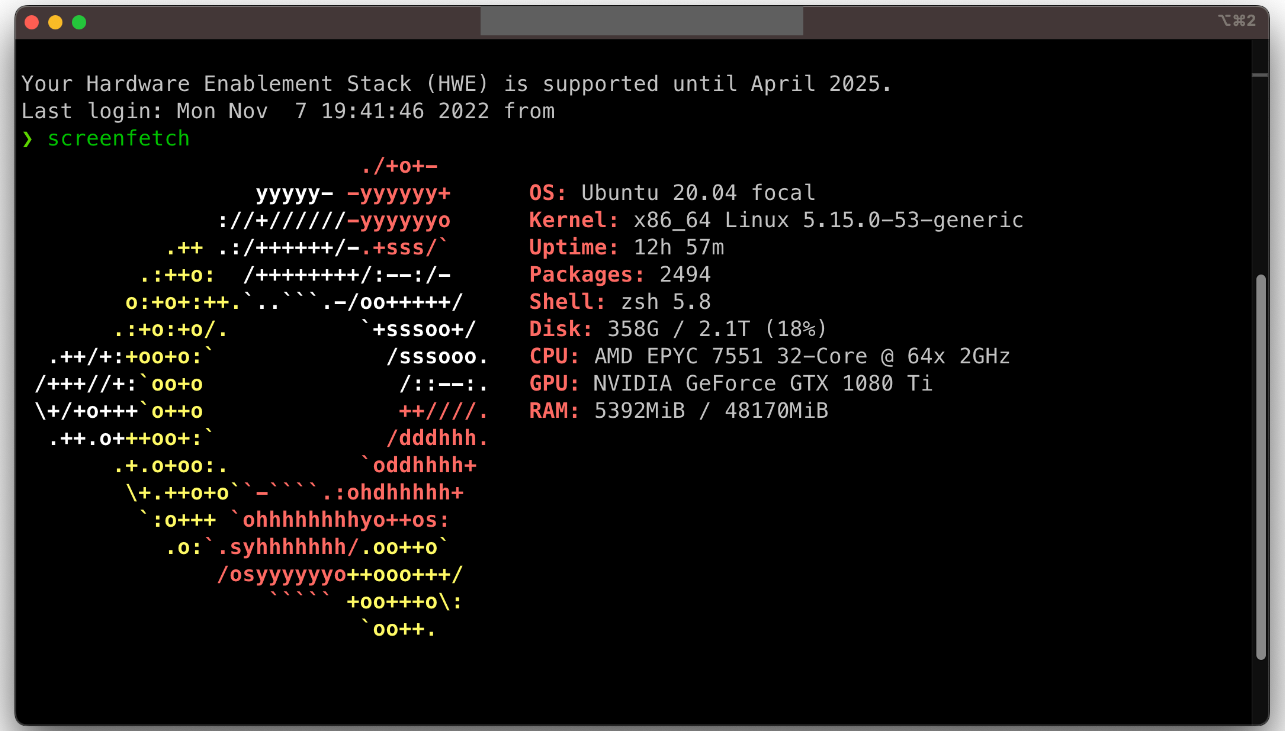
CSC 3150 Assignment 3 Report

Tianhao Shi, 120090472

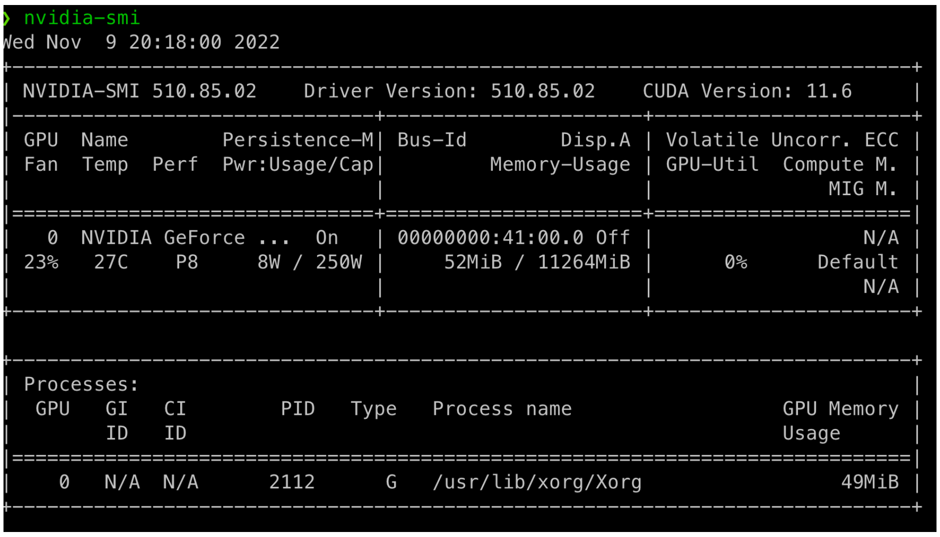
Nov, 2022

Environment



~~~
Your Hardware Enablement Stack (HWE) is supported until April 2025.
Last login: Mon Nov  7 19:41:46 2022 from
❯ screenfetch
                          ./+o+-
                  yyyyy- -yyyyyy+      OS: Ubuntu 20.04 focal
               ://+//////-yyyyyyo      Kernel: x86_64 Linux 5.15.0-53-generic
           .++ .:/++++++/-.+sss/`      Uptime: 12h 57m
         .:++o:  /+++++++++/:--:/-     Packages: 2494
        o:+o+:++.`..```.-/oo+++++/     Shell: zsh 5.8
       .:+o:+o/.          `+sssoo+/    Disk: 358G / 2.1T (18%)
  .++/+:+oo+o:`             /sssooo.   CPU: AMD EPYC 7551 32-Core @ 64x 2GHz
 /+++//+:`oo+o               /::--:.   GPU: NVIDIA GeForce GTX 1080 Ti
 \+/+o+++`o++o               ++////.   RAM: 5392MiB / 48170MiB
  .++.o+++oo+:`             /dddhhh.
       .+.o+oo:.          `oddhhhhh+
        \+.++o+o``-````.:ohdhhhhh+
         `:o+++ `ohhhhhhhhyo++os:
           .o:`.syhhhhhhh/.oo++o`
               /osyyyyyyo++ooo+++/
                   ````` +oo+++o\:
                          `oo++.
~~~

- CUDA Version: 11.6



```
❯ nvidia-smi
Wed Nov  9 20:18:00 2022
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 510.85.02    Driver Version: 510.85.02    CUDA Version: 11.6     |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|                               |                      |               MIG M. |
|===============================+======================+======================|
|   0  NVIDIA GeForce ...  On   | 00000000:41:00.0 Off |                  N/A |
| 23%   27C    P8     8W / 250W |     52MiB / 11264MiB |      0%      Default |
|                               |                      |                  N/A |
+-------------------------------+----------------------+----------------------+

+-----------------------------------------------------------------------------+
| Processes:                                                                  |
|  GPU   GI   CI        PID   Type   Process name                  GPU Memory |
|        ID   ID                                                   Usage      |
|=============================================================================|
|    0   N/A  N/A      2112      G   /usr/lib/xorg/Xorg                 49MiB |
+-----------------------------------------------------------------------------+
```

Also tested on the school's cluster

Execution

Method 1: Use makefile

```
cd Assignment_3_120090472/source
make #(compile+run)
# make build #(build only)
```

Method 2: Batch file

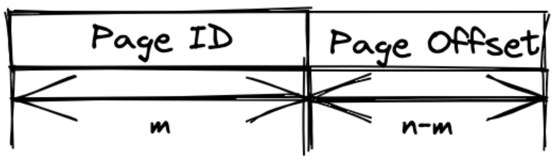
```
cd Assignment_3_120090472/source
bash ./slurm.sh
```

Design of Key Components

Inverted Page Table

A page table is a one-to-one mapping betwen the virtual address and the physical address. In an inverted table, the index is the physical address (frames) and the virtual page id is stored as elements of the table. During address translation, the CPU issues a virtual address. The address is divided into two parts, **page id** and **page offset**, using the following rule:

For a virtual address with *n* bits, if the page table entry is of size 2^m, the upper *m* bits are used as page id, and the rest *n-m* bits are used as page offset.



LRU

LRU, or "Least Recently Used", is a page replacement policy that OS use to decide which page to swap out in the inverted page table. This policy always choose, as the name suggests, the least recently used page to swap out of RAM (and page table). In my design, the underlying data structure is a **doubly linked list**. I store two pointers in the VirtualMemory struct, which tracks the *head* and *tail* of the linked list. The *head* points to the newest item in queue, while the tail points to the oldest. The update of LRU is not directly relevant to page hit/page miss. Instead, LRU is update when:

1. Page hit, but the requested page is not the latest used page.
2. Page miss, which means the page is previously not in the LRU list.

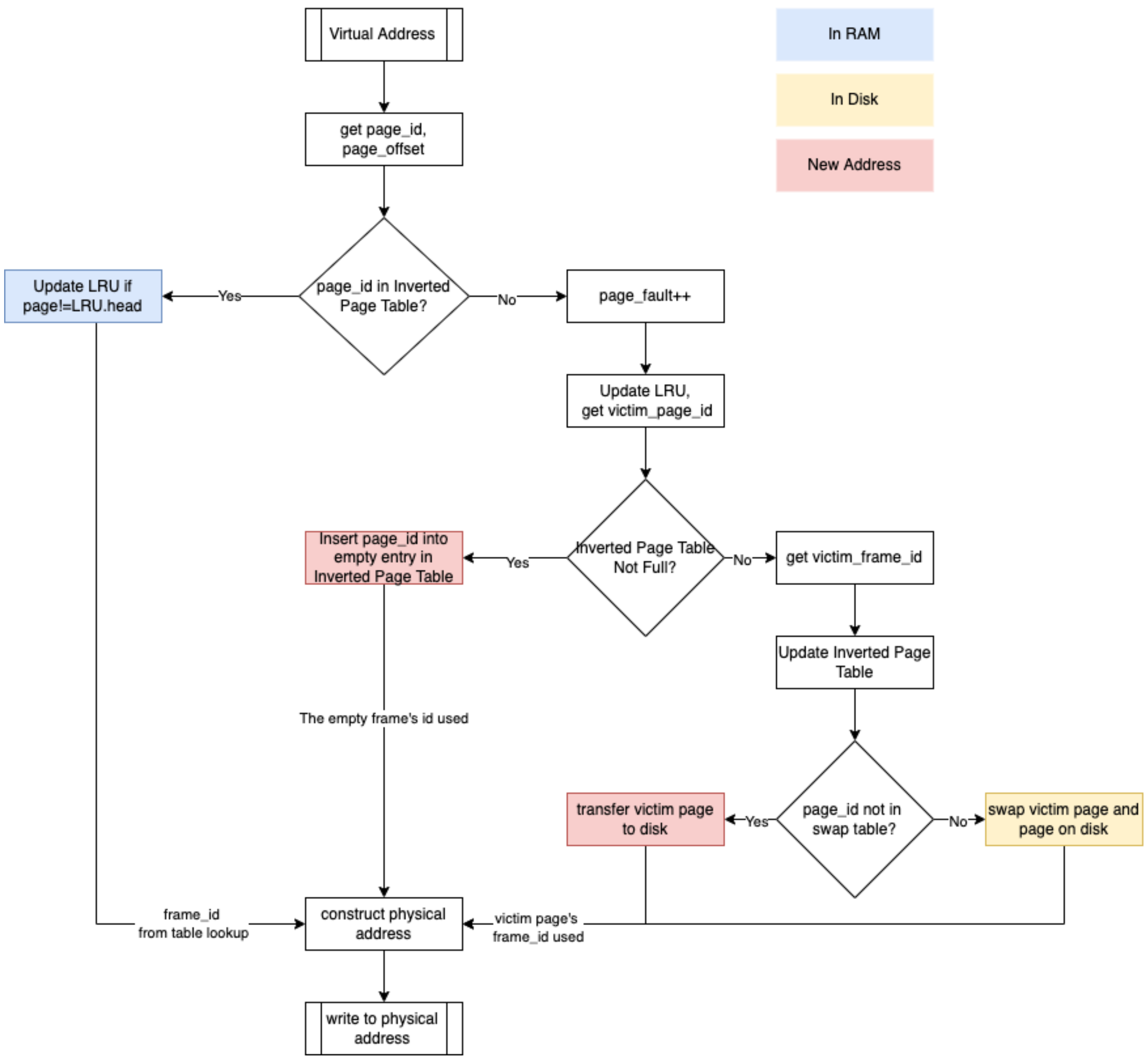
However, there are minor differences between the two scenarios. Namely, in the page hit scenario,all we need to do is traverse the LRU list and **perform a linear search**. The found node is moved to the head of the linked list.

A page miss indicates the requested page is not in the RAM. Therefore, there is no need to search. We can simply **append the node to the head** and update the list size. Notice that there is a limit for LRU size. When overflow occurs, the extraneous node is the LRU node and is removed from the linked list.

Swap table

Our virtual memory size is 160KB, but the RAM size is only 32KB. This means for any input larger than the RAM size, there must be some data that resides in disk. To find the physical address of a page in disk, a swap table is used. The structure of swap table is similar to the inverted page table. The disk address is also partitioned into 32 byte pages, giving the swap table 4096 entries. Whenever swapping happens, the page id of the queried virtual address is searched in the swap table.

Program Flow



Virtual Address
In RAM
In Disk
New Address
get page_id, page_offset
page_id in Inverted Page Table?
Yes
Update LRU if page!=LRU.head
No
page_fault++
Update LRU, get victim_page_id
Inverted Page Table Not Full?
Yes
Insert page_id into empty entry in Inverted Page Table
No
get victim_frame_id
Update Inverted Page Table
The empty frame's id used
page_id not in swap table?
Yes
transfer victim page to disk
No
swap victim page and page on disk
frame_id from table lookup
construct physical address
victim page's frame_id used
write to physical address

The `vm_write` and `vm_read` function are almost identical. Here, I use `vm_write` as an example to illustrate the flow of my program.

For any virtual address, it is pointing to one of the following places:

1. A physical address in RAM
2. A physical address in Disk
3. Unallocated

If a virtual address is pointing to RAM, then the page_id must be stored in the inverted page table. Therefore, if the return value of querying page table is not -1, then we can construct the physical address using the corresponding frame_id.

If the page_id is not stored in RAM, **a page fault occurs**, and we must increment the page_fault counter.

If the page table is not full, then we can always insert the page to the first vacant place (there is no point swapping if we can store data in RAM).

However, if the page table is full, we **must move victim a page from RAM to disk**. The selection of this 'victim' is conducted using the previously mentioned **LRU** algorithm. If the swap table contains the requested page_id, then **the desired page is moved from disk to RAM**.

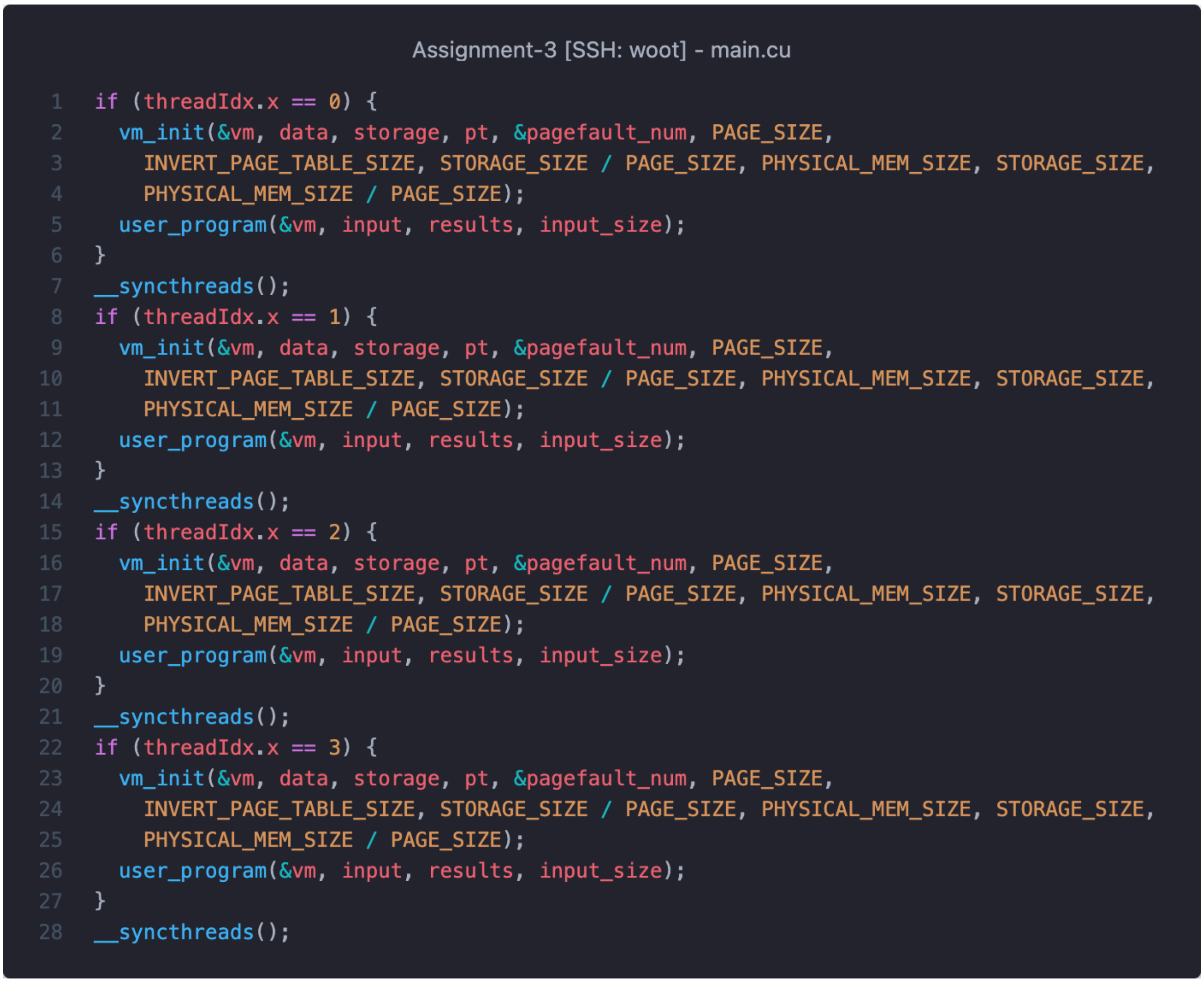
`vm_read` is almost identical except it reads from the final physical address.

Bonus Design

Due to the limitted time, I implemented the bonus using the second version described in the additional note. The code is almost identical compared to the code in the first task.

However, notibaly the following things are added:

1. In `main.cu`, we need to launch four threads. Therefore, we call `mykernel <<<1, 4, INVERT_PAGE_TABLE_SIZE >>> (input_size);`
2. It seems resolving race condition is not very straightforward (hard to implement `mutex` used in assignment 2). However, the CUDA library do provide a `__syncthreads()` function. This function serves as a barrier, blocking threads until every thread reches this function. Using this function, the threads are executed consequtively, in the order of 0->1->2->3. Each time, the vm is initialized. However, the page fault pointer is not cleaned and is continuously updating, allowing the program to count the overall fault number.


Assignment-3 [SSH: woot] - main.cu

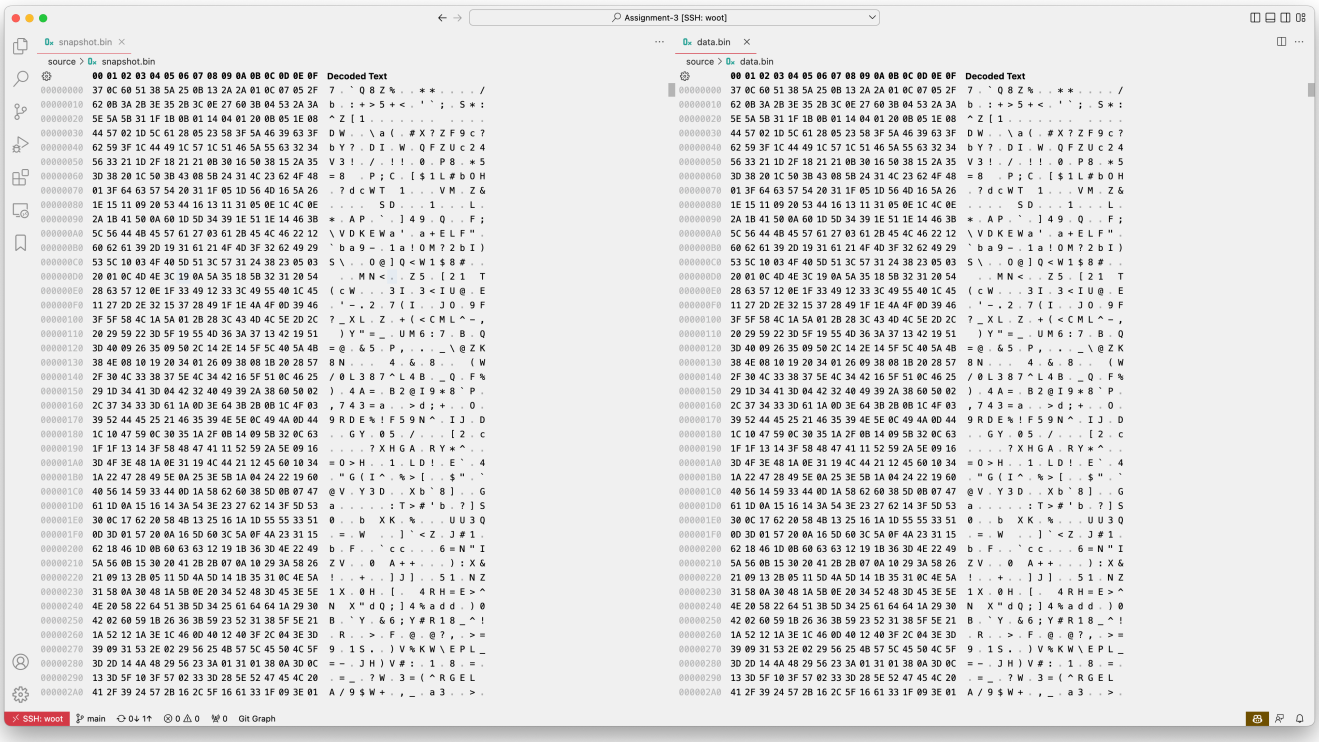
```
if (threadIdx.x == 0) {
  vm_init(&vm, data, storage, pt, &pagefault_num, PAGE_SIZE,
    INVERT_PAGE_TABLE_SIZE, STORAGE_SIZE / PAGE_SIZE, PHYSICAL_MEM_SIZE, STORAGE_SIZE,
    PHYSICAL_MEM_SIZE / PAGE_SIZE);
  user_program(&vm, input, results, input_size);
}
__syncthreads();
if (threadIdx.x == 1) {
  vm_init(&vm, data, storage, pt, &pagefault_num, PAGE_SIZE,
    INVERT_PAGE_TABLE_SIZE, STORAGE_SIZE / PAGE_SIZE, PHYSICAL_MEM_SIZE, STORAGE_SIZE,
    PHYSICAL_MEM_SIZE / PAGE_SIZE);
  user_program(&vm, input, results, input_size);
}
__syncthreads();
if (threadIdx.x == 2) {
  vm_init(&vm, data, storage, pt, &pagefault_num, PAGE_SIZE,
    INVERT_PAGE_TABLE_SIZE, STORAGE_SIZE / PAGE_SIZE, PHYSICAL_MEM_SIZE, STORAGE_SIZE,
    PHYSICAL_MEM_SIZE / PAGE_SIZE);
  user_program(&vm, input, results, input_size);
}
__syncthreads();
if (threadIdx.x == 3) {
  vm_init(&vm, data, storage, pt, &pagefault_num, PAGE_SIZE,
    INVERT_PAGE_TABLE_SIZE, STORAGE_SIZE / PAGE_SIZE, PHYSICAL_MEM_SIZE, STORAGE_SIZE,
    PHYSICAL_MEM_SIZE / PAGE_SIZE);
  user_program(&vm, input, results, input_size);
}
__syncthreads();
```

Output and Analysis:

For testcase 1, the output of my program is 8193:

```
❯ make
nvcc --relocatable-device-code=true -g -G main.cu user_program.cu virtual_memory.cu -o main
./main
pagefault number is 8193
```

snapshot output



```
snapshot.bin
source > snapshot.bin
          00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F  Decoded Text
00000000  37 0C 60 51 38 5A 25 0B 13 2A 2A 01 0C 07 05 2F  7 . ` Q 8 Z % . . * * . . . . /
00000010  62 0B 3A 2B 3E 35 2B 3C 0E 27 60 3B 04 53 2A 3A  b . : + > 5 + < . ' ` ; . S * :
00000020  5E 5A 5B 31 1F 1B 0B 01 14 04 01 20 0B 05 1E 08  ^ Z [ 1 . . . . . . .   . . . .
00000030  44 57 02 1D 5C 61 28 05 23 58 3F 5A 46 39 63 3F  D W . . \ a ( . # X ? Z F 9 c ?
00000040  62 59 3F 1C 44 49 1C 57 1C 51 46 5A 55 63 32 34  b Y ? . D I . W . Q F Z U c 2 4
00000050  56 33 21 1D 2F 18 21 21 0B 30 16 50 38 15 2A 35  V 3 ! . / . ! ! . 0 . P 8 . * 5
00000060  3D 38 20 1C 50 3B 43 08 5B 24 31 4C 23 62 4F 48  = 8   . P ; C . [ $ 1 L # b O H
00000070  01 3F 64 63 57 54 20 31 1F 05 1D 56 4D 16 5A 26  . ? d c W T   1 . . . V M . Z &
00000080  1E 15 11 09 20 53 44 16 13 11 31 05 0E 1C 4C 0E  . . . .   S D . . . 1 . . . L .
00000090  2A 1B 41 50 0A 60 1D 5D 34 39 1E 51 1E 14 46 3B  * . A P . ` . ] 4 9 . Q . . F ;
000000A0  5C 56 44 4B 45 57 61 27 03 61 2B 45 4C 46 22 12  \ V D K E W a ' . a + E L F " .
000000B0  60 62 61 39 2D 19 31 61 21 4F 4D 3F 32 62 49 29  ` b a 9 - . 1 a ! O M ? 2 b I )
000000C0  53 5C 10 03 4F 40 5D 51 3C 57 31 24 38 23 05 03  S \ . . O @ ] Q < W 1 $ 8 # . .
000000D0  20 01 0C 4D 4E 3C 19 0A 5A 35 18 5B 32 31 20 54    . . M N < . . Z 5 . [ 2 1   T
000000E0  28 63 57 12 0E 1F 33 49 12 33 3C 49 55 40 1C 45  ( c W . . . 3 I . 3 < I U @ . E
000000F0  11 27 2D 2E 32 15 37 28 49 1F 1E 4A 4F 0D 39 46  . ' - . 2 . 7 ( I . . J O . 9 F
00000100  3F 5F 58 4C 1A 5A 01 2B 28 3C 43 4D 4C 5E 2D 2C  ? _ X L . Z . + ( < C M L ^ - ,
00000110  20 29 59 22 3D 5F 19 55 4D 36 3A 37 13 42 19 51    ) Y " = _ . U M 6 : 7 . B . Q
00000120  3D 40 09 26 35 09 50 2C 14 2E 14 5F 5C 40 5A 4B  = @ . & 5 . P , . . . _ \ @ Z K
00000130  38 4E 08 10 19 20 34 01 26 09 38 08 1B 20 28 57  8 N . . .   4 . & . 8 . .   ( W
00000140  2F 30 4C 33 38 37 5E 4C 34 42 16 5F 51 0C 46 25  / 0 L 3 8 7 ^ L 4 B . _ Q . F %
00000150  29 1D 34 41 3D 04 42 32 40 49 39 2A 38 60 50 02  ) . 4 A = . B 2 @ I 9 * 8 ` P .
00000160  2C 37 34 33 3D 61 1A 0D 3E 64 3B 2B 0B 1C 4F 03  , 7 4 3 = a . . > d ; + . . O .
00000170  39 52 44 45 25 21 46 35 39 4E 5E 0C 49 4A 0D 44  9 R D E % ! F 5 9 N ^ . I J . D
00000180  1C 10 47 59 0C 30 35 1A 2F 0B 14 09 5B 32 0C 63  . . G Y . 0 5 . / . . . [ 2 . c
00000190  1F 1F 13 14 3F 58 48 47 41 11 52 59 2A 5E 09 16  . . . . ? X H G A . R Y * ^ . .
000001A0  3D 4F 3E 48 1A 0E 31 19 4C 44 21 12 45 60 10 34  = O > H . . 1 . L D ! . E ` . 4
000001B0  1A 22 47 28 49 5E 0A 25 3E 5B 1A 04 24 22 19 60  . " G ( I ^ . % > [ . . $ " . `
000001C0  40 56 14 59 33 44 0D 1A 58 62 60 38 5D 0B 07 47  @ V . Y 3 D . . X b ` 8 ] . . G
000001D0  61 1D 0A 15 16 14 3A 54 3E 23 27 62 14 3F 5D 53  a . . . . . : T > # ' b . ? ] S
000001E0  30 0C 17 62 20 58 4B 13 25 16 1A 1D 55 55 33 51  0 . . b   X K . % . . . U U 3 Q
000001F0  0D 3D 01 57 20 0A 16 5D 60 3C 5A 0F 4A 23 31 15  . = . W   . . ] ` < Z . J # 1 .
00000200  62 18 46 1D 0B 60 63 63 12 19 1B 36 3D 4E 22 49  b . F . . ` c c . . . 6 = N " I
00000210  5A 56 0B 15 30 20 41 2B 2B 07 0A 10 29 3A 58 26  Z V . . 0   A + + . . . ) : X &
00000220  21 09 13 2B 05 11 5D 4A 5D 14 1B 35 31 0C 4E 5A  ! . . + . . ] J ] . . 5 1 . N Z
00000230  31 58 0A 30 48 1A 5B 0E 20 34 52 48 3D 45 3E 5E  1 X . 0 H . [ .   4 R H = E > ^
00000240  4E 20 58 22 64 51 3B 5D 34 25 61 64 64 1A 29 30  N   X " d Q ; ] 4 % a d d . ) 0
00000250  42 02 60 59 1B 26 36 3B 59 23 52 31 38 5F 5E 21  B . ` Y . & 6 ; Y # R 1 8 _ ^ !
00000260  1A 52 12 1A 3E 1C 46 0D 40 12 40 3F 2C 04 3E 3D  . R . . > . F . @ . @ ? , . > =
00000270  39 09 31 53 2E 02 29 56 25 4B 57 5C 45 50 4C 5F  9 . 1 S . . ) V % K W \ E P L _
00000280  3D 2D 14 4A 48 29 56 23 3A 01 31 01 38 0A 3D 0C  = - . J H ) V # : . 1 . 8 . = .
00000290  13 3D 5F 10 3F 57 02 33 3D 28 5E 52 47 45 4C 20  . = _ . ? W . 3 = ( ^ R G E L
000002A0  41 2F 39 24 57 2B 16 2C 5F 16 61 33 1F 09 3E 01  A / 9 $ W + . , _ . a 3 . . > .
```

```
data.bin
source > data.bin
          00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F  Decoded Text
00000000  37 0C 60 51 38 5A 25 0B 13 2A 2A 01 0C 07 05 2F  7 . ` Q 8 Z % . . * * . . . . /
00000010  62 0B 3A 2B 3E 35 2B 3C 0E 27 60 3B 04 53 2A 3A  b . : + > 5 + < . ' ` ; . S * :
00000020  5E 5A 5B 31 1F 1B 0B 01 14 04 01 20 0B 05 1E 08  ^ Z [ 1 . . . . . . .   . . . .
00000030  44 57 02 1D 5C 61 28 05 23 58 3F 5A 46 39 63 3F  D W . . \ a ( . # X ? Z F 9 c ?
00000040  62 59 3F 1C 44 49 1C 57 1C 51 46 5A 55 63 32 34  b Y ? . D I . W . Q F Z U c 2 4
00000050  56 33 21 1D 2F 18 21 21 0B 30 16 50 38 15 2A 35  V 3 ! . / . ! ! . 0 . P 8 . * 5
00000060  3D 38 20 1C 50 3B 43 08 5B 24 31 4C 23 62 4F 48  = 8   . P ; C . [ $ 1 L # b O H
00000070  01 3F 64 63 57 54 20 31 1F 05 1D 56 4D 16 5A 26  . ? d c W T   1 . . . V M . Z &
00000080  1E 15 11 09 20 53 44 16 13 11 31 05 0E 1C 4C 0E  . . . .   S D . . . 1 . . . L .
00000090  2A 1B 41 50 0A 60 1D 5D 34 39 1E 51 1E 14 46 3B  * . A P . ` . ] 4 9 . Q . . F ;
000000A0  5C 56 44 4B 45 57 61 27 03 61 2B 45 4C 46 22 12  \ V D K E W a ' . a + E L F " .
000000B0  60 62 61 39 2D 19 31 61 21 4F 4D 3F 32 62 49 29  ` b a 9 - . 1 a ! O M ? 2 b I )
000000C0  53 5C 10 03 4F 40 5D 51 3C 57 31 24 38 23 05 03  S \ . . O @ ] Q < W 1 $ 8 # . .
000000D0  20 01 0C 4D 4E 3C 19 0A 5A 35 18 5B 32 31 20 54    . . M N < . . Z 5 . [ 2 1   T
000000E0  28 63 57 12 0E 1F 33 49 12 33 3C 49 55 40 1C 45  ( c W . . . 3 I . 3 < I U @ . E
000000F0  11 27 2D 2E 32 15 37 28 49 1F 1E 4A 4F 0D 39 46  . ' - . 2 . 7 ( I . . J O . 9 F
00000100  3F 5F 58 4C 1A 5A 01 2B 28 3C 43 4D 4C 5E 2D 2C  ? _ X L . Z . + ( < C M L ^ - ,
00000110  20 29 59 22 3D 5F 19 55 4D 36 3A 37 13 42 19 51    ) Y " = _ . U M 6 : 7 . B . Q
00000120  3D 40 09 26 35 09 50 2C 14 2E 14 5F 5C 40 5A 4B  = @ . & 5 . P , . . . _ \ @ Z K
00000130  38 4E 08 10 19 20 34 01 26 09 38 08 1B 20 28 57  8 N . . .   4 . & . 8 . .   ( W
00000140  2F 30 4C 33 38 37 5E 4C 34 42 16 5F 51 0C 46 25  / 0 L 3 8 7 ^ L 4 B . _ Q . F %
00000150  29 1D 34 41 3D 04 42 32 40 49 39 2A 38 60 50 02  ) . 4 A = . B 2 @ I 9 * 8 ` P .
00000160  2C 37 34 33 3D 61 1A 0D 3E 64 3B 2B 0B 1C 4F 03  , 7 4 3 = a . . > d ; + . . O .
00000170  39 52 44 45 25 21 46 35 39 4E 5E 0C 49 4A 0D 44  9 R D E % ! F 5 9 N ^ . I J . D
00000180  1C 10 47 59 0C 30 35 1A 2F 0B 14 09 5B 32 0C 63  . . G Y . 0 5 . / . . . [ 2 . c
00000190  1F 1F 13 14 3F 58 48 47 41 11 52 59 2A 5E 09 16  . . . . ? X H G A . R Y * ^ . .
000001A0  3D 4F 3E 48 1A 0E 31 19 4C 44 21 12 45 60 10 34  = O > H . . 1 . L D ! . E ` . 4
000001B0  1A 22 47 28 49 5E 0A 25 3E 5B 1A 04 24 22 19 60  . " G ( I ^ . % > [ . . $ " . `
000001C0  40 56 14 59 33 44 0D 1A 58 62 60 38 5D 0B 07 47  @ V . Y 3 D . . X b ` 8 ] . . G
000001D0  61 1D 0A 15 16 14 3A 54 3E 23 27 62 14 3F 5D 53  a . . . . . : T > # ' b . ? ] S
000001E0  30 0C 17 62 20 58 4B 13 25 16 1A 1D 55 55 33 51  0 . . b   X K . % . . . U U 3 Q
000001F0  0D 3D 01 57 20 0A 16 5D 60 3C 5A 0F 4A 23 31 15  . = . W   . . ] ` < Z . J # 1 .
00000200  62 18 46 1D 0B 60 63 63 12 19 1B 36 3D 4E 22 49  b . F . . ` c c . . . 6 = N " I
00000210  5A 56 0B 15 30 20 41 2B 2B 07 0A 10 29 3A 58 26  Z V . . 0   A + + . . . ) : X &
00000220  21 09 13 2B 05 11 5D 4A 5D 14 1B 35 31 0C 4E 5A  ! . . + . . ] J ] . . 5 1 . N Z
00000230  31 58 0A 30 48 1A 5B 0E 20 34 52 48 3D 45 3E 5E  1 X . 0 H . [ .   4 R H = E > ^
00000240  4E 20 58 22 64 51 3B 5D 34 25 61 64 64 1A 29 30  N   X " d Q ; ] 4 % a d d . ) 0
00000250  42 02 60 59 1B 26 36 3B 59 23 52 31 38 5F 5E 21  B . ` Y . & 6 ; Y # R 1 8 _ ^ !
00000260  1A 52 12 1A 3E 1C 46 0D 40 12 40 3F 2C 04 3E 3D  . R . . > . F . @ . @ ? , . > =
00000270  39 09 31 53 2E 02 29 56 25 4B 57 5C 45 50 4C 5F  9 . 1 S . . ) V % K W \ E P L _
00000280  3D 2D 14 4A 48 29 56 23 3A 01 31 01 38 0A 3D 0C  = - . J H ) V # : . 1 . 8 . = .
00000290  13 3D 5F 10 3F 57 02 33 3D 28 5E 52 47 45 4C 20  . = _ . ? W . 3 = ( ^ R G E L
000002A0  41 2F 39 24 57 2B 16 2C 5F 16 61 33 1F 09 3E 01  A / 9 $ W + . , _ . a 3 . . > .
```

Cmp command:

```
❯ cmp -l data.bin snapshot.bin | gawk '{printf "%08X %02X %02X\n", $1, strtonum(0$2), strtonum(0$3)}'
```

Analysis:

The user program of testcase 1 does the following things:

```
1. Write [0, 132k) of the data.bin to VM starting at address 0 in ascending order.
1. Read virtual address [132k-1, 96k-1]
1. Take a snapshot of the virtual memory with offset 0
```

During these procedures, the first step contributes 132k/32=4k page faults, as they are all new pages. The second step only contributes to 1 page fault, because it reads all the content stored in the RAM (page hit), plus one element in the disk. The final step also contributes 4k page faults, because it reads from the beginning of VM. This means the pages loaded in step 2 are all replaced starting from address 0.

Threfore, totally we have $8 \times 2^{10} + 1 = 8193$ page faults.

A visualization is as followed

For testcase 2, the output of my program is 9125

```
❯ mv user_program.c user_program.c.back
❯ mv user_program2.c user_program.c
❯ make
nvcc --relocatable-device-code=true -g -G main.cu user_program.cu virtual_memory.cu -o main
./main
pagefault number is 9125
```

Analysis:

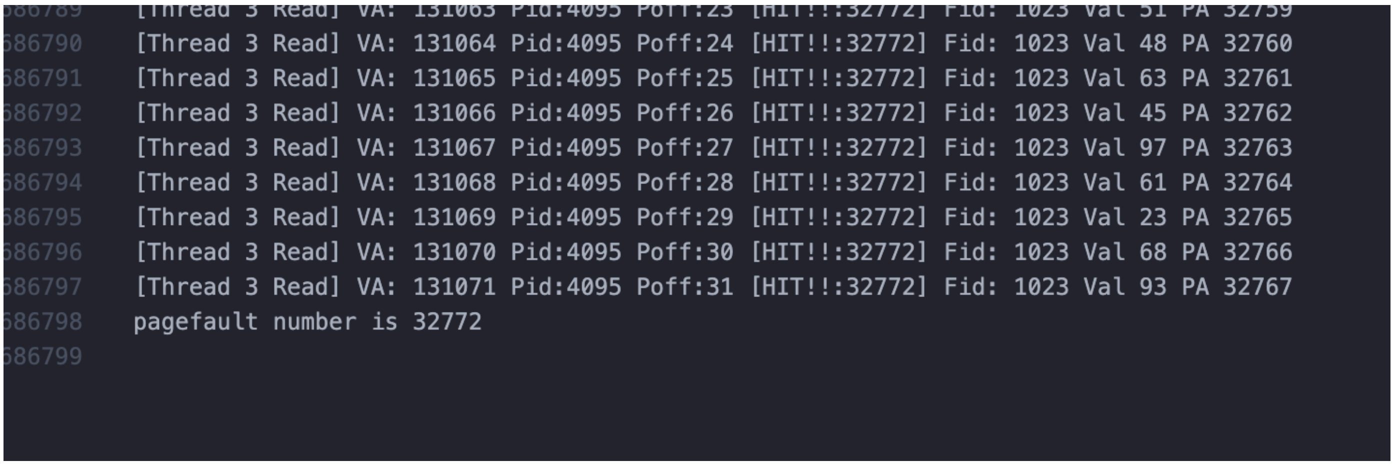
In this program the following things are done:

```
1. Write [0, 132k) of the data.bin to VM starting at address 32k in ascending order
1. Write [32k, 64k-1] of the data.bin to VM stasrting at address 0 in ascending order
1. Take a snapshot of the virtual memory with offset 32k
```

The first step populates the VM with 132K data, resulting in 4k page faults. The second step causes 1k-1 page faults, and the snapshot causes 4k page faults. Therefore, in total we have $9 * 2^{10} - 1 = 9215$ page faults.

Bonus

Only testcase 1 is used to test the bonus case. Since my code can be view as reading the .bin file four times, the total sum is 4*8193=32772 times.



```
[Thread 3 Read] VA: 131063 Pid:4095 Poff:23 [HIT!!:32772] Fid: 1023 Val 51 PA 32759
[Thread 3 Read] VA: 131064 Pid:4095 Poff:24 [HIT!!:32772] Fid: 1023 Val 48 PA 32760
[Thread 3 Read] VA: 131065 Pid:4095 Poff:25 [HIT!!:32772] Fid: 1023 Val 63 PA 32761
[Thread 3 Read] VA: 131066 Pid:4095 Poff:26 [HIT!!:32772] Fid: 1023 Val 45 PA 32762
[Thread 3 Read] VA: 131067 Pid:4095 Poff:27 [HIT!!:32772] Fid: 1023 Val 97 PA 32763
[Thread 3 Read] VA: 131068 Pid:4095 Poff:28 [HIT!!:32772] Fid: 1023 Val 61 PA 32764
[Thread 3 Read] VA: 131069 Pid:4095 Poff:29 [HIT!!:32772] Fid: 1023 Val 23 PA 32765
[Thread 3 Read] VA: 131070 Pid:4095 Poff:30 [HIT!!:32772] Fid: 1023 Val 68 PA 32766
[Thread 3 Read] VA: 131071 Pid:4095 Poff:31 [HIT!!:32772] Fid: 1023 Val 93 PA 32767
pagefault number is 32772
```

Problems Encountered & Solutions

Implementation of LRU

At first I tried to use array as the underlying datastructure for implementing LRU. Because the vm_init function seems to suggest us to use the second 1k of inverted page table as the LRU array. This idea sounds easy, as the top element is the most recently used one. However, whenever we need to update the LRU array, we always need O(N) to move elements one by one. Moreover, when I tried to debug my programs, it seems that CUDA-GDB is having some trouble displaying value of the members' values of `struct VirtualMemory`. In the end I choose to replace the data structure with doubly linked list, which allowed the code to do insertion in O(1) time.

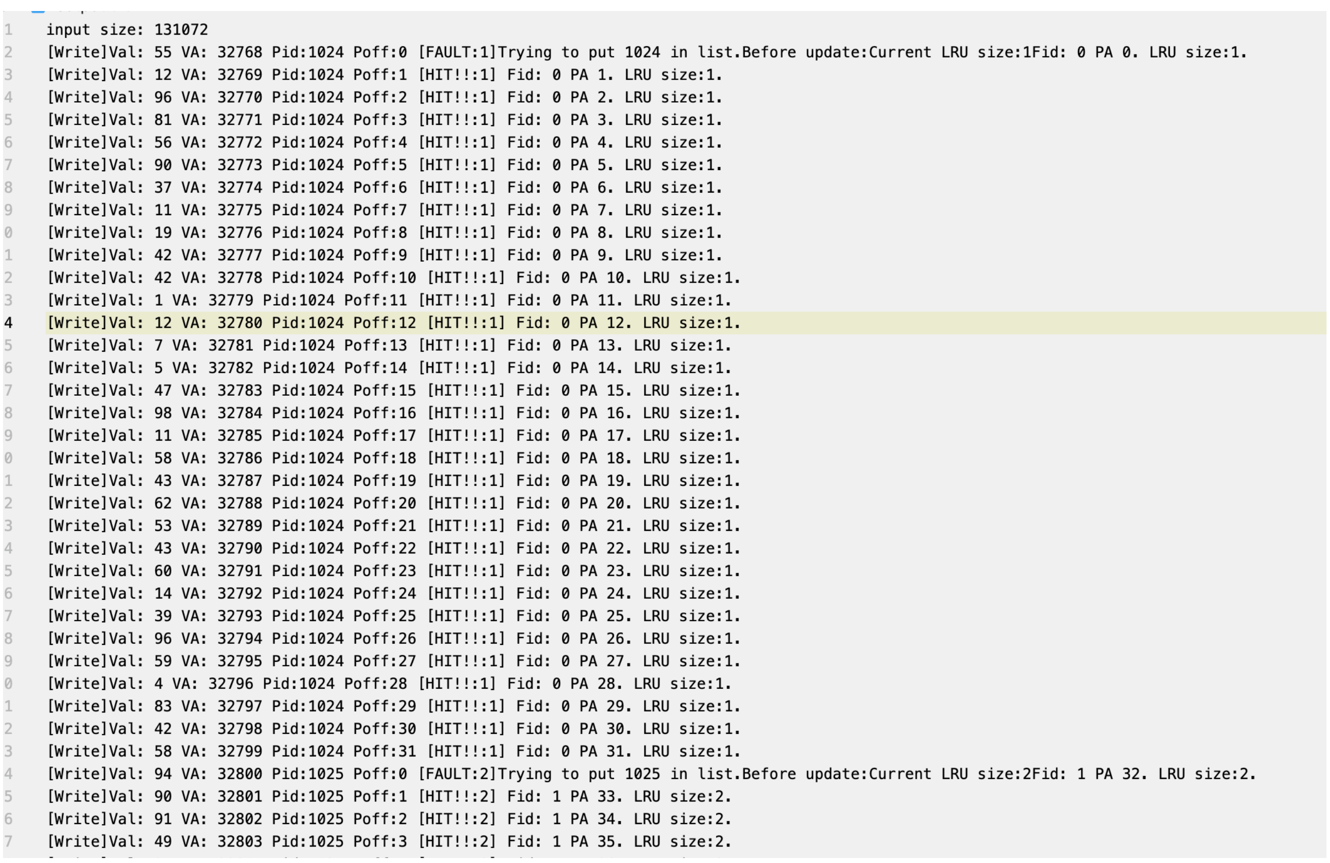
Debugging Mechanism

Considering the large input size and heterogeneous nature of CUDA, finding an efficient debugging program is very difficult. I came up with three ways to debug my program:

1. Create the `tasks.json` and `launch.json` by referring to VSCode and Nsight studio doccumentation. I uploaded these two files on github. This method allows me to debug program using VSCode GUI in a step-wise manner. Moreover, at anytime I can examine the value of expressions to get accurate information. However, this method has two serious drawbacks. First, although VSCode attaches the program to the Cuda-GDB, the running speed of program is significantly slower than other debugging methods. Second, I find that the content of `printf` is not updated immediantly. I later found out that, according to this post this is a feature of CUDA and there is nothing I can do about it. Therefore I also

use the following methods to facilitate the debugging procedure.

2. Using CUDA-GDB directly from command line. Although I am pretty new to GDB debugging and I don't know many of the commands and features, CUDA-GDB allows me to locate bugs more specifically (e.g, if SEGFAULT happens, I know which line triggers this).
3. Use rich `printf` in source code and run the executable file from the command line. Redirect stdout and stderr to a file and look for desired information in the file.



```
input size: 131072
[Write]Val: 55 VA: 32768 Pid:1024 Poff:0 [FAULT:1]Trying to put 1024 in list.Before update:Current LRU size:1Fid: 0 PA 0. LRU size:1.
[Write]Val: 12 VA: 32769 Pid:1024 Poff:1 [HIT!!:1] Fid: 0 PA 1. LRU size:1.
[Write]Val: 96 VA: 32770 Pid:1024 Poff:2 [HIT!!:1] Fid: 0 PA 2. LRU size:1.
[Write]Val: 81 VA: 32771 Pid:1024 Poff:3 [HIT!!:1] Fid: 0 PA 3. LRU size:1.
[Write]Val: 56 VA: 32772 Pid:1024 Poff:4 [HIT!!:1] Fid: 0 PA 4. LRU size:1.
[Write]Val: 90 VA: 32773 Pid:1024 Poff:5 [HIT!!:1] Fid: 0 PA 5. LRU size:1.
[Write]Val: 37 VA: 32774 Pid:1024 Poff:6 [HIT!!:1] Fid: 0 PA 6. LRU size:1.
[Write]Val: 11 VA: 32775 Pid:1024 Poff:7 [HIT!!:1] Fid: 0 PA 7. LRU size:1.
[Write]Val: 19 VA: 32776 Pid:1024 Poff:8 [HIT!!:1] Fid: 0 PA 8. LRU size:1.
[Write]Val: 42 VA: 32777 Pid:1024 Poff:9 [HIT!!:1] Fid: 0 PA 9. LRU size:1.
[Write]Val: 42 VA: 32778 Pid:1024 Poff:10 [HIT!!:1] Fid: 0 PA 10. LRU size:1.
[Write]Val: 1 VA: 32779 Pid:1024 Poff:11 [HIT!!:1] Fid: 0 PA 11. LRU size:1.
[Write]Val: 12 VA: 32780 Pid:1024 Poff:12 [HIT!!:1] Fid: 0 PA 12. LRU size:1.
[Write]Val: 7 VA: 32781 Pid:1024 Poff:13 [HIT!!:1] Fid: 0 PA 13. LRU size:1.
[Write]Val: 5 VA: 32782 Pid:1024 Poff:14 [HIT!!:1] Fid: 0 PA 14. LRU size:1.
[Write]Val: 47 VA: 32783 Pid:1024 Poff:15 [HIT!!:1] Fid: 0 PA 15. LRU size:1.
[Write]Val: 98 VA: 32784 Pid:1024 Poff:16 [HIT!!:1] Fid: 0 PA 16. LRU size:1.
[Write]Val: 11 VA: 32785 Pid:1024 Poff:17 [HIT!!:1] Fid: 0 PA 17. LRU size:1.
[Write]Val: 58 VA: 32786 Pid:1024 Poff:18 [HIT!!:1] Fid: 0 PA 18. LRU size:1.
[Write]Val: 43 VA: 32787 Pid:1024 Poff:19 [HIT!!:1] Fid: 0 PA 19. LRU size:1.
[Write]Val: 62 VA: 32788 Pid:1024 Poff:20 [HIT!!:1] Fid: 0 PA 20. LRU size:1.
[Write]Val: 53 VA: 32789 Pid:1024 Poff:21 [HIT!!:1] Fid: 0 PA 21. LRU size:1.
[Write]Val: 43 VA: 32790 Pid:1024 Poff:22 [HIT!!:1] Fid: 0 PA 22. LRU size:1.
[Write]Val: 60 VA: 32791 Pid:1024 Poff:23 [HIT!!:1] Fid: 0 PA 23. LRU size:1.
[Write]Val: 14 VA: 32792 Pid:1024 Poff:24 [HIT!!:1] Fid: 0 PA 24. LRU size:1.
[Write]Val: 39 VA: 32793 Pid:1024 Poff:25 [HIT!!:1] Fid: 0 PA 25. LRU size:1.
[Write]Val: 96 VA: 32794 Pid:1024 Poff:26 [HIT!!:1] Fid: 0 PA 26. LRU size:1.
[Write]Val: 59 VA: 32795 Pid:1024 Poff:27 [HIT!!:1] Fid: 0 PA 27. LRU size:1.
[Write]Val: 4 VA: 32796 Pid:1024 Poff:28 [HIT!!:1] Fid: 0 PA 28. LRU size:1.
[Write]Val: 83 VA: 32797 Pid:1024 Poff:29 [HIT!!:1] Fid: 0 PA 29. LRU size:1.
[Write]Val: 42 VA: 32798 Pid:1024 Poff:30 [HIT!!:1] Fid: 0 PA 30. LRU size:1.
[Write]Val: 58 VA: 32799 Pid:1024 Poff:31 [HIT!!:1] Fid: 0 PA 31. LRU size:1.
[Write]Val: 94 VA: 32800 Pid:1025 Poff:0 [FAULT:2]Trying to put 1025 in list.Before update:Current LRU size:2Fid: 1 PA 32. LRU size:2.
[Write]Val: 90 VA: 32801 Pid:1025 Poff:1 [HIT!!:2] Fid: 1 PA 33. LRU size:2.
[Write]Val: 91 VA: 32802 Pid:1025 Poff:2 [HIT!!:2] Fid: 1 PA 34. LRU size:2.
[Write]Val: 49 VA: 32803 Pid:1025 Poff:3 [HIT!!:2] Fid: 1 PA 35. LRU size:2.
```

Learning Outcome

This process allowed me to get a more thorugh and in-depth understanding of the paging policy, as well as the implementation and usage of virtual memory. It also allows me to gain some first hand-on experience on using CUDA. It is a very challenging project in terms of:

1. I have to come up how to implement a flawless paging policy from scratch
2. The input size of the binary file manes it hard to expose issues
3. Previously I have little experience in using GDB debugger (now I know how to use some of the basic features using CLI commands)
4. The CUDA-GDB itself has bugs, causing me to rely heavily on printf statements (on the bright side, I learned how to search for useful information in the log)

However, when I saw that I am able to handle 13000+ data without any mistake, I feel very proud of myself.

Miscellaneous

I kept all the `printf` commands in my code, so it may print out a lot of information. Also, I included the outut.txt as a proof that my program DO run (hard to fake a file with ten thousad lines).