**PS4 Questions**

=============

Add your answers to this file in plain text after each question. Leave a blank line between the text of the question and the text of your answer.

**CPU**

---

1. **What level of SIMD/vector support does the CPU your computer provide?**

(pasting relevant output of cpuinfo583.exe)

CPUID\_EBX\_AVX2

CPUID\_ECX\_SSE3

CPUID\_ECX\_SSSE3

CPUID\_ECX\_FMA

CPUID\_ECX\_SSE41

CPUID\_ECX\_SSE42

CPUID\_ECX\_AVX

CPUID\_EDX\_SSE

CPUID\_EDX\_SSE2

Computer has SSE, FMA and AVX support.

1. **What is the maximum operand size that your computer will support?**

256-bit operands for AVX.

128-bit operands for SIMD.

So, the CPU will support max operand size of 256 bit.

1. **What is the minimum operand size that your computer will support?**

256 bit operands for AVX.

128 bit operands for SIMD.

So max operand size of 256 bit.

1. **What is the clock speed of your CPU? You may need to look this up via "About this Mac" on MacOSX. If your Macbook is using Apple M1 Chip, try "sudo powermetrics" in the terminal, looking for "CPU frequency". Look at the "Performance" Tab in Task Manager on a Windows; and try "lscpu" in the terminal on a Linux.**

Base clock speed is 2.8 GHz and can go up to 5.8 GHz.

1. **Based on the output from bandwidth.exe on your computer, what do you expect L1 cache and L2 cache sizes to be? What are the corresponding bandwidths? How do the cache sizes compare to what "about this mac" (or equivalent) tells you about your CPU? (There is no "right" answer for this question -- but I do want you to do the experiment.)**

The machine has 8 physical cores with 2 threads per core which gives 16 logical cores

bytes/elt #elts res\_bytes ntrials usecs ttl\_bytes bytes/sec

8 16 128 67108868 15000 8589935104 5.72662e+11

8 32 256 33554436 8000 8589935616 1.07374e+12

8 64 512 16777220 4000 8589936640 2.14748e+12

8 128 1024 8388612 1000 8589938688 8.58994e+12

8 256 2048 4194308 0 8589942784 0

8 512 4096 2097156 0 8589950976 0

8 1024 8192 1048580 0 8589967360 0

8 2048 16384 524292 0 8590000128 0

8 4096 32768 131074 0 4295032832 0

8 8192 65536 65538 0 4295098368 0

8 16384 131072 32770 0 4295229440 0

8 32768 262144 16386 0 4295491584 0

8 65536 524288 8194 0 4296015872 0

8 131072 1048576 2049 0 2148532224 0

8 262144 2097152 1025 0 2149580800 0

8 524288 4194304 513 0 2151677952 0

8 1048576 8388608 257 0 2155872256 0

8 2097152 16777216 129 0 2164260864 0

8 4194304 33554432 65 0 2181038080 0

8 8388608 67108864 33 0 2214592512 0

8 16777216 134217728 17 0 2281701376 0

write

bytes/elt #elts res\_bytes ntrials usecs ttl\_bytes bytes/sec

8 16 128 67108868 64000 8589935104 1.34218e+11

8 32 256 33554436 65000 8589935616 1.32153e+11

8 64 512 16777220 65000 8589936640 1.32153e+11

8 128 1024 8388612 65000 8589938688 1.32153e+11

8 256 2048 4194308 64000 8589942784 1.34218e+11

8 512 4096 2097156 65000 8589950976 1.32153e+11

8 1024 8192 1048580 64000 8589967360 1.34218e+11

8 2048 16384 524292 65000 8590000128 1.32154e+11

8 4096 32768 131074 33000 4295032832 1.30153e+11

8 8192 65536 65538 32000 4295098368 1.34222e+11

8 16384 131072 32770 34000 4295229440 1.2633e+11

8 32768 262144 16386 33000 4295491584 1.30166e+11

8 65536 524288 8194 35000 4296015872 1.22743e+11

8 131072 1048576 2049 23000 2148532224 9.34144e+10

8 262144 2097152 1025 23000 2149580800 9.346e+10

8 524288 4194304 513 44000 2151677952 4.89018e+10

8 1048576 8388608 257 379000 2155872256 5.68832e+09

8 2097152 16777216 129 362000 2164260864 5.97862e+09

8 4194304 33554432 65 381000 2181038080 5.72451e+09

8 8388608 67108864 33 368000 2214592512 6.01791e+09

8 16777216 134217728 17 385000 2281701376 5.9265e+09

read/write

bytes/elt #elts res\_bytes ntrials usecs ttl\_bytes bytes/sec

8 8192 131072 32770 33000 4295229440 1.30158e+11

8 16384 262144 16386 33000 4295491584 1.30166e+11

8 32768 524288 8194 34000 4296015872 1.26353e+11

8 65536 1048576 2049 22000 2148532224 9.76606e+10

8 131072 2097152 1025 22000 2149580800 9.77082e+10

8 262144 4194304 513 44000 2151677952 4.89018e+10

8 524288 8388608 257 276000 2155872256 7.81113e+09

8 1048576 16777216 129 245000 2164260864 8.83372e+09

8 2097152 33554432 65 250000 2181038080 8.72415e+09

8 4194304 67108864 33 254000 2214592512 8.71887e+09

8 8388608 134217728 17 186000 2281701376 1.22672e+10

Looking at plateaus and drops in the above table we get,

L1 = 524KB (bandwidth = 126 GB/s)

L2 = 4MB (bandwidth = 49 GB/s)

Task Manager -> "Performance Tab" on my Windows machine gives following cache sizes:

L1 = 512KB

L2 = 4MB

1. **Based on the output from running this image on your computer, what do you expect L1 cache and L2 cache (if present) sizes to be? What are the corresponding bandwidths? How do the cache sizes compare to what “about this mac” (or equivalent) tells you about your CPU? (There is no “right” answer for this question – but I do want you to do the experiment.)**

The machine has 8 physical cores with 2 threads per core which gives 16 logical cores

L1 = 32KB which gives 512KB with 16 logical cores.

L2 = 512KB which gives 4MB with 16 logical cores.

L1/L2 are per core as per the machine’s CPU cache organization.

Task Manager -> "Performance Tab" on my Windows machine gives following cache sizes:

L1 = 512KB

L2 = 4MB

**Perf**

----

**Roofline: Homegrown**

-------------------

**What is the (potential) maximum compute performance of your computer? (The horizontal line.) What are the L1, L2, and RAM bandwidths?**

**How do those bandwidths correspond to what was measured above with the bandwidth program?**

$ ./roofline.exe 4194304

kernel sz res\_bytes ntrials usecs ttl\_bytes ttl\_flops intensity flops/sec bytes/sec

2 4194304 8195 1.068e+06 68744642560 8593080320 0.125 8.04596e+09 6.43676e+10

4 4194304 4099 410000 34384904192 8596226048 0.25 2.09664e+10 8.38656e+10

8 4194304 2051 266000 17205035008 8602517504 0.5 3.23403e+10 6.46806e+10

16 4194304 1027 300000 8615100416 8615100416 1 2.8717e+10 2.8717e+10

32 4194304 515 341000 4320133120 8640266240 2 2.5338e+10 1.2669e+10

64 4194304 259 433000 2172649472 8690597888 4 2.00707e+10 5.01767e+09

$ ./roofline.exe 524288

kernel sz res\_bytes ntrials usecs ttl\_bytes ttl\_flops intensity flops/sec bytes/sec

2 524288 65539 606000 68722622464 8590327808 0.125 1.41755e+10 1.13404e+11

4 524288 32771 180000 34362884096 8590721024 0.25 4.77262e+10 1.90905e+11

8 524288 16387 143000 17183014912 8591507456 0.5 6.00805e+10 1.20161e+11

16 524288 8195 188000 8593080320 8593080320 1 4.57079e+10 4.57079e+10

32 524288 4099 270000 4298113024 8596226048 2 3.18379e+10 1.59189e+10

64 524288 2051 422000 2150629376 8602517504 4 2.03851e+10 5.09628e+09

$ ./roofline.exe 262288

kernel sz res\_bytes ntrials usecs ttl\_bytes ttl\_flops intensity flops/sec bytes/sec

2 262288 131003 601000 68721029728 8590128716 0.125 1.42931e+10 1.14344e+11

4 262288 65503 176000 34361301728 8590325432 0.25 4.88087e+10 1.95235e+11

8 262288 32753 140000 17181437728 8590718864 0.5 6.13623e+10 1.22725e+11

16 262288 16378 192000 8591505728 8591505728 1 4.47474e+10 4.47474e+10

32 262288 8190 272000 4296277440 8592554880 2 3.15903e+10 1.57951e+10

64 262288 4096 420000 2148663296 8594653184 4 2.04635e+10 5.11586e+09

./roofline.exe 16777216

kernel sz res\_bytes ntrials usecs ttl\_bytes ttl\_flops intensity flops/sec bytes/sec

2 16777216 2051 5.323e+06 68820140032 8602517504 0.125 1.6161e+09 1.29288e+10

4 16777216 1027 2.622e+06 34460401664 8615100416 0.25 3.2857e+09 1.31428e+10

8 16777216 515 1.371e+06 17280532480 8640266240 0.5 6.30216e+09 1.26043e+10

16 16777216 259 731000 8690597888 8690597888 1 1.18886e+10 1.18886e+10

32 16777216 131 400000 4395630592 8791261184 2 2.19782e+10 1.09891e+10

64 16777216 67 460000 2248146944 8992587776 4 1.95491e+10 4.88728e+09

roofline.exe was run for different problem sizes. At problem size around half L1 cache size, the largest number observed for performance is around 61 GBflops/s.

L1 bandwidth obtained is around 120 Gbytes/s and L1 bandwidth obtained is around 64 Gbytes/s. DRAM bandwidth comes out to be around 12 GBytes/s.

L1 bandwith is very close but L2 bandwidth is higher than measured by bandwidth.exe. DRAM bandwidth is also higher than what is measured by bandwidth.exe.

**Based on the clock speed of your CPU and its maximum Glop rate, what is the (potential) maximum number of \*double precision\* floating point operations**

**that can be done per clock cycle? (Hint: Glops / sec :math:`\div` GHz = flops / cycle.)**

**There are several hardware capabilities that can contribute to supporting more than one operation per cycle: fused multiply add (FMA) and AVX registers.**

**Assuming FMA contributes a factor of two, SSE contributes a factor of two, AVX/AVX2 contribute a factor of four, and AVX contributes a factor of eight of eight,**

**what is the expected maximum number of floating point operations your CPU could perform per cycle, based on the capabilities your CPU advertises**

**via cpuinfo (equiv. lscpu)? Would your answer change for single precision (would any of the previous assumptions change)?**

Maximum number of \*double precision\* floating point operations = 61/5.8 = 10.51 flops / cycle (using boost speed)

CPU has SSE, FMA and AVX support giving 2\*2\*4 = 16 flops/cycle.

For single precision, the registers would be able to pack more floats (2 times) giving 32 flops/cycle.

**Roofline: Docker**

----------------

**What is the maximum compute performance of your computer? (The horizontal line.) What are the L1, L2, and DRAM bandwidths? How do those bandwidths correspond to what was measured above?**

Maximum compute performance is 44.8 GFLOPS/s.

L1 bandwidth is 132.5 GB

DRAM bandwidth is 13.2 GB/s.

Roofline plot does not show L2 cache. As explained in piazza discussion, L2 cache is disabled in docker.

**mult**

----

**Referring to the figures about how data are stored in memory, what is it about the best performing pair of loops that is so advantageous?**

Sample output of a run of ./mmult.exe

N GF/s ijk GF/s ikj GF/s jik GF/s jki GF/s kij GF/s kji

8 4.59108 3.96502 4.36152 2.4923 4.59108 2.81389

16 3.67286 4.56107 3.50675 3.38759 4.77975 3.45467

32 2.85836 4.71927 2.72922 3.95677 4.87151 3.69837

64 2.21224 4.85036 2.16472 2.5195 4.79904 2.49181

128 1.62462 4.40874 1.57134 1.41369 4.39974 1.37667

256 0.9193 4.51037 0.843012 0.347496 4.52882 0.347224

Notice that ikj and kij orderings perform best. For both index j is the innermost loop. We know C(i,j) += A(i,k) \* B(k,j).

All matrices are stored in row major order.

For ikj and kij, the inner increment on j gives us row wise access to the data which improves locality in memory over other approaches.

**What will the data access pattern be when we are executing ``mult\_trans`` in i,j,k order? What data are accessed in each if the matrices at step (i,j,k) and what data are accessed at step (i, j, k+1)? Are these accesses advantageous in any way?**

Mult\_trans implements C = A \* B\_T. For ijk order, the inner statement C(i, j) += A(i, k) \* B(j, k); will go through rows of A and rows of B\_T

resulting in locality of reference as the matrics are row major and leading to good performance.

As step (i,j,k), C(i,j), A(i,k) and B(j,k) are accessed while at step (i,j, k+1), C(i,j), A(i,k+1) and B(j,k+1) are accessed. The accesses are advantageous because they are sequential rowise leading to spatial locality.

**Referring again to how data are stored in memory, explain why hoisting ``C(i,j)`` out of the inner loop is so beneficial in mult\_trans with the "ijk" loop ordering.**

In ijk ordering, the C(i,j) is an invariant in the innermost loop that only increments k. C(i,j) is a memory write and moving it outside this inner loop reduces the number of memory accesses. This allows us to accumulate results of A(i,k)\*B(j,k) and write once to C(i,j). This increses efficiency as memory accesses are very slow. The inner index k is accessing A and B rowise in mult\_trans which is beneficial because matrics are row major.

**(AMATH 583 ONLY) What optimization is applied in going from ``mult\_2`` to ``mult\_3``?**

mult\_2 implements 2x2 tiling, hoisting.

mult\_3 implements 32 sized blocks, 2x2 tiling and hoisting.

Blocking improves cache efficiency for larger matrices where contiguous blocks can be loaded in cache and operated upon.

**How does your maximum achieved performance for ``mult`` (any version) compare to what bandwidth and roofline predicted? Show your analysis.**

Maximum achieved performance for mult = 21.09 GFLOPS/s

Roofline (docker) showed maximum comp perf as 44.8 GFLOPS/s.

**loop ordering**

-------------

**Which variant of blurring function between struct of arrays and array of structs gives the better performance? Explain what about the data layout and access pattern would result in better performance.**

Sample run on julia.bmp:

SOA inner SOA outer AOS inner AOS outer Ten inner Ten outer

35 8 5 14 31 4

Several runs on jula.bmp were tried and nummbers more or less have the same pattern as above.

SOA outer performs best for struct of arrays. For SOA outer, index k is the outer index while i,j are iterated inside k. This gives us better locality because within each k plane, we can get the pixel information from i,j combinations and load them into the caches for faster processing at once. This is because of how SOA\_Image is organized - within each k we have a vector of i,j.

AOS inner performs best for array of structs. For AOS inner, index k is the inner index which is iterated inside i,j. This gives us better locality because within for each pixel defined by i,j we can get the color information k and load it into the cache to process faster. This helps because of how AOS\_Image is organized - within each i,j pixel we have a array of k of size 3.

**Which variant of the blurring function has the best performance overall?**

Explain, taking into account not only the data layout and access pattern but also the accessor function.

Tensor outer generally comes to be the best (AOS inner tends to be close). Tensor outer iterates with k on the outside and i,j combinations inside it. The Tensor\_Image is laid out such that K is the slowest that jumps nrows\_\*ncols\_. i is the second fastest that jumps i\*ncols\_ and j is the fasted changing index in the layout. Having K on the outside, follows this pattern. For each K, it loads up a chunk of i,j vector storage into cache and uses inner j to iterate over the inner vector storage which gives better locality and lesser cache misses as compared to Tensor inner implementation.