AMATH 583 PS4 – Ben Francis

(1) What level of SIMD/vector support does the CPU your computer provide?

CPUID_ECX_SSE3
CPUID_ECX_SSSE3
CPUID_ECX_SSE4A
CPUID_ECX_SSE41
CPUID_ECX_SSE42

CPUID_ECX_AVX
CPUID_EBX_AVX2

(2/3) What is the maximum operand size that your computer will support?

256 bits (it doesn't include AVX512), 4 doubles

(4) What is the clock speed of your CPU? (You may need to look this up via "About this Mac" or "lscpu").

2.3 GHz. It can burst to 3.6 GHz. (This will be important)

(5) 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.)

L1 cache: 32,768 bytes (32 KB)

Corresponding bandwidth: 195 GB/sec

L2 cache: 262,144 bytes (256 KB)

Corresponding bandwidth: 104 GB/sec

Both are approximately equal to what "about my mac" says (and what the L1 cache is supposed to be, not listed in about my mac but can inquire size from terminal)

(6) Is Docker question

(7) 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?

Max --> 53 GF/s. This number was determined by testing with both 16KB and 32KB.
L1 --> 210 GB/s
L2 --> 115 GB/s
RAM --> 38 GB/s

For L1 and L2, this is a little bit above the previous measurement, but close.

(8) 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: GFlops / sec :math:`\div` GHz = flops / cycle.)

53 GF/s / 3.6 Gcycles/s --> 14.7 Flops per cycle.

Note the 3.6 boost speed is used for these calculations, since the -OFast optimization is used. Otherwise, the calculation would be 53 / 2.3 = 23, which is less plausible for the reasons to be considered next.

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 AVX512 contributes a factor 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)?

2 for FMA (fuse multiply add), 4 for AVX2 --> 2x4 = 8.

However, empirically I am seeing ~14, which makes me think there is another factor of 2 not being considered.

This may be due to the compiler OFast optimization switching double to single, but I do not think this is the case.

More likely, I think this is due to the fact my Mac has 2 cores, which provides the missing factor of 2:

2 for FMA, 4 for AVX2, 2 cores --> 2x4x2 = 16.

14.7 is in this expected range.

Would your answer change for single precision (would any of the previous assumptions change)?

For single precision it would double, and should be in the range of 32, instead of 16.

(9) is Docker question

(10) 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?

For both of the best performing runs the inner most loop is j. The movements along the row major matrix storage are most efficient when moving to the "right", which is to say one column over. This is because the cache, acting in accordance with the principle of spatial locality, grabs up contiguous chunks of the row-major stored matrix data, which stores neighboring columns adjacently.

This can then be stored in nearby caches in cache lines/blocks, which helps according to the principle of spatial locality.

For i and k, there is either one or two indices that "jump" in the row major storage, which results in cache misses.

(11) 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 loops through k for A(i,k) and B(j,k); since i and j are fixed, and these are row indices, this loops through the columns of a row in the k index.

This is sequential for the row major matrix storage, so the matrix is read in continguous cache lines.

At k, this accesses an element of A and B, and at k+1 it accesses the next sequentially stored element, which as mentioned above is advantageous.

(12) 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.

This is described above; then for each i and j, k is looped through for A and B, such that A(i,k+1) and B(j,k+1) reads the next element in the row major storage.

Then subsequences of A and B (or all of A and B, depending on their size) can be stored in sequential cache lines, and read without cache misses.

If indices were being read that were not stored sequentially, then if A and B were larger than a cache this would result in cache misses, and so hurt performance.

Also (and this may be a more particular answer for why hoisting is beneficial), by hoisting we don't have to read and write to C(i,j) each time we do operations with A and B where we are iterating over the index k. We define an intermediate variable t, which can be stored in a local register, instead of working with C(i,j) each time, and then only at the end give C(i,j) the value we have accrued in t.

(13) What optimization is applied in going from ``mult_2`` to ``mult_3``?

mult_3 implements blocking (in addition to hoisting and tiling), while mult_2 only has tiling (and hoisting).

The difference is blocking.

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

The maximum observed GF/s is 22.1 GF/s.

First not the processing power is 2.3 GHz, with 3.6 GHz boost. Let us consider the boost to have been used, since earlier calculations indicate this was so, though the following argument can be readily modulated to handle 2.3 GHz.

Then 22.1/3.6 = ~6 Flops per cycle.

Yet from previous considerations, we know it should be able to handle 16 flops per cycle. Where does this loss come from?

Presumably, this comes from bandwidth problems. We are no longer doing a prepackaged read/write test, but rather working with real data. This includes moving matrix data in and out of various levels of cache. As the size of these problems approaches the size of different caches, this will result in cache misses, which will hurt the computational efficiency. Furthermore, the numerical intensity is not perfect; there are read/writes in addition to just flops, which will also hurt the numerical intensity. My current thoughts are that each fused multiply add works on three doubles, each of which takes 8 bytes of storage, which needs to be read in from some memory, either cache or RAM. This also needs to be written to another variable. In this cursory glance, we seeing 4 memory operations and only one floating point operation, which gives 0.25 numerical intensity. However, we observed this value in mult_t_3, which uses blocking, tiling and hoisting, and so further muddies the exact calculation to be performed. In summary, this fall from 16 possible flops per cycle to 6 flops per cycle is attributable to a less-than-1 numerical intensity and cache misses. It may just be numerical intensity limitations, as this number of 22.1 was observed with matrix size 32, which would only be ~900 bytes, which could readily fit in my mac's L1 cache. There are also other processes occurring, which may have also muddied the exact analytical consideration.

(15) 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.

For stuct of arrays, outer is better. This is because the entire "sheet" of pixels for each color is stored in a separate vector. Then for a given k, such as k=1, the entire ij plane (a vector) can be processed, and so loaded in closer caches. Then the next vector can be loaded in. If the k was in the inner loop, it would have to exchange change vectors every single iteration, which would incur harmful read/write costs.

For array of structs, inner should be better. Since each of the values corresponding to each k (the different colors for one pixel) are stored in a struct (here a C++ array), these will be continguously stored in memory. Then looping through them can be done quickly, since three contiguous elements can be stored in a nearby cache. However, this is less harmful than the difference between inner and outer for SOA, since the structs only have three elements in them.

(16) 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 is best.

The outer is important because the different k values divide tensor into thirds; if we loop through k on the outside, this amounts to moving down the tensor's vector storage

in one-third chunks.

Each movement in j is done with continguous elements in memory, so only +/- i changes are "jumps" (since row major storage and accessing). The size of these jumps will become more harmful when the matrix is larger, since they are proportional to the number of columns, as seen in columns, as seen in the accessor function. After a critical threshold, this will prevent the "i jumps" from being contained within certain sizes of caches, and so each i jump will result in cache misses.

For tensor inner, cycling betweens k's is fairly catastrophic, since this jumps by row*column size, which further reduces the probability that the next k value is in a nearby cache. This makes the frequency of cache misses proportional to tensor size, which naturallly will not scale well and lead to poor performance.

LOGS

mmult_1.log

N	GF/s ijk	GF/s ikj	GF/s jik	GF/s jki	GF/s kij	GF/s kji
8	2.81389	2.81389	3.00795	1.85597	2.4923	1.85597
16	2.09563	2.79137	2.47462	2.04646	2.16722	1.91715
32	1.85297	3.22456	2.132	2.23729	2.39709	2.23729
64	1.93394	2.97383	1.93394	1.92983	2.13919	1.97178
128	1.73861	3.24191	1.68559	0.689658	2.25039	0.711979
256	1.15403	3.34531	1.11735	0.4356	2.2973	0.43784

mmult_2.log

N	GF/s ijk	GF/s ikj	GF/s jik	GF/s jki	GF/s kij	GF/s kji
8	2.35758	2.81389	3.00795	1.85597	2.4923	1.85597
16	2.28054	2.83676	2.48343	2.04048	2.31074	2.04646
32	2.71288	3.20177	2.11212	2.22629	2.39077	2.22629
64	2.59148	2.93533	1.90151	1.91759	2.10445	1.93807
128	2.14515	3.10197	1.68034	0.676458	2.23638	0.679657

mmult_ps3.log

N	mult_0	mult_1	mult_2	mult_3	mul_t_0	mul_t_1	mul_t_2	mul_t_3
8	2.90768	3.6346	5.81536	3.6346	3.6346	3.96502	5.4519	7.93004
16	2.10829	2.46588	6.23075	9.69227	2.12111	2.51023	5.91393	14.2417
32	2.09261	2.665	6.81279	17.0962	2.34135	2.90417	6.56595	22.1
64	1.92165	2.56946	7.03115	19.2983	2.24509	2.69145	6.87135	21.5957
128	1.73861	2.1136	6.77947	18.4263	2.0281	2.22714	6.67453	19.9618
256	1.14154	1.36283	4.73204	17.5761	1.87996	1.94433	6.85632	18.9281
512	0.769925	1.03359	3.65444	15.3392	1.75431	1.7256	6.80366	16.5268
1024	0.341972	0.293814	1.39355	13.6204	1.48643	1.64815	6.02995	15.8132