Student Name: Rohan Tiwari
AMATH 583

PS4 Questions (Homegrown Approach)

==========

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?**

Computer has SSE and AVX2 support.

Pasting output of cpuinfo583.exe:

```
SUPPORTED FEATURES


CPUID_EAX_P4_HT
CPUID_EAX_PD
CPUID_EAX_CORE_DUO
CPUID_EAX_CORE2_DUO
CPUID_EAX_XEON_3K
CPUID_EAX_CORE2_DUO_8K
CPUID_EAX_NAHALEM
CPUID_EAX_IVYBRIDGE
CPUID_EAX_SKYLAKE


CPUID_EBX_AVX2


CPUID_ECX_SSE3
CPUID_ECX_SSSE3
CPUID_ECX_FMA
```

```
CPUID_ECX_SSE41
CPUID_ECX_SSE42
CPUID_ECX_AES
CPUID_ECX_XSAVE
CPUID_ECX_AVX
CPUID_ECX_F16C
CPUID_ECX_HYPERVISOR

CPUID_EDX_MMX
CPUID_EDX_SSE
CPUID_EDX_SSE2
CPUID_EDX_INTEL64
CPUID_EDX_XD
```

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

256-bit operands for AVX2.

128-bit operands for SIMD.

So, the CPU will support max operand size of 256 bit which means it can hold up to 4 doubles (each 8 bytes) or 8 floats (each 4 bytes) .

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

256-bit operands for AVX2.

128-bit operands for SIMD.

Minimum operand size would be 128-bit which means 2 doubles (each 8 bytes) and 4 floats (4 bytes).

4. **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 2.9 GHz and can go up to 4.2 GHz.

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.)**

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

Homegrown:

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

The sizes are similar to what task manager shows.

6. **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.)**

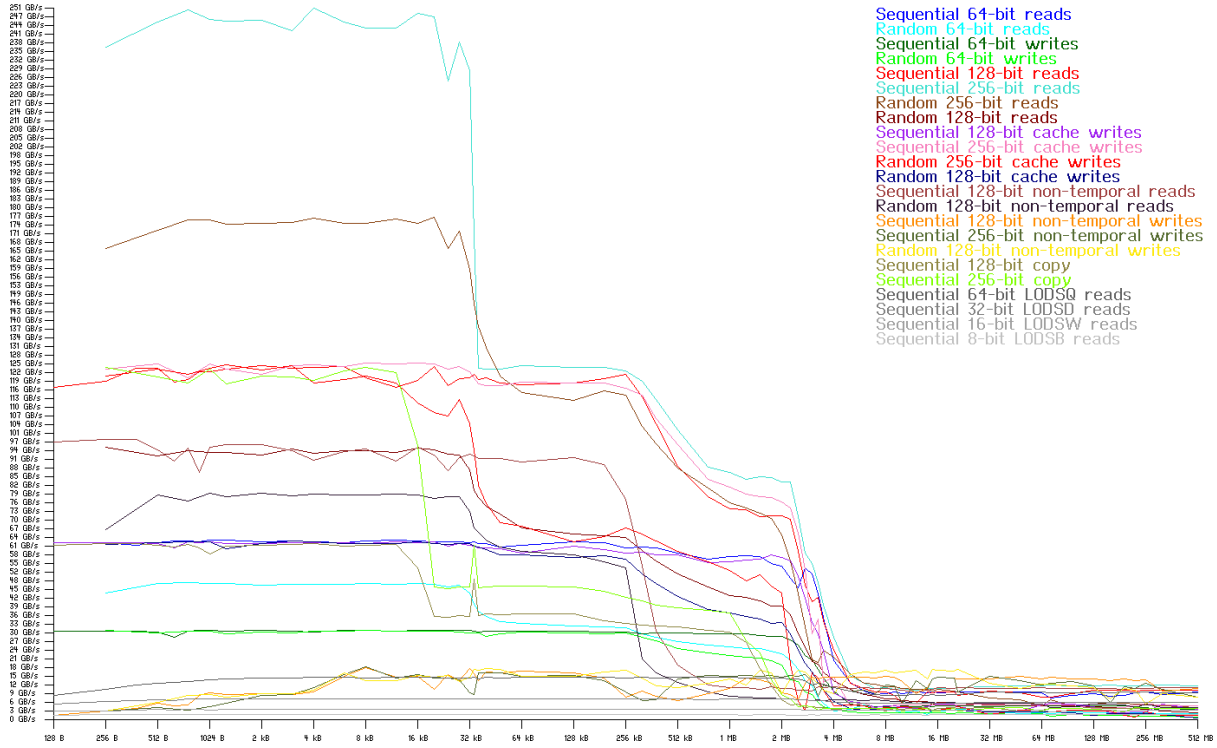

Figure 1 Bandwidth versus problem size

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. This is the first significant drop in bandwidth in Figure 1. This implies that the CPU is having to access data from farther locations like L2 cache.

L2 = 256KB which gives 4MB with 16 logical cores. This is the second significant drop in bandwidth in Figure 1. This also implies that the CPU is having to access data from farther location like L3 cache.

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

For sequential 256-bit reads, the bandwidth for L1 is around 250 GB/s and for L2 is around 128 GB/s.

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

L1 = 512KB

L2 = 4MB

The sizes multiplied by number of logical cores match what task manager shows.

Student Name: Rohan Tiwari
AMATH 583

Perf

Roofline: Homegrown

---

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?

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 L2 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 for homegrown approach. DRAM bandwidth is also higher than what is measured by bandwidth.exe for homegrown approach.

Given below are some sample runs of roofline.exe:

```
$ ./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
```

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: 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 = 61/4.2 = 14.52 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

---

9. 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?

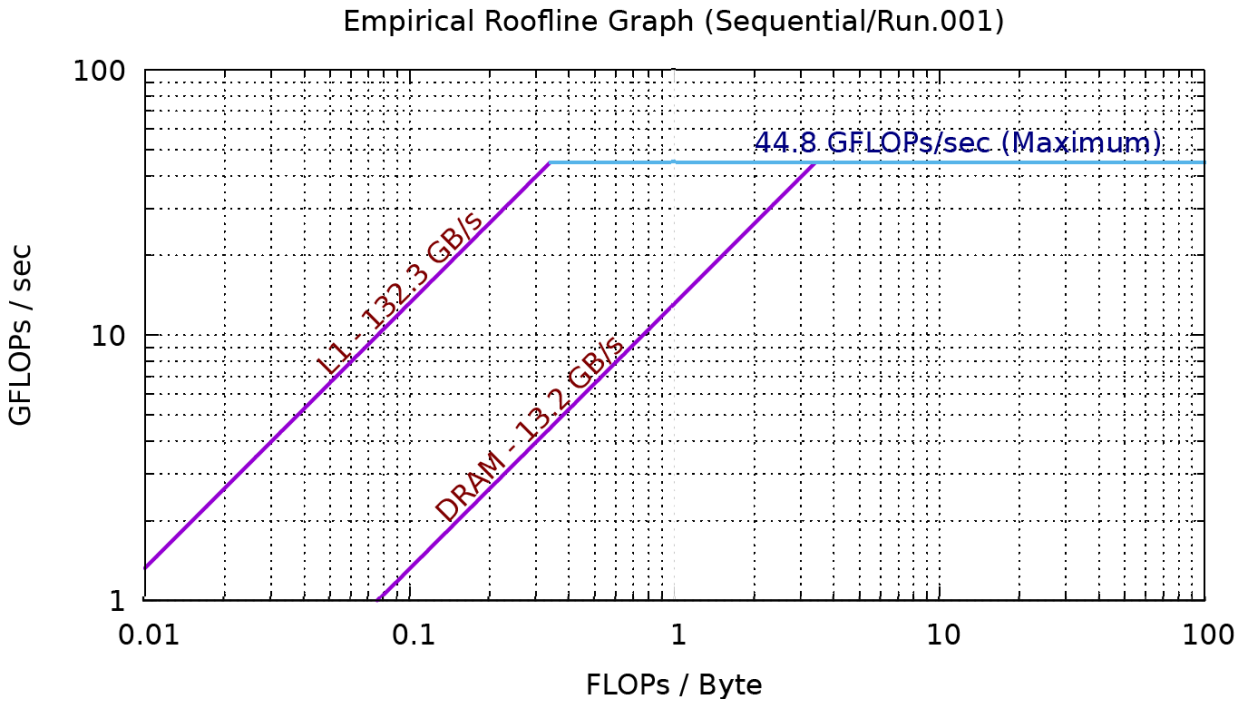

Figure 2 GFlops/s versus FLOPs/byte (roofline model)

Maximum compute performance is 44.8 GFLOPS/s.

L1 bandwidth is 132.5 GB/s.

DRAM bandwidth is 13.2 GB/s.

Both bandwidths are slightly higher than what was measured above.

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

mult

---

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?

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)` and all matrices are stored in row major order which means that elements in each row are stored next to each other and the most efficient way to access them is row wise.

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

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

To explain further, at step (i,j,k): C(i,j), A(i,k) and B(j,k) are accessed. After this, 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 row wise leading to spatial locality and faster data access.

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.

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 increases efficiency by reducing the number of memory accesses as memory accesses are very slow. The hoisting implementation writes to a register instead of C(i,j) which is relatively faster as registers are much more local to the CPU than memory. The inner index k is accessing A and B row wise in mult_trans which is beneficial because matrices are row major.

13. (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.

Mult_3 additionally implements blocking over mult_2.

Blocking improves cache efficiency for larger matrices where contiguous blocks can be loaded in cache and operated upon. It is possible that some experimentation is needed to determine the right block size for a given machine.

14. 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 = 29.2291 GFLOPS/s with a problem size of 32. Considering 4.2 GHz(boost speed), we get 29.2291/4.2=6.96 Flops/cycle. This is for a mult_3 implementation with (i,k,j) loop ordering. The problem size of 32 can easily fit in the L1 cache.

Let us look at the arithmetic intensity of this mult_3 implementation. This implementation's inner loop has 8 flops and accesses 8 doubles (8*8 = 32 bytes). This gives an arithmetic intensity of 8/32 = 0.125. We check the roofline (docker) model (L1 cache) and see that this corresponds to roughly 30 FLOPS/s which matches closely with the performance of mult_3. Please note that the additional processes / programs were shut down as much as possible while performing this analysis.

loop ordering

---

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.

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 julia.bmp were tried and numbers 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.

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 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 fastest 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.

Logs and Outputs

Deliverable 1: Save the output from a run of mmult_ps3.exe into a file 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	3.35502	3.6346	8.72305	6.71004	4.59108	3.23076	9.69227	9.69227
16	3.33897	3.43765	8.61535	7.75382	3.71193	3.38759	8.61535	16.6153
32	2.55961	2.69673	8.16307	8.09019	2.85836	2.67286	8.16307	22.1
64	2.21224	2.39319	6.66925	7.88712	2.25626	2.51252	7.25615	23.2569
128	1.66348	1.85851	6.53295	7.02239	1.89777	1.95633	6.82238	18.9112
256	0.940379	1.04561	5.6351	6.23829	1.73015	1.72074	6.45654	15.1684

Deliverable 2: Save the output from a run of mmuls with "_0" to indicate it is a run before any optimizations have been introduced.

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

Deliverable 3: Compile and run mult.exe with your modified version of mult_ijk. Save the output to a file mult_1.log

N	GF/s ijk	GF/s ikj	GF/s jik	GF/s jki	GF/s kij	GF/s kji
8	4.36152	3.96502	4.15383	2.4923	4.36152	2.81389
16	3.67286	4.59108	3.50675	3.37123	4.71516	3.27626
32	2.83156	4.71927	2.77095	4.17558	4.81968	3.33125
64	2.41872	4.72405	2.2015	2.52651	4.77378	2.52651
128	1.97063	4.39078	1.57939	1.37755	4.36411	1.34742
256	1.78452	4.5381	0.827885	0.35091	4.4649	0.347605

Deliverable 4: Compile and run mult_ps3.exe with your modified version of mult_3 with problem sizes up to 1024 (say). Save the output to a file mult_2.log

N	mult_0	mult_1	mult_2	mult_3	mul_t_0	mul_t_1	mul_t_2	mul_t_3
8	3.35502	3.48922	8.72305	14.5384	4.84614	3.35502	8.72305	10.9038
16	3.17202	3.32307	8.02119	20.5248	3.61577	3.33897	8.30766	15.8601
32	2.59628	2.61124	7.74445	**29.2291**	2.82274	2.61124	8.09019	23.8448
64	2.18559	2.37439	6.5726	27.4854	2.21224	2.46472	7.14188	20.6141
128	1.63077	1.81318	6.61311	25.3632	1.88946	1.95633	6.77947	17.8171
256	1.02385	1.23513	5.84325	22.8309	1.75067	1.78596	6.61072	16.1649
512	0.240609	0.338325	1.62093	21.5795	1.61945	1.64822	6.27808	14.801