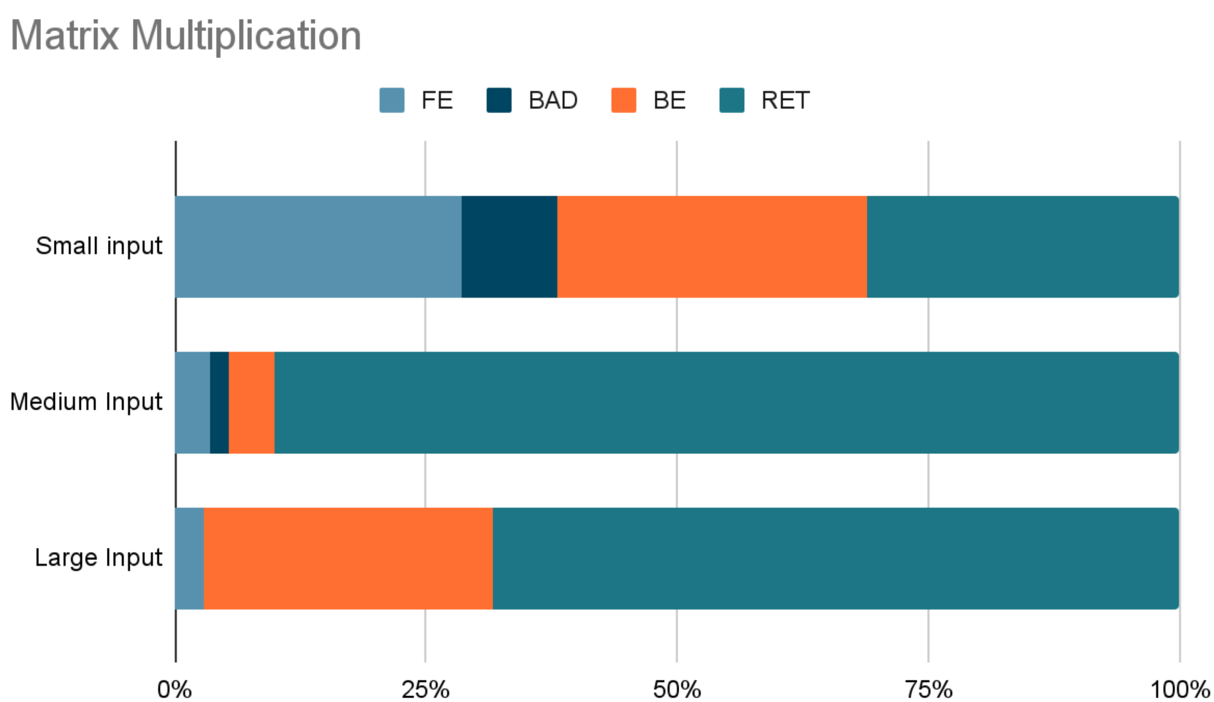
When profiling a function, I repeated the core function 10-100 times. I also profiled with 3 different input magnitudes (10, 100, 1000). I repeated the core function a lesser number of times for larger inputs and a higher number of times for smaller inputs. For all tests, I first allocated memory for each of the input variables. Then, I assigned values to them sequentially.

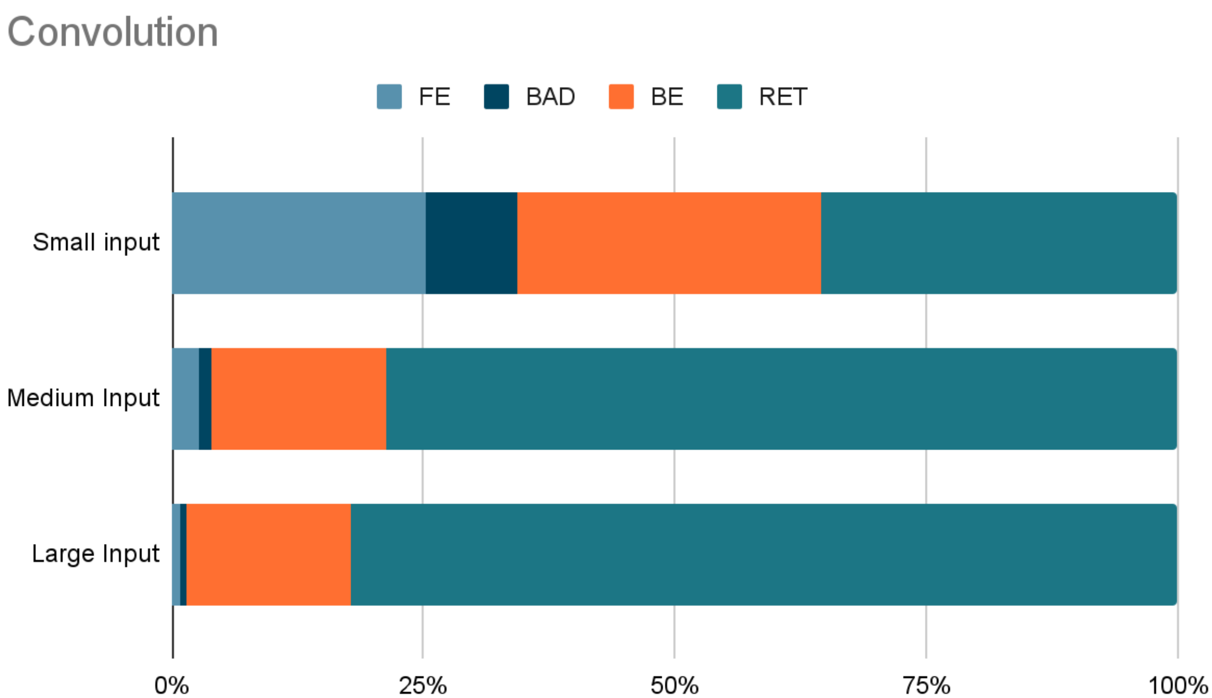
Issues in the lab

- Writing and debugging test cases was laborious as I was new to C and the process of allocating memory before using dynamically sized arrays
    - After discovering gdb, I was able to debug seg faults much faster
- The manual process of copying the values after running each profiling was also time-consuming



- Implementation - I ran 3 nested loops which iterated over the A row, B col, and the Acol/Brow
- Test - I tested with matrix sizes of (3,3), (100,100), (1000,1000)
- Bottleneck observation -
    - As I increased the input size,
        - % share of the frontend bound decreased
        - % share of bad speculation decreased
        - % share of backend-bound decreased for the medium input and increased for the large input

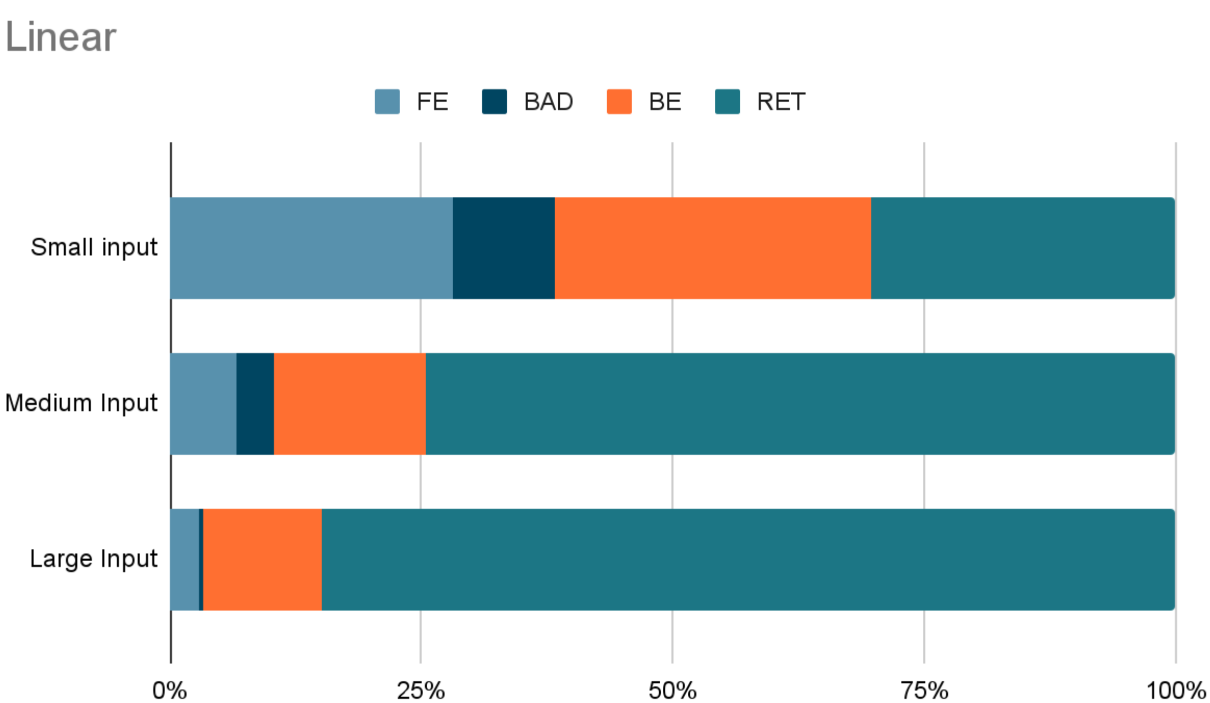
        - % share of retiring increased for medium input and decreased for large input
    - The main bottleneck seems to be retiring



- Implementation - I ran 6 nested loops. The first 3 loops iterate through each cell of the final output. The inner 3 loops loop through the row, col, and channel dimensions of the image.
- Test
    - I used the following sizes for testing
        - Channels - 3
        - Filters - 1
        - Kernel - 2
        - Input image size - 3, 100, 1000
- Bottleneck observation -
    - As I increased the input size,
        - % share of the frontend bound decreased
        - % share of bad speculation remained unchanged
        - % share of backend-bound remained unchanged
        - % share of retiring increased
    - The main bottleneck seems to be retiring

Cache data after running the largest conv output

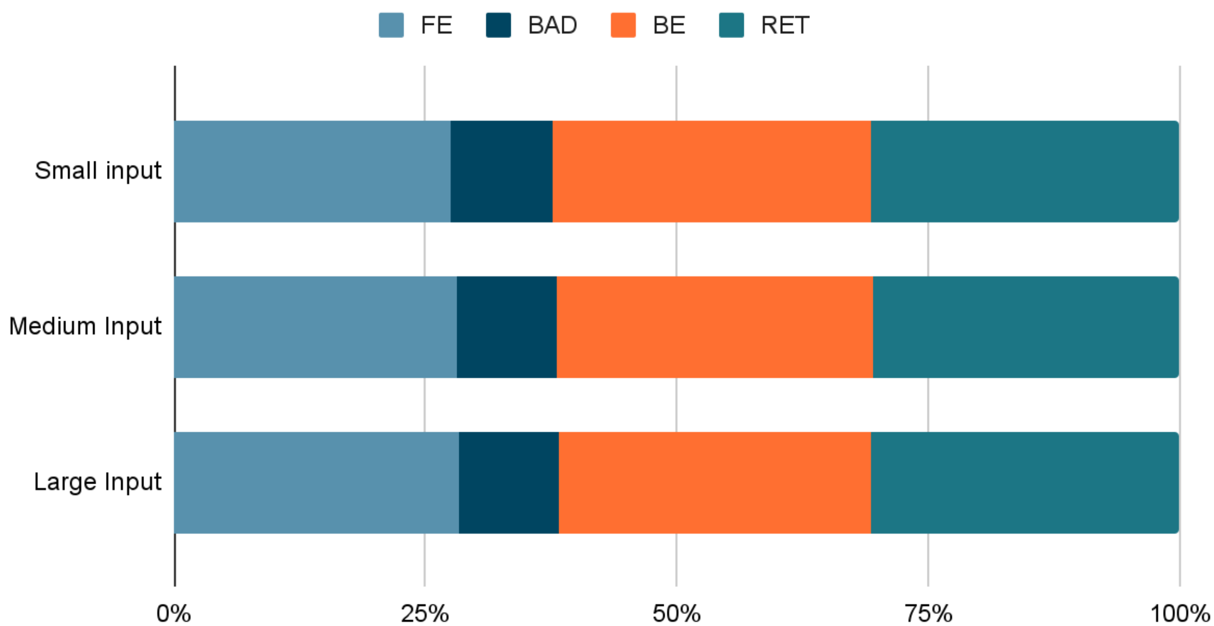
```
Architecture:            x86_64
  CPU op-mode(s):        32-bit, 64-bit
  Address sizes:         45 bits physical, 48 bits virtual
  Byte Order:            Little Endian
CPU(s):                  2
  On-line CPU(s) list:   0,1
Vendor ID:               GenuineIntel
  Model name:            Intel(R) Xeon(R) CPU E5-2660 v4 @ 2.00GHz
    CPU family:          6
    Model:               37
    Thread(s) per core:  1
    Core(s) per socket:  1
    Socket(s):           2
    Stepping:            1
    BogoMIPS:            3999.99
    Flags:               fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36 clflush mmx fxsr sse sse2 syscall nx rdtscp lm constant_tsc arch_perfmon nopl xtopology tsc_reliable nonstop_tsc cpuid tsc_known_freq pni pclmu
                         lqdq ssse3 cx16 sse4_1 sse4_2 x2apic popcnt tsc_deadline_timer aes hypervisor lahf_lm pti ssbd ibrs ibpb stibp tsc_adjust arat flush_l1d arch_capabilities
Virtualization features:
  Hypervisor vendor:     VMware
  Virtualization type:   full
Caches (sum of all):
  L1d:                   64 KiB (2 instances)
  L1i:                   64 KiB (2 instances)
  L2:                    512 KiB (2 instances)
  L3:                    70 MiB (2 instances)
NUMA:
  NUMA node(s):          1
  NUMA node0 CPU(s):     0,1
Vulnerabilities:
  Itlb multihit:         KVM: Mitigation: VMX unsupported
  L1tf:                  Mitigation; PTE Inversion
  Mds:                   Vulnerable: Clear CPU buffers attempted, no microcode; SMT Host state unknown
  Meltdown:              Mitigation; PTI
  Mmio stale data:       Unknown: No mitigations
  Retbleed:              Mitigation; IBRS
  Spec store bypass:     Mitigation; Speculative Store Bypass disabled via prctl and seccomp
  Spectre v1:            Mitigation; usercopy/swapgs barriers and __user pointer sanitization
  Spectre v2:            Mitigation; IBRS, IBPB conditional, STIBP disabled, RSB filling, PBRSB-eIBRS Not affected
  Srbds:                 Not affected
  Tsx async abort:       Not affected
hr322@slurm-instructional-login:~/ece5755-modern-computer-architecture/lab-1/lab1$
```



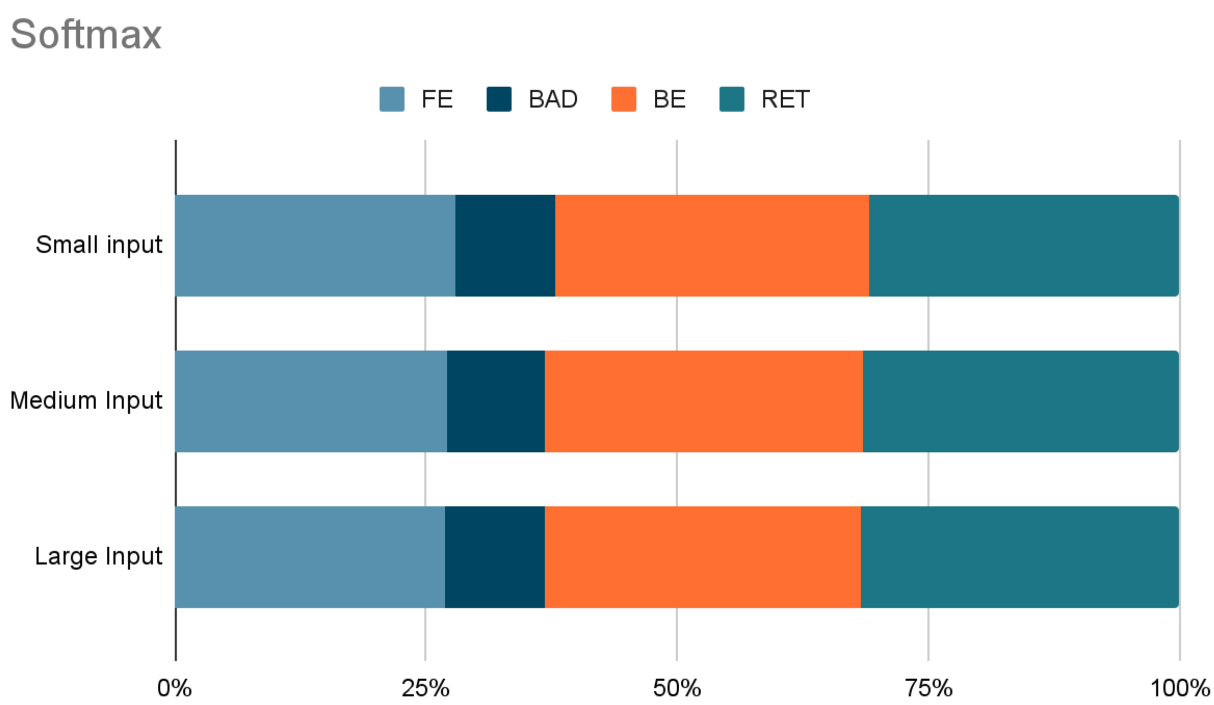
- Implementation - I ran a loop for each row of the input function. And another inner loop for multiplying and summing the cols of each input row
- Test
  - I used the following sizes for testing
    - I used the same dimensions as the input for both the row and col dimension of the weights
    - Input size - 3, 100, 1000
- Bottleneck observation -
  - As I increased the input size,
    - % share of the frontend bound decreased

        - % share of bad speculation remained unchanged
        - % share of backend-bound remained unchanged
        - % share of retiring increased
    - The main bottleneck seems to be retiring

Relu



- Implementation - I calculated the relu function via if statements
- Test
    - I used the following sizes for testing
        - I used the same dimensions as the input for both the row and col dimension of the weights
        - Input size - 3, 100, 1000
- Bottleneck observation -
    - As I increased the input size,
        - % share of the frontend remained unchanged
        - % share of bad speculation remained unchanged
        - % share of backend-bound remained unchanged
        - % share of retiring remained unchanged
    - The main bottleneck seems to be backend-bound



- Implementation - I calculated the softmax by calculating the sum via 1 loop and then the softmax values via another loop
- Test
    - I used the following sizes for testing
        - I used the same dimensions as the input for both the row and col dimension of the weights
        - Input size - 3, 100, 1000
- Bottleneck observation -
    - As I increased the input size,
        - % share of the front remained unchanged
        - % share of bad speculation remained unchanged
        - % share of backend-bound remained unchanged
        - % share of retiring remained unchanged
    - The main bottleneck seems to be backend-bound

Potential improvements

- Since most of the large input bottleneck is retiring in nature, I think the performance can be improved overall by implementing multiple threads as many of the sub-operations are independent of each other