Assignment II - Valgrind & Pytorch profiler

- assignment: https://hackmd.io/@-ZZtFnnqSZ2F1A-Uy-GMlw/By-tJ_15ke
- report: https://hackmd.io/@iTkWej8WRj2Xrnb85FtlPQ/rJBN7zY1xl

1. Memcheck

> Q. Please find out the errors from the log file. You should submit your student_ID_log with annotations (see the picture below) and explain the errors in the report.
>
> ```
> ==3527480== Invalid write of size 4                            ##error 1 : invalid write
> ==3527480==    at 0x1091C0: main (memleak.c:49)
> ==3527480==  Address 0x4a97068 is 0 bytes after a block of size 40 alloc'd
> ==3527480==    at 0x484884F: malloc (vg_replace_malloc.c:393)
> ==3527480==    by 0x10919E: main (memleak.c:46)
> ```

1. Invalid Write

- `memleak.c` 第 46 行呼叫 `malloc()` 配置了一塊 40 bytes 的記憶體（結尾是 `0x4a7d068`）。在第 49 行試圖寫入 4 個 bytes 到 `0x4a7d068`（也就是這塊記憶體之後的位置）。這個寫入操作超出了記憶體的有效範圍而導致 invalid write。

```
==24630== Invalid write of size 4 ##error 1 : invalid write
==24630==    at 0x1091C0: main (memleak.c:49)
==24630==  Address 0x4a7d068 is 0 bytes after a block of size 40 alloc'd
==24630==    at 0x484684F: malloc (vg_replace_malloc.c:393)
==24630==    by 0x10919E: main (memleak.c:46)
```

2. Invalid Read

- `memleak.c` 第 46 行呼叫 `malloc()` 配置了一塊 40 bytes 的記憶體（結尾是 `0x4a7d068`）。而在第 54 行試圖從 `0x4a7d068` 之後讀取 4 個 bytes 的資料，超出原本的配置範圍而導致 invalid read。

```
==24630== Invalid read of size 4 ##error 2 : invalid read
==24630==    at 0x1091ED: main (memleak.c:54)
==24630==  Address 0x4a7d068 is 0 bytes after a block of size 40 alloc'd
==24630==    at 0x484684F: malloc (vg_replace_malloc.c:393)
==24630==    by 0x10919E: main (memleak.c:46)
```

3. Uninitialised Value
（Use of Uninitialised Value / Conditional jump or move depends on

uninitialised value(s))

- `memleak.c` 的第 57 行呼叫 `printf()` 函數時，傳入了一個未初始化的值。

```
==24630== Conditional jump or move depends on uninitialised value(s) ##error 3
: Uninitialised Value
==24630==    at 0x48D20CB: __printf_buffer (vfprintf-process-arg.c:58)
==24630==    by 0x48D373A: __vfprintf_internal (vfprintf-internal.c:1544)
==24630==    by 0x48C81B2: printf (printf.c:33)
==24630==    by 0x109214: main (memleak.c:57)
==24630==
==24630== Use of uninitialised value of size 8 ##error 3 : Uninitialised Value
==24630==    at 0x48C70BB: _itoa_word (_itoa.c:183)
==24630==    by 0x48D1C9B: __printf_buffer (vfprintf-process-arg.c:155)
==24630==    by 0x48D373A: __vfprintf_internal (vfprintf-internal.c:1544)
==24630==    by 0x48C81B2: printf (printf.c:33)
==24630==    by 0x109214: main (memleak.c:57)
==24630==
==24630== Conditional jump or move depends on uninitialised value(s) ##error 3
: Uninitialised Value
==24630==    at 0x48C70CC: _itoa_word (_itoa.c:183)
==24630==    by 0x48D1C9B: __printf_buffer (vfprintf-process-arg.c:155)
==24630==    by 0x48D373A: __vfprintf_internal (vfprintf-internal.c:1544)
==24630==    by 0x48C81B2: printf (printf.c:33)
==24630==    by 0x109214: main (memleak.c:57)
==24630==
==24630== Conditional jump or move depends on uninitialised value(s) ##error 3
: Uninitialised Value
==24630==    at 0x48D1D85: __printf_buffer (vfprintf-process-arg.c:186)
==24630==    by 0x48D373A: __vfprintf_internal (vfprintf-internal.c:1544)
==24630==    by 0x48C81B2: printf (printf.c:33)
==24630==    by 0x109214: main (memleak.c:57)
```

4. Fishy Argument to malloc

- `memleak.c` 第 61 行呼叫 `malloc()` 時，傳入的 `size` 參數是 -40。但 `malloc()` 只能接受非負整數作為要分配的記憶體大小。

```
==24630== Argument 'size' of function malloc has a fishy (possibly negative)
value: -40 ##error 4 : Fishy Argument to malloc
==24630==    at 0x484684F: malloc (vg_replace_malloc.c:393)
==24630==    by 0x109220: main (memleak.c:61)
```

5. Invalid free()

- `memleak.c` 第 65 行呼叫了 `free()` 釋放地址為 `0x4a7d4f0`、大小為 40 bytes 的記憶體。不過這塊記憶體已經在第 64 行被釋放過一次了。它是在第 63 行透過 `malloc()` 配置的。所以發生了 double free 的錯誤。

```
==24630== Invalid free() / delete / delete[] / realloc() ##error 5 : Invalid
free()
```

```
==24630==    at 0x48490C4: free (vg_replace_malloc.c:884)
==24630==    by 0x10924A: main (memleak.c:65)
==24630==  Address 0x4a7d4f0 is 0 bytes inside a block of size 40 free'd
==24630==    at 0x48490C4: free (vg_replace_malloc.c:884)
==24630==    by 0x10923E: main (memleak.c:64)
==24630==  Block was alloc'd at
==24630==    at 0x484684F: malloc (vg_replace_malloc.c:393)
==24630==    by 0x10922E: main (memleak.c:63)
```

6. Memory Leak

- `memleak.c` 第 46 行呼叫 `malloc()` 分配了 40 bytes 記憶體，但該記憶體在程式結束時變成 definitely lost，即沒有任何指針指向它，導致無法被釋放，造成 Memory Leak。

```
==24630== HEAP SUMMARY: ##error 6 : Memory Leak
==24630==     in use at exit: 40 bytes in 1 blocks
==24630==   total heap usage: 3 allocs, 3 frees, 1,104 bytes allocated
==24630==
==24630== 40 bytes in 1 blocks are definitely lost in loss record 1 of 1
==24630==    at 0x484684F: malloc (vg_replace_malloc.c:393)
==24630==    by 0x10919E: main (memleak.c:46)
==24630==
==24630== LEAK SUMMARY:
==24630==    definitely lost: 40 bytes in 1 blocks
==24630==    indirectly lost: 0 bytes in 0 blocks
==24630==      possibly lost: 0 bytes in 0 blocks
==24630==    still reachable: 0 bytes in 0 blocks
==24630==         suppressed: 0 bytes in 0 blocks
```

2. Cachegrind

> Q. Please take screenshots of two logs and point out where the difference is and explain why this problem occurs in the report.

- good log 和 bad log 最大的差異在於 D1 Cache 的命中率。雖然兩者的 Data Refs 數量相近，但 good log 的 D1 miss rate 僅 0.9%，而 bad log 有 14.2%。這也導致 bad log 的 LL refs 顯著增加，但 LL miss 數量相近，代表資料大多仍保留在 LL cache。
- 造成這現象的原因可能是 bad 的 locality 較差，例如使用 linked list 或非連續存取陣列，讓快取不能準確預測資料。
- good log

```
==26127== Cachegrind, a cache and branch-prediction profiler
==26127== Copyright (C) 2002-2017, and GNU GPL'd, by Nicholas Nethercote et
al.
==26127== Using Valgrind-3.20.0 and LibVEX; rerun with -h for copyright info
==26127== Command: ./good
==26127== Parent PID: 24578
==26127==
--26127-- warning: L3 cache found, using its data for the LL simulation.
--26127-- warning: specified LL cache: line_size 64  assoc 1  total_size
```

```
100,663,296
--26127-- warning: simulated LL cache: line_size 64  assoc 2  total_size
134,217,728
==26127==
==26127== I   refs:      30,160,062
==26127== I1  misses:         1,060
==26127== LLi misses:         1,042
==26127== I1  miss rate:       0.00%
==26127== LLi miss rate:       0.00%
==26127==
==26127== D   refs:      14,052,880  (12,039,547 rd   + 2,013,333 wr)
==26127== D1  misses:       126,576  (    63,742 rd   +    62,834 wr)
==26127== LLd misses:        63,813  (     1,002 rd   +    62,811 wr)
==26127== D1  miss rate:        0.9% (       0.5%     +       3.1%  )
==26127== LLd miss rate:        0.5% (       0.0%     +       3.1%  )
==26127==
==26127== LL refs:          127,636  (    64,802 rd   +    62,834 wr)
==26127== LL misses:         64,855  (     2,044 rd   +    62,811 wr)
==26127== LL miss rate:         0.1% (       0.0%     +       3.1%  )
```

- bad log

```
==26138== Cachegrind, a cache and branch-prediction profiler
==26138== Copyright (C) 2002-2017, and GNU GPL'd, by Nicholas Nethercote et
al.
==26138== Using Valgrind-3.20.0 and LibVEX; rerun with -h for copyright info
==26138== Command: ./bad
==26138== Parent PID: 24578
==26138==
--26138-- warning: L3 cache found, using its data for the LL simulation.
--26138-- warning: specified LL cache: line_size 64  assoc 1  total_size
100,663,296
--26138-- warning: simulated LL cache: line_size 64  assoc 2  total_size
134,217,728
==26138==
==26138== I   refs:      30,160,090
==26138== I1  misses:         1,060
==26138== LLi misses:         1,042
==26138== I1  miss rate:       0.00%
==26138== LLi miss rate:       0.00%
==26138==
==26138== D   refs:      14,052,889  (12,039,556 rd   + 2,013,333 wr)
==26138== D1  misses:     2,001,580  ( 1,001,242 rd   + 1,000,338 wr)
==26138== LLd misses:        63,812  (     1,001 rd   +    62,811 wr)
==26138== D1  miss rate:       14.2% (       8.3%     +      49.7%  )
==26138== LLd miss rate:        0.5% (       0.0%     +       3.1%  )
==26138==
==26138== LL refs:        2,002,640  ( 1,002,302 rd   + 1,000,338 wr)
==26138== LL misses:         64,854  (     2,043 rd   +    62,811 wr)
==26138== LL miss rate:         0.1% (       0.0%     +       3.1%  )
```

3. Massif

> Q1. Please observe the relationship between time and memory allocation throughout the entire program execution, and provide one screenshot of the output file containing relevant information. (You must clearly display the total number of snapshots each time the system records the information in intervals).

- 總共有 76 個 snapshot（snapshot 0–75）。
- 追蹤用了 `--time-unit=B`，即 Massif 的「時間」是程式執行期間累積分配的記憶體位元組數。
- snapshot 0 中， `mem_heap_B` 和 `mem_heap_extra_B` 為 0。隨著程式執行，分配的記憶體逐漸上升。直到 snapshot 65 達到 peak，之後從 snapshot 66 開始到 75，可以看到 `mem_heap_B` 和 `mem_heap_extra_B` 開始下降，代表程式有釋放掉先前分配的一些堆記憶體。

```
#-----------
snapshot=65
#-----------
time=239600
mem_heap_B=239000
mem_heap_extra_B=600
mem_stacks_B=0
heap_tree=peak
n6: 239000 (heap allocation functions) malloc/new/new[], --alloc-fns, etc.
 n7: 116000 0x10919A: a (in /home/shawn/桌面/hw2/heap)
  n5: 64000 0x1091B4: b (in /home/shawn/桌面/hw2/heap)
   n4: 24000 0x1091CE: c (in /home/shawn/桌面/hw2/heap)
    n2: 8000 0x1091F7: d (in /home/shawn/桌面/hw2/heap)
     n1: 4000 0x109234: e (in /home/shawn/桌面/hw2/heap)
      n0: 4000 0x1092AF: main (in /home/shawn/桌面/hw2/heap)
     n0: 4000 0x1092C3: main (in /home/shawn/桌面/hw2/heap)
    n2: 8000 0x109201: d (in /home/shawn/桌面/hw2/heap)
     n1: 4000 0x109234: e (in /home/shawn/桌面/hw2/heap)
      n0: 4000 0x1092AF: main (in /home/shawn/桌面/hw2/heap)
     n0: 4000 0x1092C3: main (in /home/shawn/桌面/hw2/heap)
    n1: 4000 0x10922F: e (in /home/shawn/桌面/hw2/heap)
     n0: 4000 0x1092AF: main (in /home/shawn/桌面/hw2/heap)
    n0: 4000 0x1092BE: main (in /home/shawn/桌面/hw2/heap)
   n4: 24000 0x1091D8: c (in /home/shawn/桌面/hw2/heap)
    n2: 8000 0x1091F7: d (in /home/shawn/桌面/hw2/heap)
     n1: 4000 0x109234: e (in /home/shawn/桌面/hw2/heap)
      n0: 4000 0x1092AF: main (in /home/shawn/桌面/hw2/heap)
     n0: 4000 0x1092C3: main (in /home/shawn/桌面/hw2/heap)
    n2: 8000 0x109201: d (in /home/shawn/桌面/hw2/heap)
     n1: 4000 0x109234: e (in /home/shawn/桌面/hw2/heap)
      n0: 4000 0x1092AF: main (in /home/shawn/桌面/hw2/heap)
     n0: 4000 0x1092C3: main (in /home/shawn/桌面/hw2/heap)
    n1: 4000 0x10922F: e (in /home/shawn/桌面/hw2/heap)
     n0: 4000 0x1092AF: main (in /home/shawn/桌面/hw2/heap)
    n0: 4000 0x1092BE: main (in /home/shawn/桌面/hw2/heap)
   n2: 8000 0x1091F2: d (in /home/shawn/桌面/hw2/heap)
    n1: 4000 0x109234: e (in /home/shawn/桌面/hw2/heap)
     n0: 4000 0x1092AF: main (in /home/shawn/桌面/hw2/heap)
    n0: 4000 0x1092C3: main (in /home/shawn/桌面/hw2/heap)
   n1: 4000 0x109220: e (in /home/shawn/桌面/hw2/heap)
```

```
   n0: 4000 0x1092AF: main (in /home/shawn/桌面/hw2/heap)
  n0: 4000 0x1092B4: main (in /home/shawn/桌面/hw2/heap)
 n4: 24000 0x1091D3: c (in /home/shawn/桌面/hw2/heap)
  n2: 8000 0x1091F7: d (in /home/shawn/桌面/hw2/heap)
   n1: 4000 0x109234: e (in /home/shawn/桌面/hw2/heap)
    n0: 4000 0x1092AF: main (in /home/shawn/桌面/hw2/heap)
   n0: 4000 0x1092C3: main (in /home/shawn/桌面/hw2/heap)
  n2: 8000 0x109201: d (in /home/shawn/桌面/hw2/heap)
   n1: 4000 0x109234: e (in /home/shawn/桌面/hw2/heap)
    n0: 4000 0x1092AF: main (in /home/shawn/桌面/hw2/heap)
   n0: 4000 0x1092C3: main (in /home/shawn/桌面/hw2/heap)
  n1: 4000 0x10922F: e (in /home/shawn/桌面/hw2/heap)
   n0: 4000 0x1092AF: main (in /home/shawn/桌面/hw2/heap)
  n0: 4000 0x1092BE: main (in /home/shawn/桌面/hw2/heap)
 n2: 8000 0x1091FC: d (in /home/shawn/桌面/hw2/heap)
  n1: 4000 0x109234: e (in /home/shawn/桌面/hw2/heap)
   n0: 4000 0x1092AF: main (in /home/shawn/桌面/hw2/heap)
  n0: 4000 0x1092C3: main (in /home/shawn/桌面/hw2/heap)
 n2: 8000 0x109206: d (in /home/shawn/桌面/hw2/heap)
  n1: 4000 0x109234: e (in /home/shawn/桌面/hw2/heap)
   n0: 4000 0x1092AF: main (in /home/shawn/桌面/hw2/heap)
  n0: 4000 0x1092C3: main (in /home/shawn/桌面/hw2/heap)
 n1: 4000 0x109225: e (in /home/shawn/桌面/hw2/heap)
  n0: 4000 0x1092AF: main (in /home/shawn/桌面/hw2/heap)
 n1: 4000 0x10922A: e (in /home/shawn/桌面/hw2/heap)
  n0: 4000 0x1092AF: main (in /home/shawn/桌面/hw2/heap)
 n0: 4000 0x1092B9: main (in /home/shawn/桌面/hw2/heap)
n4: 48000 0x1091C9: c (in /home/shawn/桌面/hw2/heap)
 n2: 16000 0x1091F7: d (in /home/shawn/桌面/hw2/heap)
  n1: 8000 0x109234: e (in /home/shawn/桌面/hw2/heap)
   n0: 8000 0x1092AF: main (in /home/shawn/桌面/hw2/heap)
  n0: 8000 0x1092C3: main (in /home/shawn/桌面/hw2/heap)
 n2: 16000 0x109201: d (in /home/shawn/桌面/hw2/heap)
  n1: 8000 0x109234: e (in /home/shawn/桌面/hw2/heap)
   n0: 8000 0x1092AF: main (in /home/shawn/桌面/hw2/heap)
  n0: 8000 0x1092C3: main (in /home/shawn/桌面/hw2/heap)
 n1: 8000 0x10922F: e (in /home/shawn/桌面/hw2/heap)
  n0: 8000 0x1092AF: main (in /home/shawn/桌面/hw2/heap)
 n0: 8000 0x1092BE: main (in /home/shawn/桌面/hw2/heap)
n4: 32000 0x1091AF: b (in /home/shawn/桌面/hw2/heap)
 n3: 12000 0x1091CE: c (in /home/shawn/桌面/hw2/heap)
  n0: 4000 in 2 places, all below massif's threshold (1.00%)
  n1: 4000 0x1091F7: d (in /home/shawn/桌面/hw2/heap)
   n0: 4000 in 2 places, all below massif's threshold (1.00%)
  n1: 4000 0x109201: d (in /home/shawn/桌面/hw2/heap)
   n0: 4000 in 2 places, all below massif's threshold (1.00%)
 n3: 12000 0x1091D8: c (in /home/shawn/桌面/hw2/heap)
  n0: 4000 in 2 places, all below massif's threshold (1.00%)
  n1: 4000 0x1091F7: d (in /home/shawn/桌面/hw2/heap)
   n0: 4000 in 2 places, all below massif's threshold (1.00%)
  n1: 4000 0x109201: d (in /home/shawn/桌面/hw2/heap)
   n0: 4000 in 2 places, all below massif's threshold (1.00%)
 n0: 4000 in 2 places, all below massif's threshold (1.00%)
 n1: 4000 0x1091F2: d (in /home/shawn/桌面/hw2/heap)
  n0: 4000 in 2 places, all below massif's threshold (1.00%)
n2: 32000 0x1091ED: d (in /home/shawn/桌面/hw2/heap)
 n1: 16000 0x109234: e (in /home/shawn/桌面/hw2/heap)
  n0: 16000 0x1092AF: main (in /home/shawn/桌面/hw2/heap)
```

```
  n0: 16000 0x1092C3: main (in /home/shawn/桌面/hw2/heap)
 n0: 10000 0x109293: main (in /home/shawn/桌面/hw2/heap)
 n0: 1000 in 1 place, below massif's threshold (1.00%)
```

▶ massif.out.44295

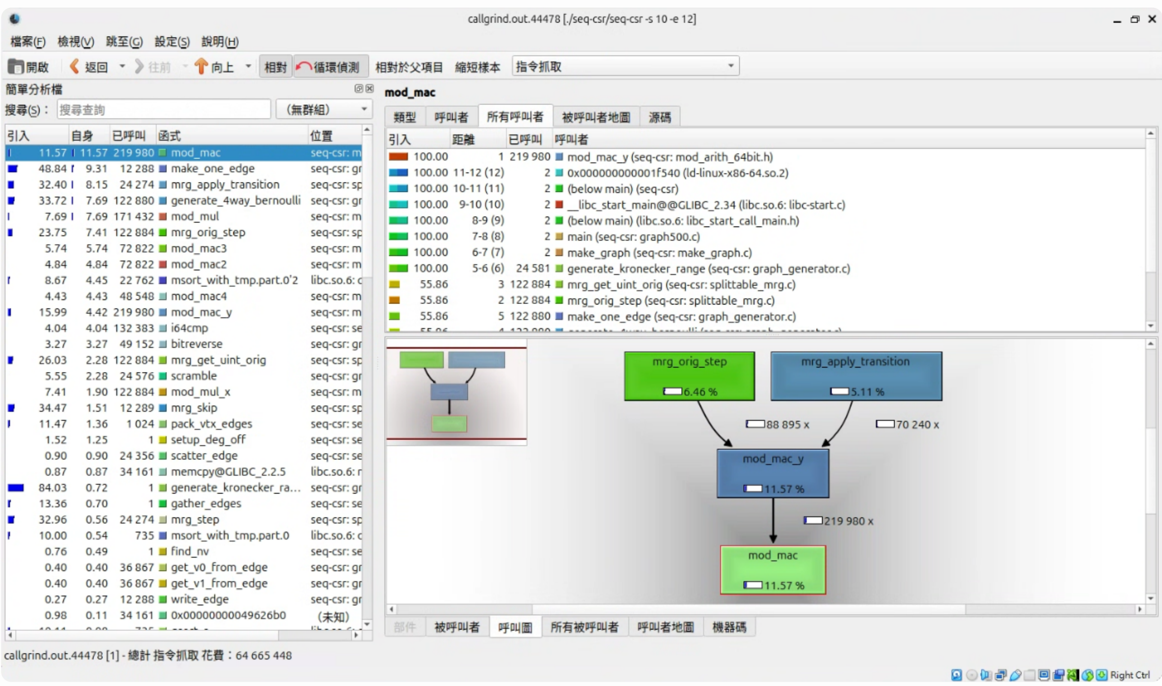
> Q2. Then, point out how many bytes are allocated and used at peak respectively.

- peak 時，累積分配了 239600 bytes。實際使用 239000 bytes，並且有 600 bytes 的額外管理開銷。

4. Callgrind

> Q1. Please use kcachegrind GUI to indicate which function is most expensive in terms of time (excluding the time of their callee functions). Please include a screenshot of the call graph.

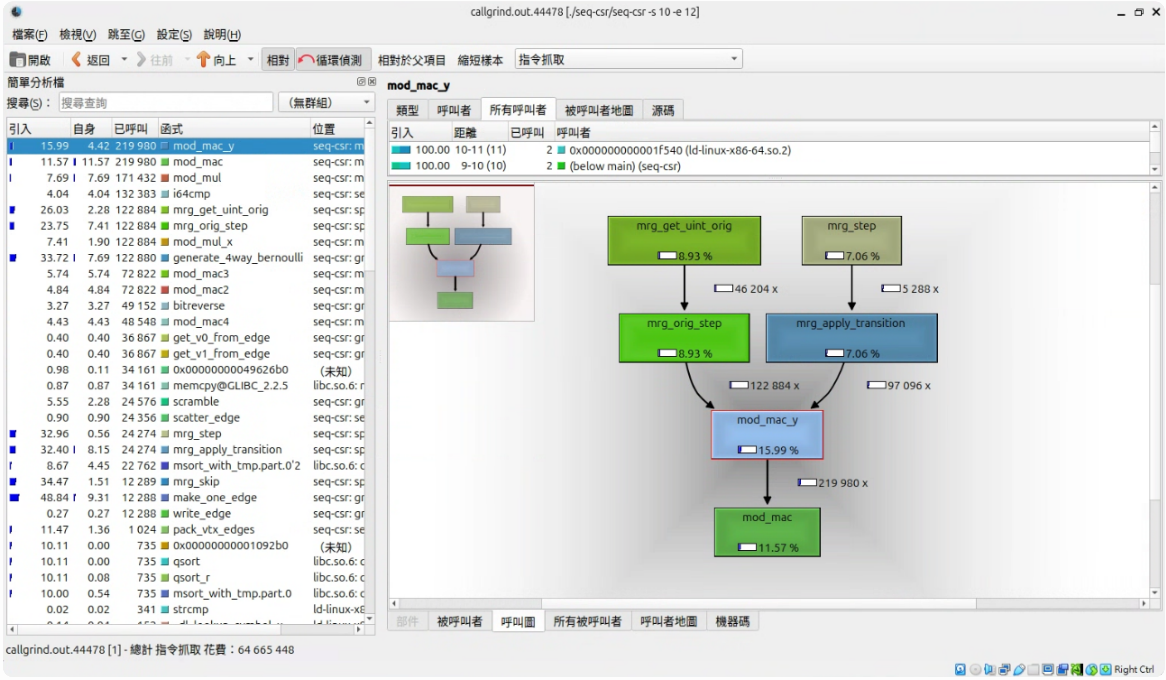
- 自身執行最久的是 `mod_mac` 函式。



> Q2. Point out which function is called most frequently, and identify its caller as well. Please include a screenshot of the call graph.

- 被 called 最多次的函式是 `mod_mac` 和 `mod_mac_y`。
  `mod_mac` 被 `mod_mac_y` call 了 219980 次。
  `mod_mac_y` 被 `mrg_orig_step` call 122884 次、被 `mrg_apply_transition` call 97096

次。



5. Pytorch profiler

> Q1. Please follow the tutorial to get the columns shown in the picture below.
>
> ```
> ymd@valgrindenv:~/Desktop/Pytorch$ python3 test2.py
> cpu
> STAGE:2023-04-25 14:30:13 4634:4634 ActivityProfilerController.cpp:311] Completed Stage: Warm Up
> STAGE:2023-04-25 14:30:13 4634:4634 ActivityProfilerController.cpp:317] Completed Stage: Collection
> STAGE:2023-04-25 14:30:13 4634:4634 ActivityProfilerController.cpp:321] Completed Stage: Post Processing
> ---------------------------  ------------  ------------  ------------  ------------  ------------  ------------  ------------  ------------
>                        Name    Self CPU %      Self CPU   CPU total %     CPU total  CPU time avg       CPU Mem  Self CPU Mem    # of Calls
> ---------------------------  ------------  ------------  ------------  ------------  ------------  ------------  ------------  ------------
> ```
>
> Please provide a screenshot of the analysis result, ensuring that the username and machinename are visible in the first line. Then, identify the top three functions in terms of CPU time excluding the time of their callee functions, except for the model label (e.g. model_inference in tutorials).

- top three functions : `aten::addmm`(49.895 ms)、`aten::copy_`(8.128 ms)、`aten::native_layer_norm`(3.011 ms)

```
(venv) shawn@shawn-MASSVM:~/桌面/hw2$ python3 pytorch_profile.py
---------------------------  ------------  ------------  ------------  ------------  ------------  ------------  ------------  ------------
                       Name    Self CPU %      Self CPU   CPU total %     CPU total  CPU time avg       CPU Mem  Self CPU Mem    # of Calls
---------------------------  ------------  ------------  ------------  ------------  ------------  ------------  ------------  ------------
                aten::addmm        54.95%      49.859ms        59.74%      54.197ms     558.737us      13.84 Mb      13.84 Mb            97
            model_inference        16.13%      14.631ms       100.00%      90.729ms      90.729ms           0 b     -34.88 Mb             1
                aten::copy_         8.96%       8.128ms         8.96%       8.128ms      36.125us           0 b           0 b           225
    aten::native_layer_norm         3.32%       3.011ms         3.49%       3.169ms     105.641us       2.94 Mb        1000 b            30
               aten::arange         2.29%       2.074ms         4.59%       4.165ms       1.041ms         320 b           0 b             4
               aten::linear         2.07%       1.876ms        62.57%      56.767ms     585.222us      13.84 Mb           0 b            97
                  aten::add         1.90%       1.719ms         1.90%       1.719ms      53.732us       3.12 Mb       3.12 Mb            32
                 aten::tril         1.81%       1.638ms         1.81%       1.638ms       1.638ms         400 b         400 b             1
            aten::clamp_min         1.17%       1.061ms         1.17%       1.061ms      88.448us       4.69 Mb       4.69 Mb            12
                  aten::bmm         0.98%     890.585us         0.99%     897.659us      24.935us       2.03 Mb       2.03 Mb            36
---------------------------  ------------  ------------  ------------  ------------  ------------  ------------  ------------  ------------
Self CPU time total: 90.729ms
```

> Q2. Output the profiling results to output.json and analyze in Chrome trace viewer. Take a screenshot of the visualization and point out which two functions (colors)

> appear the most (in terms of time), except for the model label (e.g. model_inference in tutorials).

- 在 Chrome trace viewer 中，時間最長的兩個函式（排除 `model_inference`）分別是 `aten::linear`（綠色） 和 `aten::addmm`（粉紅色）。