Performance Implications of Programming Languages

Jason H. Sauntry
Northeastern University
April 27, 2022
Final Report
EECE 5640

In 2020, I did a co-op at Purefacts Financial Solutions in our machine learning R&D team. One problem we spent a lot of time talking about was the memory footprint of our algorithm, which was so big that in it's first test with a real client's data, 64 G of ram hadn't been enoguh, and the program didn't work. Because of this we spent a lot of time optimizing our code. And yet, the code was in Python, a language well-known for being rather inefficient. We made our program work, but I always wondered how much performance we lost by our choice of programming language.

The question I wanted to answer in my project is a generalization of the above: how much does one's choice of programming language affect performance? Everyone knows that C is faster than Python, but how much faster? And where do other languages fit in?

Specifically, I measured the following langauges:

- Python
- C
- Java

Using the following benchmarks:

- Cannonical Matrix Multiplication
- Optimized Matrix Multiplication
- Sorting (via standard library)
- Heap (measured by sorting a list)

Performance was measured by the following metrics:

- Runtime (wall-clock)
- Peak memory usage

1 Testing Methodology

I evaluated each language over a series of benchmarks, intended to represent a variety of general-purpose yet performance-intensive computing workloads.

The source code of all benchmarks are publically availaible on GitHub, along with instructions describing how to run them.

1.1 Choice of Benchmarks

1.1.1 Cannonical Matrix Multiplication

Matrix multiplication is a common benchmark in high-performance computing and is a useful task in many applications. It's memory-intensive but has somewhat good caching perfermance due to the storage of data in 2d arrays.

I implemented this multiplication myself, based on the following psuedocode:

```
matmul(matrix a, matrix b, matrix c) {
    for (i = 0; i < N; i++) {
        for (j = 0; j < N; j++) {
            c[i][j] = 0;
            for (k = 0; j < N; k++) {
                c[i][j] += a[i][k] * b[k][j];
            }
        }
    }
}
```

No manual optimizations were applied, but C code was be compiled with `-O3`.

1.1.2 Optimized Matrix Multiplication

In addition to seeing what a language can do in typical use, I was also interested in what it can do when it has been optimized to the highest extent possible. I again used matrix multiplication for this benchmark, since it's common enough to have highly-optimized libraries in each language I used. Specifically:

Python Numpy

C OpenBLAS

Java EJML

1.1.3 Sorting

Another common task in software is sorting a list. This is a performance-intensive task, and is included in the standard libraries of each language I am evaluating. This allowed me to evaluate the efficiency of standard library functions (as opposed to 3rd-party libraries evaluated in matrix multiplication).

1.1.4 Heap

The above benchmarks all use data organized in arrays. Arrays are good for caching since they have high spatial locality. But I was also interested in what happens when operating on a pointer-heavy data structure, where closely related blocks of data may be far apart in physical memory address. I did this by implementing a heap as a binary tree.

To exercise this data structure, I sorted a list. I generated a list of floats, inserted each into the heap, then popped them all into a new list.

1.2 Data Extraction

There were two data points collected for exery benchmark: runtime and peak memory usage. Runtime was memasured directly by the language under test. Memory usage, on the other hand, was measured using Valgrind's Massif tool.

In both cases benchmarks were managed—and data collected—using a Python script. Another Python script generated graphs for analysis.

2 Results

2.1 Cannonical Matrix Multiplication

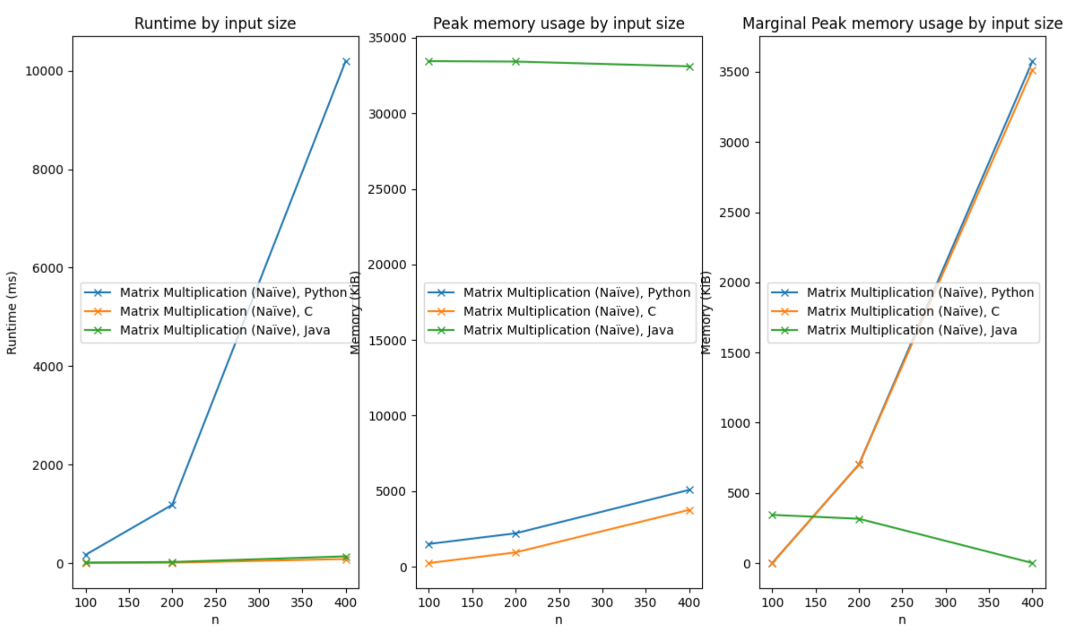

Figure 1: Performance of cannonical matrix multiplication.

Figure 1 shows the performance of cannoncical matrix multiplication. The leftmost graph shows the runtime of the benchmark. This *only* measures the benchmark itself, not any setup or tear-down. In this case, runtime includes multiplying two matrices, but not allocating the input matrices and filling them with random data.

The center graph shows peak memory usage, measured using Valgrind. This measurement includes the entire benchmark program, including both setup and tear-down. This is because there is no good way in Valgrind to determine what part of the program is running at any given time. The rightmost graph shows *marginal* memory usage, which is the peak memory usage minus the lowest peak memory usage of that program and benchmark, for any input. It is intented to show the memory associated with increased input size, without any constant overhead.

C did very well in this benchmark. At all input sizes, it was both the fastest and used the least memory.

Python did very poorly on speed, taking 10 times the runtime as C on larger input sizes. It performed much better on memory. Although Python carried a 1 MB overhead, this overhead was constant. In marginal memory usage Python was very slightly worse than C.

Java on the other hand performed very well in runtime, with runtimes only doubling that of C on larger inputs. Java's limitation was in terms of memory, with performance *considerable* worth than Python or C. This memory usage was an overhead. Java's marginal memory usage appears very good, but that is probably due to random variance in the memory overhead making the marginal memory usage an invalid measurement.

2.2 Heap

Figure 2 shows the performance of my heap implementation. As with 2.1 Cannonical Matrix Multiplication, C's performance was boring and efficient.

Python's performance was similar to 2.1 Cannonical Matrix Multiplication: memory was efficiency, constant overhead notwithstanding, but runtime was very poor.

Java's performance[1] was unremarkable. Like previously, it is roughly comparable to C.

2.3 Standard Library Sorting

Figure 3 shows the performance of sorting via the various language's standard library. As usual, C was fast and efficient.

Python's performance on this benchmark was interesting. The runtime performance was about double that of C. That doesn't seem impressive—it's still slower—but in previous benchmarks it was orders of magnitude slower. As usual, Python's memory performance was quite good. In fact, it was slightly better than C. There are two main reasons for this: firstly,

[1] Java's results for this benchmark are different that what I presented in class on April 22, 2022. I noticed a bug after the presentation where the Java implementation skipped all the computation.

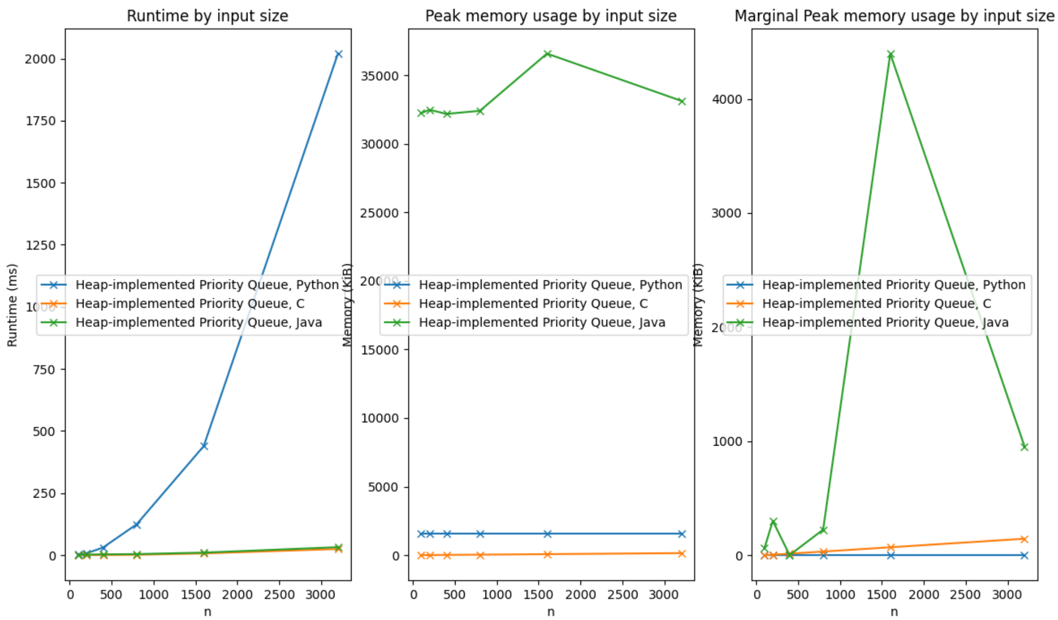

Figure 2: Performance of a heap.

the memory requirements of storing the input are high enough that the constant overhead is not noticable. This is because Python's sort implementation is in-place[2], whereas C's implementation may not be in-place[3] and thus may require additional heap space.

Java's runtime performance was similar to how the language behaved previously: worse than C, but not drastically so. It's memory performance on the other hand was impossibly good. This is discussed in **??**.

2.4 Optimized Matrix Multiplication

Figure 4 shows the performance of sorting via heavily-optimized third-party libries. This benchmark is intended to show what the language can do when it is optimized to the greatest extend possible.

C's matrix multiplicaiton was done using the OpenBLAS[4] library. As usual, C did reasonably well, but it was *not* the fastest in all cases.

Python used the popular Numpy library, although as discussed in **??**, this isn't *quite* correct. Impressively, Python's performance at the largest input size was slightly that C. It's marginal

[2] Per Python's documentation.
[3] In the GNU specification.
[4] Basic Linear Algebra System

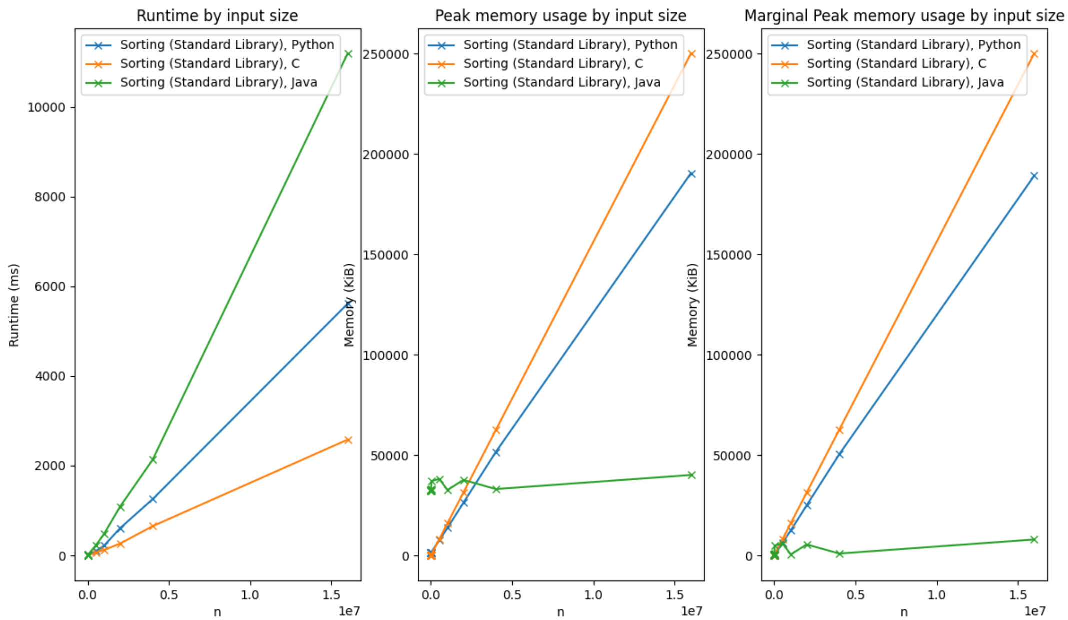

Figure 3: Performance of sorting via standard library functions.

memory performance was also very good, almost exactly equal to that of C[5].

Java used the EJML linear algebra library. It's performance was unimpressive, with runtimes larger than C, but within the same order of magnitude.

3 Analysis

3.1 Why Python is so Slow

Something that's very clear from 2.2 Heap and 2.1 Cannonical Matrix Multiplication is that Python code is really, *really* slow.

On a high level, the reasons for this is rather simple. C and Java are both ahead-of-time compiled languages: transformantion of textual code to computer-friently bytecode is done before runtime, by a compiler (either GCC or the Java Compiler). Python, on the other hand, is a JIT langauge, or just-in-time compiled. Compilation is done *at* runtime. This adds a time overhead.

Python is also an interpreted language. C's bytecode is run directly on the CPU's hardware,

[5] This is why the Python line isn't visible on the marginal memory usage graph: The Python line is exactly underneath the C line.

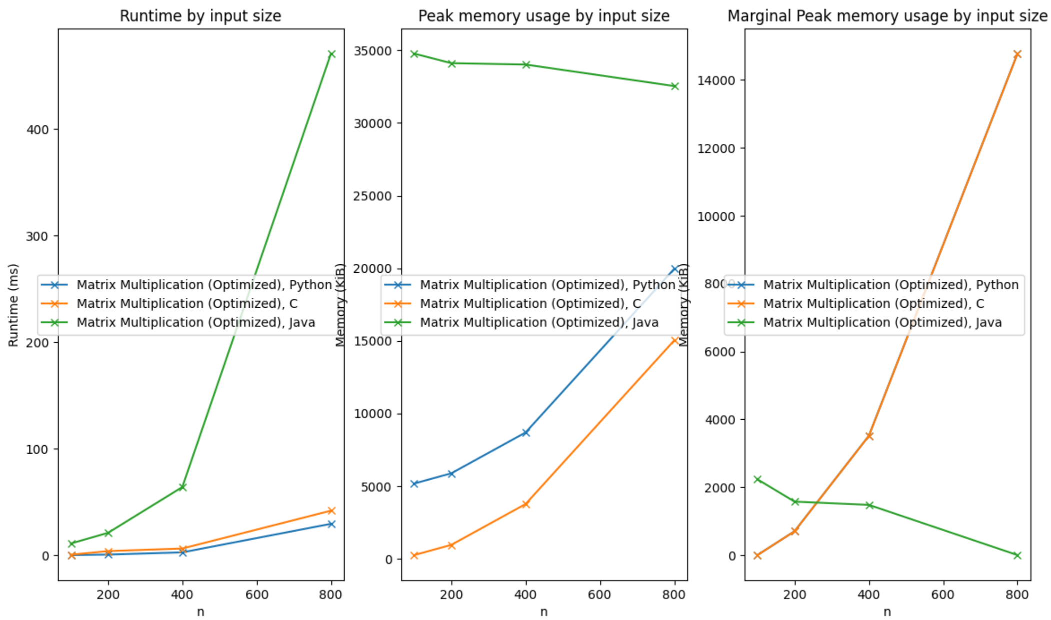

Figure 4: Performance of optimized matrix multiplication.

with no layer of indirection.[6] But Python runs in the Python Interpreter, a program that interprets and runs your code for you. Need to reference a variable? Your code has to ask the interpreter to do it for you. Want to call a function? Again, the interpret adds a (slow) layer in between your code and a function call on the CPU. All this adds up to a significant slowdown.

Python isn't the only interpreted language in my sample: Java is also interpreted: it runs in the Java Virtual Machine (JVM). But the JVM is more efficient than the Python Interpreter, probably because it's a thinner abstraction.

3.2 Java Memory Overhead

It's very apparent from all benchmarks that before allocating *any* significant data, Java has a large memory overhead, consistently about 30 MB. This is because of the JVM. As it turns out, the JVM has to store a lot of data about the code itself: compiled classes, field names and function signatures, as well as the garbage collector. In particular, compressed class space grows proportionally to the number of classes loaded[7], and can easily dwarf the heap.

For more information, see Memory footprint of the JVM, an article by Spring Blog that

[6] Except for the indirection built into CPU itself, but that is common to all programs.
[7] Which in turn is roughly proportional to the complexity of your code.

delves into this topic in some detail. For my purposes, the main takeaway is that every Java process that gets launched will immediatly take up 30 MB of RAM.

The question I'm more interested in is this: is this a significant loss of available memory? Well, it depends. Certainly it's problematic in embedded applications where available memory is sometimes measured in bytes. But Java is not commonly used for embedded tasks. Java is normally run on modern machines like telephones and computers, where gigabytes of memory are available. Suppose an application is being developed that will run on a server with 128 GB of ram available. Is this 30 MB overhead significant? Again, it depends. If the deployment plan is to run a handful of processes that each do a lot of work, the memory overhead probably isn't significant. If the plan is to run microservices, with thousands of process each doing a tiny piece of work, then this tiny overhead is now taking up gigabytes of memory, and that probably *is* a problem.

3.3 Why Sorting in Python *isn't* Terrible

As noted in 3.1, Python is rather painfully slow. Except, in 2.3 Sorting[8], it isn't. Why? Because the sorting isn't done in Python code. It's done directly by the Python Interpreter. Or, to be more precise, it's done by the CPython interpreter, since CPython is the Python implementation under test. CPython is implemented in C. So Python's sorting is implemented in C[9][10] which, as this project has made abaundantly clear, is quite fast.

So speaking generally, what are the implications of this? It means that although Python code runs slowly, standard library functions run quite quickly. So will a performance-intensive application run in a reasonable time in Python? It depends. If most of the computation happens within a standard library function, it might run quite quickly.

3.4 Why Python's Numpy is Fast

So far, we've established that Python code is very slow, but Python standard library functions can be fast, though not *as* fast as C. So one would expect that Python's optimized matrix multiplication runs very slowly. It's certainly not a standard library function. And yet, Numpy's performance was approximately the same as C, both in terms of runtime and in terms of marginal memory usage.

How is this possible? Well, Python (or, at least, the popular CPython implementation) has a rarely-used but enormously powerful feature: it can interface nicely with C and C++ code[11]. This means that a Python program with a performance-intensive component can implement that component in C, taking advantage of C's efficiency. This is exactly how

[8] It also wasn't slow in 2.4. I'll address this in **??**.

[9] For reference, see the header sorting where sorting is declared and the implementation.

[10] Presumably, if I tested Jython, it would be implemented in Java.

[11] Extending Python with C or C++

Numpy implements matrix multiplication: in C. In fact, it uses OpenBLAS. Most of the work in the Python matrix multiplication benchmark happens in a call to `cblas_dgemm`, the exact same function used in my C benchmark. So of *course* Python's performance was similar to C's: on some level, it was running exactly the same code.

4 Conclusion

Before starting this project, I had a vague idea that C was very efficient and Python was very slow, with most languages in the middle. In other words, to program in a easy-to-use language like Python, you have to pay the price of worse performance. My goal in this project was to quantify this cost, with the ultimate goal of helping answer the qustion of whether that cost is worth it.

To absolutely no one's surprise, the answer is that it depends.

The performance penalty will depend not only on what language is used, but also on *how* that language is used. Python is the most extreme example of this. A pure-Python implementation might be very slow. But perhaps an application is 90 % Python, 10 % C? If the performance-intensive part is in C, you can sort of get the best of both worlds: most of your code is built in the easy-to-use Python, but code that has to run fast is built in the more efficient C.

Ultimately, every language will have advantages and disadvantages. To intelligently choose the best language for a particular task, one needs to know what these advantages and disadvantages are. To that end, here are listed the specific, objective results of this project:

- The runtime of Python code can be over 100 times the runtime of equivalent C code.
- The runtime of Java code is generally about double the runtime of equivalent C code.
- Each Python process introduces a memory overhead of approximately 100 kB.
- Each Java process introduces a memory overhead of approximately 30 MB.
- The marginal cost of allocating more data in Python is not materially different than allocating equivalent data in C.
- The marginal cost of allocating more data in Java is difficult to measure.
- Python standard library functions can be nearly as fast or as fast as their C equivalents. The same is true of third-party libraries that have implemented some of their functionality in C.
- CPython allows easy integration with C code, meaning that a Python codebase can, in some cases, be easily accelerated by implementing a small portion of the codebase in C.

A Computer Specifications

These benchmarks can be run on any AMD-64 computer. Since runtimes are only compared to each other, running on different computers shouldn't meaningfully affect results. Additionally, since all benchmarks are single-threaded, the number of available cores shouldn't affect performance.

However, the specifications of the test machine are listed below, collected using Inxi via `inxi --admin --verbosity=7 --filter`.

```
System:
  Kernel: 5.13.0-37-generic x86_64 bits: 64 compiler: N/A
  parameters: BOOT_IMAGE=/@/boot/vmlinuz-5.13.0-37-generic
  root=UUID=4b08d9d7-9ad0-4036-b964-208d6ae91032 ro rootflags=subvol=@ quiet
  splash
  Desktop: Cinnamon 5.0.7 wm: muffin 5.0.2 dm: LightDM 1.30.0
  Distro: Linux Mint 20.2 Uma base: Ubuntu 20.04 focal
Machine:
  Type: Laptop System: Razer
  product: Blade 15 Advanced Model (Early 2020) - RZ09-033 v: 5.04
  serial: <filter> Chassis: type: 10 serial: <filter>
  Mobo: Razer model: CH551 v: 4 serial: <filter> UEFI: Razer v: 1.01
  date: 04/16/2020
Battery:
  ID-1: BAT0 charge: 81.3 Wh condition: 81.3/80.2 Wh (101%) volts: 15.8/15.4
  model: Razer Blade type: Unknown serial: <filter> status: Full
  Device-1: hidpp_battery_23 model: Logitech Wireless Mouse MX Master 3
  serial: <filter> charge: 55% (should be ignored) rechargeable: yes
  status: Discharging
Memory:
  RAM: total: 15.53 GiB used: 11.89 GiB (76.6%)
  RAM Report: permissions: Unable to run dmidecode. Root privileges required.
CPU:
  Topology: 8-Core model: Intel Core i7-10875H bits: 64 type: MT MCP arch: N/A
  family: 6 model-id: A5 (165) stepping: 2 microcode: EA L2 cache: 16.0 MiB
  bogomips: 73598
  Speed: 4084 MHz min/max: 800/5100 MHz Core speeds (MHz): 1: 4034 2: 3750
  3: 3989 4: 3809 5: 3881 6: 3897 7: 3776 8: 967 9: 2679 10: 1270 11: 2852
  12: 4201 13: 4475 14: 4330 15: 3172 16: 2983
  Flags: 3dnowprefetch abm acpi adx aes aperfmperf apic arat arch_capabilities
  arch_perfmon art avx avx2 bmi1 bmi2 bts clflush clflushopt cmov constant_tsc
  cpuid cpuid_fault cx16 cx8 de ds_cpl dtes64 dtherm dts epb ept ept_ad erms
  est f16c flexpriority flush_l1d fma fpu fsgsbase fxsr ht hwp hwp_act_window
  hwp_epp hwp_notify ibpb ibrs ibrs_enhanced ida intel_pt invpcid
  invpcid_single lahf_lm lm mca mce md_clear mmx monitor movbe mpx msr mtrr
  nonstop_tsc nopl nx ospke pae pat pbe pcid pclmulqdq pdcm pdpe1gb pebs pge
  pku pln pni popcnt pse pse36 pts rdrand rdseed rdtscp rep_good sdbg sep smap
  smep smx ss ssbd sse sse2 sse4_1 sse4_2 ssse3 stibp syscall tm tm2
  tpr_shadow tsc tsc_adjust tsc_deadline_timer vme vmx vnmi vpid x2apic
  xgetbv1 xsave xsavec xsaveopt xsaves xtopology xtpr
```

```
  Vulnerabilities: Type: itlb_multihit status: KVM: VMX disabled
  Type: l1tf status: Not affected
  Type: mds status: Not affected
  Type: meltdown status: Not affected
  Type: spec_store_bypass
  mitigation: Speculative Store Bypass disabled via prctl and seccomp
  Type: spectre_v1
  mitigation: usercopy/swapgs barriers and __user pointer sanitization
  Type: spectre_v2 mitigation: Enhanced IBRS, IBPB: conditional, RSB filling
  Type: srbds status: Not affected
  Type: tsx_async_abort status: Not affected
Graphics:
  Device-1: NVIDIA vendor: Razer USA driver: nvidia v: 510.47.03
  bus ID: 01:00.0 chip ID: 10de:1e93
  Display: x11 server: X.Org 1.20.13 driver: nvidia
  resolution: 1920x1080~60Hz, 1920x1080~240Hz
  OpenGL: renderer: llvmpipe (LLVM 12.0.0 256 bits) v: 4.5 Mesa 21.2.6
  compat-v: 3.1 direct render: Yes
Audio:
  Device-1: Intel Comet Lake PCH cAVS vendor: Razer USA driver: snd_hda_intel
  v: kernel bus ID: 00:1f.3 chip ID: 8086:06c8
  Device-2: NVIDIA TU104 HD Audio vendor: Razer USA driver: snd_hda_intel
  v: kernel bus ID: 01:00.1 chip ID: 10de:10f8
  Sound Server: ALSA v: k5.13.0-37-generic
Network:
  Device-1: Intel Wi-Fi 6 AX201 driver: iwlwifi v: kernel bus ID: 00:14.3
  chip ID: 8086:06f0
  IF: wlo1 state: up mac: <filter>
  IP v4: <filter> type: dynamic noprefixroute scope: global
  broadcast: <filter>
  IF-ID-1: gpd0 state: down mac: N/A
  IF-ID-2: tun0 state: unknown speed: 10 Mbps duplex: full mac: N/A
  IP v4: <filter> scope: global
  WAN IP: <filter>
Drives:
  Local Storage: total: 953.87 GiB used: 313.23 GiB (32.8%)
  SMART Message: Required tool smartctl not installed. Check --recommends
  ID-1: /dev/nvme0n1 vendor: LITE-ON model: CA5-8D1024 size: 953.87 GiB
  block size: physical: 512 B logical: 512 B speed: 31.6 Gb/s lanes: 4
  serial: <filter> rev: CQA0901 scheme: GPT
  Message: No Optical or Floppy data was found.
RAID:
  Message: No RAID data was found.
Partition:
  ID-1: / raw size: 46.57 GiB size: 102.45 GiB (220.00%)
  used: 53.92 GiB (52.6%) fs: btrfs dev: /dev/nvme0n1p9 label: N/A
  uuid: 4b08d9d7-9ad0-4036-b964-208d6ae91032
  ID-2: /boot/efi raw size: 100.0 MiB size: 96.0 MiB (96.00%)
  used: 30.2 MiB (31.5%) fs: vfat dev: /dev/nvme0n1p2 label: SYSTEM
  uuid: 0EA5-BB51
```

```
ID-3: /home raw size: 139.70 GiB size: 383.24 GiB (274.34%)
used: 246.31 GiB (64.3%) fs: btrfs dev: /dev/nvme0n1p8 label: N/A
uuid: 933a0fe4-8eaa-4cd1-82d1-e723b7872a8d
ID-4: /run/timeshift/backup raw size: 46.57 GiB size: 102.45 GiB (220.00%)
used: 53.92 GiB (52.6%) fs: btrfs dev: /dev/nvme0n1p9 label: N/A
uuid: 4b08d9d7-9ad0-4036-b964-208d6ae91032
ID-5: /snap/anbox/186 raw size: 373.8 MiB size: <superuser/root required>
used: <superuser/root required> fs: squashfs dev: /dev/loop0 label: N/A
uuid: N/A
ID-6: /snap/authy/8 raw size: 64.6 MiB size: <superuser/root required>
used: <superuser/root required> fs: squashfs dev: /dev/loop1 label: N/A
uuid: N/A
ID-7: /snap/authy/9 raw size: 64.6 MiB size: <superuser/root required>
used: <superuser/root required> fs: squashfs dev: /dev/loop6 label: N/A
uuid: N/A
ID-8: /snap/bare/5 raw size: 4 KiB size: <superuser/root required>
used: <superuser/root required> fs: squashfs dev: /dev/loop15 label: N/A
uuid: N/A
ID-9: /snap/blender/2090 raw size: 215.4 MiB size: <superuser/root required>
used: <superuser/root required> fs: squashfs dev: /dev/loop11 label: N/A
uuid: N/A
ID-10: /snap/blender/2106 raw size: 215.4 MiB
size: <superuser/root required> used: <superuser/root required> fs: squashfs
dev: /dev/loop14 label: N/A uuid: N/A
ID-11: /snap/core/12834 raw size: 110.6 MiB size: <superuser/root required>
used: <superuser/root required> fs: squashfs dev: /dev/loop7 label: N/A
uuid: N/A
ID-12: /snap/core/12941 raw size: 111.6 MiB size: <superuser/root required>
used: <superuser/root required> fs: squashfs dev: /dev/loop2 label: N/A
uuid: N/A
ID-13: /snap/core18/2284 raw size: 55.5 MiB size: <superuser/root required>
used: <superuser/root required> fs: squashfs dev: /dev/loop4 label: N/A
uuid: N/A
ID-14: /snap/core18/2344 raw size: 55.5 MiB size: <superuser/root required>
used: <superuser/root required> fs: squashfs dev: /dev/loop19 label: N/A
uuid: N/A
ID-15: /snap/gnome-3-28-1804/145 raw size: 162.9 MiB
size: <superuser/root required> used: <superuser/root required> fs: squashfs
dev: /dev/loop9 label: N/A uuid: N/A
ID-16: /snap/gnome-3-28-1804/161 raw size: 164.8 MiB
size: <superuser/root required> used: <superuser/root required> fs: squashfs
dev: /dev/loop5 label: N/A uuid: N/A
ID-17: /snap/gnome-3-34-1804/77 raw size: 219.0 MiB
size: <superuser/root required> used: <superuser/root required> fs: squashfs
dev: /dev/loop16 label: N/A uuid: N/A
ID-18: /snap/gtk-common-themes/1515 raw size: 65.1 MiB
size: <superuser/root required> used: <superuser/root required> fs: squashfs
dev: /dev/loop13 label: N/A uuid: N/A
ID-19: /snap/gtk-common-themes/1519 raw size: 65.2 MiB
size: <superuser/root required> used: <superuser/root required> fs: squashfs
```

```
  dev: /dev/loop12 label: N/A uuid: N/A
  ID-20: /snap/hello-world/29 raw size: 20 KiB size: <superuser/root required>
  used: <superuser/root required> fs: squashfs dev: /dev/loop3 label: N/A
  uuid: N/A
  ID-21: /snap/nordpass/130 raw size: 75.2 MiB size: <superuser/root required>
  used: <superuser/root required> fs: squashfs dev: /dev/loop20 label: N/A
  uuid: N/A
  ID-22: /snap/nordpass/131 raw size: 76.5 MiB size: <superuser/root required>
  used: <superuser/root required> fs: squashfs dev: /dev/loop18 label: N/A
  uuid: N/A
  ID-23: /snap/slack/60 raw size: 94.9 MiB size: <superuser/root required>
  used: <superuser/root required> fs: squashfs dev: /dev/loop21 label: N/A
  uuid: N/A
  ID-24: /snap/slack/61 raw size: 103.1 MiB size: <superuser/root required>
  used: <superuser/root required> fs: squashfs dev: /dev/loop17 label: N/A
  uuid: N/A
  ID-25: /snap/spotify/58 raw size: 169.6 MiB size: <superuser/root required>
  used: <superuser/root required> fs: squashfs dev: /dev/loop10 label: N/A
  uuid: N/A
  ID-26: /snap/spotify/60 raw size: 169.4 MiB size: <superuser/root required>
  used: <superuser/root required> fs: squashfs dev: /dev/loop23 label: N/A
  uuid: N/A
  ID-27: /tmp raw size: 46.57 GiB size: 45.58 GiB (97.89%)
  used: 405.9 MiB (0.9%) fs: ext4 dev: /dev/nvme0n1p7 label: N/A
  uuid: a27a7500-ed82-4d93-ad1d-1525b9dcf578
  ID-28: swap-1 size: 29.80 GiB used: 12.57 GiB (42.2%) fs: swap
  swappiness: 60 (default) cache pressure: 100 (default) dev: /dev/nvme0n1p12
  label: swap1 uuid: 5ee4d08d-45a6-4399-99c0-017ee9a7528e
Unmounted:
  ID-1: /dev/nvme0n1p1 size: 100.0 MiB fs: ntfs label: RazerRecPar
  uuid: 2484A35584A3286E
  ID-2: /dev/nvme0n1p10 size: 1000.0 MiB fs: ntfs label: Winre
  uuid: FC30B63B30B5FCA8
  ID-3: /dev/nvme0n1p11 size: 103.85 GiB fs: btrfs label: N/A
  uuid: 933a0fe4-8eaa-4cd1-82d1-e723b7872a8d
  ID-4: /dev/nvme0n1p3 size: 16.0 MiB fs: <superuser/root required> label: N/A
  uuid: N/A
  ID-5: /dev/nvme0n1p4 size: 390.62 GiB fs: ntfs label: Windows-primary
  uuid: 8C5CB5AB5CB59086
  ID-6: /dev/nvme0n1p5 size: 55.88 GiB fs: btrfs label: N/A
  uuid: 4b08d9d7-9ad0-4036-b964-208d6ae91032
  ID-7: /dev/nvme0n1p6 size: 139.70 GiB fs: btrfs label: N/A
  uuid: 933a0fe4-8eaa-4cd1-82d1-e723b7872a8d
USB:
  Hub: 1-0:1 info: Full speed (or root) Hub ports: 16 rev: 2.0 speed: 480 Mb/s
  chip ID: 1d6b:0002
  Device-1: 1-1:2
  info: Corsair CORSAIR K95 RGB PLATINUM XT Mechanical Gaming Keyboard
  type: Keyboard,HID,Mouse driver: hid-generic,usbhid interfaces: 4 rev: 2.0
  speed: 12 Mb/s chip ID: 1b1c:1b89 serial: <filter>
```

```
  Device-2: 1-7:3 info: IMC Networks Integrated Camera type: Video
  driver: uvcvideo interfaces: 4 rev: 2.0 speed: 480 Mb/s chip ID: 13d3:56d5
  serial: <filter>
  Device-3: 1-8:4 info: Razer USA Razer Blade type: Keyboard,Mouse
  driver: hid-generic,usbhid interfaces: 3 rev: 2.0 speed: 12 Mb/s
  chip ID: 1532:0253
  Device-4: 1-14:5 info: Intel type: Bluetooth driver: btusb interfaces: 2
  rev: 2.0 speed: 12 Mb/s chip ID: 8087:0026
  Hub: 2-0:1 info: Full speed (or root) Hub ports: 8 rev: 3.1 speed: 10 Gb/s
  chip ID: 1d6b:0003
  Hub: 3-0:1 info: Full speed (or root) Hub ports: 2 rev: 2.0 speed: 480 Mb/s
  chip ID: 1d6b:0002
  Hub: 4-0:1 info: Full speed (or root) Hub ports: 4 rev: 3.1 speed: 10 Gb/s
  chip ID: 1d6b:0003
  Hub: 5-0:1 info: Full speed (or root) Hub ports: 2 rev: 2.0 speed: 480 Mb/s
  chip ID: 1d6b:0002
  Hub: 6-0:1 info: Full speed (or root) Hub ports: 2 rev: 3.1 speed: 10 Gb/s
  chip ID: 1d6b:0003
Sensors:
  System Temperatures: cpu: 62.0 C mobo: N/A gpu: nvidia temp: 63 C
  Fan Speeds (RPM): N/A
Info:
  Processes: 693 Uptime: 6d 12h 09m Init: systemd v: 245 runlevel: 5
  Compilers: gcc: 9.4.0 alt: 8/9 Shell: bash v: 5.0.17 running in: server
  inxi: 3.0.38
```