Masterpraktikum Scientific Computing

High Performance Computing Solution Sheet

Session 1: Vectorization

Authors

Mantosh Kumar (Matriculation number : 03662915)
Chen-Yin Wu (Matriculation number : 03673331)
João Trindade (Matriculation number : 03673251)

Course of Study: Master of Science Informatics

Exercise 1: “Auto-vectorization”

Reference: https://software.intel.com/en-us/node/522569

Q1: Which kinds of loops can be vectorized automatically?

- Loops need to be countable (the trip count must be know at the start. loop exit shouldn’t be data-dependant)
- Single Entry - Single Exit (same as above. no data-dependant exits)
- Contain straight-line code (no flow-control structures - if, switch) (some if’s are allowed - masked assignments - https://software.intel.com/en-us/forums/intel-c-compiler/topic/563952)
- Innermost loop of a nest
- without function calls (printfs is sufficient to prevent a loop from getting vectorized. some intrinsic math functions are allowed sin(), log(), fmax(), etc)

Q2: Which datatypes and operations are allowed in order to enable auto-vectorization of loops?

Vector data only.
Prefer Structure of Arrays instead Arrays of Structures. RRRBBBGGG is better than RGBRGBRGB. A structure with 3 arrays works better than an array of structure with 3 values.
Supported operations include addition, subtraction, multiplication and division. Other mathematical operation are also software-supported by a run-time library.

Q3: Which types of dependency analysis do the compiler perform?

Data Dependencies:

- Flow-Dependency: when a variable is written in one iteration and read in a subsequent one. (eg. A[i] = A[i-1] + 1. in this case A[1] has to wait for A[0]. not vectorizable)
- Write-After-Read Dependency: when a variable is read in one iteration and written in a subsequent one (eg. A[j-1] = A[j] + 1. not safe because iteration with write may execute before the iteration with the read)
- Read-after-Read: are not dependencies. fine.
- Write-after-Write: are unsafe for the same reasons as first 2
- Potential Dependency (eg. c[i] = a[i] * b[i] . in this case the compiler needs to see if for every i, c[i] and a[i] don’t ever refer to the same memory location. otherwise we have read-after-write)

Q4: How does programming style influence auto-vectorization?

"The style in which you write source code can inhibit vectorization"
"Many stylistic issues that prevent automatic vectorization by compilers are found in loop structures. The ambiguity arises from the complexity of the keywords, operators, data references, pointer arithmetic, and memory operations within the loop bodies."
Some things that affect auto-vectorization:

- Complexity of "for loops”
- Use of flow-constructs (switch, goot, unmasked ifs)
- Use of array notation instead of pointers

Q5: Is there a way to assist the compiler through language extensions? If yes, please give details!

Yes, by using the SIMD Pragma we can give hints to guide the compiler on how it should proceed to vectorize the instructions.

- Vector declaration: __declspec(vector) (Windows*) or __attribute__((vector)) (Linux*). This indicates the compiler that a vector should be vectorized, despite the fact that a not intrinsic function is being called on that vector
- #pragma ivdep: Instructs the compiler to ignore assumed vector dependencies
- #pragma simd: Enforces vectorization of loops

Q6: Which loop optimizations are performed by the compiler in order to vectorize and pipeline loops?

Within HLO, loop transformation techniques include:

- Loop Permutation or Interchange
- Loop Distribution
- Loop Fusion
- Loop Unrolling
- Data Prefetching
- Scalar Replacement
- Unroll and Jam
- Loop Blocking or Tiling
- Partial-Sum Optimization
- Predicate Optimization
- Loop Reversal
- Profile-Guided Loop Unrolling
- Loop Peeling
- Data Transformation: Malloc Combining and Memset Combining
- Loop Rerolling
- Memset and Memcpy Recognition
- Statement Sinking for Creating Perfect Loop Nests

Exercise 2: "Dependency Checks and Vectorization"

Please provide your personal opinion w.r.t. vectorization for each function!
Which reports does the compiler provide? Which loops does the compiler vectorize
Please re-write functions/loops and use compiler-directives in order to vectorize additional loops!
Hand-in the different compiler outputs (for compilations of `vector.c` with and without manipulations)

Function number - Analysis

1. **Function name** : bsp1_
   Personal opinion: The condition within the loop is masked assignment, so this loop may be vectorized.
   Compiler report:
   LOOP WAS VECTORIZED.
   scalar loop cost: 35
   vector loop cost: 10.000
   estimated potential speedup: 3.130
   Estimate of max trip count of loop=12

   There is only one loop inside this function which gets vectorized.
   Compiler report after changes:
   In this case, technically vectorization is possible, that is why to ensure compiler vectorize the loop, we included vectorization hint (#pragma vector always). Also there is no dependencies, to make a subtle hint to compiler, we included (#pragma ivdep)
   LOOP WAS VECTORIZED
   scalar loop cost: 35
   vector loop cost: 10.000
   estimated potential speedup: 3.160
   Estimate of max trip count of loop=12

2. **Function name**: bsp2_
   Personal opinion: Though Intel compiler documentation "Guidelines for Writing Vectorizable Code" clearly states that goto statements prevent vectorization as it could become the cause to force premature exit from loops and thus should be avoided. But the source code inside the loop is not very complex and if we look at it carefully it is a simple implementation of any canonical loop such as while, for. Control flow is pretty simple and the code lines after goto statement level is not performing any conditional check and is following single entry single exit rule. So it is safe to deduce that it can not force premature exit and thus this simulated loop should be vectorized.
   Compiler report:
   LOOP WAS VECTORIZED
   scalar loop cost: 19
   vector loop cost: 7.500
   estimated potential speedup: 2.320
   Estimate of max trip count of loop=12

   GOTO is implemented as a canonical loop and this gets vectorized.
   Compiler report after changes:
   No changes. Same result as above.

3. **Function name**: bsp3_

Personal opinion: Data can be vectorized as it seems there will not be any data dependencies.
Compiler report:

LOOP WAS VECTORIZED
scalar loop cost: 8
vector loop cost: 2.000
estimated potential speedup: 3.600
Estimate of max trip count of loop=50

There is only one loop inside this function which gets vectorized.

Compiler report after changes:

No changes. Same result as above.

4. **Function name**: bsp4_

Personal opinion: Data can be vectorized as long as l is -2 or smaller.
Compiler report:

LOOP WAS VECTORIZED
scalar loop cost: 8
vector loop cost: 1.500
estimated potential speedup: 4.620
Estimate of max trip count of loop=50

There is only one loop inside this function which gets vectorized.

Compiler report after changes:

No changes. Same result as above.

5. **Function name**: bsp5_

Personal opinion: The data can be vectorized since they are write after read. However, if we modified l = 2 and m =1 inside main function, then there will be read-after-write dependency, and it can't be vectorized.
Compiler report:

LOOP WAS VECTORIZED
scalar loop cost: 13
vector loop cost: 3.000
estimated potential speedup: 3.670
Estimate of max trip count of loop=12

There is only one loop inside this function which gets vectorized.

Compiler report after changes:

No changes. Same result as above.

6. **Function name**: bsp6_

Personal opinion: The two instructions are dependent. The compiler splits the loop into two. "Loop Distributed (2 way)"
Compiler report:

PARTIAL LOOP WAS VECTORIZED
scalar loop cost: 7
vector loop cost: 1.500
estimated potential speedup: 4.120
Estimate of max trip count of loop=49

There is only one loop in this function which is partially vectorized.

Compiler report after changes:

No changes. Same result as above.

7. **Function name**: bsp7_

Personal opinion: There's no data dependency, the loop can be vectorized. The compiler doesn't split the loop into two since the first statement is redundant and compiler would just ignore the first statement.

Compiler report:

LOOP WAS VECTORIZED
scalar loop cost: 11
vector loop cost: 3.500
estimated potential speedup: 2.770
Estimate of max trip count of loop=12

There is only one loop inside this function which gets vectorized.

Compiler report after changes:

No changes. Same result as above.

8. **Function name**: bsp8_

Personal opinion: "Loop multiversioned for Data Dependence" - the compiler does not know if the x vector, and e vector overlap, so it creates two loops and uses a runtime test to decide whether it is safe to vectorize the loop. Can be improved by using **#pragma ivdep** or **#pragma omp simd.** Source:https://software.intel.com/en-us/articles/getting-the-most-out-of-your-intel-compiler-with-the-new-optimization-reports

Compiler report:

LOOP WAS VECTORIZED
scalar loop cost: 30
vector loop cost: 14.000
estimated potential speedup: 2.010
Estimate of max trip count of loop=12

There is only one loop inside this function which compiler splits in two loop and then decides at runtime about vectorization.

Compiler report after changes:

No changes. Same result as above.

9. **Function name**: bsp9_

Personal opinion: Loop not vectorized - Dependency of y[ix[i]]

Compiler report:

loop was not vectorized: vector dependence prevents vectorization.

Compiler report after changes:

No changes as nothing could yield effective vectorization in case of vector dependence. Same result as above.

10. **Function name**: bsp10_

Personal opinion: The compiler only auto-vectorize a loop if its internal heuristics indicate that a speed-up is likely. But because of low iteration count it seems compiler might decide that vectorization might not likely going to be beneficial. But inside this function it is evident that there is no data dependency and it is safe. To make sure that the loop gets vectorized, we must override the compiler cost model by inserting the directive #pragma simd before the loop, as a hint to the compiler.

Compiler report:

loop was not vectorized: vectorization possible but seems inefficient.
scalar loop cost: 109
vector loop cost: 115.000
estimated potential speedup: 0.940

No loop gets vectorized inside this function.

Compiler report after changes:

Included vectorization hint (#pragma simd).
SIMD LOOP WAS VECTORIZED
scalar loop cost: 109
vector loop cost: 114.000
estimated potential speedup: 0.950
Estimate of max trip count of loop=12

11. **Function name**: bsp11_
    Personal opinion: By default the innermost loop should be vectorized and also this loop does the heavy operation(divide), so it could be advised that the vectorization of this loop is more important than outer loop. No changes need to be made.
    Compiler report:

    Outer loop was not vectorized: inner loop was already vectorized.
    scalar loop cost: 30
    vector loop cost: 19.000
    estimated potential speedup: 1.560
    Estimate of max trip count of loop=12

    Inner loop gets vectorized inside this function.

    Compiler report after changes:

    No changes. Same result as above.

12. **Function name**: bsp12_
    Personal opinion: The inner loop is not accessing memory with unit stride and thus is very expensive in comparison with outer loop. By default inner loop should be getting vectorized by compiler. Also if outer loop will get vectorized then there is a grave danger of allowing data dependencies especially Read After Write and Write After Write. There is no need to give any hint to compiler.
    Compiler report:

    PERMUTED LOOP WAS VECTORIZED
    scalar loop cost: 6
    vector loop cost: 3.000
    estimated potential speedup: 1.990
    Estimate of max trip count of loop=1237

    Mostly inner loop inside this function gets vectorized.

    Compiler report after changes:

    No changes. Same result as above.

13. **Function name**: bsp13_
    Personal opinion: The two innermost loops can be fused together effectively as they get executed for the same number of iteration for each iteration of outermost loop. So this fused loop will become innermost loop and should be vectorized. Outer loop should not be compelled to get vectorized as in that case it will lead to

data dependency Read After Write. There is no need to give hint to compiler by programmer, compiler would be able to vectorize it on its own.

Compiler report:

FUSED LOOP WAS VECTORIZED
scalar loop cost: 18
vector loop cost: 6.500
estimated potential speedup: 2.510
Estimate of max trip count of loop=12

Two innermost loops will be fused together and will be vectorized.

Compiler report after changes:

No changes. Same result as above.

14. **Function name**: bsp14_

Personal opinion: Only the inner loops can be vectorized. They can probably be fused to use the same index. After testing, the compiler reports that data is unaligned. setting **__declspec(align(64))** fixes the issue. Also, the compiler does fuse both inner loops, and then distributes the work by 2 loops. fuse-loop then fission-loop.it's an interesting behavior but seems not to be out of the ordinary (https://software.intel.com/en-us/forums/intel-fortran-compiler-for-linux-and-mac-os-x/topic/437882)

Compiler report:

LOOP WAS VECTORIZED
scalar loop cost: 8
vector loop cost: 2.500
estimated potential speedup: 2.770
Estimate of max trip count of loop=12

Outer loop was not vectorized: inner loops which are fused together were already vectorized.

Compiler report after changes:

No changes. Same result as above.

15. **Function name**: bsp15_

Personal opinion: Loop order is changed. Probably because of the Row-Major Order Memory allocation. Loops are collapsed only in case 1. But in both cases, the loops are vectorized. Possible improvements might include changing the loop orders manually.

Compiler report:

PERMUTED LOOP WAS VECTORIZED
scalar loop cost: 9
vector loop cost: 4.500
estimated potential speedup: 1.990
Estimate of max trip count of loop=1250

Based on the order of the loop, the innermost loop gets selected and is vectorized.

Compiler report after changes:

No changes. Same result as above.

16. **Function name**: bsp16_

Personal opinion: Data is not vectorized because of possible dependency and indirect addressing. However, since the content of ix is every integer from 100 to 1. This means that the whole vector will be re-written. The first line can be removed without changing the output. Forcing the vectorization of the two lines may lead to inconsistent behaviour/unpredictable results.

Compiler report:

loop was not vectorized: vector dependence prevents vectorization.

Compiler report after changes:

No changes. Same result as above.

17. **Function name**: bsp17_

Personal opinion: s variable is shared but the compiler should identify this “reduction idiom” and vectorize it safely. Data will be vectorized, there is no room for improvements.

Compiler report:

LOOP WAS VECTORIZED
scalar loop cost: 7
vector loop cost: 2.500
estimated potential speedup: 2.350
Estimate of max trip count of loop=6

There is only one loop inside this function which is vectorized.

Compiler report after changes:

No changes. Same result as above.

18. **Function name**: bsp18_

Personal opinion: Loop can not be vectorized as it has multiple exits. To make it vectorize we need to rewrite the loop in such a way that it follows single entry - single exit rule.

Compiler report:

loop was not vectorized: loop with multiple exits cannot be vectorized unless it meets search loop idiom criteria.

Compiler report after changes:

No changes. There is no way this function could be rewritten in such a way that it satisfies single entry- single exit rule. Same result as above.

19. **Function name**: bsp19_

Personal opinion: Loop can not be vectorized as it has multiple exits. To make it vectorize we need to rewrite the loop in such a way that it follows single entry - single exit rule. There is no way we could rewrite in that way as another exit is being checked first before doing anything.

Compiler report:

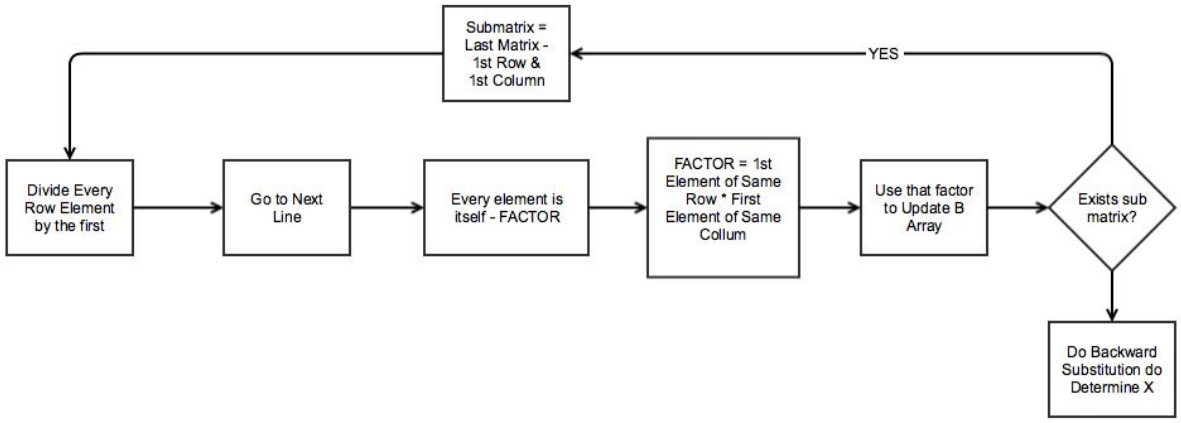
loop was not vectorized: loop with multiple exits cannot be vectorized unless it meets search loop idiom criteria.

Compiler report after changes:

No changes. There is no way this function could be rewritten in such a way that it satisfies single entry- single exit rule. Same result as above.

Exercise 3:
"Vectorization across equal-shaped problems/subtasks"

Sketch of the Gaussian Algorithm



Why can a single system of equations of rank 3 not be vectorized?
The overhead that vectorization would introduce would immensely surpass the theoretical gains. In the calculation of a single system of equations no more than 10 - 20 double precision memory addresses would be involved. This methods of vectorization are best applied to vast amounts of data, where the improvements in performance surpass the added complexity overhead.

Our Solution
Intel proposes that data to be optimally vectorized, structure of arrays should be prefered instead of arrays of structures.

> "Use structure of arrays (SoA) instead of array of structures (AoS): An array is the most common type of data structure that contains a contiguous collection of data items that can be accessed by an ordinal index. You can organize this data as an array of structures (AoS) or as a structure of arrays (SoA). While AoS organization is excellent for encapsulation it can be a hindrance for use of vector processing. To make vectorization of the resulting code more effective, you can also select appropriate data structures."
> in Intel Programing Guidelines for Vectorization

With this in mind, we approached this problem by, defining a way to optimize the location of our data in a way that the compiler could take best use out of it. The most significant amount of data, in terms of size, that we are going to process are the right-hand-side vectors. In the example code provided, there are 32800 * 3 records to be stored and later processed. This totals close to 100 000 double precision records.
We decided that the optimal way to store this data, was to create a structure of arrays. Our struct definition is as follows:

```
struct SoA{
        double x[NGLS];
        double y[NGLS];
        double z[NGLS];
};
```

Each of the arrays holds the data for one of the coordinates of the right hand side array. This means that all the data will be aligned therefore optimized for automatic vectorization by the Intel Compiler.

This meant that several changes would have to be made in the given algorithm, but always with the Intel Auto Vectorization Guidelines in mind. For instance, flow-control structures were avoided where possible, which represented an additional challenge.

This solution proved itself successful by decreasing the time of execution from X to Y.

We also decided to see how a different approach to the calculation of equation systems by processing matrixes would behave against the vectorization solution. To that extent, we decided to implement, out of pure scientific curiosity, the LU-Factorization Method. This method factorizes the initial matrix into two triangular matrixes, one upper-triangular (U-Matrix), and one lower-triangular (L-Matrix). These to matrixes allow the calculation of the X vector, taking into account any right hand side vector (B Vector). The big advantage is that, once the L & U Matrixes are calculated, the rest of the processing is trivial for the given B arrays. On the other hand, the original, provided method, enforces a re-calculation of the entire A matrix for each and every B-Array.

This solution proved itself also successful in terms of performance. However, he noticed a loss of precision that we weren't, until the time that this report is being written, able to totally understand and minimize.

Results

Method	Average Time (s)	Average Capacity (GFLOPS)
Original Gauss	**0.00303**	**11772864.87**
Vectorized Gauss	**0.0016228**	**21767951.25**
LU-Factorization	**0.0013848**	**777.2818277**

For the capacity calculation, the following complexity approximations were considered: Original Gauss O(n^3), Vectorized Gauss O(n^3), LU-Factorization O(n^2).
The complete dataset is also provided in the attached spreadsheet.

Conclusions

The Vectorized implementation of the Gauss method showed an time reduction of nearly 50% over the original Gauss method. And the LU-Factorization even surpassed the 50% time reduction.
We can therefore conclude that:

- When done properly, vectorization can in fact improve dramatically the execution cost of a task
- It's better to have a good algorithm without vectorization than a bad algorithm vectorized. By good algorithm, we mean an algorithm that is more appropriate for the problem in question.

Exercise 4: "Matrix-Matrix-Multiplication I

Q1. Examine the memory access of the given implementation.

The algorithm used in the implementation of matrix-matrix multiplication is fairly simple and is obviously not commutative. The complexity of this algorithm is (2* n^3) which is not optimal one. In this exercise we need to examine the memory access of the given implementation.

Generally computer programmers take only arithmetic operations such as additions and multiplications in account while analysing complexity and ignores the data movement operations(load/store). But in reality, it turns out that loads and stores, i.e. moving data from memory to registers (which is where all the arithmetic and other "useful work" takes place) are the most expensive, sometimes costing hundreds of cycles or more. The reason is that any computer memory is organized as a memory hierarchy. Whenever some arithmetic operation is performed on data which are not available in registers, those data is fetched from slower memory, which consumes considerable amount of time. So it is clear that merely counting arithmetic operations is not the whole story; we should try to minimize memory accesses instead.

In the case of given matrix multiplication implementation, if we assume there is some small number of memory accesses for each statement like ci,j = ci,j + ai,k * bk,j (load ai,k, bk,j , ci,j ; store ci,j), then counting arithmetic is nearly the same as counting memory accesses. But it turns out that if there is a memory hierarchy then one can organize the algorithm to minimize the number of accesses to the slower levels of the hierarchy, since these are the most expensive operations. So it is possible and needed to organize the matrix multiplication to reuse data already present in fast memory, decreasing the number of slow memory accesses.

So now we first establish few assumptions.

1. We know arithmetic (and logic) can only be done on data residing in fast memory. Let's assume that each such operation takes time "tL".
2. Moving a word of data from fast to slow memory, or slow to fast, takes time "tM" (tM >>tL). So if a program does "M" memory moves and "L" arithmetic operations, the total time it takes is "T" (T= L * tL + M * tM), where M * tM may be >> L * tL.

We also assume that the slow memory is large enough to contain our three n-by-n matrices A, B and C, but the fast memory is too small for this. Otherwise, if the fast memory were large enough to contain A, B and C simultaneously, then our algorithm would be following.

a. Move A, B and C from slow to fast memory.
b. Compute C = C + A * B entirely in fast memory
c. Move the result C back to slow memory

The number of slow memory accesses for this algorithm is 4 * n^2 [3 * n^2 for loads of A, B and C into fast memory and n^2 for storing C to slow memory].
Thus T = 2 * (n^3) * tL + 4 * (n^2) * tM. Clearly, no algorithm doing 2*(n^3) arithmetic operations can run faster.

At the extreme where the fast memory is very small (M = 1), then there will be at least 1 memory reference per operand for each arithmetic operation involving entries of A and B, for a running time of at least T = 2 * (n^3) * tL + [2* (n^3) + 2 * (n^3)] * tM.

For practical question for large matrices, since real caches have thousands of entries, the size of fast memory M satisfies 1 << M << 3 * (n^2).

As we just saw, the worst case running time is 2 * (n^3) * tL + [2* (n^3) + 2 * (n^3)] * tM, and the best we can hope for is 2 * (n^3) * tL + 4 * (n^2) * tM, which can be almost n/2 times faster when tM >> tL.

Now we analyze the given matrix multiplication algorithm below, including descriptions of when data moves from slow to fast memory and back. As we have stated earlier A, B and C all start in slow memory, and that the results C must be finally stored in slow memory.

```
Matrix multiplication C = C + A * B
procedure MM(A,B,C)
for i = 1 to n
        for j = 1 to n
                for k = 1 to n
                        Load ci,j into fast memory
                        Load ai,k into fast memory
                        Load bk,j into fast memory
                        Store ci,j into slow memory
                end for
        end for
end for
```

Let “m_MM” denote the number of slow memory references in above algorithm. Then

```
m_MM = (n^3)      ... for loading each entry of A n times.
      +(n^3)      ... for loading each entry of B n times.
      +(2* n^3) ... for loading and storing each entry of C once.
      = 4* n^3
```

or about as many slow memory references as arithmetic operations. Thus the running time (in worst case) is "t_MM", where t_MM = 2* n^3 + (2* n^3 + 2* n^3)* tM

It is interesting to know that according to 13th Symposium on the Theory of Computing, any implementation of matrix multiplication using 2* n^3 arithmetic operations performs at least $\Omega(n\ 3/\ \sqrt{}\ M)$ slow memory references. Here M denotes the size of fast memory.

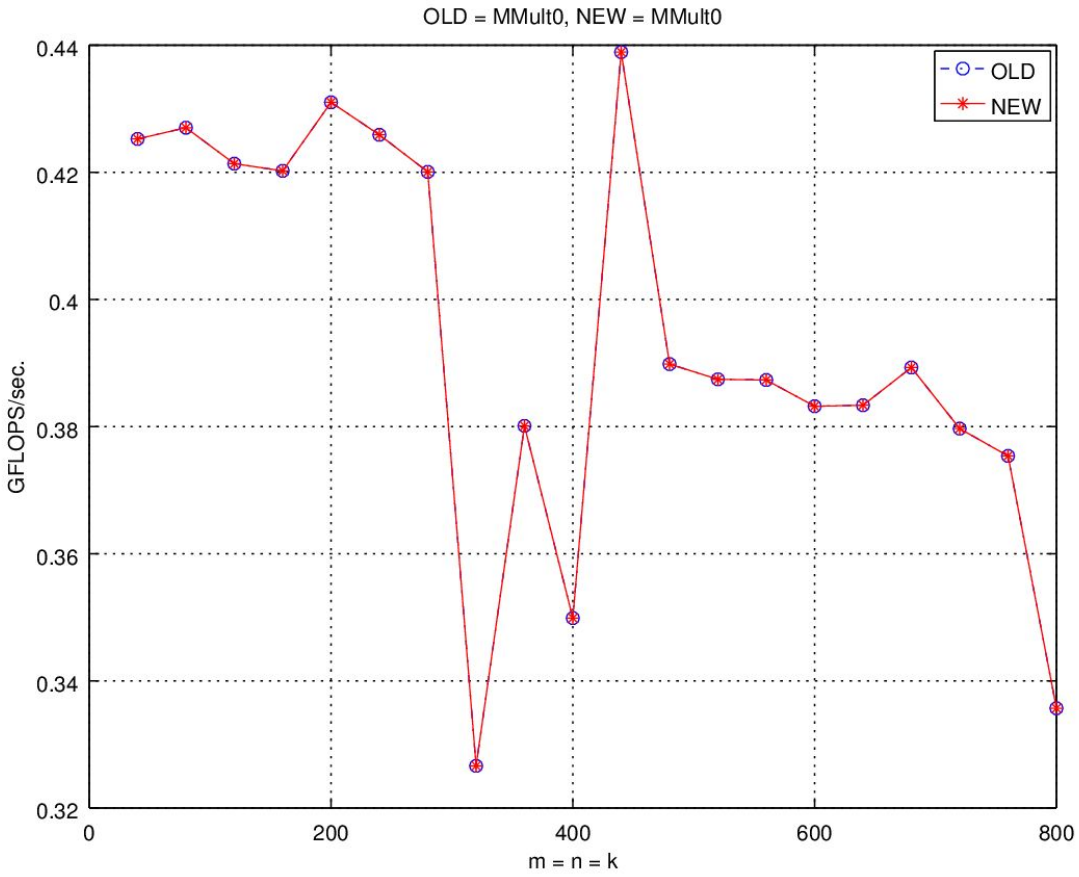
Q2. Use different compiler-directive (Intel compiler) in order to vectorize the given code.

icc timer.c -c
icc dgemm.c -o matrix timer.o

Q3. Measure MFLOPS rates and cache-miss-rates for different problems sizes and plot your results. Please use Valgrind for measuring the cache-miss rate!

valgrind --tool=cachegrind --D1=1024,1,64 --log-file=cg.out ./matrix PROBLEM_SIZE
where PROBLEM_SIZE is the size of the problem; such as 1000, 2000.

Problem size	MFLOPS rate	Cache-miss-rate
2000	-3.351748e+00	3.2%
1800	-4.366551e+00	3.2%
1500	-8.051436e+00	3.1%
1200	-1.439073e+01	3.2%
1000	2.232166e+01	3.1%
800	2.477814e+01	3.1%
600	2.296245e+01	0.0%
400	2.489093e+01	0.0%
200	2.296452e+01	0.0%



Plot between GFLOPS rates and problem sizes