**Intel® System Studio**

**Lab: Vectorization with Intel® Compiler and**

**Intel® Math Kernel Library**

# Overview

This lab shows how to improve the application performance using vectorization optimizations in Intel® Compiler and Intel® Math Kernel Library.

Parts 1 – 5 of the lab focus on Intel® Compiler optimization options. The sample application used in this part of the lab implements a matrix by vector multiplication function.

Part 6 of the lab shows how to use Intel® Math Kernel Library to optimize General Matrix Multiplication (GEMM).

# Prerequisites

Set the environment variables (Linux\*):

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| $ | | sourc | e | |
|  | /opt/intel/system\_studio\_2019/compilers\_and\_libraries\_2019/linux/bin/compilervars.sh intel64 | | |  |

# Part 1: Using Optimization Level Options

This part of the lab describes Intel® Compiler optimization level options, and shows the impact they have on the sample application’s performance.

Intel® Compiler supports the following options to set the optimization level (Linux\* and macOS\*):

* **-O0** – Disables all optimizations. Can be useful for the application debugging.
* **-O1** - Enables optimizations for speed and disables some optimizations that increase code size and affect speed.
* **-02** – Enables optimizations for speed. This is the generally recommended optimization level. Vectorization (using SSE2 by default) is enabled at O2 and higher levels (this is the default optimization level)
* **-03** - Performs O2 optimizations and enables more aggressive loop transformations such as Fusion, Block-Unroll-and-Jam, and collapsing IF statements.

The source files for this part of the lab are located in ~/vec\_samples/src. Please change the current directory to this path using the following command:

|  |  |  |
| --- | --- | --- |
| $ | cd ~/vec\_samples/src |  |
| vec\_samples/src$ |

Using the commands below compile and run the sample application using different optimization level options, and observe the execution time:

vec\_samples/src$ icc -O0 -no-vec Driver.c Multiply.c -o MatVector.o0 vec\_samples/src$ ./MatVector.o0

ROW:101 COL: 101

Execution time is **39.150** seconds

GigaFlops = 0.521129

Sum of result = 195853.999899

vec\_samples/src$ icc -O1 Driver.c Multiply.c -o MatVector.o1 vec\_samples/src$ ./MatVector.o1

ROW:101 COL: 101

Execution time is **10.110** seconds

GigaFlops = 2.018024

Sum of result = 195853.999899

vec\_samples/src$ icc -O2 Driver.c Multiply.c -o MatVector.o2 vec\_samples/src$ ./MatVector.o2

ROW:101 COL: 101

Execution time is **9.854** seconds

GigaFlops = 2.070460

Sum of result = 195853.999899

# Part 2: Generating a Vectorization Report

The Intel® Compiler’s optimization report tells the programmer which optimizations were performed and why other optimizations were not performed. A programmer can use this feedback to tune code to enable additional compiler optimizations and further enhance application performance.

The generation of the Intel® Compiler optimization report can be enabled and controlled using the following options:

* **-qopt-report[=N]** - Enables the report; N=1-5 specifies an increasing level of detail (default N=2)
* **-qopt-report-file=stdout | stderr | filename** - Controls where the report is written (default is to file with extension .optrpt)
* **-qopt-report-phase=phase1[,phase2,…]** - Optimization information is provided only for the specified optimization phases. Some of the supported phases are: o **vec** - Automatic and explicit vectorization using SIMD instructions o **loop** - Memory, cache usage and other loop optimizations o **all** - Reports on all optimization phases (default)

Using the command below compile the sample application while enabling the optimization report for vectorization and loop phases (**-qopt-report-phase=vec,loop** and **-qopt-report=2** options).

vec\_samples/

src$

icc

-

O2

-

qopt

-

report

-

phase=vec

,loop

-

qopt

-

report=2

Driver.c

Multiply.c

-

o

MatVector

.o2

Check the optimization report. Observe that the loop at the line 49 was not vectorized due to a possible vector dependence (aliasing). The aliasing happens when two pointers point to the same memory location.

|  |  |  |
| --- | --- | --- |
| vec\_samples/src$                LOOP BEGIN at Multiply.c(37,5)      LOOP BEGIN at Multiply.c(    b[i] (50:13)  LOOP END      <Remainder>  LOOP END  LOOP END | cat Multiply.optrpt | us/intel-advisor-xe" for details.  Begin optimization report for: matvec(int, int, double (\*)[\*], double \*, double \*)    -vectorized: consider using SIMD directive  **remark #15344: loop was not vectorized: vector dependence prevents vectorization.**  assumed FLOW dependence between b[i] (50:13) and    =============== |
| Intel(R) Advisor can now assist with vectorization and show optimization report messages with your source code.  See "https://software.intel.com/en-  Report from: Loop nest & Vector optimizations [loop, vec]    remark #15541: outer loop was not auto  **49**,9)  First dependence is shown below. Use level 5 report for details remark #15346: vector dependence: remark #25439: unrolled with remainder by 2  LOOP BEGIN at Multiply.c(49,9)  ============================================================ |
|  | | |

The here are the respective lines of the **matvec()** function defined in ***Multiply.c*** file. The compiler assumes that the destination array **b[]** may overlap with the source arrays **a[]** and **x[]**, which will lead to an incorrect result if the loop is vectorized:

|  |
| --- |
| 32 void matvec(int size1, int size2, FTYPE a[][size2], FTYPE b[], FTYPE x[]) <...>   1. for (j = 0;j < size2; j++) { 2. b[i] += a[i][j] \* x[j]; 51 } |

# Part 3: Improving Performance by Disabling Pointer Aliasing

As explained in the previous lab, two pointers are aliased if both point to the same memory location. In the case of a possible pointer aliasing the compiler takes the conservative approach, and does not vectorize the loop. If the developer is certain that the arrays are not overlapping, and thus the aliasing will not occur, it is possible to inform the compiler that it is indeed the case using **restrict** type qualifier and either **-restrict** or **-std=c99** compiler options.

Using the command below compile the sample application while enabling the optimization report for vectorization phase (**-qopt-report-phase=vec** and **-qopt-report=2** options). The **-D NOALIAS** enables the following **matvec()** function’s signature with the **restrict** type qualifier:

void matvec(int size1, int size2, FTYPE a[][size2], FTYPE b[**restrict**], FTYPE x[])

Run the application and observe the execution time.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| vec\_samples/src$ | | icc -std=c99 -qopt-report=2 -qopt-report-phase=vec -D NOALIAS Driver. | | c |
|  | Multiply.c -o MatVector.noalias | |  |
| vec\_samples/src$ ./MatVector.noalias    ROW:101 COL: 101  Execution time is **3.364** seconds  GigaFlops = 6.064092  Sum of result = 195853.999899 | |

Check the optimization report. Observe that the loop at the line 49 was vectorized this time:

|  |  |  |
| --- | --- | --- |
| vec\_samples/src$    <...>  LOOP BEGIN at Multiply.c(    LOOP END  <...> | cat Multiply.optrpt | =============================== |
| **49**,2)  **remark #15300: LOOP WAS VECTORIZED**  ============================================ |

# Part 4: Improving Performance by Aligning the Data

The compiler can vectorize the code more efficiently when operating on aligned data.

This part of the lab shows how to improve performance by aligning the arrays **a[]**, **b[]**, and **x[]** defined in ***Driver.c*** file on a 16-byte boundary so that the compiler can use aligned load instructions for all arrays rather than the slower unaligned load instructions, and that it can avoid runtime tests of alignment. Using the **ALIGNED** macro will modify the declarations of **a[]**, **b[]**, and **x[]** using the **aligned** attribute keyword, which has the following syntax:

float array[30] \_\_attribute\_\_((**aligned**(base, [offset])));

This instructs the compiler to create an array that it is aligned on a "base"-byte boundary with an "offset" (Default=0) in bytes from that boundary. Example:

FTYPE a[ROW][COLWIDTH] \_\_attribute\_\_((**aligned**(16)));

In addition, the row length of the matrix **a[]**, needs to be padded to be a multiple of 16 bytes, so that each individual row of it is 16-byte aligned. It is also necessary to inform the compiler that the arrays in the **matvec()** function are aligned by using **#pragma vector aligned**.

Using the commands below compile the sample application while enabling the optimization report for vectorization phase (**-qopt-report-phase=vec** and **-qopt-report=4** options). Run the application and observe the execution time.

vec\_samp

les/src$

icc

-

std=c99

-

qopt

-

report=4

-

qopt

-

report

-

phase=vec

-

D NOALIAS

-

D ALIGNE

D

Multiply.c Driver.c

-

o MatVector.noalias.aligned

vec\_samples/src$ ./MatVector.noalias.aligned

ROW:101 COL: 102

Execution time is

**3.183**

seconds

GigaFlops = 6.409127

Sum of r

esult = 195853.999899

Check the optimization report. Observe the estimated potential speedup.

|  |  |  |
| --- | --- | --- |
| vec\_samples/src$  <...>    [ Multiply.c(50,21)    [ Multiply.c(50,31)    remark #15300:  remark #15475: remark #15476: remark #15477: remark #15478: remark #15488:  LOOP END      <    [ Multiply.c(50,21)    [ Multiply.c(50,31)    inefficient. Use vector always d    remark #15475:    remark #15488:  LOOP END  <...> | cat Multiply.optrpt | remark #15388: vectorization support: reference a[i][j] has aligned access  erence x[j] has aligned access      remark #15309: vectorization support: normalized vectorization overhead 0.594    begin vector cost summary ---      **2.410**  end vector cost summary ---  >  remark #15388: vectorization support: reference a[i][j] has aligned access remark #15388: vectorization support: reference x[j] has aligned access  remark #15335: remainder loop was not vectorized: vectorization possible but seems  irective or -vec-threshold0 to override  remark #15309: vectorization support: normalized vectorization overhead 2.417  begin vector cost summary ---        end vector cost summary --- |
| LOOP BEGIN at Multiply.c(49,2)  ]  remark #15388: vectorization support: ref  ]  remark #15305: vectorization support: vector length 2 remark #15399: vectorization support: unroll factor set to 4  LOOP WAS VECTORIZED  remark #15448: unmasked aligned unit stride loads: 2  --- scalar cost: 10 vector cost: 4.000 estimated potential speedup:  ---  LOOP BEGIN at Multiply.c(49,2)  Remainder loop for vectorization  ]  ]  remark #15305: vectorization support: vector length 2  remark #15448: unmasked aligned unit stride loads: 2  ---  remark #15476: scalar cost: 10 remark #15477: vector cost: 4.000  remark #15478: estimated potential speedup: 2.410 --- |

Note: The “vector cost” numbers can be thought of as the vectorizer’s rough estimates of the execution time per iteration of the original loop, in arbitrary units, for the original scalar version and for the vectorized loop version.

# Part 5: Enabling AVX-512 Vector Instructions

The latest Intel® Xeon™ Processors add support for AVX-512 vector instructions. One way to verify the SIMD instructions are being used by the compiler is to generate the assembly output using **-S** option, and to check the vector registers used in the relevant assembly output files.

Using the command below compile the ***Multiply.c*** file and generate the assembly output:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| vec\_samples/src$ | | | icc -S -std=c99 -qopt-report=4 -qopt-report-phase=vec -D NOALIAS -D ALIGNE | D |
|  | Multiply.c |  | |

Check the ***Multiply.s*** assembly output file report. Observe that complier used **xmm** registers (SSE instruction set).

|  |  |  |
| --- | --- | --- |
| vec\_samples/src$    <...>      pxor %    movaps % movaps % movaps %  .alig  **xmm0 xmm1 xmm2 xmm3**  movups                addpd % addpd % addpd % addpd %        <...> | vi Multiply.s | # Execution count [4.50e+00]  movl %esi, %ebx #49.2 movq %r11, %r14 #49.2  #50.13  #50.21 #50.13  #50.13  #50.13  # LOE rax rdx rcx rbp rdi r8 r9 r10 r11 r12 r13 r14 ebx esi    # Execution count [2.50e+01]  #50.21 **xmm5**  #50.21 **xmm6** #50.21 **xmm7** #50.21  addq $64, %rbp #49.2 **xmm4** #50.31 **xmm5** #50.31  , %**xmm6** #50.31 **xmm7** #50.31 #50.13  #50.13  #50.13  #50.13  addq $8, %r14 #49.2  #49.2  jb ..B1.6 # Prob 82% #49.2 |
| ..B1.5: # Preds ..B1.4  **xmm3**, %**xmm3**  movq %rdx, %rbp  **xmm3**, %**xmm2 xmm2**, %**xmm1 xmm1**, %**xmm0**  n 16,0x90    ..B1.6: # Preds ..B1.6 ..B1.5  (%rbp), %**xmm4**  movups 16(%rbp), % movups 32(%rbp), % movups 48(%rbp), %  mulpd (%r8,%r14,8), % mulpd 16(%r8,%r14,8), % mulpd 32(%r8,%r14,8) mulpd 48(%r8,%r14,8), %  **xmm4**, %**xmm3 xmm5**, %**xmm2 xmm6**, %**xmm1 xmm7**, %**xmm0**  cmpq %rax, %r14 |

Recompile ***Multiply.c*** adding **-xcore-avx512** and **-qopt-zmm-usage=high** compiler options to enable more aggressive AVX-512 instruction set 512-bit SIMD vectorization using ZMM registers.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| vec\_samples/src$ | | icc -S -std=c99 -qopt-report=4 -qopt-report-phase=vec -D NOALIAS -D ALIGNE | | D |
|  | Multiply.c -xcore-avx512 -qopt-zmm-usage=high | |  |

Check the ***Multiply.s*** assembly output file report. Observe that this time complier used **zmm** registers (AVX-512 instruction set).

|  |  |  |  |
| --- | --- | --- | --- |
| vec\_samples/src$ | vi Multiply.s |  | |
| <...>  ..B1.5: # Preds ..B1.4  # Execution count [4.50e+00] vmovaps %**zmm5**, %**zmm4** #50.13 movl %eax, %ebx #49.2 vmovaps %**zmm4**, %**zmm3** #50.13 movq %r11, %r13 #49.2 vmovaps %**zmm3**, %**zmm2** #50.13 movq %rdx, %r12 #50.21 movq -16(%rsp), %r14 #50.21[spill]  .align 16,0x90  # LOE rdx rcx rbp rsi rdi r8 r9 r11 r12 r13 r14 eax ebx r10d ymm0 ymm1 **zmm2 zmm3 zmm4 zmm5**  ..B1.6: # Preds ..B1.6 ..B1.5 # Execution count [2.50e+01] vmovups (%r8,%r13,8), %**zmm6** #50.31 vmovups 64(%r8,%r13,8), %**zmm7** #50.31 vmovups 128(%r8,%r13,8), %**zmm8** #50.31 vmovups 192(%r8,%r13,8), %**zmm9** #50.31 vfmadd231pd (%r12), %**zmm6**, %**zmm5** #50.13 vfmadd231pd 64(%r12), %**zmm7**, %**zmm4** #50.13 vfmadd231pd 128(%r12), %**zmm8**, %**zmm3** #50.13 vfmadd231pd 192(%r12), %**zmm9**, %**zmm2** #50.13 addq $32, %r13 #49.2 addq $256, %r12 #49.2 cmpq %r14, %r13 #49.2 jb ..B1.6 # Prob 82% #49.2 <...> | | |

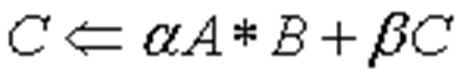
Check the optimization report. Observe the estimated potential speedup.

|  |  |  |
| --- | --- | --- |
| vec\_samples/src$  <...>    [ Multiply.c(50,21)    [ Multiply.c(50,31)        remark #15475:    remark #15488:  LOOP END  <...> | cat Multiply.optrpt | remark #15388: vectorization support: reference a[i][j] has aligned access remark #15388: vectorization support: reference x[j] has aligned access  r length 8  remark #15309: vectorization support: normalized vectorization overhead 0.893 ds: 2  begin vector cost summary ---    **7.860**  end vector cost summary --- |
| LOOP BEGIN at Multiply.c(49,9)  ]  ]  remark #15305: vectorization support: vecto  remark #15399: vectorization support: unroll factor set to 4  remark #15300: LOOP WAS VECTORIZED  remark #15448: unmasked aligned unit stride loa  ---  remark #15476: scalar cost: 8 remark #15477: vector cost: 0.870 remark #15478: estimated potential speedup: --- |

# Part 6: Intel® Math Kernel Library – Multiplying Matrices Using dgemm

General Matrix Multiplication (GEMM) is a part of Basic Linear Algebra Subprograms (BLAS). It is widely used in Deep Neural Networks (DNNs), and it is one of the most computationally expensive parts of these networks.

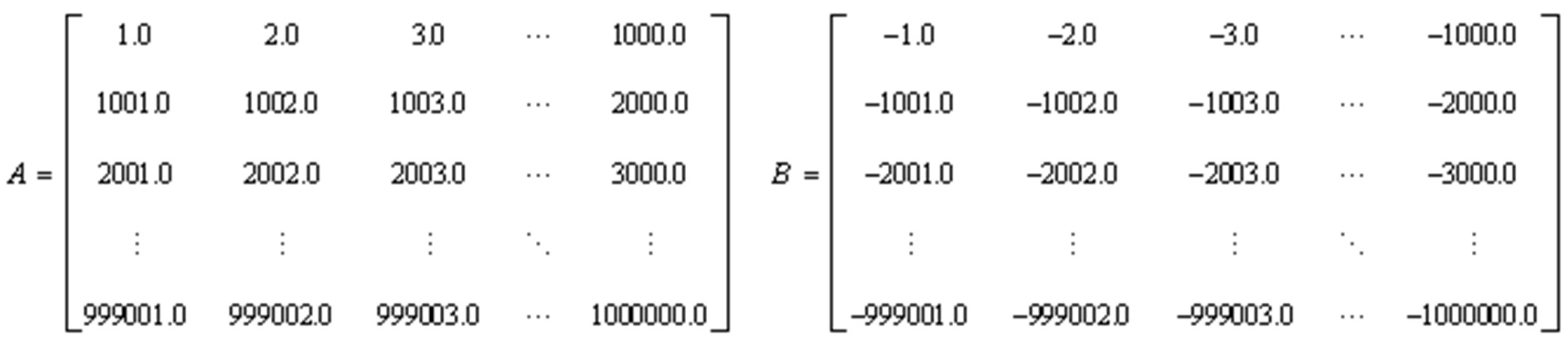
Intel® Math Kernel Library (MKL) provides several routines for multiplying matrices. The most widely used is the **dgemm()** routine, which calculates the product of double precision matrices:



The latest Intel® Cascade Lake processors include the AVX-512 Vector Neural Network Instructions (VNNI) that specifically intended to accelerate GEMM algorithm. Intel® MKL can autodetect AVX-512 VNNI support, and make use of it to improve the GEMM algorithm performance.

This part of the lab compares two implementations of the matrix multiplication:

1. Naïve triple nested loop implementation - ***matrix\_multiplication.c***
2. Intel® MKL based implementation using dgemm function - ***dgemm\_with\_timing.c*** It uses the following input matrices:



One-dimensional arrays are used to store the matrices by placing the elements of each column in the successive elements of the arrays.

The source files for this part of the lab are located in ~/matrix\_multiplication/src. Please change the current directory to this path using the following command:

vec\_samples/src$

cd

~/matrix\_multipli

cation/src

matrix\_multiplication/src

$

## Naïve Triple Nested Loop Implementation

1. printf (" Measuring performance of matrix product using triple nested loop \n\n");
2. s\_initial = dsecnd();
3. for (r = 0; r < LOOP\_COUNT; r++) {
4. for (i = 0; i < m; i++) {
5. for (j = 0; j < n; j++) {
6. sum = 0.0;
7. for (k = 0; k < p; k++)
8. sum += A[p\*i+k] \* B[n\*k+j];
9. C[n\*i+j] = sum;
10. }
11. }
12. }

99

s\_elapsed = (dsecnd()

-

s\_initial) / LOOP\_COUNT;

100

101

printf (

" == Matrix multiplication using triple nested loop completed ==

\

n

"

102

" == at

%.5f

milliseconds ==

\

n

\

n

"

, (s\_elapsed \*

1000

))

;

Using the commands below compile and run ***matrix\_multiplication.c***

|  |  |  |
| --- | --- | --- |
| matrix\_multiplication/src$ matrix\_multiplication    matrix\_multiplication/src$              Initializing data for matrix multipl  A(2000x200) and matrix B(200x1000)    Allocating memory for matrices aligned on 64 performance    Initializing matrix data    to get stable run        ==  == at **294.61551 milliseconds**    Deallocating memory    Example completed. | icc -mkl -static-intel -O2 matrix\_multiplication.c - | o |
| ./matrix\_multiplication  This example measures performance of rcomputing the real matrix product C=alpha\*A\*B+beta\*C using a triple nested loop, where A, B, and C are matrices and alpha and beta are double precision scalars  ication C=A\*B for matrix    -byte boundary for better  Making the first run of matrix product using triple nested loop time measurements  Measuring performance of matrix product using triple nested loop  Matrix multiplication using triple nested loop completed ==  == |

Note the execution time of this sample.

## Intel® MKL Based Implementation Using dgemm Function

1. printf (" Measuring performance of matrix product using Intel(R) MKL dgemm function \n"
2. " via CBLAS interface \n\n");
3. s\_initial = dsecnd();
4. for (r = 0; r < LOOP\_COUNT; r++) {
5. cblas\_dgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans,
6. m, n, p, alpha, A, p, B, n, beta, C, n);
7. }
8. s\_elapsed = (dsecnd() - s\_initial) / LOOP\_COUNT;

86

87 printf (" == Matrix multiplication using Intel(R) MKL dgemm completed == \n" 88 " == at %.5f milliseconds == \n\n", (s\_elapsed \* 1000));

Using the commands below compile and run ***dgemm\_with\_timing.c***

|  |  |  |
| --- | --- | --- |
| matrix\_multiplication/src$ dgemm\_with\_timing    matrix\_multiplication/src$      are matrices an      A(2000x200) and matrix B(200x1000)    Allocating memory for matrices aligned on 64 performance    Initializing matrix data          via CBLAS interface    == Matrix multiplication  == at **12.34390 milliseconds**    Deallocating memory    Example completed. | icc -mkl -static-intel -O2 dgemm\_with\_timing.c - | o |
| ./dgemm\_with\_timing  This example measures performance of Intel(R) MKL function dgemm computing real matrix C=alpha\*A\*B+beta\*C, where A, B, and C d alpha and beta are double precision scalars  Initializing data for matrix multiplication C=A\*B for matrix    -byte boundary for better  Making the first run of matrix product using Intel(R) MKL dgemm function via CBLAS interface to get stable run time measurements  Measuring performance of matrix product using Intel(R) MKL dgemm function  using Intel(R) MKL dgemm completed == == |

Note the execution time of this sample. Compare the execution time of two matrix multiplication samples.

# References

* Quick Reference Guide of Intel® Compliers: https://software.intel.com/sites/default/files/Quick-

Reference-Guide-Intel-Compilers-v19.FINAL\_.pdf

* What are PEEL and REMAINDER loops? (Fortran and C vectorization support) :

https://software.intel.com/en-us/articles/what-are-peel-and-remainder-loops-fortranvectorization-support

* Intel® Software Development Products Samples and Tutorials : https://software.intel.com/enus/product-code-samples
* Tutorial: Using Auto Vectorization: https://software.intel.com/en-us/cpp-compiler-autovectorization-tutorial-tutorial-linux-and-macos-version
* Tutorial: Using Intel® Math Kernel Library 2019 for Matrix Multiplication – C:

https://software.intel.com/en-us/download/tutorial-using-intel-math-kernel-library-2019-formatrix-multiplication-c