Compiler Options and Code Optimizations

Objectives.

- To run programs using different compiler options and analyze their performance
- To reduce the algorithmic complexity of programs using code optimizations
- To use the roofline performance model in order to identify performance issues
- To Improve cache utilization and performance of programs using code optimizations

Aim

The **aim** of this lab session is to acquaint yourselves with code optimizations, cache efficiency and performance issues. We will be focusing on memory bound algorithms.

Leveraging on the compiler's options to improve performance

Task1: Download the '*Image_processing*' file. This is the main kernel of an image processing application. This routine has been written by an inexperienced developer and thus there is much room for improvement. Compile the code by using the following options and measure the execution time in each case. Keep in mind that this option is available in Visual Studio too.

- -Ofast (optimize very aggressively to the point of breaking standard compliance)
- '-O3' option
- '-O2' option
- '-O1' option
- '-O0' option (this is the default option)

Gcc provides about 200 optimizations, which can be found in https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html . Keep in mind that not all the optimizations aim on reducing the program's execution time, e.g., some optimizations relate to debugging and security.

As you might have seen in the link above (https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html), '-O3' includes all the optimizations included in '-O2' plus some additional ones. In rare cases, '-O3' might generate slower code than '-O2'. In general, a specific optimization might improve performance of a specific code block, on a specific processor and for a specific input size, but it might degrade performance for another input size or processor. Furthermore, the order that the optimizations are applied strongly impacts performance. '-O3' option applies a number of optimizations in a fixed order, but we could manually apply those in a different order and further improve performance. You can see which optimizations are enabled by the compiler by using the following command in the compilation *gcc source.c –o exec -O2 -Q --help=optimizers --help=params*

The problem of identifying which optimizations to apply, their parameters as well as their order, remains unsolved. So, **using the right optimizations is a matter of experience and expertise.**

Compiler Optimization options in Visual Studio: You can enable/disable the optimizations options by visiting the 'Project tab --> properties --> C/C++ --> Optimization'. Furthermore, in Visual Studio, you can use either debug mode or release mode. The first one applies no optimization, while the latter forces the compiler to generate the fastest binary possible.

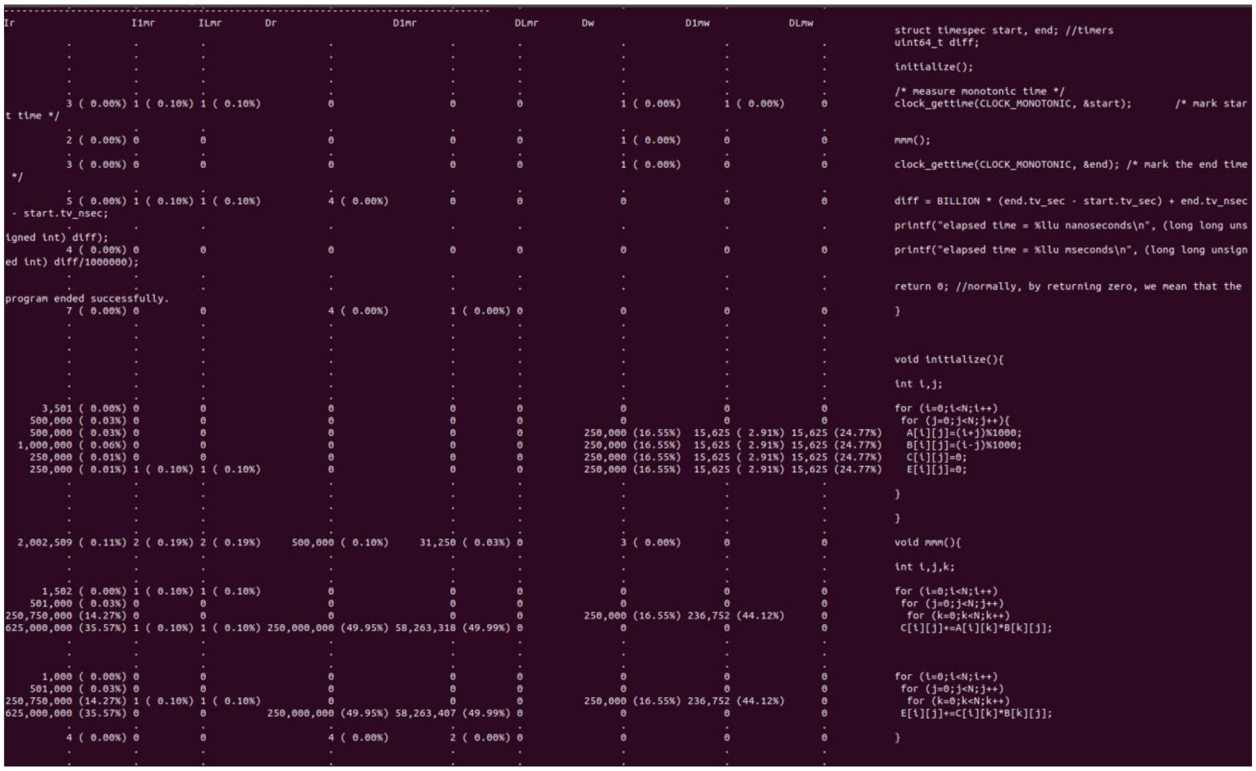
Task2: Compile the '*Image_processing*' file using '-O2' and any other optimization you like. Find experimentally a good optimization set. An example is shown below:

gcc image_processing.c -o p -O2 -funroll-loops -floop-unroll-and-jam

Improving Cache Utilization and Performance

Case study. Matrix-Matrix Multiplication (MMM)

Task1. Download and study 'mmm.c' file. Revisit the cache analysis we had for MMM previous week as well as the roofline model. 'mmm.c' is a simpler version of the 2mmm.c program we studied last week. 'mmm.c' contains just one MMM loop kernel, while 2mmm.c contains two MMM loop kernels. Fig.1 shows the dL1 statistics of 2mmm() program; the statistics of 'mmm.c' program are included in Fig.1, so there is no need for profiling the 'mmm.c' program again.



```
--------------------------------------------------------------------------------------------------------
Ir                I1mr      ILmr      Dr                 D1mr              DLmr      Dw               D1mw             DLmw
                                                                                                                                  struct timespec start, end; //timers
          .          .          .                  .                  .          .                .                .          uint64_t diff;
          .          .          .                  .                  .          .                .                .
          .          .          .                  .                  .          .                .                .          initialize();
          .          .          .                  .                  .          .                .                .
          .          .          .                  .                  .          .                .                .          /* measure monotonic time */
          3 ( 0.00%) 1 ( 0.10%) 1 ( 0.10%)         0                  0          0                1 ( 0.00%)       1 ( 0.00%)       0          clock_gettime(CLOCK_MONOTONIC, &start);       /* mark star
t time */
          .          .          .                  .                  .          .                .                .
          2 ( 0.00%) 0          0                  0                  0          0                1 ( 0.00%)       0                0          mmm();
          .          .          .                  .                  .          .                .                .
          3 ( 0.00%) 0          0                  0                  0          0                1 ( 0.00%)       0                0          clock_gettime(CLOCK_MONOTONIC, &end); /* mark the end time
 */
          .          .          .                  .                  .          .                .                .
          5 ( 0.00%) 1 ( 0.10%) 1 ( 0.10%)         4 ( 0.00%)         0          0                0                0                0          diff = BILLION * (end.tv_sec - start.tv_sec) + end.tv_nsec
 - start.tv_nsec;
          .          .          .                  .                  .          .                .                .          printf("elapsed time = %llu nanoseconds\n", (long long uns
igned int) diff);
          4 ( 0.00%) 0          0                  0                  0          0                0                0                0          printf("elapsed time = %llu mseconds\n", (long long unsign
ed int) diff/1000000);
          .          .          .                  .                  .          .                .                .
          .          .          .                  .                  .          .                .                .          return 0; //normally, by returning zero, we mean that the
program ended successfully.
          7 ( 0.00%) 0          0                  4 ( 0.00%)         1 ( 0.00%) 0                0                0                0          }
          .          .          .                  .                  .          .                .                .
          .          .          .                  .                  .          .                .                .
          .          .          .                  .                  .          .                .                .
          .          .          .                  .                  .          .                .                .          void initialize(){
          .          .          .                  .                  .          .                .                .
          .          .          .                  .                  .          .                .                .          int i,j;
          .          .          .                  .                  .          .                .                .
      3,501 ( 0.00%) 0          0                  0                  0          0                0                0                0          for (i=0;i<N;i++)
    500,000 ( 0.03%) 0          0                  0                  0          0                0                0                0           for (j=0;j<N;j++){
    500,000 ( 0.03%) 0          0                  0                  0          0      250,000 (16.55%)  15,625 ( 2.91%) 15,625 (24.77%)   A[i][j]=(i+j)%1000;
  1,000,000 ( 0.06%) 0          0                  0                  0          0      250,000 (16.55%)  15,625 ( 2.91%) 15,625 (24.77%)   B[i][j]=(i-j)%1000;
    250,000 ( 0.01%) 0          0                  0                  0          0      250,000 (16.55%)  15,625 ( 2.91%) 15,625 (24.77%)   C[i][j]=0;
    250,000 ( 0.01%) 1 ( 0.10%) 1 ( 0.10%)         0                  0          0      250,000 (16.55%)  15,625 ( 2.91%) 15,625 (24.77%)   E[i][j]=0;
          .          .          .                  .                  .          .                .                .
          .          .          .                  .                  .          .                .                .          }
          .          .          .                  .                  .          .                .                .
          .          .          .                  .                  .          .                .                .          }
  2,002,509 ( 0.11%) 2 ( 0.19%) 2 ( 0.19%)   500,000 ( 0.10%)    31,250 ( 0.03%) 0                3 ( 0.00%)       0                0          void mmm(){
          .          .          .                  .                  .          .                .                .
          .          .          .                  .                  .          .                .                .          int i,j,k;
          .          .          .                  .                  .          .                .                .
      1,502 ( 0.00%) 1 ( 0.10%) 1 ( 0.10%)         0                  0          0                0                0                0          for (i=0;i<N;i++)
    501,000 ( 0.03%) 0          0                  0                  0          0                0                0                0           for (j=0;j<N;j++)
250,750,000 (14.27%) 0          0                  0                  0          0      250,000 (16.55%) 236,752 (44.12%)       0            for (k=0;k<N;k++)
625,000,000 (35.57%) 1 ( 0.10%) 1 ( 0.10%) 250,000,000 (49.95%) 58,263,318 (49.99%) 0                0                0                0           C[i][j]+=A[i][k]*B[k][j];
          .          .          .                  .                  .          .                .                .
          .          .          .                  .                  .          .                .                .
          .          .          .                  .                  .          .                .                .
      1,000 ( 0.00%) 0          0                  0                  0          0                0                0                0          for (i=0;i<N;i++)
    501,000 ( 0.03%) 0          0                  0                  0          0                0                0                0           for (j=0;j<N;j++)
250,750,000 (14.27%) 1 ( 0.10%) 1 ( 0.10%)         0                  0          0      250,000 (16.55%) 236,752 (44.12%)       0            for (k=0;k<N;k++)
625,000,000 (35.57%) 0          0          250,000,000 (49.95%) 58,263,407 (49.99%) 0                0                0                0           E[i][j]+=C[i][k]*B[k][j];
          .          .          .                  .                  .          .                .                .
          4 ( 0.00%) 0          0                  4 ( 0.00%)         2 ( 0.00%) 0                0                0                0          }
          .          .          .                  .                  .          .                .                .
```

Fig.1. Valgrind Output for 2mmm.c file

Roofline Model: The roofline model [1] provides an easy way to get performance bounds for compute-bound and memory-bound loop kernels. It allows us to know how far the achieved performance is from the optimum. It is based on the concept of computational intensity, sometimes also called arithmetic or operational Intensity. The arithmetic intensity is given by the following formula: '*FP.arithmetical.instructions / number.of.bytes.loaded.stored*'. This model has several limitations [1], e.g., does not consider all features of modern processors and ignores integer computations. Note that an extension of the roofline model already exists supporting integer operations too. This can be found in the intel Vtune Advisor tool.

Roofline Model for MMM algorithm: MMM has N^3 iterations and each iteration contains 4 Floating Point (FP) L/S operations and 2 FP arithmetical operations. So, the arithmetical intensity of MMM (the ratio between FP arithmetical operations and number of bytes loaded/stored), when the arrays are of type 'float', is 2/(4*4bytes)=1/8.

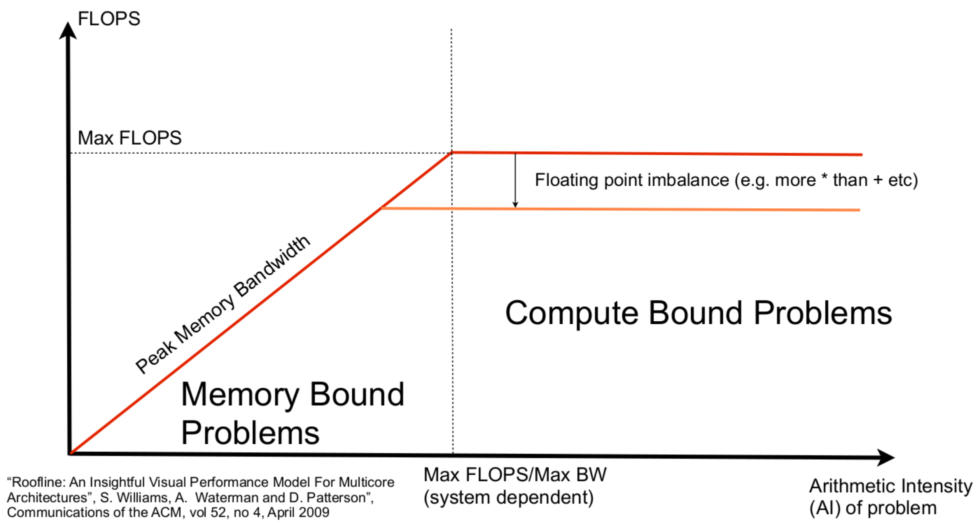

FLOPS
Max FLOPS
Floating point imbalance (e.g. more * than + etc)
Peak Memory Bandwidth
Compute Bound Problems
Memory Bound Problems
Max FLOPS/Max BW (system dependent)
Arithmetic Intensity (AI) of problem
"Roofline: An Insightful Visual Performance Model For Multicore Architectures", S. Williams, A. Waterman and D. Patterson", Communications of the ACM, vol 52, no 4, April 2009

Fig.2. The Roofline Model [1]

Algorithms that have a low arithmetical intensity are **memory-bounded,** while algorithms that have a high arithmetical intensity are **compute-bounded**. Memory-bound means that their performance is bounded on the memory latency and bandwidth values; in simple words, no matter how fast the CPU is, it stays idle waiting for the data to be loaded/stored from/to memory. The maximum performance we can get is given by the following formula

Attainable GFLOPS = min(Peak Floating Point Performance, Peak Memory Bandwidth x Arithmetical Intensity)

The peak FP performance is the maximum we can get, and is CPU dependent. The peak memory bandwidth depends on the DDR and memory controller hardware characteristics. Furthermore, if the data fit in a cache memory and they are mostly accessed from there, the peak memory bandwidth is the cache bandwidth, not the DDR bandwidth. So, if the peak DDR bandwidth is 21GBytes/sec (this value can be extracted from the memory specifications), then the maximum MMM performance will be 21GB/sec x (1/8)=2.65gigaflops. If the arrays fit in the precious cache memories, then the memory bandwidth is higher and thus performance is increased.

How can we improve the performance of memory-bounded algorithms? By applying code optimizations that reduce the number of memory accesses through the whole memory hierarchy; this relates to reducing the number of cache-misses too. Another strategy is to use software prefetching. The above can be achieved by using code optimizations such as loop tiling, register blocking, array copying, loop merge/distribution etc.

Task2. Measure the GFLOPS achieved on your CPU for the mmm() routine. Use different input sizes. Make a graph where the y-axis is FLOPs and x-axis is different input sizes. Use the '-O2' compiler option. You will realize that as the input size increases, performance is reduced. This is because the arrays can no longer fit either in the dL1 or L2 of L3 cache.

Task3. Repeat Task2 using '-O3' compiler option. The performance achieved is higher as auto-vectorization is applied as well as loop optimizations such as loop tiling.

Task4. Run 'flops.c' program; you will see printed the number of flops achieved. Why GFLOPS value of this program is that high, comparing to MMM? Recall the roofline model to answer this. Note that Flops() routine is written using AVX x86-64 intrinsics. Although these intrinsics will be further explained next week, in this module, we will not be focusing on SSE/AVX programming, so remain calm. Flops() routine is just a code example I wrote to get very high CPU performance.

You can develop a loop kernel of your choice (if you want) which uses a high arithmetic intensity value and measure the FLOPs value being achieved.

I have applied the procedure of task2, task3 and task4 on my PC and the results are shown in Fig.3. 'flops.c' program has an arithmetic intensity value of 2, while MMM a value of 1/8. **This means that flops() routine is compute-bound while MMM is memory-bound.** Furthermore, flops() routine has an excellent balance between add and multiply instructions, allowing us to use fused add/multiply instructions. All the fmadd instructions feed the pipeline one after another maximizing performance. To sum up, flops() routine gets the most out of our CPU performance (81 GigaFLOPS) while MMM is far away from it (we can improve its performance though).

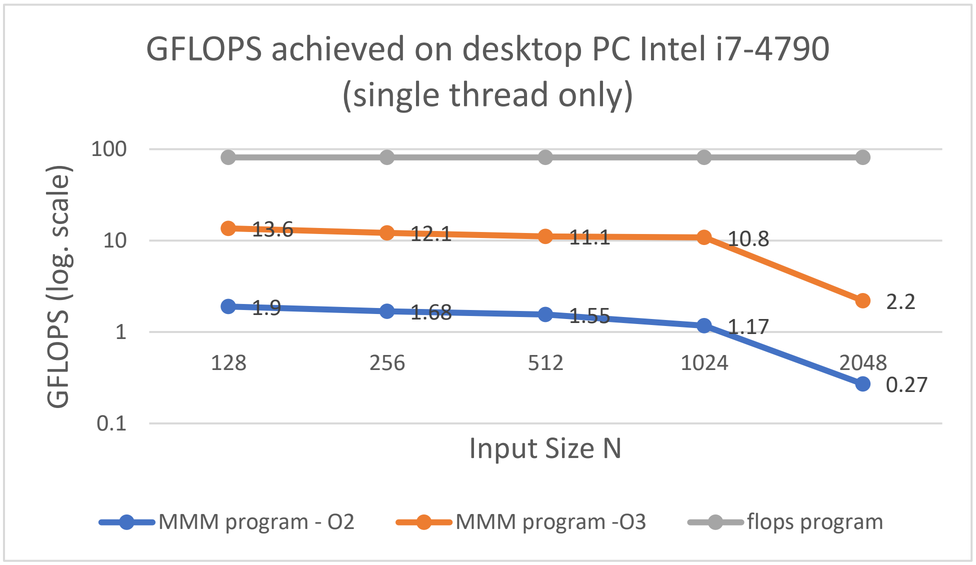


Fig.3. Results from Task2, Task3 and Task4

What is the Peak FLOPs value on your PC? The maximum FLOPs value is system dependent; for x86-64 architectures it is given by the following formula (single precision computations are assumed):

Peak.FLOPs = num.CPU.cores x CPU.freq. x simd.length.in.bits/32 x num.FMA.units x 2 *(Eq.1)*

Where *num.CPU.cores* is the number of CPU cores, CPU.freq. is the CPU's frequency, simd.length.in.bits is the length of the vector instructions (e.g., AVX technology uses 256 bits) and num.FMA.units is the number of Fuzzed-Multiply-Accumulate (FMA) units supported. The number of FMA units is normally either one or two, meaning that one or two SSE/AVX instructions can be executed in parallel. The last term of Eq.1 is two because a FMA instruction can execute 2 instructions (a multiply and an addition, together). Eq.1 will become clearer next week.

Register Blocking as a solution to memory-bound algorithms - Reduce the number of memory accesses

Task5: Apply register blocking transformation to mmm() and check the a) the performance gain (using the timer), b) the total number of memory accesses ('Dr' and 'Dw' columns), c) the gain in FLOPS. Use 'O2' option, not 'O3'. Apply register blocking only to j loop with a factor of 4, 8 and 12. Use '-O2' option. Compare the results you found with the results prior to register blocking. Pay attention to the 'Dr' (data reads) column. It is important to note that the register blocking factor must perfectly divide N. Make sure your code generates the right output before you use valgrind. To this end, you could perhaps write another function that compares the results.

Below the code is shown with an unroll factor of 4

```
for (i=0; i<N; i++)
 for (j=0; j<N; j+=4){
  c0=C[i][j];
  c1=C[i][j+1];
  c2=C[i][j+2];
  c3=C[i][j+3];
  for (k=0; k<N; k++) {
   a=A[i][k];
   c0+=a*B[k][j];
   c1+=a*B[k][j+1];
   c2+=a*B[k][j+2];
   c3+=a*B[k][j+3];
}
C[i][j]=c0;
C[i][j+1]=c1;
C[i][j+2]=c2;
C[i][j+3]=c3;
}
```

Array copying as a solution - Reduce the number of memory accesses

In the previous lab session, you have seen that accessing B[][] array in a column-wise order is not efficient; if you are still not sure about the reason, revisit the notes of the previous lab session. We can improve the cache performance by applying array copying transformation. The new code will be the following

```
for (i=0;i<N;i++)
 for (j=0;j<N;j++)
Btranspose[i][j]=B[j][i];

for (i=0;i<N;i++)
 for (j=0;j<N;j++)
  for (k=0;k<N;k++)
 C[i][j]+=A[i][k]*Btranspose[j][k];
```

An extra loop kernel is added copying the columns of B[][] in Btranspose[][]. This loop kernel adds extra overhead to the program but as you will find out by measuring its execution time, it pays off. In the new version of the program a row of A is multiplied by many rows of Btranspose (not columns).

Task6: Apply the array copying optimization and compare this code with mmm() in terms of a) execution time, b) dL1 accesses and misses, c) FLOPS. Compare the results.

Loop tiling as a solution - Reduce the number of memory accesses

Run the '*mmm_tiling_bad()*' routine and measure the performance for different tile sizes. The tile size must perfectly divide N in this implementation. In mmm_tiling_bad() routine, loop tiling has been applied to all the three loops. The tiles are square of size TILExTILE. Which tile size is the most efficient? It is important to note that in this case loop tiling has not been applied in an efficient way and there is much room for improvement; the efficient application of loop tiling is an advanced topic and will not be discussed here. For those who want to learn more, you can read the following article [3]; however, this is out of the scope of this lab session. Furthermore, loop tiling can be applied together with array copying and register blocking and thus highly improve performance.

(advanced topic) The main reason that the application of loop tiling is not efficient here is because the tiles' elements are not written in consecutive main memory locations (all the tiles' sub-rows are stored into no consecutive memory locations). Thus, although the aggregated tiles size is smaller than the cache, the tiles sub-rows conflict with each other and they cannot remain in the cache. Still, even using loop tiling in an inefficient way, performance is improved as some of the sub-rows remain in the cache.

Software prefetching as a solution (optional)

This paragraph is for your information only as it refers to more advanced topics. Next week, we will learn how to use SSE/AVX x86-64 intrinsics. These include prefetch instructions. All the prefetch instructions supported for x86-64 architectures can be found here https://software.intel.com/sites/landingpage/IntrinsicsGuide/#expand=173,5533,3505,1449,3505,2940,2024&text=prefetch. An example of a software prefetch instruction is shown below

_mm_prefetch(&C[i][j], _MM_HINT_T0);

The instruction above pre-fetches the cache line containing C[i][j] from DDR. No value is written back to a register, and we do not have to wait for the instruction to be completed. The cache line is loaded in the background.

Reducing the algorithmic complexity (optional)

Task3 (optional): '*inefficient_routine()*' is a routine written in an inefficient way. Try to simplify the code (reduce its complexity), firstly, by eliminating redundant operations and secondly, by applying as many optimization techniques you have been taught so far (have a look at the lectures).

Some of the optimizations you can apply are:

- the loop kernel in line 64 can be simplified
- the if condition in line 69 can be eliminated
- gauss_x_compute[][][] and gauss_xy_compute[][][] arrays are redundant and thus can be eliminated
- function inline

References and Further Reading:

1. Samuel Williams, Andrew Waterman, and David Patterson. 2009. Roofline: an insightful visual performance model for multicore architectures. Commun. ACM 52, 4 (April 2009), 65-76. DOI=10.1145/1498765.1498785, available at https://people.eecs.berkeley.edu/~kubitron/cs252/handouts/papers/RooflineVyNoYellow.pdf
2. Options That Control Optimization, available at https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html
3. V.I. Kelefouras, A. Kritikakou, I. Mporas and V. Kolonias, "A high performance Matrix-Matrix Multiplication methodology for CPU and GPU architectures", Journal of Supercomputing, Springer, available at https://link.springer.com/article/10.1007/s11227-015-1613-7
4. Optimizing compilers for modern architectures: a dependence-based approach, book, available at https://liveplymouthac-my.sharepoint.com/:b:/g/personal/vasilios_kelefouras_plymouth_ac_uk/EVy4Laj_1W9Hr7D3W57CBuQBeohd0M9iVVT7x5n91PcDyg?e=RGnRCa