtk-3-dev libcpuid-dev libncurses5-dev
   libprocps-dev libpci-dev libatasmart-dev dmidecode
git clone https://github.com/X0rg/CPU-X.git
cd CPU-X
mkdir build
cd build
cmake ..
make
sudo make install
```

Una vez instalado, ejecutarlo simplemente requiere ejecutarlo desde una terminal. Algunas de sus funcionalidades requieren permisos de superusuario, por lo que se recomienda ejecutarlo con sudo:

Ejecución de CPU-X:

```
sudo cpu-x
```

2.2. ¿Cómo usar CPU-X?

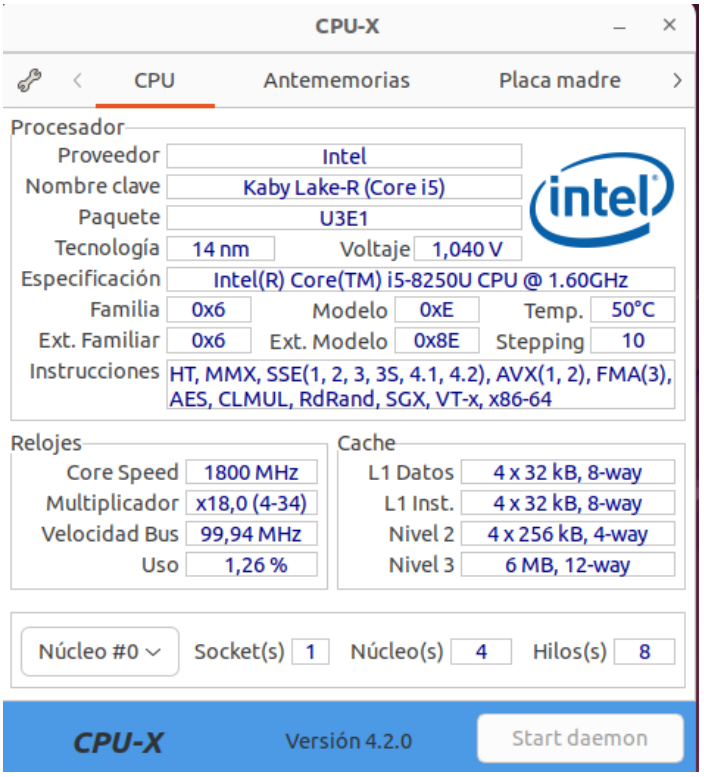

Figura 1: Interfaz CPU-X

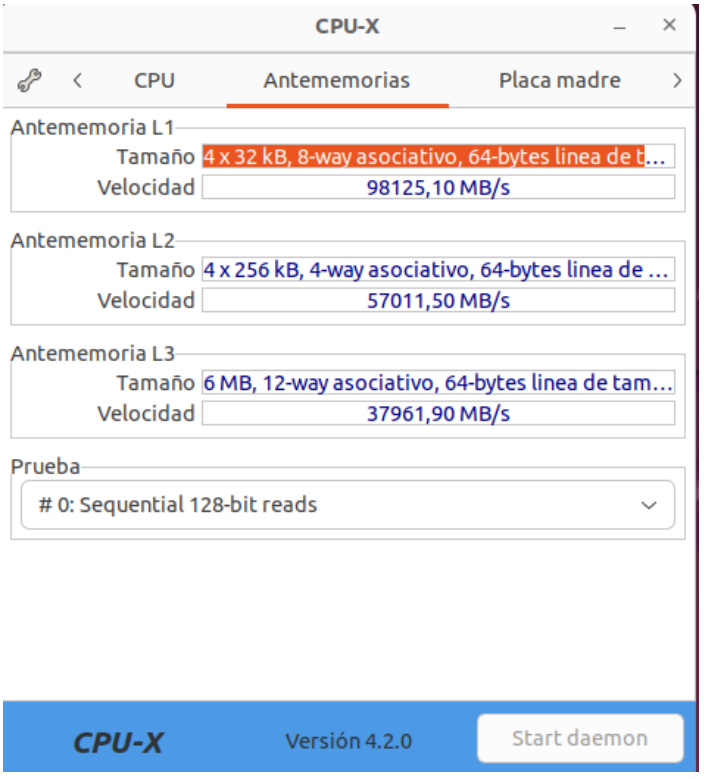

Figura 2: Antememorias: Benchmark de jerarquía caché

Al iniciar CPU-X podemos observar un menú que muestra información acerca de nuestra cpu de forma similar a lo que hace el comando *lscpu*, sin embargo, para ver los resultados del benchmark de memoria caché en tiempo real, debemos ir al apartado *Antememorias* y ahí aparecerán varios apartados para cada nivel de caché informando de su tamaño, asociatividad y, de forma más relevante para nuestro informe, la velocidad del ancho de banda.

Es en este apartado donde pasaremos el mayor tiempo junto a CPU-X, ya que no solo tenemos un benchmark como en otros programas, sino que en el apartado *Prueba* se nos permite cambiar el tipo de benchmark de una serie de generosas opciones, pudiendo elegir tipo de acceso secuencial o aleatorio y tipo de prueba: copia, lectura o escritura con un tamaño de entre 8 y 256 bits dependiendo del tipo. Es importante indicar que no algunas combinaciones no están soportadas, por ejemplo, no hay copias de 64 bits o escrituras de 32 bits.

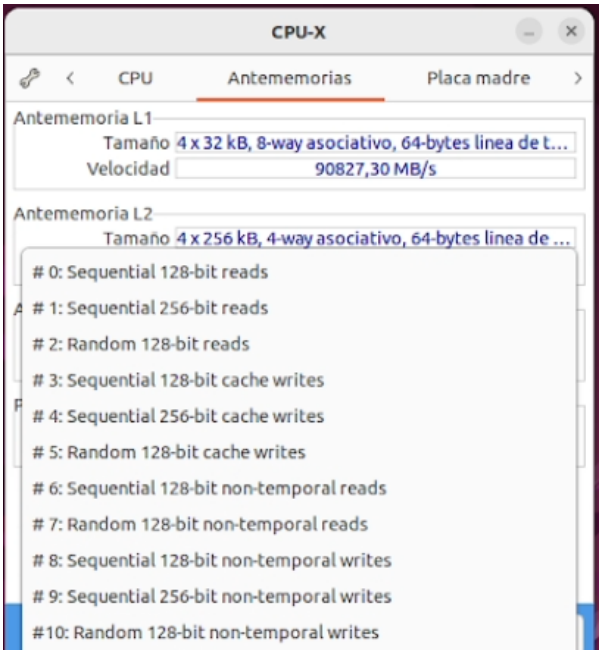

Figura 3: Selección de los 10 primeros tipos de benchmarks para caché en CPU-X

En conclusión, de cara al siguiente apartado, el gran abanico de posibles tests que nos propone CPU-X, nos va a permitir realizar comparaciones entre nuestros programas escritos íntegramente en c++ y un benchmark probado que ofrece resultados de alta fiabilidad, además de un uso fácil e intuitivo.

3. Tests

En primer lugar, vamos a separar las pruebas para obtener el ancho de banda de caché en dos, pruebas de acceso secuencial y aleatorio. Esta distinción es muy importante, ya que, utilizar una u otra para las mismas condiciones exactas lleva a resultados vastamente distintos.

Vamos a centrarnos en el tamaño de 64 bits ejecutando tests de copias, lecturas y escrituras. Estos tests basados en el programa *peak-mem* adaptan el tamaño de un array para que sea de un tamaño adecuado para el nivel de caché indicado y, a continuación, copian, leen o escriben dicho array cambiando la función *calculate()* en consonancia con el objetivo de evaluar el ancho de banda de la caché.

Generalmente, todos los tests comparten una serie de similitudes, por ejemplo, al comienzo de cada programa hay una serie de constantes definidas en las que se indica el tamaño de cada uno de los tres niveles de caché que el usuario debe adaptar a su sistema. Debemos indicar el tamaño de las caché de cada core y no en total, salvo que sea L3, puesto que, CPU-X utiliza un solo hilo y el objetivo de este benchmark es evaluar el ancho de banda de una caché, no el ancho de banda usando las 4 cachés (una por core). Además, los tests seguirán una estructura similar compartiendo las funciones: *init(), calculate(), measure()* y *executeTest()* cada una adaptada al tipo de test que el programa ejecute.

Para ejecutar los tests existe en un makefile similar al de *peak-mem*, pero en el que se recogen todos los test *peak-mem* que se van a explicar a continuación. Con el objetivo de facilitar la ejecución, se proporciona un script para el compilador gcc *(gcc-run-peak-mem.sh)* en el cual se puede sustituir el peak-mem-cache predeterminado (*peak-mem-random-read-64-a*), por alguno de los programas disponibles o ir sobreescribiendo el archivo *peak-mem*. Vamos a utilizar gcc junto al script *gcc-run-peak-mem.sh* durante el resto de test debido a que los otros compiladores presentan pocas diferencias en algunos casos y dan resultados incoherentes o incorrectos en otros. En definitiva, el objetivo de este informe es ver como se comportan los diferentes test al interactuar con la caché y gcc es el compilador seleccionado debido a proporcionar los resultados más coherentes y precisos.

Nota: Todas las ejecuciones han de hacerse dentro del contendor docker de la asignatura.

3.1. Acceso secuencial a caché

3.1.1. Test de copia secuencial (peak-mem-seq-copy-128.cpp y peak-mem-seq-copy64)

```
1 //Benchmark: Sequential 128 bit copy test
2 #include "util.h"
3 #include <cstdio>
4 #include <cstring>
```

```
5  #include <memory>
6  #include <cmath>
7
8  using namespace std;
9
10 #define L1_CACHE_SIZE 32*1024  // Size of L1 cache to be
    ↪  tested in bytes.
11 #define L2_CACHE_SIZE 256*1024 // Size of L2 cache to be
    ↪  tested in bytes.
12 #define L3_CACHE_SIZE 6*1024*1024 // Size of L3 cache to be
    ↪  tested in bytes.
13 #define REDUCE_SIZE 0.4
14
15 struct {
16   size_t repeat_times = 7; // total, including warmup
17   size_t warmup_times = 2;
18   bool print_each_time = true;
19   size_t array_size; // size of the array in number of
    ↪  elements
20   size_t threads = 1;
21 } options;
22
23 long double* a = nullptr;
24 long double* b = nullptr;
25
26 void init() {
27   a = static_cast<long double*>(aligned_alloc(64,
    ↪  options.array_size * sizeof(a[0])));
28   b = static_cast<long double*>(aligned_alloc(64,
    ↪  options.array_size * sizeof(a[0])));
29   for (size_t i = 0; i < options.array_size; ++i) {
30     a[i] = i;
31   }
32 }
33
34 template<bool use_memcpy>
35 void calculate() {
36   long double *la = assume_aligned<64>(a);
37   long double *lb = assume_aligned<64>(b);
```

```
38    if (use_memcpy) {
39      size_t per_thread_elements = options.array_size /
    ↪   options.threads;
40      memcpy(lb + per_thread_elements * omp_get_thread_num(),
41             la + per_thread_elements * omp_get_thread_num(),
42             per_thread_elements * sizeof(a[0]));     // usando
    ↪   std::copy se obtienen los mismos resultados
43    } else {
44      for (size_t i = 0; i < options.array_size; ++i) {
45        lb[i] = la[i];
46      }
47    }
48  }
49
50  template<bool use_memcpy>
51  void measure() {
52    long n_bytes = 2 * options.array_size * sizeof(a[0]);
53    printf("Measuring time to copy %ld KiB with %zu threads
    ↪   %s:\n", n_bytes / 2 / 1024, options.threads, use_memcpy ?
    ↪   "using memcpy" : "using a loop");
54    vector<double> times;
55    vector<double> bps;
56
57    for (size_t i = 0; i < options.repeat_times; ++i) {
58      double elapsed_time =
    ↪   measure_time(calculate<use_memcpy>);
59      if (i >= options.warmup_times) {
60        times.push_back(elapsed_time);
61        bps.push_back(n_bytes / elapsed_time);
62      }
63      if (options.print_each_time) {
64        printf("    Run %2ld/%2ld: %7.2fs  %7.2f GiB/s %7.2f
    ↪   GB/s  %s\n", i + 1, options.repeat_times, elapsed_time,
    ↪   n_bytes / elapsed_time / (1024*1024*1024), n_bytes /
    ↪   elapsed_time / (1000*1000*1000), i < options.warmup_times
    ↪   ? "(warmup)" : "");
65      }
66    }
67
```

```
68    double average_time = vector_average(times);
69    double stddev_time = vector_stddev(times);
70    printf("Average time (s): %7.2f±%.2f      threads = %zu\n",
      ↪  average_time, stddev_time, options.threads);
71    double average_bps = vector_average_harmonic(bps);
72    double stddev_bps = vector_stddev_harmonic(bps);
73    printf("Average GiB/s:    %7.2f±%.2f      threads = %zu\n",
      ↪  average_bps / (1024*1024*1024), stddev_bps /
      ↪  (1024*1024*1024), options.threads);
74    printf("Average GB/s:     %7.2f±%.2f      threads =
      ↪  %zu\n\n", average_bps / (1000*1000*1000), stddev_bps /
      ↪  (1000*1000*1000), options.threads);
75  }
76
77  void executeTest(string cache_type, size_t array_size) {
78    printf("Testing %s cache\n", cache_type.c_str());
79    options.array_size = array_size;
80    init();
81    measure<false>();
82    measure<true>();
83    free(a);
84    free(b);
85  }
86
87  int main(int argc, char** argv) {
88    for (int i = 1; i < argc; ++i) {
89      if (!parse_bool_arg(argv[i], "print-each-time",
      ↪  options.print_each_time)
90          && !parse_size_arg(argv[i], "repeat-times",
      ↪  options.repeat_times)
91          && !parse_size_arg(argv[i], "warmup-times",
      ↪  options.warmup_times)
92          && !parse_size_arg(argv[i], "threads",
      ↪  options.threads)) {
93        fprintf(stderr, "Incorrect argument: %s\n", argv[i]);
94        return 1;
95      }
96    }
97
```

```
98  printf("Benchmark: Sequential 128 bit copy test\n");
99      executeTest("L1", (L1_CACHE_SIZE / sizeof(long double)) *
     ↪  REDUCE_SIZE);
100     executeTest("L2", (L2_CACHE_SIZE / sizeof(long double)) *
     ↪  REDUCE_SIZE);
101     executeTest("L3", (L3_CACHE_SIZE /sizeof(long double)) *
     ↪  REDUCE_SIZE);
102
103   return 0;
104 }
105
```

```
1  /************Sequential 64 bit copy test**************/
2
3
4
5  #include "util.h"
6  #include <cstdio>
7  #include <cstring>
8  #include <memory>
9  #include <cmath>
10
11 using namespace std;
12
13 #define L1_CACHE_SIZE 32*1024  // Size of L1 cache to be
     ↪  tested in bytes.
14 #define L2_CACHE_SIZE 256*1024 // Size of L2 cache to be
     ↪  tested in bytes.
15 #define L3_CACHE_SIZE 6*1024*1024 // Size of L3 cache to be
     ↪  tested in bytes.
16 #define REDUCE_SIZE 0.4
17
18 struct {
19   size_t repeat_times = 7; // total, including warmup
20   size_t warmup_times = 2;
21   bool print_each_time = true;
22   size_t array_size; // size of the array in number of
     ↪  elements
```

```
23   size_t threads = 1;
24 } options;
25
26 double* a = nullptr;
27 double* b = nullptr;
28
29 void init() {
30   a = static_cast<double*>(aligned_alloc(64,
     ↪  options.array_size * sizeof(a[0])));
31   b = static_cast<double*>(aligned_alloc(64,
     ↪  options.array_size * sizeof(a[0])));
32   for (size_t i = 0; i < options.array_size; ++i) {
33     a[i] = i;
34   }
35 }
36
37 template<bool use_memcpy>
38 void calculate() {
39   double *la = assume_aligned<64>(a);
40   double *lb = assume_aligned<64>(b);
41   if (use_memcpy) {
42     size_t per_thread_elements = options.array_size /
     ↪  options.threads;
43     memcpy(lb + per_thread_elements * omp_get_thread_num(),
44            la + per_thread_elements * omp_get_thread_num(),
45            per_thread_elements * sizeof(a[0]));     // usando
     ↪  std::copy se obtienen los mismos resultados
46   } else {
47     for (size_t i = 0; i < options.array_size; ++i) {
48       lb[i] = la[i];
49     }
50   }
51 }
52
53 template<bool use_memcpy>
54 void measure() {
55   long n_bytes = 2 * options.array_size * sizeof(a[0]);
```

```
56    printf("Measuring time to copy %ld KiB with %zu threads
    ↪  %s:\n", n_bytes / 2 / 1024, options.threads, use_memcpy ?
    ↪  "using memcpy" : "using a loop");
57    vector<double> times;
58    vector<double> bps;
59
60    for (size_t i = 0; i < options.repeat_times; ++i) {
61      double elapsed_time =
    ↪  measure_time(calculate<use_memcpy>);
62      if (i >= options.warmup_times) {
63        times.push_back(elapsed_time);
64        bps.push_back(n_bytes / elapsed_time);
65      }
66      if (options.print_each_time) {
67        printf("    Run %2ld/%2ld: %7.2fs  %7.2f GiB/s %7.2f
    ↪  GB/s  %s\n", i + 1, options.repeat_times, elapsed_time,
    ↪  n_bytes / elapsed_time / (1024*1024*1024), n_bytes /
    ↪  elapsed_time / (1000*1000*1000), i < options.warmup_times
    ↪  ? "(warmup)" : "");
68      }
69    }
70
71    double average_time = vector_average(times);
72    double stddev_time = vector_stddev(times);
73    printf("Average time (s): %7.2f±%.2f      threads = %zu\n",
    ↪  average_time, stddev_time, options.threads);
74    double average_bps = vector_average_harmonic(bps);
75    double stddev_bps = vector_stddev_harmonic(bps);
76    printf("Average GiB/s:    %7.2f±%.2f      threads = %zu\n",
    ↪  average_bps / (1024*1024*1024), stddev_bps /
    ↪  (1024*1024*1024), options.threads);
77    printf("Average GB/s:     %7.2f±%.2f      threads =
    ↪  %zu\n\n", average_bps / (1000*1000*1000), stddev_bps /
    ↪  (1000*1000*1000), options.threads);
78  }
79
80  void executeTest(string cache_type, size_t array_size) {
81    printf("Testing %s cache\n", cache_type.c_str());
82    options.array_size = array_size;
```

```
  init();
  measure<false>();
  measure<true>();
  free(a);
  free(b);
}

int main(int argc, char** argv) {
  for (int i = 1; i < argc; ++i) {
    if (!parse_bool_arg(argv[i], "print-each-time",
    ↪ options.print_each_time)
        && !parse_size_arg(argv[i], "repeat-times",
    ↪ options.repeat_times)
        && !parse_size_arg(argv[i], "warmup-times",
    ↪ options.warmup_times)
        && !parse_size_arg(argv[i], "threads",
    ↪ options.threads)) {
      fprintf(stderr, "Incorrect argument: %s\n", argv[i]);
      return 1;
    }
  }

printf("Benchmark: Sequential 64 bit copy test\n");
    executeTest("L1", (L1_CACHE_SIZE / sizeof(double)) *
    ↪ REDUCE_SIZE);
    executeTest("L2", (L2_CACHE_SIZE / sizeof(double)) *
    ↪ REDUCE_SIZE);
    executeTest("L3", (L3_CACHE_SIZE /sizeof(double)) *
    ↪ REDUCE_SIZE);

  return 0;
}
```

Estos programas son una adaptación directa de *peak-mem*, ya que tan solo cambia el tamaño del array a copiar según el tamaño de caché y el tipo del array. El tipo del array es interesante para copiar 64 bits (double) o 128 bits (long double), ya que, CPU-X solo tiene copia de 128 y 256 bits, por lo que adaptar el programa para compararlo con CPU-X resultaba interesante.

A continuación compararemos los resultados obtenidos con los de CPU-X e intentaremos

justificar las diferencias:

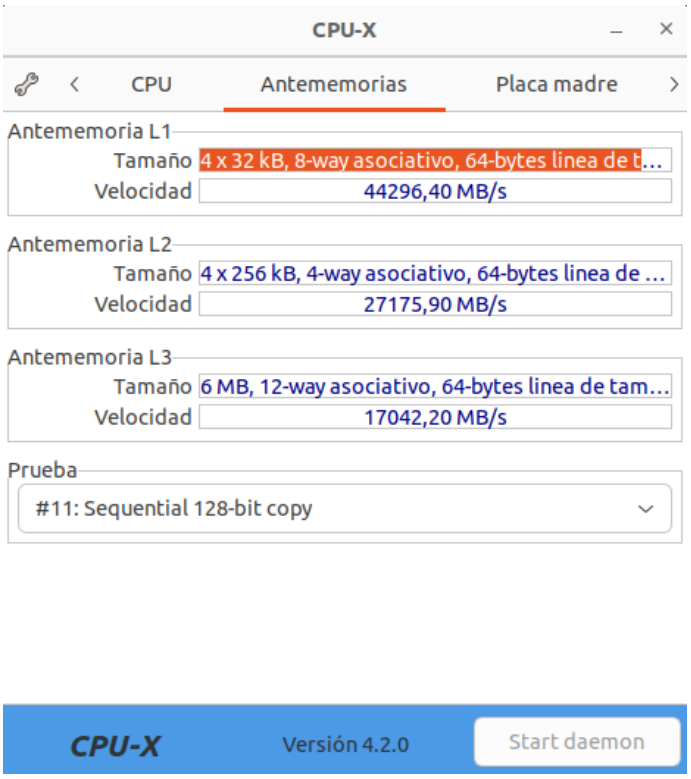

Figura 4: Resultados de CPU-X para copias secuenciales de 128 bits

	L1	L2	L3
Run 1	67,3	47,15	25,54
Run 2	50,52	29,07	40,8
Run 3	54,85	46,74	44,99
Run 4	55,38	29,23	46,1
Run 5	50,54	31,64	44,88
Media	55,72	36,77	40,46
Maximo	67,3	47,15	46,1
Mínimo	50,52	29,07	25,54

Figura 5: Resultados de bandwidth (GB/s) de peak-mem-seq-copy-128.cpp para copias secuenciales de 128 bits con memcpy

	L1	L2	L3
Run 1	92,94	55,6	41,49
Run 2	97,28	48,23	43,52
Run 3	62,7	49,54	48,67
Run 4	98,82	48,65	42,07
Run 5	55,71	47,8	41,86
Media	81,49	49,96	43,52
Maximo	98,82	55,6	48,67
Mínimo	55,71	47,8	41,49

Figura 6: Resultados de bandwidth (GB/s) de peak-mem-seq-copy64.cpp para copias secuenciales de 64 bits con memcpy

Para este caso vemos que tanto si copiamos utilizando 64 o 128 bits, el resultado sigue siendo superior al ofrecido por CPU-X, lo cual puede significar que se estén realizando optimizaciones en nuestro programa que no se realizan en CPU-X o que, aprovechándose de la naturaleza secuencial del programa, se esté realizando prefetching reduciendo los tiempos en memoria y aumentando el ancho de banda.

En cuando a *peak-mem-seq-copy64.cpp*, podemos apreciar que los resultados de ancho de banda son superiores a su contraparte de 128 bits y además más consistentes con la realidad, puesto que en las 5 ejecuciones del programa, este ha conseguido que L1 siempre resulte más rápida que L2 y que L2 sea más rápida que L3, mientras que, por el contrario, *peak-mem-seq-copy128.cpp* ofrece un resultado medio peor para L2 que para L3, aunque el resultado máximo es ligeramente mejor, por lo que es posible que debido al tamaño superior de L3, se pueda aprovechar más el prefetching de datos de 128 bits.

Otro patrón que se repite es la cierta variación entre las iteraciones o *runs*, ya que en algunos casos estos resultados pueden variar mucho según la ejecución, como se ve en la diferencia entre máximo y mínimos para cada caché en cada tabla.

3.1.2. Test de lectura secuencial (peak-mem-seq-read64.cpp)

```
1  /************Sequential 64 bit read test AOC
    ↪  style***************/
2
3
4
5  #include "util.h"
6  #include <cstdio>
7  #include <cstring>
8  #include <memory>
9  #include <cmath>
10
11 using namespace std;
12
13 #define L1_CACHE_SIZE 32*1024  // Size of L1 cache to be
    ↪  tested in bytes.
14 #define L2_CACHE_SIZE 256*1024 // Size of L2 cache to be
    ↪  tested in bytes.
15 #define L3_CACHE_SIZE 6*1024*1024 // Size of L3 cache to be
    ↪  tested in bytes.
16 #define REDUCE_SIZE 0.4
17
```

```
18 struct {
19   size_t repeat_times = 7; // total, including warmup
20   size_t warmup_times = 2;
21   bool print_each_time = true;
22   size_t array_size; // size of the array in number of
     ↪  elements
23   size_t threads = 1;
24 } options;
25
26 double* a = nullptr;
27 double* b = nullptr;
28
29 void init() {
30   a = static_cast<double*>(aligned_alloc(64,
     ↪  options.array_size * sizeof(a[0])));
31   b = static_cast<double*>(aligned_alloc(64,
     ↪  options.array_size * sizeof(a[0])));
32   for (size_t i = 0; i < options.array_size; ++i) {
33     a[i] = i;
34   }
35 }
36
37 template<bool use_memcpy>
38 void calculate() {
39   double *la = assume_aligned<64>(a);
40   volatile double volatil;
41     for (size_t i = 0; i < options.array_size; ++i) {
42       volatil = la[i];
43     }
44   }
45
46 template<bool use_memcpy>
47 void measure() {
48   long n_bytes = options.array_size * sizeof(a[0]);
49   printf("Measuring time to read %ld KiB with %zu threads
     ↪  %s:\n", n_bytes / 1024, options.threads, use_memcpy ?
     ↪  "using memcpy" : "using a loop");
50   vector<double> times;
51   vector<double> bps;
```

```
52
53    for (size_t i = 0; i < options.repeat_times; ++i) {
54      double elapsed_time =
      ↪  measure_time(calculate<use_memcpy>);
55      if (i >= options.warmup_times) {
56        times.push_back(elapsed_time);
57        bps.push_back(n_bytes / elapsed_time);
58      }
59      if (options.print_each_time) {
60        printf("    Run %2ld/%2ld: %7.2fs  %7.2f GiB/s %7.2f
      ↪  GB/s  %s\n", i + 1, options.repeat_times, elapsed_time,
      ↪  n_bytes / elapsed_time / (1024*1024*1024), n_bytes /
      ↪  elapsed_time / (1000*1000*1000), i < options.warmup_times
      ↪  ? "(warmup)" : "");
61      }
62    }
63
64    double average_time = vector_average(times);
65    double stddev_time = vector_stddev(times);
66    printf("Average time (s): %7.2f±%.2f      threads = %zu\n",
      ↪  average_time, stddev_time, options.threads);
67    double average_bps = vector_average_harmonic(bps);
68    double stddev_bps = vector_stddev_harmonic(bps);
69    printf("Average GiB/s:    %7.2f±%.2f      threads = %zu\n",
      ↪  average_bps / (1024*1024*1024), stddev_bps /
      ↪  (1024*1024*1024), options.threads);
70    printf("Average GB/s:     %7.2f±%.2f      threads =
      ↪  %zu\n\n", average_bps / (1000*1000*1000), stddev_bps /
      ↪  (1000*1000*1000), options.threads);
71  }
72
73  void executeTest(string cache_type, size_t array_size) {
74  printf("Testing %s cache\n", cache_type.c_str());
75    options.array_size = array_size;
76    init();
77    measure<false>();
78    free(a);
79    free(b);
80  }
```

```
81
82  int main(int argc, char** argv) {
83    for (int i = 1; i < argc; ++i) {
84      if (!parse_bool_arg(argv[i], "print-each-time",
    ↪   options.print_each_time)
85          && !parse_size_arg(argv[i], "repeat-times",
    ↪   options.repeat_times)
86          && !parse_size_arg(argv[i], "warmup-times",
    ↪   options.warmup_times)
87          && !parse_size_arg(argv[i], "threads",
    ↪   options.threads)) {
88        fprintf(stderr, "Incorrect argument: %s\n", argv[i]);
89        return 1;
90      }
91    }
92
93  printf("Benchmark: Sequential 64 bit read test AOC style\n");
94      executeTest("L1", (L1_CACHE_SIZE / sizeof(double)) *
    ↪   REDUCE_SIZE);
95      executeTest("L2", (L2_CACHE_SIZE / sizeof(double)) *
    ↪   REDUCE_SIZE);
96      executeTest("L3", (L3_CACHE_SIZE / sizeof(double)) *
    ↪   REDUCE_SIZE);
97
98    return 0;
99  }
100
```

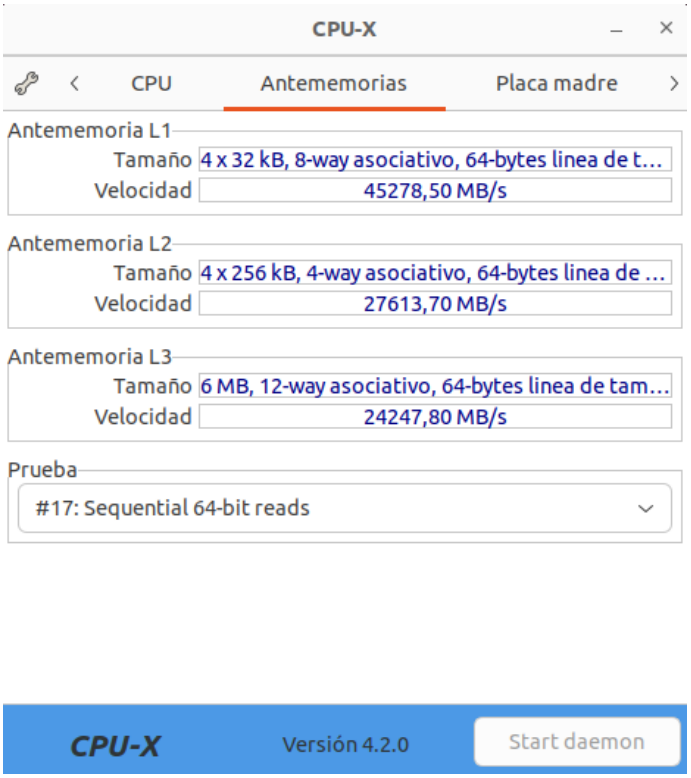

Figura 7: Resultados de CPU-X para lecturas secuenciales de 64 bits

	L1	L2	L3
Run 1	23,76	17,25	17,25
Run 2	24,35	12,56	15,29
Run 3	23,18	12,76	18,39
Run 4	23,62	22,54	16,77
Run 5	23,83	22,83	17,85
Media	23,75	17,59	17,11
Maximo	24,35	22,83	18,39
Mínimo	23,18	12,56	15,29

Figura 8: Resultados de bandwidth (GB/s) de peak-mem-seq-read-64.cpp para lecturas secuenciales de 64 bits

A diferencia del caso anterior, si ejecutamos la copia secuencial, los resultados de CPU-X son superiores a los de nuestro programa. El motivo de este suceso puede ser que en el caso anterior utilizábamos *memcpy*, lo cual puede facilitar al compilador aumentar el rendimiento de la caché; sin embargo, ahora estamos utilizando una lectura normal como se puede ver en el código de *peak-mem-seq-read-64.cpp*, así que es posible que no se puedan aprovechar dichas optimizaciones y obtengamos un rendimiento inferior para el ancho de banda.

Como en el caso anterior, sigue habiendo cierta variación entre los máximos y mínimos resultados, especialmente en L2, pero vemos más estabilidad con una menor diferencia y si nos concentramos en los resultados máximos, podemos ver que se cumple que L1 tiene mayor bandwidth que L2 y L2 que L3.

3.1.3. Test de escritura secuencial (peak-mem-seq-write-64.cpp)

```
/************Sequential 64 bit write test AOC style***************/


#include "util.h"
#include <cstdio>
#include <cstring>
#include <memory>
#include <omp.h>
#include <cmath>

using namespace std;

#define L1_CACHE_SIZE 32*1024  // Size of L1 cache to be tested in bytes.
#define L2_CACHE_SIZE 256*1024 // Size of L2 cache to be tested in bytes.
#define L3_CACHE_SIZE 6*1024*1024 // Size of L3 cache to be tested in bytes.
#define REDUCE_SIZE 0.4

struct {
  size_t repeat_times = 7; // total, including warmup
  size_t warmup_times = 2;
  bool print_each_time = true;
  size_t array_size; // size of the array in number of elements
  size_t threads = 1;
} options;

double* a = nullptr;
double* b = nullptr;

void init() {
  a = static_cast<double*>(aligned_alloc(64, options.array_size * sizeof(a[0])));
```

```
32    b = static_cast<double*>(aligned_alloc(64,
    ↪  options.array_size * sizeof(a[0])));
33    for (size_t i = 0; i < options.array_size; ++i) {
34      a[i] = i;
35    }
36  }
37
38  template<bool use_memcpy>
39  void calculate() {
40    double *la = assume_aligned<64>(a);
41    volatile double volatil;
42      for (size_t i = 0; i < options.array_size; ++i) {
43        la[i]=1.0;
44      }
45    }
46
47  template<bool use_memcpy>
48  void measure() {
49    long n_bytes = options.array_size * sizeof(a[0]);
50    printf("Measuring time to write %ld KiB with %zu threads
    ↪  %s:\n", n_bytes / 1024, options.threads, use_memcpy ?
    ↪  "using memcpy" : "using a loop");
51    vector<double> times;
52    vector<double> bps;
53
54    for (size_t i = 0; i < options.repeat_times; ++i) {
55      double elapsed_time =
    ↪  measure_time(calculate<use_memcpy>);
56      if (i >= options.warmup_times) {
57        times.push_back(elapsed_time);
58        bps.push_back(n_bytes / elapsed_time);
59      }
60      if (options.print_each_time) {
61        printf("    Run %2ld/%2ld: %7.2fs  %7.2f GiB/s %7.2f
    ↪  GB/s  %s\n", i + 1, options.repeat_times, elapsed_time,
    ↪  n_bytes / elapsed_time / (1024*1024*1024), n_bytes /
    ↪  elapsed_time / (1000*1000*1000), i < options.warmup_times
    ↪  ? "(warmup)" : "");
62      }
```

```
63    }
64
65    double average_time = vector_average(times);
66    double stddev_time = vector_stddev(times);
67    printf("Average time (s): %7.2f±%.2f      threads = %zu\n",
      ↪  average_time, stddev_time, options.threads);
68    double average_bps = vector_average_harmonic(bps);
69    double stddev_bps = vector_stddev_harmonic(bps);
70    printf("Average GiB/s:    %7.2f±%.2f      threads = %zu\n",
      ↪  average_bps / (1024*1024*1024), stddev_bps /
      ↪  (1024*1024*1024), options.threads);
71    printf("Average GB/s:     %7.2f±%.2f      threads =
      ↪  %zu\n\n", average_bps / (1000*1000*1000), stddev_bps /
      ↪  (1000*1000*1000), options.threads);
72  }
73
74  void executeTest(string cache_type, size_t array_size) {
75  printf("Testing %s cache\n", cache_type.c_str());
76    options.array_size = array_size;
77    init();
78    measure<false>();
79    free(a);
80    free(b);
81  }
82
83  int main(int argc, char** argv) {
84    for (int i = 1; i < argc; ++i) {
85      if (!parse_bool_arg(argv[i], "print-each-time",
      ↪  options.print_each_time)
86          && !parse_size_arg(argv[i], "repeat-times",
      ↪  options.repeat_times)
87          && !parse_size_arg(argv[i], "warmup-times",
      ↪  options.warmup_times)
88          && !parse_size_arg(argv[i], "threads",
      ↪  options.threads)) {
89        fprintf(stderr, "Incorrect argument: %s\n", argv[i]);
90        return 1;
91      }
92    }
```

```
93
94  printf("Benchmark: Sequential 64 bit write test AOC
    ↪  style\n");
95      executeTest("L1", (L1_CACHE_SIZE / sizeof(double)) *
    ↪  REDUCE_SIZE);
96      executeTest("L2", (L2_CACHE_SIZE / sizeof(double)) *
    ↪  REDUCE_SIZE);
97      executeTest("L3", (L3_CACHE_SIZE / sizeof(double)) *
    ↪  REDUCE_SIZE);
98
99    return 0;
100 }
101
```

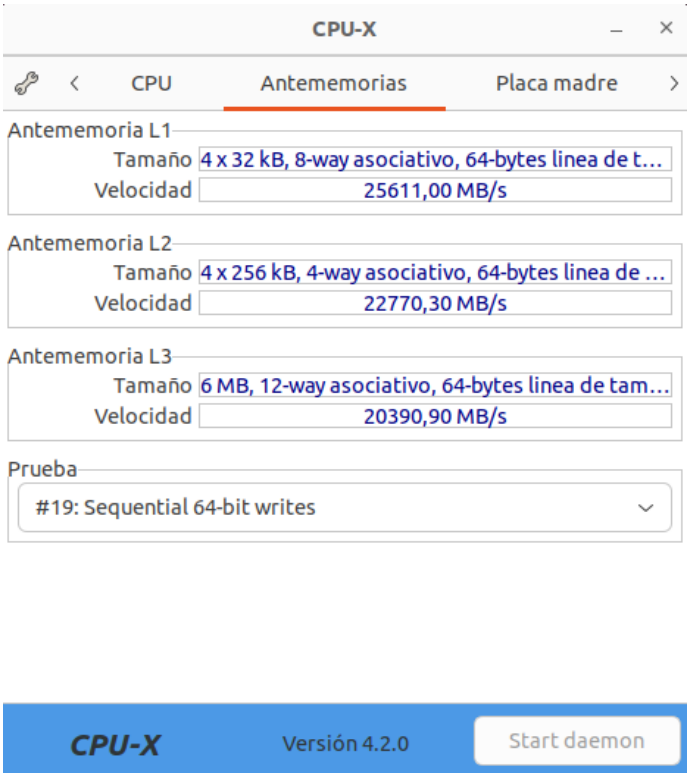

Figura 9: Resultados de CPU-X para escrituras secuenciales de 64 bits

	L1	L2	L3
Run 1	31,84	36,82	28,86
Run 2	45,79	58,25	29,88
Run 3	43,25	37,54	27,53
Run 4	29,43	39,16	28,4
Run 5	55,34	40,41	29,73
Media	41,13	42,44	28,88
Maximo	55,34	58,25	29,88
Mínimo	29,43	36,82	27,53

Figura 10: Resultados de bandwidth (GB/s) de peak-mem-seq-copy-64.cpp para escrituras secuenciales de 64 bits

Para este test obtenemos unos resultados inusuales que no se corresponden con los de CPU-X al tiempo que son superiores a estos y presentan en comportamiento medio una L2 igual o más rápida que L1. Es posible que la mayor cantidad de datos del array que se usa para probar L2 permita un prefetching superior al tiempo que se pueda realizar un mayor número de escrituras simultáneas, sin embargo, este test no nos ofrece ni unos resultados que se correspondan con CPU-X ni unos resultados que se acerquen al ancho de banda que puede obtenerse de las cachés con otras operaciones.

3.2. Acceso aleatorio a caché

3.2.1. Test de lectura aleatoria (peak-mem-random-read-64-a.cpp)

```
1  /************Random 64 bit read test**************/
2  //a-spec
3  #include "util.h"
4  #include <cstdio>
5  #include <cstring>
6  #include <memory>
7  #include <random>
8  #include <algorithm>
9  #define L1_CACHE_SIZE 32*1024  // Size of L1 cache to be
   ↪  tested in bytes.
10 #define L2_CACHE_SIZE 256*1024 // Size of L2 cache to be
   ↪  tested in bytes.
11 #define L3_CACHE_SIZE 6*1024*1024 // Size of L3 cache to be
   ↪  tested in bytes.
12 #define REDUCE_SIZE 0.4
13
14
```

```
15 using namespace std;
16
17 //RANDOM
18
19 // Define a structure "options" to store program
    ↪ configuration options
20 struct
21 {
22   size_t repeat_times = 7; // total number of repetitions,
    ↪ including warmup repetitions
23   size_t warmup_times = 2; // number of warmup repetitions
24   bool print_each_time = true; // indicates whether the
    ↪ result of each repetition should be printed
25   size_t array_size; // size of the array in number of
    ↪ elements
26   size_t threads = 1; // number of threads used in parallel
    ↪ execution
27 } options;
28
29 // Declare two pointers of type 'block', called 'a' and 'b'
30 double* a = nullptr;
31
32 // We'll use a separate array to hold indices
33 size_t* indices = nullptr;
34
35 void init() {
36   // Allocate aligned memory for array 'a'
37   a = static_cast<double*>(aligned_alloc(64,
    ↪ options.array_size * sizeof(double)));
38
39   // Allocate memory for the indices array
40   indices = static_cast<size_t*>(aligned_alloc(64,
    ↪ options.array_size * sizeof(size_t)));
41
42   // Fill the array 'a' with sequential values and indices
    ↪ with sequential numbers
43   for (size_t i = 0; i < options.array_size; ++i) {
44     a[i] = static_cast<double>(i);
45     indices[i] = static_cast<size_t>(i);
```

```
46    }
47
48    // Shuffle the indices to randomize the order
49    random_device rd;
50    mt19937 g(rd());
51    shuffle(indices, indices + options.array_size, g);
52  }
53
54  // This function reads data from an array in a random
    ↪  manner, following the indices stored in the elements of
    ↪  the array
55  template <bool use_memcpy>
56  void calculate() {
57    // Use assume_aligned to tell the compiler that 'a' is
    ↪  aligned
58    double *la = assume_aligned<64>(a);
59    size_t *lindices = assume_aligned<64>(indices);
60
61    // Iterate over all elements in a random order defined by
    ↪  the shuffled indices
62    for (size_t i = 0; i < options.array_size; ++i) {
63      // Access 'a' in a random way using shuffled indices
64      size_t idx = lindices[i];
65      volatile double temp = la[idx];
66    }
67  }
68
69  template <bool use_memcpy>
70  void measure()
71  {
72    long n_bytes = options.array_size * sizeof(double);
73    printf("Measuring time to read %ld KiB with %zu threads
    ↪  %s:\n", n_bytes / 1024, options.threads, use_memcpy ?
    ↪  "using memcpy" : "using a loop");
74
75    vector<double> times;
76    vector<double> bps;
77
78    for (size_t i = 0; i < options.repeat_times; ++i)
```

```
79    {
80      double elapsed_time =
    ↪   measure_time(calculate<use_memcpy>);
81
82      if (i >= options.warmup_times)
83      {
84        times.push_back(elapsed_time);
85        bps.push_back(n_bytes / elapsed_time);
86      }
87
88      if (options.print_each_time)
89      {
90        printf("    Run %2ld/%2ld: %7.10fs  %7.2f GiB/s %7.2f
    ↪   GB/s  %s\n", i + 1, options.repeat_times, elapsed_time,
    ↪   n_bytes / elapsed_time / (1024 * 1024 * 1024), n_bytes /
    ↪   elapsed_time / (1000 * 1000 * 1000), i <
    ↪   options.warmup_times ? "(warmup)" : "");
91      }
92    }
93
94    double average_time = vector_average(times);
95    double stddev_time = vector_stddev(times);
96    printf("Average time (s): %7.2f±%.10f      threads =
    ↪   %zu\n", average_time, stddev_time, options.threads);
97
98    double average_bps = vector_average_harmonic(bps);
99    double stddev_bps = vector_stddev_harmonic(bps);
100   printf("Average GiB/s:    %7.2f±%.2f      threads = %zu\n",
    ↪   average_bps / (1024 * 1024 * 1024), stddev_bps / (1024 *
    ↪   1024 * 1024), options.threads);
101   printf("Average GB/s:     %7.2f±%.2f      threads =
    ↪   %zu\n\n", average_bps / (1000 * 1000 * 1000), stddev_bps
    ↪   / (1000 * 1000 * 1000), options.threads);
102 }
103
104 void executeTest(string cache_type, size_t array_size) {
105   printf("Testing %s cache\n", cache_type.c_str());
106   options.array_size = array_size;
107   init();
```

```
108   measure<false>();
109   free(a);
110   free(indices);
111 }
112
113 int main(int argc, char **argv)
114 {
115   for (int i = 1; i < argc; ++i)
116   {
117     if (!parse_bool_arg(argv[i], "print-each-time",
    ↪  options.print_each_time) &&
118       !parse_size_arg(argv[i], "repeat-times",
    ↪  options.repeat_times) &&
119         !parse_size_arg(argv[i], "warmup-times",
    ↪  options.warmup_times) &&
120           !parse_size_arg(argv[i], "array-size",
    ↪  options.array_size) &&
121             !parse_size_arg(argv[i], "threads", options.threads))
122     {
123       fprintf(stderr, "Incorrect argument: %s\n", argv[i]);
124       return 1;
125     }
126   }
127
128   printf("Benchmark: Random reads 64 bits spec-a\n");
129   executeTest("L1", (L1_CACHE_SIZE / sizeof(double)) *
    ↪  REDUCE_SIZE);
130   executeTest("L2", (L2_CACHE_SIZE / sizeof(double)) *
    ↪  REDUCE_SIZE);
131   executeTest("L3", (L3_CACHE_SIZE / sizeof(double)) *
    ↪  REDUCE_SIZE);
132
133   return 0;
134 }
135
```

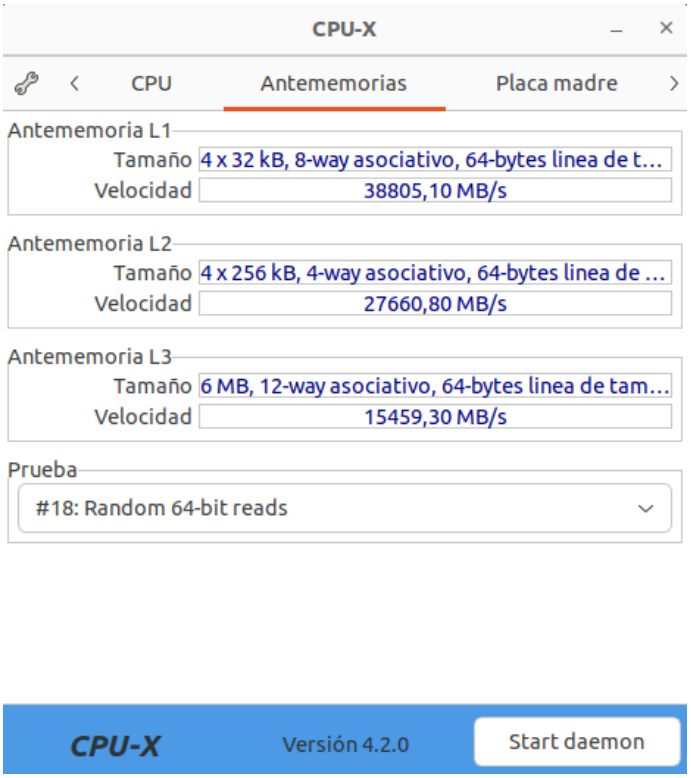

Figura 11: Resultados de CPU-X para lecturas aleatorias de 64 bits

	L1	L2	L3
Run 1	16,15	11,99	4,22
Run 2	17,25	7,29	4,05
Run 3	16,25	8,01	3,84
Run 4	15,77	6,34	3,95
Run 5	16,11	6,61	4,06
Media	16,31	8,05	4,02
Maximo	17,25	11,99	4,22
Mínimo	15,77	6,34	3,84

Figura 12: Resultados de bandwidth (GB/s) de peak-mem-random-read-64-a.cpp para lecturas aleatorias de 64 bits

Este test es el primer test de acceso aleatorio a la caché, lo cual significa que accede a los elementos del array siguiendo un orden completamente aleatorio y no puede aprovechar las optimizaciones de prefetching u otras mejoras que sí se pueden aprovechar cuando el acceso es secuencial. En particular, se consigue el acceso aleatorio siguiendo un array de índices ordenado aleatoriamente que se accede de forma secuencial.

Los resultados obtenidos son de alrededor de un 30-40 % de los obtenidos con CPU-X y aunque L1 y L2 no tienen una caída demasiado grande respecto a *peak-mem-seq-read-64.cpp*, L3 se ve la más perjudicada, ya que al cambiar a acceso aleatorio el no poder realizar optimizaciones respecto a los datos, se ve lastrada por sus mayores tiempos de lectura.

Sin embargo, estos resultados, aunque no obtienen el máximo rendimiento de la caché, sí que consiguen ser muy regulares, con una variación general "bajaz que muestra que L1 es consistentemente la caché con mayor ancho de banda, seguida por L2 y, finalmente, L3. Así que podemos observar que este benchmark pinta una situación más realista del ancho de banda en las cachés y como afecta usar una u otra.

3.2.2. Test de lectura aleatoria. B-spec (peak-mem-random-read-64-b.cpp)

```
1  /************Random 64 bit read test**************/
2  //b-spec
3  #include "util.h"
4  #include <cstdio>
5  #include <cstring>
6  #include <memory>
7  #include <random>
8  #include <algorithm>
9  #define L1_CACHE_SIZE 32*1024  // Size of L1 cache to be
   ↪  tested in bytes.
10 #define L2_CACHE_SIZE 256*1024 // Size of L2 cache to be
   ↪  tested in bytes.
11 #define L3_CACHE_SIZE 6*1024*1024 // Size of L3 cache to be
   ↪  tested in bytes.
12 #define REDUCE_SIZE 0.4
13
14
15 using namespace std;
16
17 //RANDOM
18
19 //RANDOM
20
21 // Define a structure "options" to store program
   ↪  configuration options
22 struct
23 {
24   size_t repeat_times = 7; // total number of repetitions,
   ↪  including warmup repetitions
25   size_t warmup_times = 2; // number of warmup repetitions
```

```
26    bool print_each_time = true; // indicates whether the
    ↪  result of each repetition should be printed
27    size_t array_size; // size of the array in number of
    ↪  elements
28    size_t threads = 1; // number of threads used in parallel
    ↪  execution
29  } options;
30
31  // Declare two pointers of type 'block', called 'a' and 'b'
32  long long int* a = nullptr;
33
34  void init() {
35    // Allocate aligned memory for array 'a'
36    a = static_cast<long long int*>(aligned_alloc(64,
    ↪  options.array_size * sizeof(long long int)));
37
38    // Fill the array 'a' with sequential values
39    for (size_t i = 0; i < options.array_size; ++i) {
40      a[i] = static_cast<long long int>(i);
41    }
42
43    // Shuffle the indices to randomize the order
44    random_device rd;
45    mt19937 g(rd());
46    shuffle(a, a + options.array_size, g);
47  }
48
49
50  //This function reads data from an array in a random manner,
    ↪  following the indices stored in the elements of the
    ↪  array
51  template <bool use_memcpy>
52  void calculate() {
53    // Use assume_aligned to tell the compiler that 'a' is
    ↪  aligned
54    long long int *la = assume_aligned<64>(a);
55    volatile long long int idx = 0;
56    size_t cont=0;
57
```

```
58    while(cont < options.array_size){
59      idx = la[idx];
60      cont++;
61    }
62  }
63
64  template <bool use_memcpy>
65  void measure()
66  {
67    long n_bytes = options.array_size * sizeof(long long int);
68    printf("Measuring time to read %ld KiB with %zu threads
    ↪  %s:\n", n_bytes / 1024, options.threads, use_memcpy ?
    ↪  "using memcpy" : "using a loop");
69
70    vector<double> times;
71    vector<double> bps;
72
73    for (size_t i = 0; i < options.repeat_times; ++i)
74    {
75      double elapsed_time =
    ↪  measure_time(calculate<use_memcpy>);
76
77      if (i >= options.warmup_times)
78      {
79        times.push_back(elapsed_time);
80        bps.push_back(n_bytes / elapsed_time);
81      }
82
83      if (options.print_each_time)
84      {
85        printf("    Run %2ld/%2ld: %7.10fs  %7.2f GiB/s %7.2f
    ↪  GB/s  %s\n", i + 1, options.repeat_times, elapsed_time,
    ↪  n_bytes / elapsed_time / (1024 * 1024 * 1024), n_bytes /
    ↪  elapsed_time / (1000 * 1000 * 1000), i <
    ↪  options.warmup_times ? "(warmup)" : "");
86      }
87    }
88
89    double average_time = vector_average(times);
```

```
90    double stddev_time = vector_stddev(times);
91    printf("Average time (s): %7.2f±%.10f      threads =
      %zu\n", average_time, stddev_time, options.threads);
92
93    double average_bps = vector_average_harmonic(bps);
94    double stddev_bps = vector_stddev_harmonic(bps);
95    printf("Average GiB/s:    %7.2f±%.2f      threads = %zu\n",
      average_bps / (1024 * 1024 * 1024), stddev_bps / (1024 *
      1024 * 1024), options.threads);
96    printf("Average GB/s:     %7.2f±%.2f      threads =
      %zu\n\n", average_bps / (1000 * 1000 * 1000), stddev_bps
      / (1000 * 1000 * 1000), options.threads);
97  }
98
99  void executeTest(string cache_type, size_t array_size) {
100   printf("Testing %s cache\n", cache_type.c_str());
101   options.array_size = array_size;
102   init();
103   measure<false>();
104   free(a);
105 }
106
107 int main(int argc, char **argv)
108 {
109   for (int i = 1; i < argc; ++i)
110   {
111     if (!parse_bool_arg(argv[i], "print-each-time",
      options.print_each_time) &&
112       !parse_size_arg(argv[i], "repeat-times",
      options.repeat_times) &&
113         !parse_size_arg(argv[i], "warmup-times",
      options.warmup_times) &&
114           !parse_size_arg(argv[i], "array-size",
      options.array_size) &&
115           !parse_size_arg(argv[i], "threads", options.threads))
116     {
117       fprintf(stderr, "Incorrect argument: %s\n", argv[i]);
118       return 1;
119     }
```

```
120     }
121
122     printf("Benchmark: Random reads 64 bits spec-b\n");
123     executeTest("L1", (L1_CACHE_SIZE / sizeof(long long int)) *
        ↪  REDUCE_SIZE);
124     executeTest("L2", (L2_CACHE_SIZE / sizeof(long long int)) *
        ↪  REDUCE_SIZE);
125     executeTest("L3", (L3_CACHE_SIZE / sizeof(long long int)) *
        ↪  REDUCE_SIZE);
126
127     return 0;
128 }
```

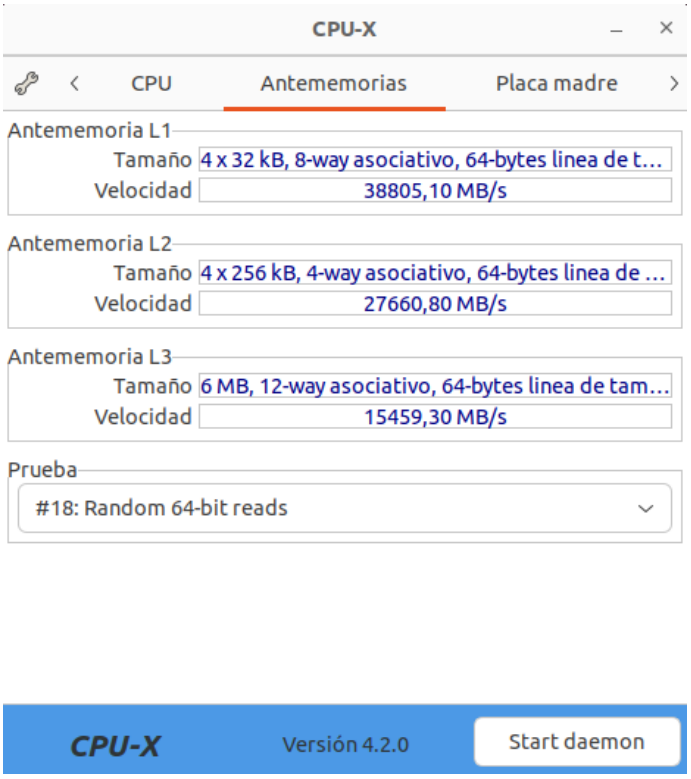

Figura 13: Resultados de CPU-X para lecturas aleatorias de 64 bits

	L1	L2	L3
Run 1	2,8	1,58	0,49
Run 2	2,81	1,15	0,49
Run 3	2,12	1,31	0,51
Run 4	2,54	1,36	0,5
Run 5	2,78	1,16	0,51
Media	2,61	1,31	0,5
Maximo	2,81	1,58	0,51
Mínimo	2,12	1,15	0,49

Figura 14: Resultados de bandwidth (GB/s) de peak-mem-random-read-64-b.cpp para lecturas aleatorias de 64 bits

Este programa es muy similar al anterior, salvo porque el acceso aleatorio no se realiza a través de un array de índices, sino que cada valor leído del array es el índice que contiene el siguiente elemento a leer. Sin embargo, esta aproximación tiene diversos problemas. En primer lugar, los resultados son peores que en el caso anterior, aun evitando realizar dos lecturas(aunque una sea secuencial), pero esos accesos secuenciales permitían guardarse en memoria y aunque los accesos a memoria fuesen aleatorios, se podía optimizar dicho acceso, ya que se conocía el orden de antemano, pero al utilizar un índice que hay que leer para obtener, el rendimiento cae debido a que hasta que no se realice una lectura, no se puede proceder a la siguiente.

No obstante, este test presenta muy poca variación entre los valores máximos y mínimos, siendo muy constante y coherente con L1 como la más rápida y tras ella L2 y L3.

3.2.3. Test de escritura aleatoria (peak-mem-random-write-64.cpp)

```
/************Random 64 bit write test**************/
//a-spec

#include "util.h"
#include <cstdio>
#include <cstring>
#include <memory>
#include <random>
#include <algorithm>
#define L1_CACHE_SIZE 32*1024  // Size of L1 cache to be tested in bytes.
#define L2_CACHE_SIZE 256*1024 // Size of L2 cache to be tested in bytes.
#define L3_CACHE_SIZE 6*1024*1024 // Size of L3 cache to be tested in bytes.
#define REDUCE_SIZE 0.4


using namespace std;

//RANDOM

// Define a structure "options" to store program configuration options
struct
{
  size_t repeat_times = 7; // total number of repetitions, including warmup repetitions
  size_t warmup_times = 2; // number of warmup repetitions
  bool print_each_time = true; // indicates whether the result of each repetition should be printed
  size_t array_size; // size of the array in number of elements
```

```
27    size_t threads = 1; // number of threads used in parallel
      ↪  execution
28  } options;
29
30  // Declare two pointers of type 'block', called 'a' and 'b'
31  double* a = nullptr;
32
33  // We'll use a separate array to hold indices
34  size_t* indices = nullptr;
35
36  void init() {
37    // Allocate aligned memory for array 'a'
38    a = static_cast<double*>(aligned_alloc(64,
      ↪  options.array_size * sizeof(double)));
39
40    // Allocate memory for the indices array
41    indices = static_cast<size_t*>(aligned_alloc(64,
      ↪  options.array_size * sizeof(size_t)));
42
43    // Fill the array 'a' with sequential values and indices
      ↪  with sequential numbers
44    for (size_t i = 0; i < options.array_size; ++i) {
45      a[i] = static_cast<double>(i);
46      indices[i] = static_cast<size_t>(i);
47    }
48
49    // Shuffle the indices to randomize the order
50    random_device rd;
51    mt19937 g(rd());
52    shuffle(indices, indices + options.array_size, g);
53  }
54
55  // This function writes data from an array in a random
      ↪  manner, following the indices stored in the elements of
      ↪  the array
56  template <bool use_memcpy>
57  void calculate() {
58    // Use assume_aligned to tell the compiler that 'a' is
      ↪  aligned
```

```
59    double *la = assume_aligned<64>(a);
60    size_t *lindices = assume_aligned<64>(indices);
61
62    // Iterate over all elements in a random order defined by
      ↪  the shuffled indices
63    for (size_t i = 0; i < options.array_size; ++i) {
64      // Access 'a' in a random way using shuffled indices
65      size_t idx = lindices[i];
66      la[idx] = 1.0 ;
67    }
68  }
69
70  template <bool use_memcpy>
71  void measure()
72  {
73    long n_bytes = options.array_size * sizeof(double);
74    printf("Measuring time to write %ld KiB with %zu threads
      ↪  %s:\n", n_bytes / 1024, options.threads, use_memcpy ?
      ↪  "using memcpy" : "using a loop");
75
76    vector<double> times;
77    vector<double> bps;
78
79    for (size_t i = 0; i < options.repeat_times; ++i)
80    {
81      double elapsed_time =
      ↪  measure_time(calculate<use_memcpy>);
82
83      if (i >= options.warmup_times)
84      {
85        times.push_back(elapsed_time);
86        bps.push_back(n_bytes / elapsed_time);
87      }
88
89      if (options.print_each_time)
90      {
```

```
91        printf("    Run %2ld/%2ld: %7.10fs  %7.2f GiB/s %7.2f
    ↪  GB/s  %s\n", i + 1, options.repeat_times, elapsed_time,
    ↪  n_bytes / elapsed_time / (1024 * 1024 * 1024), n_bytes /
    ↪  elapsed_time / (1000 * 1000 * 1000), i <
    ↪  options.warmup_times ? "(warmup)" : "");
92      }
93    }
94
95    double average_time = vector_average(times);
96    double stddev_time = vector_stddev(times);
97    printf("Average time (s): %7.2f±%.10f      threads =
    ↪  %zu\n", average_time, stddev_time, options.threads);
98
99    double average_bps = vector_average_harmonic(bps);
100   double stddev_bps = vector_stddev_harmonic(bps);
101   printf("Average GiB/s:    %7.2f±%.2f      threads = %zu\n",
    ↪  average_bps / (1024 * 1024 * 1024), stddev_bps / (1024 *
    ↪  1024 * 1024), options.threads);
102   printf("Average GB/s:     %7.2f±%.2f      threads =
    ↪  %zu\n\n", average_bps / (1000 * 1000 * 1000), stddev_bps
    ↪  / (1000 * 1000 * 1000), options.threads);
103 }
104
105 void executeTest(string cache_type, size_t array_size) {
106   printf("Testing %s cache\n", cache_type.c_str());
107   options.array_size = array_size;
108   init();
109   measure<false>();
110   free(a);
111   free(indices);
112 }
113
114 int main(int argc, char **argv)
115 {
116   for (int i = 1; i < argc; ++i)
117   {
118     if (!parse_bool_arg(argv[i], "print-each-time",
    ↪  options.print_each_time) &&
```

```
119       !parse_size_arg(argv[i], "repeat-times",
    ↪  options.repeat_times) &&
120         !parse_size_arg(argv[i], "warmup-times",
    ↪  options.warmup_times) &&
121           !parse_size_arg(argv[i], "array-size",
    ↪  options.array_size) &&
122             !parse_size_arg(argv[i], "threads", options.threads))
123       {
124         fprintf(stderr, "Incorrect argument: %s\n", argv[i]);
125         return 1;
126       }
127     }
128
129     printf("Benchmark: Random writes 64 bits\n");
130     executeTest("L1", (L1_CACHE_SIZE / sizeof(long long int)) *
    ↪  REDUCE_SIZE);
131     executeTest("L2", (L2_CACHE_SIZE / sizeof(long long int)) *
    ↪  REDUCE_SIZE);
132     executeTest("L3", (L3_CACHE_SIZE / sizeof(long long int)) *
    ↪  REDUCE_SIZE);
133
134     return 0;
135   }
```

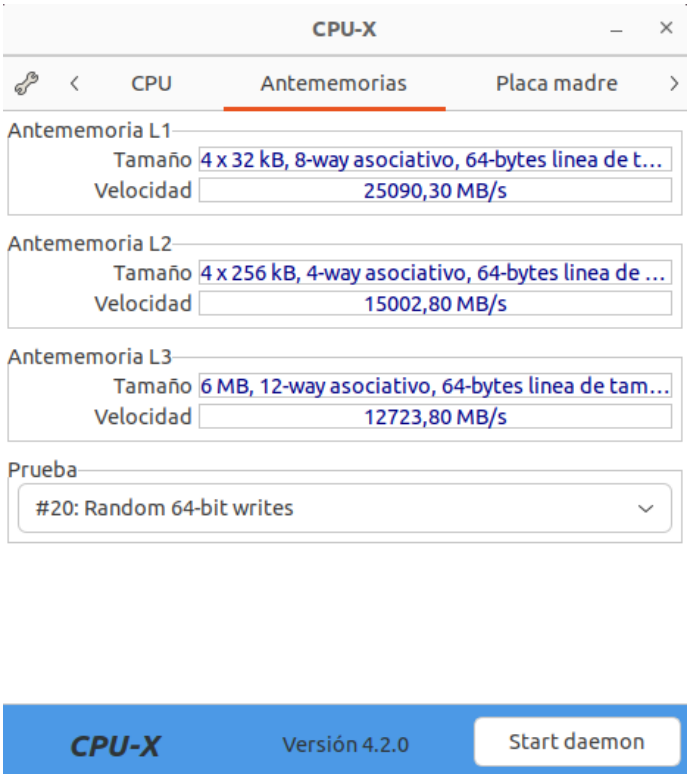

Figura 15: Resultados de CPU-X para escrituras aleatorias de 64 bits

	L1	L2	L3
Run 1	11,84	4,39	3,13
Run 2	10,02	4,46	3,12
Run 3	11,07	4,88	3,15
Run 4	13,37	4,9	3,17
Run 5	10,35	4,45	3,02
Media	11,33	4,62	3,12
Maximo	13,37	4,9	3,17
Mínimo	10,02	4,39	3,02

Figura 16: Resultados de bandwidth (GB/s) de peak-mem-random-write-64.cpp para escrituras aleatorias de 64 bits

Finalmente, el test de escrituras aleatorias produce unos resultados que se sitúan alrededor de un 30 % de los obtenidos por CPU-X. Estos resultados también son coherentes a lo largo de las ejecuciones con las cachés obteniendo bandwidth mayores o menores según su tipo. Si comparamos estos resultados con los obtenidos por *peak-mem-seq-write-64.cpp* vemos una gran caída de rendimiento, obteniendo aproximadamente un cuarto del rendimiento secuencial, mientras que el test de lectura aleatoria no experimentó una caída tan pronunciada, lo cual puede indicar que si realizamos escrituras aleatorias, el peso de las optimizaciones es aún más vital para obtener un buen rendimiento.

3.3. Conclusiones generales de los test

Tras analizar los resultados de los diferentes test y evaluarlos frente a sus contrapartidas de CPU-X, hemos llegado a ciertas conlusiones.

En primer lugar, los tests de acceso aleatorio son más fiables a la hora de dar resultados coherentes que los de acceso secuencial. Con un resultado coherente nos referimos a que L1 sea más rápida que L2 y esta que L3, lo cual es obtenido con facilidad en estos test frente a los secuenciales que obtienen resultados más elevados que llevan más cerca del límite a la caché, pero que se pueden ver afectados por diversas optimizaciones y mecanismos de mejora del rendimiento de cpu como el prefetching. Sin embargo, es posible que estos test también puedan dar un resultado incoherente en algún momento, ya que debido a la naturaleza de estos test, ya sean aleatorios o secuenciales, es fácil que otros procesos interfieran con la caché aun ejecutando el benchmark en docker y limpiando las cachés. Simplemente, el acceso aleatorio hace depender al benchmark mucho más de la velocidad pura de lectura y escritura de la caché y eso permite obtener resultados más coherentes con la teoría.

Respecto a los resultados pico, podemos comprobar que los programas de copia secuencial obtienen los mejores resultados, en particular, el benchmark de copia de 64 bits ofrece un alto rendimiento pico, además de ser menos errático que su hermano de 128 bits, aunque debido a su naturaleza secuencial, es posible obtener algún que otro resultado incoherente en una ejecución concreta. Si decidimos compararlo con el benchmark predeterminado de CPU-X, lectura secuencial de 64 bits, los resultados obtenidos son bastante similares, aunque la relativamente alta variación de los resultados, hace que la media sea ligeramente más baja, no así los máximos, que resultan muy cercanos.

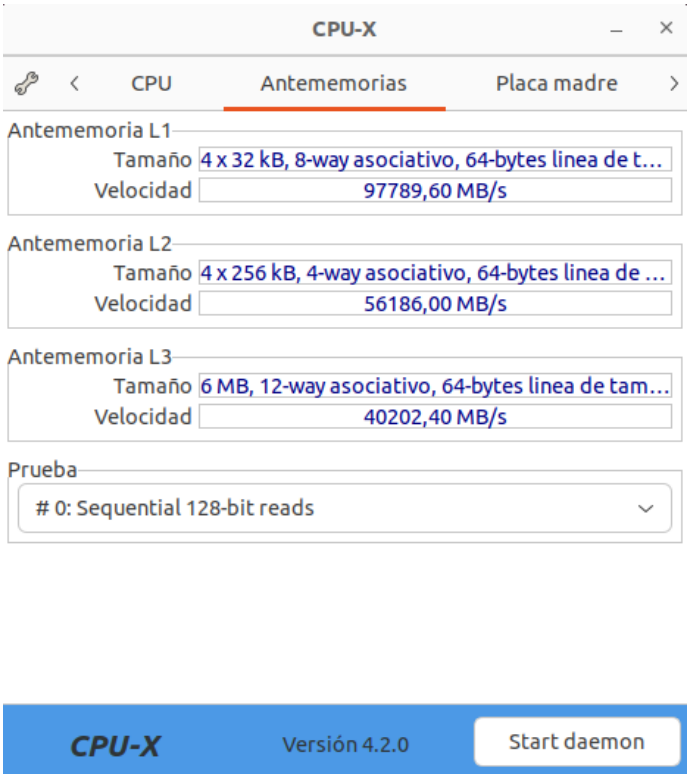

Figura 17: Resultados de bandwidth (GB/s de CPU-X para lecturas secuenciales de 128 bits)

	L1	L2	L3
Run 1	92,94	55,6	41,49
Run 2	97,28	48,23	43,52
Run 3	62,7	49,54	48,67
Run 4	98,82	48,65	42,07
Run 5	55,71	47,8	41,86
Media	81,49	49,96	43,52
Maximo	98,82	55,6	48,67
Mínimo	55,71	47,8	41,49

Figura 18: Resultados de bandwidth (GB/s) de peak-mem-seq-copy64.cpp para copias secuenciales de 64 bits con memcpy

En conclusión, si queremos obtener el máximo ancho de banda de la caché podemos utilizar *peak-mem-seq-read-64.cpp* y si queremos algo más fiable a la hora de obtener resultados a la hora de menor precisión del rendimiento máximo, pero más próximo al rendimiento en una situación real, *peak-mem-random-read-64-a.cpp* es nuestra mejor elección.

4. Bibliografía

1. **Arquitectura y Organización de Computadores.** Tema 1. Arquitectura de un multiprocesador-en-un-chip. Departamento de Ingeniería y Tecnología de Computadores. Universidad de Murcia. [Consultado: 24 de junio de 2023].

2. **Arquitectura y Organización de Computadores.** Práctica 1 - Introducción a la computación de alto rendimiento. Departamento de Ingeniería y Tecnología de Computadores. Universidad de Murcia.[Consultado: 24 de junio de 2023].

3. **Darkcrizt**, "CPU-X: muestra información sobre la CPU, motherboard y más" Linux Adictos, Aug. 03, 2018. Disponible en: `https://www.linuxadictos.com/cpu-x-muestra-informacion-sobre-la-cpu-motherboard-y-mas.html`. [Consultado: 25 de junio de 2023]

4. **TheTumultuousUnicornOfDarkness**, "GitHub - TheTumultuousUnicornOfDarkness/CPU-X: CPU-X es un software libre que recopila información sobre la CPU, la placa base y más", GitHub, 02 de abril de 2023. Disponible en: `https://github.com/TheTumultuousUnicornOfDarkness/CPU-X`. [Consultado: 25 de junio de 2023]