CS5239 Group 10 Project
Topic 1: CPU Monitoring

Jaume Ferrer Mayol *A0256701J*
Khurush Khushrov Bengali *A0268410J*
Sanchari Saha *A0281052N*
Michael Trent Hellman *A0254413M*
Tan Jin Ming *A0254293B*

Introduction

In this report, we focus on the CPU and the available means by which one can monitor its use in computational tasks. We begin by exploring two such tools, *vmstat* and *mpstat*. Next we perform measurement of a zip task in the Linux setting. Thereafter we take a look into how to access hardware performance counters. Finally, we perform an optimisation of a matrix multiplication operation.

Throughout the report, in the interest of facilitating work amongst the report contributors, different machines might be used to obtain findings in particular sections. In general, this will not significantly impact the findings in each section, where this might be the case we will denote as such.

Scripts and code we use throughout this report are archived at the following **link**. Simple bash scripts may otherwise be found as part of a screen capture within this report document.

Vmstat

In this section, we explore use of the *vmstat* tool. Running from the command-line, *vmstat* displays an array of information from various sub-domains of the computer system such as memory, paging, processes, IO, CPU, and disk scheduling. Of particular interest to us will be the 'system' and 'cpu' tables, which contain the following (as detailed from the *vmstat* manual):

- in: The number of interrupts per second, including the clock.
- cs: The number of context switcher per second.
- us: Time spent running non-kernel code. (user time, including nice time)
- sy: Time spent running kernel code. (system time)
- id: Time spent idle. Prior to Linux 2.5.41, this includes IO-wait time.

Task A - Monitoring stress

We use the 'stress' tool to generate synthetic workloads on all cores:

```
dextcy@dextcy-ThinkPad-T430:~$ stress --cpu $(nproc)
stress: info: [32860] dispatching hogs: 8 cpu, 0 io, 0 vm, 0 hdd
```

running stress on all CPU cores with command substitution

stress generates synthetic workload that maximises CPU utilisation, which we can clearly observe from 'us' readings being near 100 percent, indicating that a process in the user space

is dominating CPU processing apart from system calls. Some characteristics of the load are also shown, manifesting as a much higher amount of interrupts and context switches occurring under *stress*.

```
procs -----------memory---------- ---swap-- -----io---- -system-- ------cpu-----
 r  b   swpd   free   buff  cache   si   so    bi    bo   in   cs us sy id wa st
 1  0      0 8512372  55232 5491364    0    0     0    32  299  554  0  0 99  0  0
 0  0      0 8512372  55232 5491364    0    0     0     0  263  512  0  0 99  0  0
 0  0      0 8511376  55232 5491364    0    0     0     0  444 1033  1  0 98  0  0
 0  0      0 8511152  55232 5491364    0    0     0     0  309  576  1  0 99  0  0
 0  0      0 8511152  55240 5491364    0    0     0   392  339  621  0  0 99  0  0
 0  0      0 8511152  55240 5491364    0    0     0   548  310  594  1  0 99  0  0
 0  0      0 8511404  55240 5491364    0    0     0     0  299  535  1  0 99  0  0
 0  0      0 8511152  55240 5491364    0    0     0     0  283  554  1  0 99  0  0
 0  0      0 8511152  55240 5491364    0    0     0    40  357  591  1  0 99  0  0
 0  0      0 8511152  55240 5491380    0    0     0     0  288  561  1  0 99  0  0
procs -----------memory---------- ---swap-- -----io---- -system-- ------cpu-----
 r  b   swpd   free   buff  cache   si   so    bi    bo   in   cs us sy id wa st
 8  0      0 8634452  55036 5389572    0    0     0     0 5956 13908 98  2  0  0  0
 8  0      0 8634452  55044 5389068    0    0     0    32 2471 1766 100  0  0  0  0
11  0      0 8602160  55044 5427544    0    0   112     0 2845 4708 98  2  0  0  0
15  0      0 8516516  55052 5506608    0    0     8     0 3464 10906 97  3  0  0  0
13  0      0 8461348  55052 5556084    0    0     0     0 3498 10202 98  2  0  0  0
 8  0      0 8489332  55052 5529132    0    0     0     0 5690 11102 98  2  0  0  0
10  0      0 8485312  55060 5529332    0    0     0   124 4492 6674 99  1  0  0  0
 8  0      0 8502008  55060 5516440    0    0     0     0 2774 2942 99  1  0  0  0
 8  0      0 8501492  55060 5516536    0    0     0     0 2440 2323 100  0  0  0  0
 8  0      0 8482100  55060 5534740    0    0     0     0 3566 8610 98  2  0  0  0
```

Normal vs stressed workload

Task B - How does it work?

The statistics from *vmstat* output are not usually obtainable from user space. As a preliminary investigation, we run *strace* on *vmstat*. The following output block shows a portion of the trace of system calls that *vmstat* performs.

```
write(1, "procs -----------memory---------"..., 81) = 81
write(1, " r  b   swpd   free   buff  cach"..., 81) = 81
openat(AT_FDCWD, "/proc/sys/kernel/osrelease", O_RDONLY) = 3
newfstatat(3, "", {st_mode=S_IFREG|0444, st_size=0, ...}, AT_EMPTY_PATH) = 0
read(3, "6.2.0-26-generic\n", 1024)     = 17
close(3)                                = 0
openat(AT_FDCWD, "/proc/meminfo", O_RDONLY) = 3
lseek(3, 0, SEEK_SET)                   = 0
read(3, "MemTotal:       16062120 kB\nMemF"..., 8191) = 1475
openat(AT_FDCWD, "/proc/stat", O_RDONLY) = 4
read(4, "cpu  231500 3287 53204 736584 58"..., 131071) = 1549
openat(AT_FDCWD, "/proc/vmstat", O_RDONLY) = 5
lseek(5, 0, SEEK_SET)                   = 0
read(5, "nr_free_pages 2814664\nnr_zone_in"..., 8191) = 3584
write(1, " 1  0      0 11258656 111640 367"..., 84) = 84
close(1)                                = 0
close(2)                                = 0
exit_group(0)                           = ?
+++ exited with 0 +++
```

strace output

Specifically, the *meminfo*, stat and *vmstat* subdirectories in the */proc/* pseudo-filesystem are opened and read from. In the case of *meminfo*, it is successfully opened then 1475 bytes are read from the file. Hence, it is through using this *proc* filesystem as an interface that enables *vmstat* access to kernel data structures.

A look into the */proc/meminfo* file shows information obtained by vmstat's read system call. From the meminfo.c linux kernel file, the kernel uses its memory management subsystem to obtain statistical data from zones and nodes through calls. **Zones** represent specific regions in memory that are used to optimize memory access and allocation.**Nodes** in Linux memory management are typically Non-Uniform Memory Access (NUMA) nodes, which denote a division of memory into regions each associated with specific CPUs and are an architectural feature.

Task C - USO flashbacks (1)

Finally, we use *vmstat* to produce disk statistics by implementing the -d flag. We further sort the output in descending order by the second column 'total reads' values. With this, it can be

```
dextcy@dextcy-ThinkPad-T430:~$ cat /proc/meminfo
MemTotal:       16062120 kB
MemFree:         8507216 kB
MemAvailable:   13066152 kB
Buffers:           60860 kB
Cached:          5222116 kB
SwapCached:            0 kB
Active:          2059032 kB
Inactive:        4309340 kB
Active(anon):       5064 kB
Inactive(anon):  1744992 kB
Active(file):    2053968 kB
Inactive(file):  2564348 kB
Unevictable:      571732 kB
Mlocked:              16 kB
SwapTotal:       2097148 kB
SwapFree:        2097148 kB
Zswap:                 0 kB
Zswapped:              0 kB
Dirty:              1860 kB
Writeback:             0 kB
AnonPages:       1657224 kB
Mapped:           675880 kB
Shmem:            664660 kB
KReclaimable:     277976 kB
Slab:             450920 kB
SReclaimable:     277976 kB
SUnreclaim:       172944 kB
KernelStack:       14432 kB
PageTables:        27480 kB
SecPageTables:         0 kB
NFS_Unstable:          0 kB
Bounce:                0 kB
WritebackTmp:          0 kB
CommitLimit:    10128208 kB
Committed_AS:    7129412 kB
VmallocTotal:   34359738367 kB
VmallocUsed:       55680 kB
VmallocChunk:          0 kB
Percpu:             7584 kB
HardwareCorrupted:     0 kB
AnonHugePages:         0 kB
ShmemHugePages:        0 kB
ShmemPmdMapped:        0 kB
FileHugePages:         0 kB
FilePmdMapped:         0 kB
HugePages_Total:       0
HugePages_Free:        0
HugePages_Rsvd:        0
HugePages_Surp:        0
Hugepagesize:       2048 kB
Hugetlb:               0 kB
DirectMap4k:      357948 kB
DirectMap2M:    16109568 kB
```

/proc/meminfo

seen that the most read disk by far in this system is disk B, or sdb, followed by various loopback devices and disk A.

```
dextcy@dextcy-ThinkPad-T430:~$ vmstat -d | tail -n +3 | sort -k2,2nr
sdb     47797  19457 3904986   25582 145917 131696 9599306  111346      0     99
loop11   4119      0   30966     104      0      0       0       0      0      0
loop10   3572      0  136610     338      0      0       0       0      0      1
loop0    3228      0  241718     679      0      0       0       0      0      2
loop7    1355      0  136626     300      0      0       0       0      0      1
sda       735      0   61272     508      0      0       0       0      0      0
loop16    603      0   43798     126      0      0       0       0      0      1
loop6     541      0   14364      74      0      0       0       0      0      0
loop3     301      0   18742      67      0      0       0       0      0      0
loop14    239      0    7128      38      0      0       0       0      0      0
loop8      58      0    2216      26      0      0       0       0      0      0
loop9      57      0    2132      31      0      0       0       0      0      0
loop4      56      0    2152      17      0      0       0       0      0      0
loop5      53      0    2170      21      0      0       0       0      0      0
loop13     45      0     308      15      0      0       0       0      0      0
loop2      45      0     698      11      0      0       0       0      0      0
loop15     43      0     696      14      0      0       0       0      0      0
loop12     39      0     266      11      0      0       0       0      0      0
loop17     32      0     628       4      0      0       0       0      0      0
loop1      14      0      34       0      0      0       0       0      0      0
loop18     11      0      28       0      0      0       0       0      0      0
```

vmstat output

Mpstat

Task A - Python recursion depth

In this section, we use *mpstat* to analyse python depth recursion in the *fact_rcrs.py* program, which performs a recursive factorial computation. The code provided is somewhat incomplete in the sense that it calculates the factorials but doesn't print the results. To do so, we modify the code to print the factorial values for the given range of numbers.

```
Traceback (most recent call last):
  File "/home/e1154502/fact_rcrs.py", line 21, in <module>
    result = factorial(i)
  File "/home/e1154502/fact_rcrs.py", line 15, in factorial
    else n * factorial(n - 1)
  File "/home/e1154502/fact_rcrs.py", line 15, in factorial
    else n * factorial(n - 1)
  File "/home/e1154502/fact_rcrs.py", line 15, in factorial
    else n * factorial(n - 1)
  [Previous line repeated 995 more times]
  File "/home/e1154502/fact_rcrs.py", line 14, in factorial
    return 1 if n == 0 \
RecursionError: maximum recursion depth exceeded in comparison
```

Python recursion error

The script crashes with a "RecursionError" when we pass a large number like 1000 as a command-line argument because Python has a maximum recursion depth limit, which prevents

deeply recursive functions from causing a stack overflow and crashing the program. (shown above)

This recursion depth limit is in place to protect against infinite recursion and stack overflow errors. To increase the recursion depth limit in Python, we can use the *sys* module to access and modify it. With this modification, the factorial code does not crash. This code is archived at fact_rcrs_1.py

Task B - CPU affinity

When we ran the script again, this time passing 10000, the code crashes reporting the following error

```
Traceback (most recent call last):
  File "/home/e1154502/fact_rcrs.py", line 22, in <module>
    print('factorial(%d) = %d' % (i, result))
ValueError: Exceeds the limit (4300) for integer string conversion; use sys.set_int_max_str_digits() to increase the limit
```

The ”Exceeds the limit (4300) for integer string conversion” error indicates that the value we are trying to format as a string using the ‘%d‘ format specifier is too large for the default string conversion limit. In this case, the result of a large factorial calculation exceeds the default limit for integer string conversion.

To address this issue, we use the ‘sys.set_int_max_str_digits()‘ function to increase the limit for integer string conversion. This code is archived at fact_rcrs_2.py.

We observe *htop* activity for our program and find good CPU utilization for only one of the cores. This core with close to 100% load is the one running our script.

htop: activity on only one core

mpstat output monitored at 1s intervals (below) also confirms the *htop* findings, with core 11 showing close to 100% load

```
e1154502@soctf-pdc-002:~$ mpstat -P ALL 1
Linux 5.15.0-87-generic (soctf-pdc-002.d1.comp.nus.edu.sg)      10/29/23        _x86_64_        (20 CPU)

18:47:21     CPU    %usr   %nice    %sys %iowait    %irq   %soft  %steal  %guest  %gnice   %idle
18:47:22     all    7.05    0.30    0.55   72.74    0.00    0.00    0.00    0.00    0.00   19.36
18:47:22       0    0.00    0.00    0.00  100.00    0.00    0.00    0.00    0.00    0.00    0.00
18:47:22       1    0.00    0.00    0.99   99.01    0.00    0.00    0.00    0.00    0.00    0.00
18:47:22       2    0.00    0.00    0.99    0.00    0.00    0.00    0.00    0.00    0.00   99.01
18:47:22       3    0.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00  100.00
18:47:22       4    0.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00  100.00
18:47:22       5    0.00    0.00    0.00  100.00    0.00    0.00    0.00    0.00    0.00    0.00
18:47:22       6   10.00    1.00    1.00    0.00    0.00    0.00    0.00    0.00    0.00   88.00
18:47:22       7    0.00    0.00    0.00  100.00    0.00    0.00    0.00    0.00    0.00    0.00
18:47:22       8    0.00    0.00    0.00  100.00    0.00    0.00    0.00    0.00    0.00    0.00
18:47:22       9    0.00    1.01    0.00   98.99    0.00    0.00    0.00    0.00    0.00    0.00
18:47:22      10    0.00    0.00    0.00  100.00    0.00    0.00    0.00    0.00    0.00    0.00
18:47:22      11   98.99    0.00    1.01    0.00    0.00    0.00    0.00    0.00    0.00    0.00
18:47:22      12    0.00    0.00    1.00   99.00    0.00    0.00    0.00    0.00    0.00    0.00
18:47:22      13    0.00    1.00    0.00   99.00    0.00    0.00    0.00    0.00    0.00    0.00
18:47:22      14    0.99    0.00    0.99   98.02    0.00    0.00    0.00    0.00    0.00    0.00
18:47:22      15    0.00    0.00    0.00  100.00    0.00    0.00    0.00    0.00    0.00    0.00
18:47:22      16    0.00    0.00    0.00  100.00    0.00    0.00    0.00    0.00    0.00    0.00
18:47:22      17   31.68    0.00    0.99   67.33    0.00    0.00    0.00    0.00    0.00    0.00
18:47:22      18    0.00    1.01    0.00   98.99    0.00    0.00    0.00    0.00    0.00    0.00
18:47:22      19    0.00    1.98    3.96   94.06    0.00    0.00    0.00    0.00    0.00    0.00
```

mpstat: core 11 near max utilisation

Next, we create a bash script to find the number of CPU cores on our system, then use stress to create N-1 CPU workers, where N is the number of cores on our system. We use taskset

to set the CPU affinity of the N-1 workers to CPUs 1-(N-1). Finally, it runs our factorial Python code with taskset on CPU 0. This script is archived at: task4_2_B.sh.

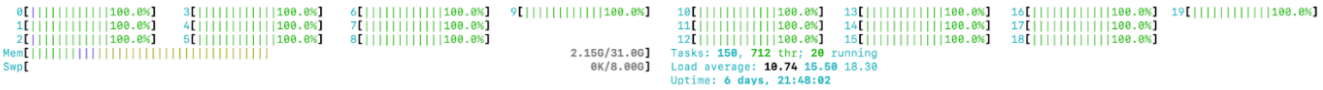
```
top - 19:26:56 up 6 days, 21:46,  6 users,  load average: 8.54, 16.85, 18.99
Tasks: 387 total,  21 running, 364 sleeping,   2 stopped,   0 zombie
%Cpu(s): 99.4 us,  0.5 sy,  0.1 ni,  0.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
MiB Mem :  31735.8 total,  19862.0 free,   2185.0 used,   9688.7 buff/cache
MiB Swap:   8192.0 total,   8192.0 free,      0.0 used.  29097.8 avail Mem

    PID USER      PR  NI    VIRT    RES    SHR S  %CPU  %MEM     TIME+ COMMAND
1075441 e1154502  20   0    3572    100      0 R 100.0   0.0   0:16.43 stress
1075460 e1154502  20   0    3572    104      0 R 100.0   0.0   0:16.41 stress
1075444 e1154502  20   0    3572    100      0 R 100.0   0.0   0:16.42 stress
1075449 e1154502  20   0    3572    100      0 R 100.0   0.0   0:16.40 stress
1075451 e1154502  20   0    3572    104      0 R 100.0   0.0   0:16.42 stress
1075454 e1154502  20   0    3572    100      0 R 100.0   0.0   0:16.40 stress
1075457 e1154502  20   0    3572    100      0 R 100.0   0.0   0:16.43 stress
1075458 e1154502  20   0    3572    100      0 R 100.0   0.0   0:16.43 stress
1075459 e1154502  20   0    3572    104      0 R 100.0   0.0   0:16.43 stress
1075461 e1154502  20   0    3572    104      0 R 100.0   0.0   0:16.40 stress
1075437 e1154502  20   0    3572    100      0 R  99.7   0.0   0:16.39 stress
1075446 e1154502  20   0    3572    100      0 R  99.7   0.0   0:16.35 stress
1075450 e1154502  20   0    3572    104      0 R  99.3   0.0   0:16.41 stress
1075456 e1154502  20   0    3572    104      0 R  99.3   0.0   0:16.35 stress
1075448 e1154502  20   0    3572    104      0 R  97.7   0.0   0:15.78 stress
1075453 e1154502  20   0    3572    104      0 R  97.0   0.0   0:15.87 stress
1075442 e1154502  20   0    3572    100      0 R  95.7   0.0   0:15.99 stress
1075452 e1154502  20   0    3572    104      0 R  95.7   0.0   0:15.45 stress
1075455 e1154502  20   0    3572    100      0 R  95.4   0.0   0:16.28 stress
1075475 e1154502  20   0   17664   8876   5768 R  90.1   0.0   0:10.38 python3
```

core loads

Monitoring *top* activity (above) shows only one out of the 20 cores is running Python load while all other cores are busy running stress commands.

Following are the *htop* activity and *mpstat* output observed for this task:



htop

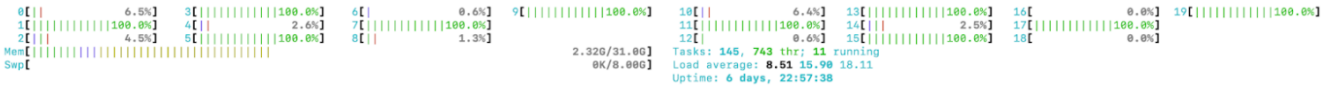
```
Average:      CPU    %usr   %nice    %sys %iowait    %irq   %soft  %steal  %guest  %gnice   %idle
Average:      all   99.41    0.09    0.49    0.00    0.00    0.01    0.00    0.00    0.00    0.00
Average:        0   97.72    0.00    2.28    0.00    0.00    0.00    0.00    0.00    0.00    0.00
Average:        1   99.49    0.00    0.51    0.00    0.00    0.00    0.00    0.00    0.00    0.00
Average:        2   95.96    1.26    2.78    0.00    0.00    0.00    0.00    0.00    0.00    0.00
Average:        3   99.75    0.00    0.25    0.00    0.00    0.00    0.00    0.00    0.00    0.00
Average:        4   99.75    0.00    0.25    0.00    0.00    0.00    0.00    0.00    0.00    0.00
Average:        5   99.49    0.00    0.51    0.00    0.00    0.00    0.00    0.00    0.00    0.00
Average:        6  100.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00
Average:        7  100.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00
Average:        8   99.49    0.00    0.51    0.00    0.00    0.00    0.00    0.00    0.00    0.00
Average:        9  100.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00
Average:       10   98.74    0.00    1.26    0.00    0.00    0.00    0.00    0.00    0.00    0.00
Average:       11   98.48    0.51    0.76    0.00    0.00    0.25    0.00    0.00    0.00    0.00
Average:       12  100.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00
Average:       13  100.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00
Average:       14  100.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00
Average:       15  100.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00
Average:       16   99.75    0.00    0.25    0.00    0.00    0.00    0.00    0.00    0.00    0.00
Average:       17   99.75    0.00    0.25    0.00    0.00    0.00    0.00    0.00    0.00    0.00
Average:       18   99.75    0.00    0.25    0.00    0.00    0.00    0.00    0.00    0.00    0.00
Average:       19  100.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00
```

mpstat

Task C - USO flashbacks (2)

We create a bash script to perform the task of binding CPU stress workers on our odd-numbered cores (i.e.: 1,3,5,…). The list of cores and the number of stress workers are constructed based on nproc. This script is archived at: task4_2_C.sh.

Through *htop* CPU utilization, we are able to highlight the load usage of odd-numbered cores of the system
top output (below) shows 10 out of the 20 cores running stress, as expected.



htop: stress on odd numbered cores

```
top - 20:38:02 up 6 days, 22:57,  8 users,  load average: 10.49, 16.78, 18.44
Tasks: 382 total,  11 running, 369 sleeping,   2 stopped,   0 zombie
%Cpu(s): 50.1 us,  0.7 sy,  0.3 ni, 14.4 id, 34.5 wa,  0.0 hi,  0.0 si,  0.0 st
MiB Mem :  31735.8 total,  19668.7 free,   2372.1 used,   9694.9 buff/cache
MiB Swap:   8192.0 total,   8192.0 free,      0.0 used.  28910.6 avail Mem

    PID USER      PR  NI    VIRT    RES    SHR S  %CPU  %MEM     TIME+ COMMAND
1088112 e1154502  20   0    3704    108      0 R 100.0   0.0   0:08.04 stress
1088103 e1154502  20   0    3704    112      0 R 100.0   0.0   0:08.04 stress
1088104 e1154502  20   0    3704    112      0 R 100.0   0.0   0:08.04 stress
1088105 e1154502  20   0    3704    112      0 R 100.0   0.0   0:08.04 stress
1088106 e1154502  20   0    3704    108      0 R 100.0   0.0   0:08.04 stress
1088107 e1154502  20   0    3704    108      0 R 100.0   0.0   0:08.04 stress
1088108 e1154502  20   0    3704    112      0 R 100.0   0.0   0:08.04 stress
1088109 e1154502  20   0    3704    112      0 R 100.0   0.0   0:08.04 stress
1088110 e1154502  20   0    3704    108      0 R 100.0   0.0   0:08.04 stress
1088111 e1154502  20   0    3704    112      0 R 100.0   0.0   0:08.04 stress
```

top: stress on odd numbered cores

We provide the *mppstat* capture to prove that our code is working, with 100% usr seen for all odd-numbered cores.

```
e1154502@soctf-pdc-002:~$ mpstat -P ALL 1
Linux 5.15.0-87-generic (soctf-pdc-002.d1.comp.nus.edu.sg)      10/29/23        _x86_64_        (20 CPU)

20:39:42     CPU    %usr   %nice    %sys %iowait    %irq   %soft  %steal  %guest  %gnice   %idle
20:39:43     all   50.20    0.30    0.40   34.42    0.00    0.00    0.00    0.00    0.00   14.68
20:39:43       0    0.00    2.04    7.14   90.82    0.00    0.00    0.00    0.00    0.00    0.00
20:39:43       1  100.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00
20:39:43       2    0.00    1.01    1.01    0.00    0.00    0.00    0.00    0.00    0.00   97.98
20:39:43       3  100.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00
20:39:43       4    0.00    1.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00   99.00
20:39:43       5  100.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00
20:39:43       6    0.00    0.00    0.00    3.00    0.00    0.00    0.00    0.00    0.00   97.00
20:39:43       7  100.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00
20:39:43       8    1.01    0.00    0.00   98.99    0.00    0.00    0.00    0.00    0.00    0.00
20:39:43       9  100.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00
20:39:43      10    0.00    1.00    0.00   99.00    0.00    0.00    0.00    0.00    0.00    0.00
20:39:43      11  100.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00
20:39:43      12    1.00    0.00    0.00   99.00    0.00    0.00    0.00    0.00    0.00    0.00
20:39:43      13  100.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00
20:39:43      14    0.00    1.00    0.00   99.00    0.00    0.00    0.00    0.00    0.00    0.00
20:39:43      15  100.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00
20:39:43      16    0.00    0.00    0.00  100.00    0.00    0.00    0.00    0.00    0.00    0.00
20:39:43      17  100.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00
20:39:43      18    0.00    0.00    0.00  100.00    0.00    0.00    0.00    0.00    0.00    0.00
20:39:43      19  100.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00
```

mpstat: 100% usr on odd numbered cores

Zip with compression levels

Task A - Measurements

In this task, we take measurements of zip compression operations. Here, we perform compression on three files, two bitmaps and one text. We create a shell script which compresses the 3 mentioned files, where the output text file contains time for each compression level and more details (such as archive size) from the zip. Following are the key commands used in the script:

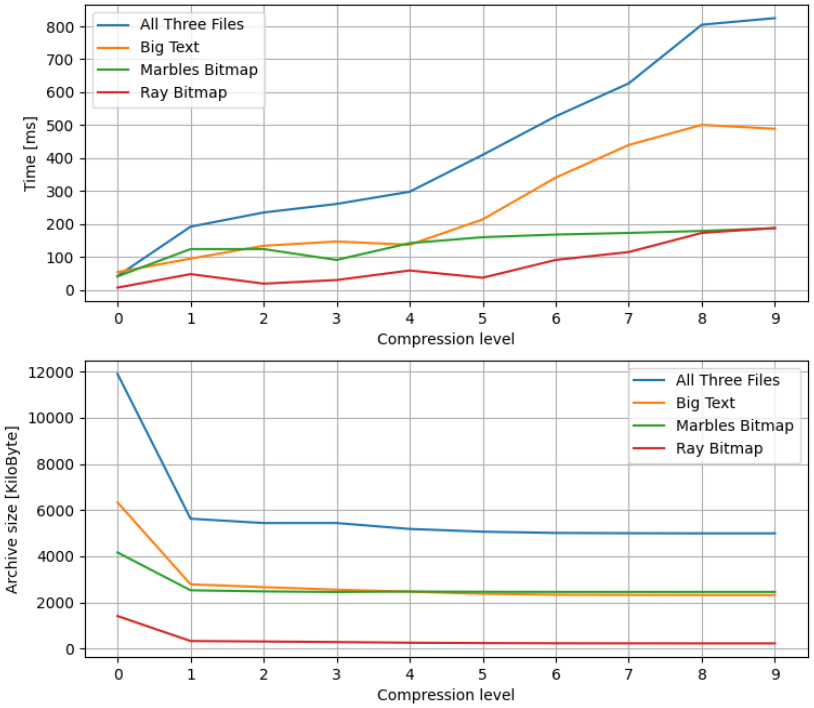
- zip: Used to create archive. Compression levels can be specified by standard input (-0 to -9)
- unzip: Used to print a verbose output which also contains the archive size.
- time: For measuring time while running the zip command.

This script is archived at: task4_3.sh.

Task B - Plot

The provided python plotting code is updated with results from the earlier script. We can see as the compression is increased, time for compression is also increased gradually.

Based on the figure below, the most balanced compression level is 6, resulting in an slightly higher average time with almost the smallest archive size. Note, this might change depending on number of files and size of each file.
Compression level 0 performs no compression and yields the fastest packing time. On the other hand, level 9 requires the most time but provides the best compression.
A salient point is that the most significant reduction in archive size relative to increase in compression time is obtained at level 1. This makes it the ideal setting for batch compression in time-sensitive contexts.



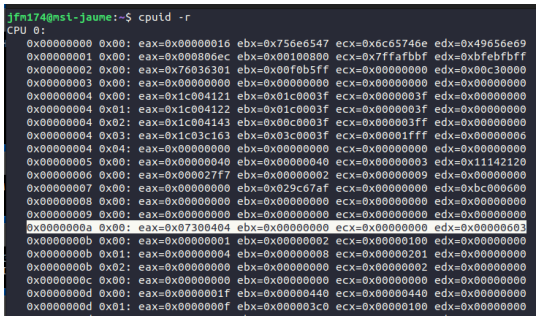
Effect of Compression Levels on Time and Archive Size

The updated python code is archived at: task4_3_plot.py.

Hardware counters

Task A - Hardware Counters

In this task, we're required to extract general information about our CPU architecture. Concretely, we want to identify the number of general purpose counter, the micro-architecture version ID and the number of fixed function counters defined for the CPU. We do this by using the command cpuid -r, and looking into the 0Ah leaf from the raw data:
We observe the following values:

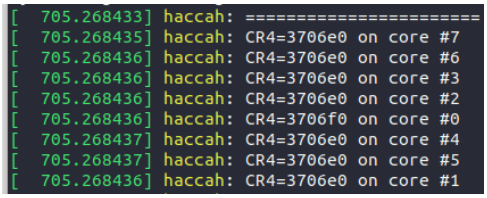


cpuid -r

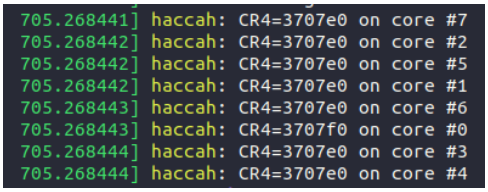
- **Version ID:** 0x04, **General Purpose Counters**: 0x04, **Fixed Counters**: 0x03.

Task B - Unlock RDPMS in ring3

In order to use performance monitoring counters, we first need to enable the corresponding flag in the control registers. If set, the performance counters could be read at any privilege level. We observe that once we load the kernel module that modifies CR4 flags, the bit 8 (PCE) changes to 1:



dmesg before module is loaded: bit 8 of CR4 is 0.



dmesg after module is loaded: bit 8 of CR4 is 1.

To proceed with the following tasks, this module will not be removed until the exercise is completed.

Task C - Configure IA32_PERF_GLOBAL_CTRL

In this subtask we aim to to use a single write instruction to enable the General Purpose counters utilizing the x86 architecture's IA32_PERF_GLOBAL_CTRL register. Typically, these bits are set to default values during the booting process. However, it's advised to check and, if necessary, modify the IA32_PERF_GLOBAL_CTRL register. We use the **rdmsr** and **wrmsr** tools to read and write into the register, respectively. For this tools to work, the corresponding module has to be loaded into the kernel.

Kernel loaded modules after msr module is inserted.

Following, the corresponding bits for the IA32_PERF_GLOBAL_CTRL register can be enable through the **wrmsr** command (note that the bits 1:0 must be enabled). We observe that the command modified the first bits in each logical core (8 cores):

```
root@msi-jaume:/home/jfm174# rdmsr -a 0x38f
70000000f
70000000f
70000000f
70000000f
70000000f
70000000f
70000000f
70000000f
```

IA32_PERF_GLOBAL_CTRL value for each core.

Task D - Configure IA32_PERFEVENTSELx

Now, we must define the events we want to monitor through counters. In order to start the event counting, we use the first of the counters defined in the architecture: IA32_PERFEVENTSEL0 (0x186) is used to configure the monitored event of a counter, in this case, counter 1. According to Intel's manual, the values for UMASK and EVSEL used to monitor the LLC misses correspond to UMASK= 0x41 and EVSEL=0x2e.

We reset the count for the first of the counters (0xc1) and we write the configuration into the first counter definition (0x186) for each of the CPUs:

```
jfm174@msi-jaume:~/Desktop/
 wrmsr -a 0x186 0x41412e
jfm174@msi-jaume:~/Desktop/
 rdmsr -a 0x186
41412e
41412e
41412e
41412e
41412e
41412e
41412e
41412e
```

IA32_PERFEVENTSEL0 (0x186)

```
jfm174@msi-jaume:~/Desktop/
 rdmsr -a 0xc1
0
0
0
0
0
0
0
0
```

Before IA32_PERFEVENTSEL0 bits are set.

```
jfm174@msi-jaume:~/Desktop/
 rdmsr -a 0xc1
4853c99
64fd0b5
3ddc6a6
374877b
4fa5820
3d5f9c7
37cd9ad
1c02381
```

After IA32_PERFEVENTSEL0 bits are set.

Task E - Ring3 cache performance evaluation

In this task, we want to monitor the cache misses by performing a matrix multiplication. In this example, we define two different calculation modes for the multiplication, one more efficient than the other in terms of memory. In theory, the second optimized multiplication is designed to produce less cache misses than the first one. Access to memory should be then faster, and so it would finish earlier.

We observe that the default code brings us to segmentation fault errors during the execution, this could be produced due to incompatibility of the defined **rdpmc** function with the current architecture. For this reason, we consider the alternative case of using the **system(rdmsr 0xc1)** instruction in contrast to the current method.

After tasks 4.4.1, 4.4.2, 4.4.3 and 4.4.4, the program is ready to read the HPC counts:

```
root@msi-jaume:/home/jfm174/Desktop/CS5239-PerfAnalysis/cs5239_Project01/hw_coun
ter# taskset 0x01 ./mat_mul_02_sys 1024
Reading L2 cache misses before first multiplication.
1e662
Reading L2 cache misses after first multiplication.
641fdf78
Multiplication 1 finished in  19.62 s
Reading L2 cache misses before second multiplication.
64220a84
Reading L2 cache misses after second multiplication.
679694d1
Multiplication 2 finished in   3.45 s
```

Alternative mat_mul.c execution output. Refer to mat_mul_02_sys.c for the source code.

- **LLC Miss ratio (slow)**: 85,610,902.65 misses/s

- **LLC Miss ratio (fast)**: 16,802,675.07 misses/s
- **Slow/fast** = 5.1

The miss ratio shows that the second multiplication method is indeed much lower than the first one for N=1024, up to five times.
Note (1): For this task we first assume that LLC misses are a good way to compare the memory efficiency of the operations, despite the problem originally considers L2_misses. This approach is because we couldn't extract a L2 misses event selector from the user Intel's manual that matched with our architecture.
Note (2): A second observation is regarding the use of **system(<cmd>)** as an alternative to the **asm** code which produced the access error. We understand this function from **<stdlib.h>** performs a **fork()** operation internally. Thus, we assume that due to the use of taskset to bind the process to a single core, the impact of this is negligible and the performance is minimally affected by context switches.

Optimising mat_mul.c

Motivation

Matrix multiplication is a fundamental mathematical operation in various fields, including machine learning, robotics, and scientific computing.

In this task, we aim to optimize this operation. Optimizing matrix multiplication is a complex task with various techniques and considerations for increasing operation speed and efficacy. It is also a well-documented problem space. What follows is an exploration of known avenues for optimization.

A naive implementation in C

We begin with a naive implementation of matrix multiplication written in C as our baseline: naive_matmul.c. Here, elements (i, j) of the result matrix are obtained as a summation of element-wise multiplications over the rows of matrix 1 and the columns of matrix 2 via a triple-nested loop. It is not hard to see that this method is inefficient, but will nonetheless serve as the foundation for our analysis.

We consider two versions, a "slow" and "fast" method (in truth, both are painfully slow). In the slower method, the loop order is a truly naive i, j, k. In the faster, the loop order is k, i, j.//

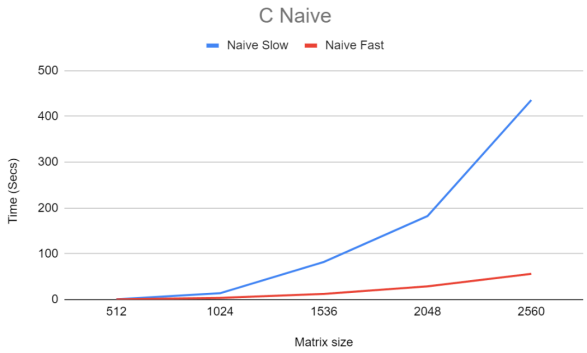


Figure 3: Naive slow vs fast

Figure 3 shows a comparison of operation completion time between both implementations across different matrix multiplication sizes. Indeed, the fast method lives true to its name and achieves better results across the board. To investigate, we employ *valgrind* to check cache miss rates.

	512		1024		2048	
	C Naive Slower	C Naive Faster	C Naive Slower	C Naive Faster	C Naive Slower	C Naive Faster
I1 miss rate	0%	0%	0%	0%	0%	0%
LLi miss rate	0%	0%	0%	0%	0%	0%
D1 miss rate	4.40%	0.60%	4.40%	0.50%	4.40%	1.10%
LLd miss rate	0%	0%	4.40%	0.50%	4.30%	0.50%
LL miss rate	0%	0%	1.50%	0.20%	1.50%	0.20%

Figure 4: Naive valgrind outputs

From the results above, we can see that cache miss rates are significantly improved in the faster method, leading to more efficient computation. Hence, changing the loop order serves to optimise the algorithm's cache access patterns, resulting in improved performance. From this, we surmise that memory will have a significant role to play in our optimization efforts.
In either case, these naive approaches have a time complexity of O(N^3) due to the triple-nested loops. They are straightforward but are inefficient for large matrix sizes.

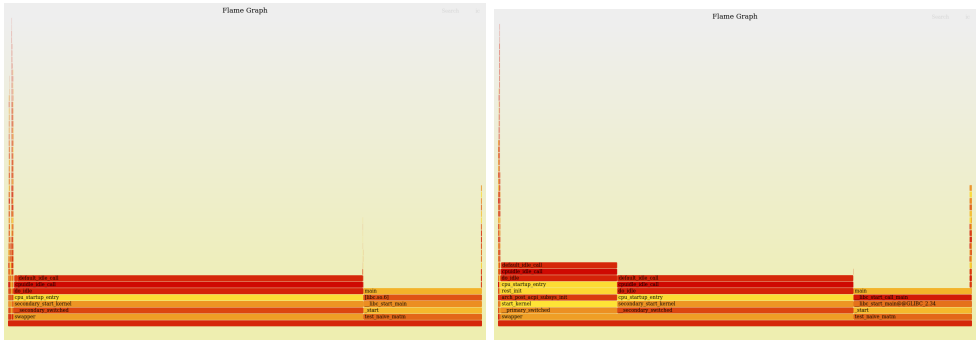


Figure 5: Flame graphs for naive slow vs fast at matrix size 2048

We also generate flamegraphs in flamegraphs.sh for both naive methods to observe their call-stacks. Neither graph revealed anything of note, which is to be expected as the programs are relatively simple.

Justification for valgrind

Before proceeding, we must outline our reasoning behind using *valgrind* for memory profiling instead of hardware performance counters.

As noted in the prior task, the **rdpmc** function proved unusable, while the uncertainty regarding use of system instructions in lieu dissuaded our implementation of such. Furthermore, prior studies have shown the instability of hardware performance counters due to non-deterministic behaviour and contamination from external processes.

Hence, we choose to use *valgrind*, or more specifically the *cachegrind* subtool, for our memory profiling purposes owing to it's ease of use and comprehensive measurements. *Valgrind* functions via insertion of instrumentation code into base code which is then run on simulated core. This process injects overheads to the client programs, hence we use *valgrind* exclusively as a comparative tool for memory usage amongst our different implementations. Other performance measure are derived outside of the *valgrind* environment.

Optimization 1: Pre-computing the transpose

Taking from the results of the prior section, we further attempt to optimise cache access in transpose_matmul.c.

The key idea of this optimization strategy is through pre-computation of the second matrix's transpose. By doing so, the second operand matrix can be accessed row-wise instead of column-wise through the transpose, resulting in contiguous memory access. In theory, this improves cache locality and reduces cache misses during multiplication. The transposition step incurs an initial cost but pays off during multiplication, especially for large matrices.

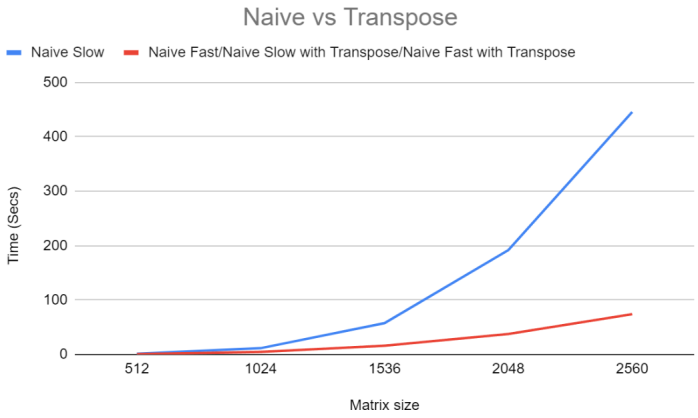


Figure 6: Naive slow vs fast with transpose

The above figure shows a comparison between slow and fast naive methods and the effect of matrix transposition on both. To note, the results for naive fast, naive slow with transpose and naive fast with transpose chart extremely similarly (to be elaborated on shortly), causing the lines to overlap and obfuscating results. Thus we combine all three method outputs in a single line for clarity.

Observations:

- Improvement to naive slow: We can see that the slow method is improved by performing the transpose pre-computation. However, our results reveal a surprising outcome in the overlapping performance (differing in fractions of a second) amongst the slow naive with transpose, fast naive without and fast naive with transpose.
- Code analysis: As aforementioned, the slow method's i, j, k loop order has a greater inefficiency in terms of cache access. Hence, it is here that the transpose has a more pronounced effect. In the case of the fast method, the k, i, j loop order represents a degree of optimisation in cache access, explaining the relatively less performance gain.

The following *valgrind* output verifies our claim:

	512		1024		2048	
	Tranpose Slow	Transpose Fast	Tranpose Slow	Transpose Fast	Tranpose Slow	Transpose Fast
I1 miss rate	0%	0%	0%	0%	0%	0%
LLi miss rate	0%	0%	0%	0%	0%	0%
D1 miss rate	0.60%	0.60%	0.60%	0.60%	0.60%	0.60%
LLd miss rate	0%	0%	0.60%	0.60%	0.60%	0.60%
LL miss rate	0%	0%	0.20%	0.20%	0.20%	0.20%

Figure 7: Naive slow vs fast with transpose valgrind outputs

From this finding, we conclude that the performance gain from pre-computation of transposes is algorithm-specific, algorithms with sub-optimal cache access patterns can benefit more from this strategy.

Optimization 2: Parallelization

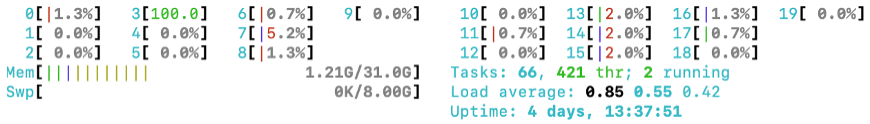


Figure 8: htop captured during naive implementation showing activity on 1 core

We return to the naive methods, using *htop* to analyse CPU activity while running. From figure 8, *htop* indicates suboptimal CPU resource utilization during execution as only one core is being utilized.

Hence, we look into a parallelization strategy. This allows simultaneous computation of different elements in the matrices, leveraging the available CPU cores and mitigating the cubic time complexity. In other words, parallelization harnesses the full computational power of our modern processors, thereby significantly improving the overall performance of matrix multiplication.

In omp_matmult.c, we make the following considerations:

- Multi-threading vs. Multi-processing: Parallelization can be executed as multi-threading or multi-processing. Multi-threading suits CPU-bound tasks like matrix multiplication, but synchronization is crucial to prevent data races. Multi-processing involves separate processes, offering simplicity but may have higher overhead.
- Frameworks or Libraries: Leverage parallel processing libraries or frameworks (e.g., OpenMP for multi-threading, MPI for multi-processing) for easier implementation.

Parallelization: OpenMP with(out) Tiling

Accounting for prior considerations, we implement the OpenMP library for it's robust multi-threading support. In our implementation, 8 threads are defined for parallel execution and multi-threading is applied to the naive matrix multiplication for-loops. By incorporating private iterative variables and collapsing inner loops, computational speed is enhanced.

We further seek to bolster OpenMP's performance by simultaneously employing a tiling/blocking strategy. Here, a given matrix of size N is intelligently divided into tiles for parallel computation. We determine the optimal tile size by experimentation based on the given matrix size and specified number of threads. Hence, we have 2 permutations of the slow and fast methods each, namely OMP with and without tiling. Figure 9 shows an execution time comparison against the unmodified slow and fast methods.

Observations:

- Matrix size and Overheads: At higher matrix sizes beyond 1024 we can see a greater divergence in performance. Fast OMP w/o tiling consistently yields the best performance at matrix sizes 1536 and above, with it's non OMP fast counterpart achieving the best results at 1024 and below. From this we hypothesize that OMP introduces overheads

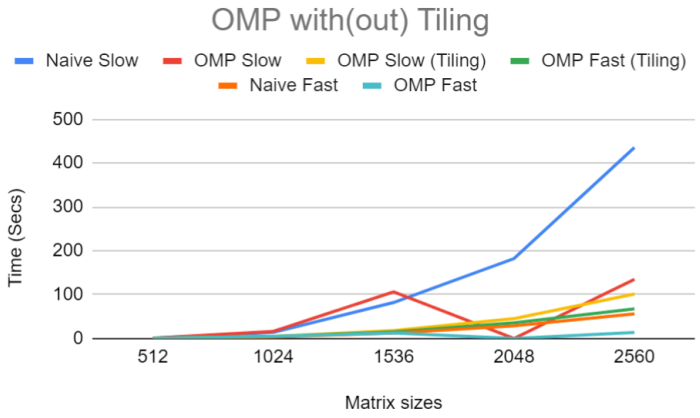


Figure 9: Naive vs OpenMP vs OpenMP with Tiling

that are inefficient for smaller matrix sizes, only proving to be optimal at larger matrix sizes above 1024.

- Inefficiency of Tiling: A noteworthy development is in both tiling implementations failing to consistently outperform the naive faster multiplication across all matrix multiplication sizes. Our prior hypothesis extends to the tiling case, with the lack of improvement implying that tiling overheads outstripped any performance gains.
- Anomalous Behaviour: Some seemingly anomalous results are the substantial improvements to runtime of both slow and fast OMP w/o tiling methods at matrix size 2048, completing the operation in fractions of a second. This is likely a functional quirk of OpenMP, which we leave to others for further analysis.

	512			
	OMP Slow	OMP Fast	OMP Slow Tiling	OMP Fast Tiling
I1 miss rate	0%	0%	0%	0%
LLi miss rate	0%	0%	0%	0%
D1 miss rate	0.60%	0.60%	0.42%	0.60%
LLd miss rate	0%	0%	0%	0%
LL miss rate	0%	0%	0%	0%

	1024			
	OMP Slow	OMP Fast	OMP Slow Tiling	OMP Fast Tiling
I1 miss rate	0%	0%	0%	0%
LLi miss rate	0%	0%	0%	0%
D1 miss rate	4.60%	0.60%	4.20%	3%
LLd miss rate	4.50%	0.60%	0%	3%
LL miss rate	1.40%	1.20%	0%	1%

	2048			
	OMP Slow	OMP Fast	OMP Slow Tiling	OMP Fast Tiling
I1 miss rate	0%	0%	0%	0%
LLi miss rate	0%	0%	0%	0%
D1 miss rate	3%	7.30%	4.20%	0.60%
LLd miss rate	3%	7.30%	0%	0%
LL miss rate	1%	1.10%	0%	0%

Figure 10: Naive vs OpenMP vs OpenMP with Tiling valgrind outputs

Hence, we conclude that parallelization is often a good strategy. At smaller matrix sizes (aside from those below 512), the gap between naive fast and fast OMP implementations is marginal, whereas at higher matrix sizes the gap is widens significantly with OMP being much more efficient.

Unfortunately, the experimental results indicate tiling to be sub-optimal for our setting. Further testing on larger matrix sizes outside our testing parameters could elicit better results. For example, smarter use of tiling can be seen in other implementations such as with the famous Strassen's algorithm, but this lies outside our intended scope for this report.

Optimization 3: Accelerating with SIMD

Next, we consider a parallization optimization at the instruction level through SIMD and the AVX family. SIMD (Single Instruction, Multiple Data) is a CPU instruction set that facilitates parallel processing by performing multiple operations simultaneously using a single instruction.

AVX (Advanced Vector Extensions): AVX2 and AVX-512 represent advanced extensions of SIMD, designed to enhance CPU parallel processing capabilities by enabling simultaneous execution of a greater number of operations on larger data vectors. The source code for both approaches can be find here: avx2_matmul.c, avx512_matmul.c.

Figure 11 shows a comparison between naive and AVX implementations.

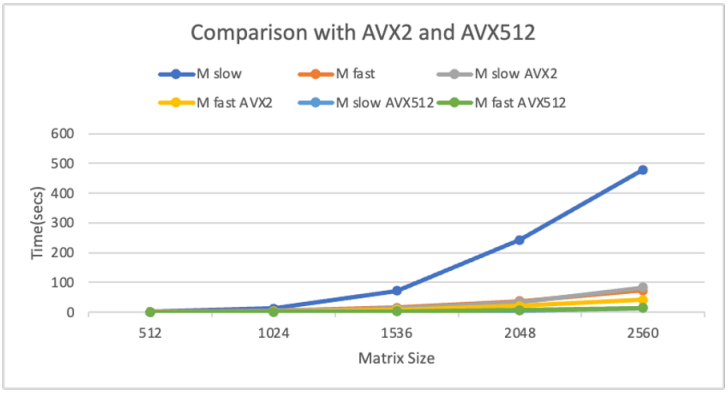


Figure 11: Naive vs AVX2 vs AVX512

Observations:

- Upward Trend with Matrix Size: As the matrix size increases, the execution time generally exhibits an upward trend for all methods. This behavior is expected due to the increased computational load associated with larger matrices.
- Consistent Outperformance of AVX2 and AVX-512: Notably, the AVX2 and AVX-512 variations consistently outperform their non-AVX counterparts across all matrix sizes. This trend underscores the benefits of advanced vector extensions in the context of matrix multiplication.

Hence, we demonstrate the effect of parallelization on optimizing matrix multiplication operations. With SIMD and AVX implementations, there are important considerations. Data alignment, vectorization, and compatibility with target architecture are paramount. Ensuring that data is properly aligned in memory and taking advantage of vectorized instructions are essential for achieving optimal performance.

Optimization 4: CUDA

Next implementation consists of using GPUs, specifically Nvidia GPUs with CUDA cores. Due to large number of CUDA cores available on GPU, matrix multiplication can take the full advantage of parallelization and tiling/blocking.

Time calculation for CUDA implementation (cuda_matmult.c) is divided into three sections:

- cudaMemcpy(HtoD): This function is used to move initial matrix from host to device (CPU to GPU).

- Mat_Mul_Shared«<dim_grid, dim_block»>: Global device function responsible for matrix multiplication.
- cudaMemcpy(DtoH): This function is used to move result matrix from device to host (GPU to CPU).

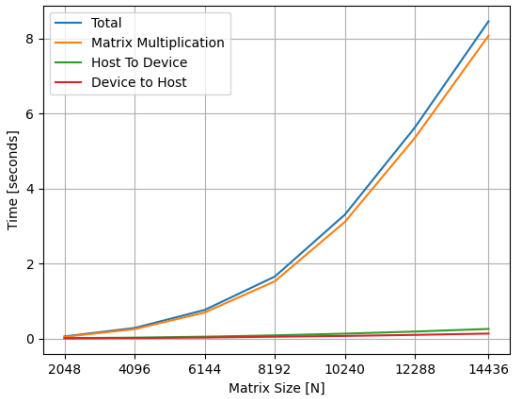


Figure 12: CUDA Implementation (with tiling)

As seen in Figure 12, cudaMemcpy(HtoD) and cudaMemcpy(DtoH) execute relatively quick. Hence not a significant factor for time plot. Comparing with all previous implementation to improve matrix multiplication, using GPU with CUDA cores is a significant improvement as expected.

Observations:

- Grid and Tiling Advantages: Compared to OpenMP, tiling in CUDA is significantly more faster. Matrix is broken down into grid, where each grid has a set of fixed tiles/blocks. This can be further optimized by using shared memory accesses. Cache efficiency is also improved by choosing appropriate tiling size.
- Faster Shared Memory Access: CUDA kernel is using shared memory is more beneficial as it reduces global memory accesses, which in turn takes longer time. Shared memory is directly present in GPU and CUDA cores can access it directly.

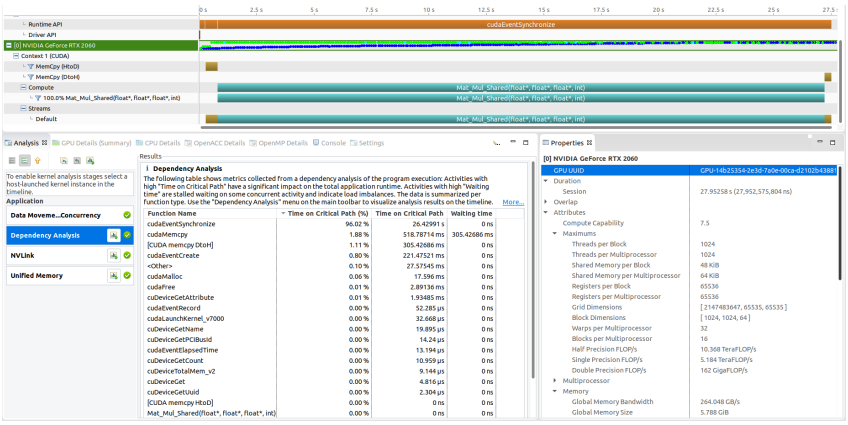


Figure 13: CUDA Summary

Outside C: Implementations in Python

We dedicate this section to a comparative analysis of matrix multiplication implementations in C and Python to discern performance trends and assess the impact of language-specific optimizations, thereby highlighting the differences in execution efficiency between the two programming languages. Another primary motivation for this comparison is to observe the effect of having access to Python libraries such as NumPy and Numba that are sources of active development.

NumPy

NumPy is a well-known Python library that adds support for large data structures and mathematical functions. It enables highly optimized matrix multiplication operations (np.dot or the @ operator) that leverage underlying C implementations.

Advantages:

- Avoiding Unnecessary Array Initialization: The optimized version skips unnecessary array initializations. In the original code, matrices m1 and m2 were initialized with values and immediately overwritten. In the optimized version, initialization is done once, and subsequent values are directly computed using NumPy operations.
- Simplified Code: The optimized version removes explicit loops and temporary arrays, creating a more concise and readable code.
- Efficient Memory Usage: The optimized version avoids unnecessary memory allocations and operations by directly updating the result matrix r using the NumPy np.dot function.

Numba

Next, we consider a parallelization method in Python as a reflection of our prior OMP work done in C. We leverage the Numba library to do so, optimize performance through parallel execution of the matrix multiplication.

Advantages:

- Numba JIT Compilation: The @njit(parallel=True) decorator is applied to the matrix multiplication function. This decorator instructs Numba to use Just-In-Time (JIT) compilation, which translates the Python code into machine code for faster execution.
- Parallel Execution: The prange(N) construct in the nested loops indicates to Numba that the iterations can be executed in parallel. This allows the work to be distributed across multiple threads, utilizing the capabilities of multi-core processors.
- Thread-Level Parallelism: Numba's parallelization primarily targets loop-level parallelism. In this specific context, the innermost loop (for k in range(N)) is parallelized. Each thread handles a portion of the work, independently updating elements in the result matrix.
- Efficient Memory Access: Numba's parallelization takes care of efficiently managing memory access patterns to avoid data hazards and contention. This is crucial for performance, especially when dealing with large matrices.

To start, we recreate the naive methods from before in Python. Then, we create variants of naive methods utilising NumPy operators. Finally, the naive methods are boosted by Numba parallelization.

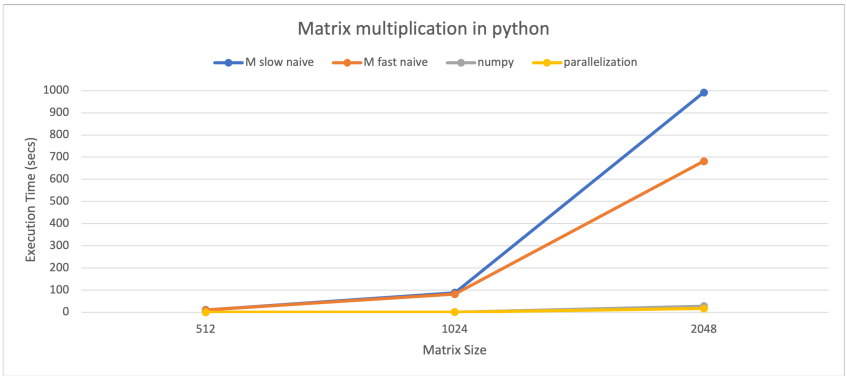


Figure 14: Naive Python vs NumPy vs Numba

Figure 14 shows that both library implementations outperform the naive Python implemented at pythonmul_naive.py. At larger matrix sizes, Numba parallelization from pythonmul_parallel.py exhibits improved performance over the NumPy modified methods (pythonmul_numpy.py and pythonmul_numpyparallel.py).

Analysis

Profiling Python code is essential for identifying performance bottlenecks and optimizing critical sections. In this section, we employed various profiling tools to analyze the matrix multiplication implementations

Tools Used:

- cProfile: Python's built-in cProfile module was utilized to collect function-level performance statistics.
- SnakeViz: The SnakeViz tool was employed to visualize the profiling data generated by cProfile.
- Valgrind: The Valgrind tool provided cache and memory profiling to analyze the Python script's performance and identify areas for optimization.

Profiling Results: The visual output generated by SnakeViz provides a detailed breakdown of function execution times, revealing areas of the code that may benefit from optimization. Key metrics include function calls, cumulative time, and exclusive time, aiding in the identification of performance bottlenecks.

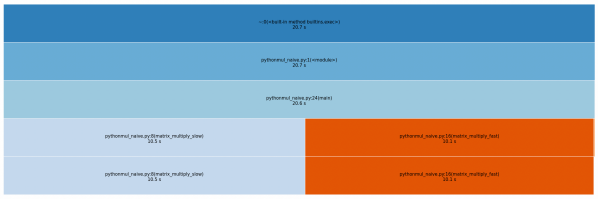


Figure 15: SnakeViz visualization of Python code profiling using cProfile for naive matrix multiplication with a matrix size of 512

These visualizations provide insight to the code and optimization opportunities by hotspot identification as they highlight specific functions that consume significant execution time.

Through these visualizations, we can see that NumPy's optimized functions outperform the naive approach by utilizing low-level, highly efficient C implementations. Additionally, parallelization in Numba effects a reduction in call stack depth, indicating more efficient computation.

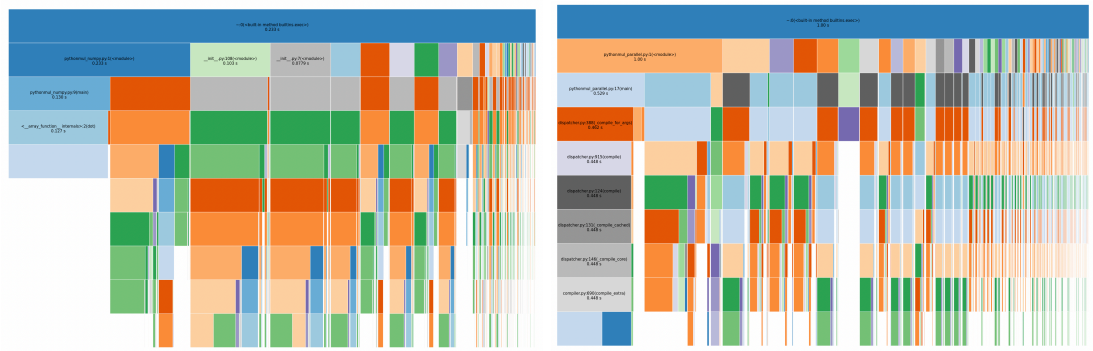


Figure 16: SnakeViz visualizations of Python code profiling using cProfile for NumPy(left) and Numba(right) based matrix multiplication at matrix size 512

From figure 14, we found that parallelization implementation outperforms NumPy implementation at larger matrix sizes. as before, we perform memory profiling using *valgrind*. The output provides insights into the cache and memory behavior of the two Python implementations (NumPy and parallelized) for matrix size 2048.

	numpy	parallel
I1 miss rate	0.01%	0.09%
LLi miss rate	0.00%	0.01%
D1 miss rate	50.1%	46.2%
LLd miss rate	49.3%	45.7%
LL miss rate	9.9%	7.9%

Figure 17: python implementations valgrind output

Inferences:

- Instruction Cache (I-cache) Misses: Both implementations have low I-cache miss rates (0.01% and 0.09%), indicating that the instruction cache is effectively utilized. This suggests that the code has good instruction cache locality.
- Data Cache (D-cache) Misses: The numpy implementation has a higher D-cache miss rate (50.1%) compared to the parallel implementation (46.2%). A lower D-cache miss rate implies better utilization of the data cache and improved cache locality.
- Last Level Cache (LLC) Misses: The numpy implementation has a higher LLC miss rate (49.3%) compared to the parallel implementation (45.7%). A lower LLC miss rate suggests better utilization of the last-level cache, leading to improved overall cache performance.
- LL Miss Rate vs D-cache Miss Rate: The lower LL miss rate for the parallel implementation (7.9%) compared to the numpy implementation (9.9%) suggests that the parallel implementation benefits from improved cache performance.
- Overall Performance: The parallel implementation appears to have better cache utilization, resulting in lower cache miss rates. Lower cache miss rates are generally indicative of better performance, as they suggest that the CPU can access data and instructions more efficiently.

Based on the provided cache and memory statistics, the parallel implementation demonstrates superior cache performance compared to the numpy implementation.

Optimized Parallel Matrix Multiplication with Numba and NumPy

Vectorization: NumPy supports vectorized operations, and we leveraged this to further optimize the code. In this optimized version, the innermost loop has been replaced with a NumPy operation (np.sum) to take advantage of vectorization. The result is an optimized parallel matrix multiplication, benefiting from Numba's just-in-time compilation and NumPy's vectorized operations. This approach is particularly effective for large matrices, where parallelization and vectorization can lead to significant performance improvements.

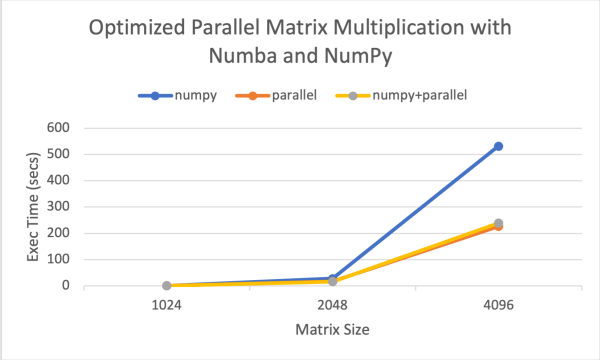


Figure 18

Conclusion and Future work

We summarise our work with the following points:

- Naive multiplication is computationally and memory intensive as each element of the output matrix is computed through a sequence of operations that are accessed inefficiently.
- We demonstrate that methods like pre-computing the transpose and parallelization can improve performance. However, one should make considerations for each strategy such as matrix size for parallelization and existence of cache inefficiencies for transpose pre-computation.
- Our report shows the importance of leveraging hardware features like SIMD and AVX for computational efficiency, especially for intensive tasks like matrix multiplication.
- We also show a comparison between C and Python implementations that underscore the impact of language and library choices on performance. Python, with libraries like NumPy and Numba, can offer competitive performance to compiled languages such as C.
- The greatest optimization gains may lie outside the CPU, such as the case of CUDA-based GPU optimization which outperformed other methods for larger matrices.

This report is far from exhaustive as the perennial usage of matrix multiplication demands constant new work and research into this optimization problem. Within this report, we hypothesize that a flexible approach to matrix multiplication may be truly optimal across all matrix sizes, one that can adaptively implement strategies such as parallelization, CUDA or even the naive approach given a small enough matrix size.