Assignment 1

Yusheng Zhao

February 13, 2023

1 Problem 1

We will be using the following formula to calculate FLOPS of my CPU

$$FLOPS = cpu\_speed \times num\_cores \times avx\_factor \times fma\_factor$$

Now, we will collect information of above parameters from our CPU. However, MacOS does not support `lsmem` and `lscpu`. The best alternative I could find on the internet is to execute the following scirpt

```
sysctl -a | grep machdep.cpu | fold -w60
```

```
machdep.cpu.cores_per_package: 10
machdep.cpu.core_count: 10
machdep.cpu.logical_per_package: 10
machdep.cpu.thread_count: 10
machdep.cpu.brand_string: Apple M1 Max
machdep.cpu.features: FPU VME DE PSE TSC MSR PAE MCE CX8 API
C SEP MTRR PGE MCA CMOV PAT PSE36 CLFSH DS ACPI MMX FXSR SSE
 SSE2 SS HTT TM PBE SSE3 PCLMULQDQ DTSE64 MON DSCPL VMX EST
TM2 SSSE3 CX16 TPR PDCM SSE4.1 SSE4.2 AES SEGLIM64
machdep.cpu.feature_bits: 151121000215084031
machdep.cpu.family: 6
```

However, I could not find any wanted parameters here. Therefore, I just `ssh` into another linux machine and got its info instead. Here is the output for `lscpu` on that machine.

```
Architecture:                    x86_64
CPU op-mode(s):                  32-bit, 64-bit
```

```
Address sizes:                   46 bits physical, 57 bits v
irtual
Byte Order:                      Little Endian
CPU(s):                          8
On-line CPU(s) list:             0-7
Vendor ID:                       GenuineIntel
Model name:                      Intel Xeon Processor (Icela
ke)
CPU family:                      6
Model:                           134
Thread(s) per core:              1
Core(s) per socket:              1
Socket(s):                       8
Stepping:                        0
BogoMIPS:                        5985.93
Flags:                           fpu vme de pse tsc msr pae
mce cx8 apic sep mtrr pge mca cmov pat pse36 clflush mmx fxs
r sse sse2 ss syscall nx pdpe1gb rdtscp lm constant_tsc rep_
good nopl xtopology cpuid tsc_known_freq pni pclmulqdq vmx s
sse3 fma cx16 pcid sse4_1 sse4_2 x2apic movbe popcnt tsc_dea
dline_timer aes xsave avx f16c rdrand hypervisor lahf_lm abm
 3dnowprefetch cpuid_fault invpcid_single ssbd ibrs ibpb sti
bp ibrs_enhanced tpr_shadow vnmi flexpriority ept vpid ept_a
d fsgsbase tsc_adjust bmi1 avx2 smep bmi2 erms invpcid avx51
2f avx512dq rdseed adx smap avx512ifma clflushopt clwb avx51
2cd sha_ni avx512bw avx512vl xsaveopt xsavec xgetbv1 xsaves
wbnoinvd arat avx512vbmi umip pku ospke avx512_vbmi2 gfni va
es vpclmulqdq avx512_vnni avx512_bitalg avx512_vpopcntdq la5
7 rdpid fsrm md_clear arch_capabilities
Virtualization:                  VT-x
Hypervisor vendor:               KVM
Virtualization type:             full
L1d cache:                       256 KiB (8 instances)
L1i cache:                       256 KiB (8 instances)
L2 cache:                        32 MiB (8 instances)
L3 cache:                        128 MiB (8 instances)
NUMA node(s):                    1
NUMA node0 CPU(s):               0-7
Vulnerability Itlb multihit:     Not affected
Vulnerability L1tf:              Not affected
```

```
Vulnerability Mds:                Not affected
Vulnerability Meltdown:           Not affected
Vulnerability Mmio stale data:    Vulnerable: Clear CPU buffe
rs attempted, no microcode; SMT Host state unknown
Vulnerability Retbleed:           Not affected
Vulnerability Spec store bypass: Mitigation; Speculative Sto
re Bypass disabled via prctl and seccomp
Vulnerability Spectre v1:         Mitigation; usercopy/swapgs
 barriers and __user pointer sanitization
Vulnerability Spectre v2:         Mitigation; Enhanced IBRS,
IBPB conditional, RSB filling, PBRSB-eIBRS Not affected
Vulnerability Srbds:              Not affected
Vulnerability Tsx async abort:    Mitigation; TSX disabled
```

I notice, I have 8 cores and they support `avx512f` and `fma`. According to this answer, we adjust `avx_fct` and `fma_fct` used in calculation accordingly.

I notice the `lscpu` reports `BogoMIPS` instead of `cpu speed`. So I did `cat /proc/cpuinfo | grep cpu` to find out the cpu clock speed is in fact `2992.969 MHz`.

The FLOPS is calculated by the following code.

```
begin
    cpu_speed = 2.992969; # GHz
    num_cores = 8;
    avx_fct = 512 / 64; # 2 floats
    fma_fct = 2;
    println("The computing power of my CPU is \
            $(cpu_speed * num_cores * avx_fct * fma_fct) GFLOPS")
end

The computing power of my CPU is 383.100032 GFLOPS
```

Thas is extremely fast. However, we also need to consider data I/O bottleneck. Meaning, we need to take data from **DRAM** to **SRAM** on the CPU. According to the lecture, this is about `100 GB/s`. Each GFLOP uses roughly `32 *2 GB` of data on `x64` machine.

```
begin
    io_speed = 100 #GB/s
    gflop_data = 32 * 2 #GB , we operate on two floats
    println("data I/O bottleneck caps the machine speed to \
```

```
                $(io_speed/gflop_data) GFLOPS")
 end
```

```
data I/O bottleneck caps the machine speed to 1.5625 GFLOPS
```

Lastly, if the program requires multiple data I/O, we will then hit the latency problem. The output of `lsmem` shows that we have `64GB` of ram, the OS will take away approximately `1GB`. So we could use `63GB` to do useful things. Every `63GB` of data, we need extra `50ns` of time to let the machine know we need new data!

```
RANGE                                  SIZE  STATE REMOVABLE
BLOCK
0x0000000000000000-0x00000000bfffffff   3G online       yes
  0-2
0x0000000100000000-0x000000103fffffff  61G online       yes
 4-64

Memory block size:         1G
Total online memory:      64G
Total offline memory:      0B
```

```
begin
    mem_size = 63; #GB
    gflop_data = 32 * 2 ; #GB
    latency = 50 /(10^9); # s
    println("Latency issue caps our machine speed at \
         $(mem_size/gflop_data/(1+latency)) GFLOPS")
end
```

```
Latency issue caps our machine speed at 0.9843749507812526 GFLOPS
```

In conclusion, for a short amount of time, our machine could do work at `383.100032 GFLOPS`. Then it will hit a data I/O bottleneck and degrade to `1.52625 GFLOPS`. Eventually, it will hit the latency bottleneck and degrade to less than `0.9843749507812526 GFLOPS`.

2 Problem 2

Let us assume information is propogating at the speed of light. And latency time is the time it took starting from CPU issues a data request til the time data starts to arrive at the cache.

```
begin
    c = 29979245800 # cm/s
    dist = 10 # cm
    cpu_freq =  2.992969 # GHz
    println("The latency is $(dist*2/c) seconds")
    println("The CPU clock time is $(1/(cpu_freq * 10^9)) seconds")
end

The latency is 6.671281903963041e-10 seconds
The CPU clock time is 3.341163907811942e-10 seconds
```

The minimum latency time is **roughly twice** as the cpu clock. But we were told in class that it's much longer. Probably due to the need to actually find and retrieve those data in DRAM.

3 Problem 3

NPU is a processor that specializes in the operations over machine learning tasks. Regarding the machine learning tasks, it will be more perfomant than a conventional CPU and more power efficient and performant than the CPU and GPU combination. To achieve such advantage on machine learning tasks, it has two major differences compared to a CPU. Firstly, it supports way less instructions compare to a CPU. But for the instructions it supports, its architecture makes sure it's really efficient and powerful at it. For example, it is good at multiply-accumulate calculation which is used a lot in machine learning tasks. The supported calculation is also of low precission, (8 bits compare to 64 and 32 bit supported in CPU), which further decreases the calculation load of it. Secondly, it supports paralized computation much better than CPU does. NPU beats CPU and GPU combination in terms of data processing speed since it does not require data to be transferred from the CPU to GPU. Everything is handled inside NPU. Therefore, it is good for machine learning tasks.

According to this Wiki Chip page, the TPU is a specialized type of NPU. It is good at deep learning tasks where the NPU is good at machine learning tasks in general. They are also proprietary. Google develops and only provides it on GCP. According to this page, TPU is highly parallizable and programmable.

FPGA, like the name suggests, is programmable and could be used for prototyping devices. Customers could purchase the device long before they figure out what exactly their functionality needs to be. The programmability

of FPGA is achieved via Programmable interconnect and programmable logic block (See first figure on this site). From the homogenous composition of the structure of FPGA, we may infer thatit is very good at parallel computation.

3.1 Sources

Here are a few pages that I referenced. I did not bother to put them in bibtex form. - NPU

- NPU2
- FPGA
- TPU

4 Problem 4

In conclusion, **4** axioms are violated. I will discuss about each of them.

Note, the following discussion does not consider the three special values `Inf`, `-Inf`, and `NaN` as the starting point of operation. But if any of them is reached during the calculation, it's fine. If you include them, **all of the axioms are violates, some are not well-defined**

4.1 Associativity of addition

Satisfied

```
begin
    trials = 100000
    rand_float_pairs = rand(Float32,(2,trials))
    for ii in 1:trials
        a,b = rand_float_pairs[:,ii]
        # we probably want to use == in this case since we are testing
        # if the same EXACTLY
        @assert bitstring(a + b) == bitstring(b + a) "Addition is not \
                                                      associative for floats"
    end
    println("Addition is associative for floats")
    if !(NaN32 + 0.5 == 0.5 + NaN32)
        # had to do this org babel does not output err
        println("NaN violates this axiom!")
```

```
    end
end

Addition is associative for floats
NaN violates this axiom!
```

4.2 Exitence of additive identity

Not satisfied, we see 0 is not approximated in the bit-wise representation of a float. However, there are two additive identity 0.0 and -0.0!

```
zero = 0.0
other_zero = -0.0
if bitstring(zero) != bitstring(other_zero)
    println("Additive identity is not unique!")
end
if !(NaN32 - NaN32 == 0.0)
    println("NaN violates this axiom")
end

0.0
-0.0
Additive identity is not unique!
NaN violates this axiom
```

4.3 Existence of additive inverses

Satisfied, we could always invert the sign bit of a float to create its additive inverse.

```
begin
    trials = 1000000
    rand_floats = rand(Float32,trials)
    for i in 1:trials
        @assert rand_floats[i] - rand_floats[i] == 0.0 "Additive \
                                                Inverse does not exist"
        a_str = bitstring(rand_floats[i])
        b_str = bitstring(- rand_floats[i])
        @assert  a_str[2:32] == b_str[2:32] && a_str[1] != b_str[1] "Additive\
              Inverse does not only differs by sign"
    end
```

```
    println("There exists additive inverse for all floats")
    if !(NaN32 - NaN32 == 0.0 )
        println("NaN violates this axiom")
    end
end
```

```
There exists additive inverse for all floats
NaN violates this axiom
```

4.4 Commutativity of multiplication

Satisfied

```
begin
    trials = 1000000
    rand_floats = rand(Float32,(2,trials))
    for i in 1:trials
        @assert (rand_floats[1,i] * rand_floats[2,i] ==
                    rand_floats[2,i] * rand_floats[1,i])
        "multiplication is not commutative"
    end
    println("Multiplication is commutative")
    if !(NaN32 * 1.0 == 1.0 * NaN32)
        println("NaN violates this axiom")
    end
end
```

```
Multiplication is commutative
NaN violates this axiom
```

4.5 Associativity of Multiplication

Not satisfied, consider a overflowing case

```
begin
    a = prevfloat(typemax(Float32));
    b = 2^5;
    c = 1/2^5;
    (a * b) * c == a * (b * c)
end
```

```
false
```

4.6 Existence of multiplicative identity

Satisfied, in theory, 1.0 is well represented in floats, i.e no approximation. So we could always implement a special operation rule such that when it sees 1.0 been multiplied to something, it simply returns that thing.

```
begin
    trials = 1000000
    rand_floats = rand(Float32,trials)
    for i in 1:trials
        @assert rand_floats[i] * 1.0 == rand_floats[i] "There's no \
                 identity for multiplication"
    end
    println("1.0 is the multiplicative identity")
end
#again NaN violates it but I won't write it out
```

```
1.0 is the multiplicative identity
```

4.7 Existence of multiplicative inverses

Not satisfied, there exists number whose inverse can only be approximately represented. When they get multiplied together, you don't get 1.0.

```
begin
    a = 9/11
    a_inv = 11/9
    a * a_inv == 1.0
end
```

```
false
```

4.8 Distributive law

Not satisfied, like associativity of multiplication, overflow could be a big problem.

```
begin
    a = prevfloat(typemax(Float32))
    b = - a / 2
    c = 100
    (a + b) * c  == a * c + b * c
end
```

```
false
```

4.9 Zero-one law

Satisfied, 1.0 does not equal to either 0.0 or -0.0.