Performance portability on CPU and GPU using SYCL

Bruno Roberto
Department of Computer Science
University of Salerno
Italy

***Abstract*—Over recent years heterogeneous systems have become more prevalent across HPC systems, with almost all supercomputers incorporating GPUs or other accelerators. These hardware platforms have different characteristics and optimization requirements and often vendors provide libraries targeting their devices. Here born the portability's problem and in this paper using SYCL, a programming model which allows users to write heterogeneous programs using standard C++, we want to compare the performance achieved by portable SYCL kernels for matrix multiplication against vendor libraries specially optimized for specific architectures.**

I. Introduction

Motivation. Due to the heterogeneity of hardware, many vendors provide highly specialized libraries, and as heterogeneous systems become more and more prevalent, the problem of having to write applications which handle multiple different libraries for each available accelerator will only add further complexity. This is where the performance portability's issue arises.
Various definitions of it were given at the 2016 DOE Center of Excellence (COE) meeting [1] in Phoenix, such as "(Performance portability means) the same source code will run productively on a variety of different architectures." (Larkin). However, the definition we decided to use in this paper is the following:
"An application is performance portable if it achieves a consistent ratio of the actual time to solution to either the best-known or the theoretical best time to solution on each platform with minimal platform specific code required."
Therefore in the paper we analyze the problem in SYCL using the matrix multiplication problem, which is widely used in large scientific applications, and often it is an important factor of the overall performance of a HPC application.
We implemented various versions of it in SYCL using different optimization techniques [2] and tested them both on CPU and GPU, compared the results with vendor libraries (CuBLAS and MKL) showing how much far we are from the "peak".

Related work. The performance portability's issue, as we have said, is becoming more and more prevalent in recent years, and there have been various studies done about it which focus on performance portability in SYCL. As in our work, Lawson and his team [3] address the problem using highly parameterized SYCL Kernels, and compare the performance obtained by their implementations with the performance of the respective vendors libraries. In their work they also focused on two problems: Matrix Multiplication and Convolution, while in this paper we analyze only the first one trying additional optimization techniques to those already applied by them. We also focus only on the study of performance portability on two of the most widely used hardware, CPU and GPU.
The work [4] of Johnston and his team, instead, focuses more on evaluating the performance portability of the various SYCL implementations. In fact, they compare how performance varies among the various available implementations. In our work, however, we decided to ignore this aspect and to use only the hipSYCL implementation to run the tests, with, of course, the possibility of a future extension of the experiment on others implementation as well. Finally, we used the work [5] of Pennycook and his team as reference for choosing the metric to evaluate the performance portability of the implemented versions.

II. Background

In this section we will first have a brief overview about the Matrix multiplication problem and some of its typical optimizations, and then a brief description of SYCL programming model.

Matrix Multiplication. Dense matrix-matrix multiplication is a problem widely present in large scientific applications, and therefore it represents an important factor for the overall performance.
The problem definition is very simple:

given two matrices A and B of dimensions MxK and KxN respectively, we want the matrix C of dimension MxN, where:

$$C_{ij} = \sum_{r=1}^{K} A_{ir} * B_{rj}, \;\; i = 1, ..., M \;\; j = 1, ..., N$$

For its importance there are many papers which propose improvements of existing algorithms or completely new algorithms. But all of them must confront with the memory access problem. In fact, its worst bottleneck is the cache usage and each algorithm, due to this memory-intensive and more importantly cache-intensive nature, should try to use the cache hierarchy as effectively and efficiently as possible, avoiding cache misses and strided memory access where is possible. Therefore reducing the memory access by improving the usage of data that is already stored in cache is one of the most used techniques to improve the overall performance of the algorithm.

Optimization techniques. Knowing the bottlenecks of matrix multiplication algorithms, we will briefly show some of the techniques usually used to try to avoid them and boost the performance.[7]

Blocking (or Tiling): is a well-proven technique, which deals with optimizing the cache usage by reducing the memory access. The idea is very simple: we divide the matrix in blocks or tiles of size b and instead of multiplying the entire row of the first matrix with the entire column of the second we multiplies these blocks. In this way we reuse the data and when we remove the data from the cache we have already done all the computations on them and therefore they don't need to be loaded again. Very important is the choice of the block size because to achieve better performances it should be chosen so that the data fills the cache size.

Loop Unrolling: is a loop transformation technique that attempts to optimize a program's execution speed. The transformation can be undertaken manually by the programmer or by an optimizing compiler and simply consist in the literally unrolling of a loop in its instructions. The goal of loop unrolling is to increase the performance by reducing or eliminating instructions that control the loop, such as to hide latency and increase instruction level parallelism. On modern processors, loop unrolling can be also counterproductive, as the increased code size can cause more cache misses, and therefore is very important choose the right loop unrolling factor.

Thread coarsening: is an optimization technique in which the code that is normally executed by several different threads is merged into a single thread. The goal is to execute a smaller number of larger, more coarse-grained, threads than before. Thread coarsening affects a reduction in parallelism in the application, which can have both a beneficial and a detrimental effect on runtime requirements. The choice of the coarsening factor is very important because its increasing allows to exploit the available parallelism efficiently but it also raises resource consumption.

SYCL. SYCL [8] is an open standard maintained by the Khronos Group. It provides a highly parallel programming model designed to abstract away much of the complexity of traditional parallel programming models like OpenCL. Therefore using SYCL instead of a lower level interface allows the developer to write code using modern C++ features, like templates which allow to provide constants at compile time. All these makes SYCL a strong parallel programming framework for performance portability across different platforms.

III. Proposed method

In this section we will briefly talk about the various implementations of matrix multiplication done in SYCL and how they have been tuned for the different platform. We will then talk about the baseline used and show which are the metrics that have been used to evaluate the portability of the applications.

Implementation. First of all the different versions of Matrix multiplication have been implemented as highly parameterized kernel, like the following one.

```
template<int tile_size, int coarse_factor_x, int
    coarse_factor_y>
class MatMulKernel { ... }
```

So in such way it's possible to pass the various parameters of tuning at the compile time and therefore the same code can be easily tuned in different ways according to the architecture. We implemented the various versions of Matrix multiplication using the three optimizations discussed above incrementally. Thus we produced a total of 8 versions, whose complete implementation can be seen at the public repository in the references [9].

Here we'll briefly show only part of three versions of the 8 versions to avoid taking up too much space with examples of implementations of well-known optimization techniques.

```
...
float acc = 0;
for (size_t i = 0; i < M; i++)
    acc += A_acc[i + row * M] * B_acc[col + i * K];
...
```

Listing 1: Simple matrix multiplication without any optimization

```
...
float Csub = 0.0f;
for(int a = aBegin, b = bBegin; a <= aEnd; a +=
    aStep, b += bStep) {
    tileA[tx][ty] = A_acc[a + M * tx + ty];
    tileB[tx][ty] = B_acc[b + K * tx + ty];
    it.barrier(access::fence_space::local_space);

    #ifndef UNROLL_STEP_SIZE
        #pragma unroll
    #else
        #pragma unroll UNROLL_STEP_SIZE
    #endif
    for(int k = 0; k < tile_size; k++)
        Csub += tileA[tx][k] * tileB[k][ty];
    it.barrier(access::fence_space::local_space);

}
...
```

Listing 2: Matrix multiplication using tiling/blocking and loop unrolling

```
...
float acc[c_factor_x][c_factor_y] {};
for(int i = 0; i < M; i++)
    #pragma unroll
    for(int j = 0; j < c_factor_x; j++)
        #pragma unroll
        for(int k = 0; k < c_factor_y; k++) {
            acc[j][k] += A_acc[i + row[j] * M] *
    B_acc[col[k] + i * K];
        }
...
```

Listing 3: Matrix multiplication using only thread coarsening

The first one is a very naive version in which none of the previously cited techniques is used and in fact represents the basic algorithm to which optimizations were subsequently

applied. In addition in this version we can easily see one of the cited problems about memory access. All the access to the B matrix, in fact, are uncoalesced memory access.

While in the second snippet are applied together tiling/blocking and loop unrolling. The first implemented using the SYCL local memory, which allows to avoid the uncoalesced memory access problem. And the second one implemented using the *pragma* directive to explicitly tell the compiler to perform unrolling with the specified unrolling factor (no parameter means complete unrolling of the loop).

In the third snippet instead we can see the application of the thread coarsening technique to the base algorithm. In fact, each thread doesn't compute only one element of the C matrix, but a number of elements which depends on the coarsening factor parameters. Note in order to make the choice of coarsening factor variable, we chose to use the *pragma unroll* directive so as to unroll the small internal loops at compile time.

Tuning. Each of the 8 versions has been tuned using the tool, *HyperMapper* [10]. It's a black-box optimizer based on Bayesian Optimization created by L. Nardi and his group. We decided to use it to autotune each of the versions and find the best parameters for each of the platforms, CPU and GPU.
Here an example of the optimization process made by the tool on the version with tiling, coarsening and unrolling on CPU and GPU.

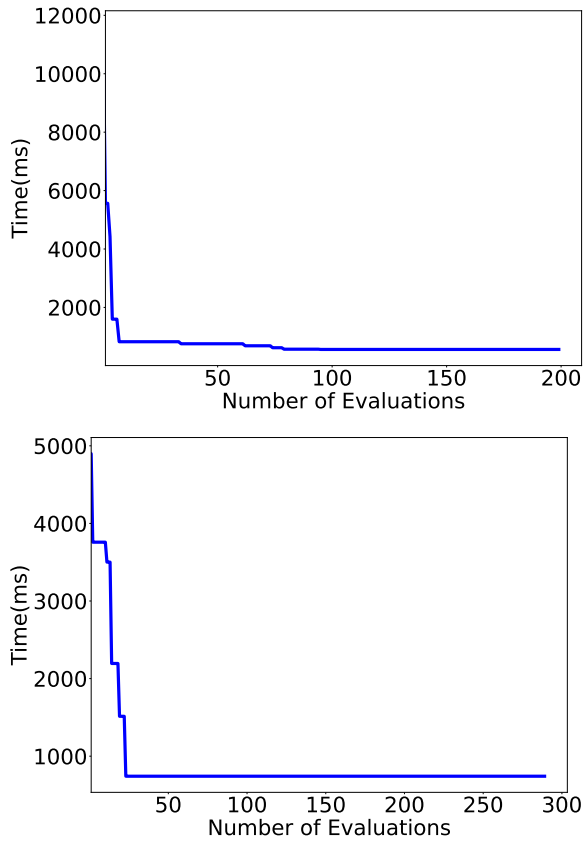

Fig. 1: Tuning on GPU(above) and CPU(below).

These plots have been created using the same data generated by the tool and the x-axis represents the number of evaluations done by the tool, where each evaluation is a "point" in the search space (aka a combinations of the three parameters in this particular case), while the y-axis represents the time in ms calculated as the arithmetic mean over 5 executions of the respective point.
Below there are the tuning parameters found by the tool for each of the 8 versions on both CPU and GPU.

TABLE I: Tuning parameters found by HyperMapper on GPU

Version	Block size	Tile size	Unrolling factor	Coarsening factor
Naive	16x32	-	-	-
Naive + unroll	16x32	-	16	-
Naive + coarsening	8x16	-	-	8x8
Naive + unroll + coarsening	4x32	-	8	8x8
Tiling	16x16	16	-	-
Tiling + unroll	16x16	16	Complete	-
Tiling + thread coarsening	8x16	64	-	8x4
Tiling + unroll + thread coarsening	6x16	64	2	8x4

TABLE II: Tuning parameters found by HyperMapper on CPU

Version	Block size	Tile size	Unrolling factor	Coarsening factor
Naive	128x128	-	-	-
Naive + unroll	128x128	-	4	-
Naive + coarsening	4x32	-	-	16x4
Naive + unroll + coarsening	64x64	-	Complete	8x2
Tiling	32x32	32	-	-
Tiling + unroll	32x32	32	32	-
Tiling + thread coarsening	64x16	128	-	2x8
Tiling + unroll + thread coarsening	64x16	128	4	2x8

Baseline and metrics. After tuning each of the 8 versions on both the architectures, we compared them with our two baselines, *oneAPI MKL* for CPU and *CuBLAS* for GPU. *Intel oneAPI Math Kernel Library* is a mathematical computation library with highly optimized and extensively parallelized routines for CPU and GPU. While the *cuBLAS library* is an implementation of BLAS (Basic Linear Algebra Subprograms) on top of the Nvidia CUDA runtime. To compare the implementations we took the time using the C++ ⟨clock⟩ library, and in particular by taking the average time over 5 runs of the algorithm. Afterwards to evaluate the "efficiency" of our implementations towards the baseline on a single architecture we simply considered the ratio between the baseline time and our implementation's time. While to evaluate the performance portability of the various solutions on both the architectures we decided to use the same metric proposed by Pennycook and his team [5]:

$$PP(a,H) = \begin{cases} \frac{|H|}{\sum_{i \in H} \frac{1}{e_i(a)}} & \text{if } i \text{ is supported } \forall i \in H \\ 0 & \text{otherwise} \end{cases}$$

where for us H is the set of platform, in our case $H = \{\text{CPU, GPU}\}$, a one of the 8 versions implemented and $e_i(a)$ the efficiency of the tuned version a on the i platform. Therefore in the end it's just a simple harmonic mean of the application's efficiency on the various platforms.

IV. Experimental Results

In this section we will show, with some explanatory charts, the results acquired comparing them with the baselines defined before.

Experimental setup. All our experiments have been executed on a *NVIDIA GeForce RTX 3060* with:

- Memory Size: 12 GB;
- Memory Bus width: 192 bit;
- Peak Bandwidth: 360 GB/s;
- Base clock speed: 1320 MHz
- Compute capability: 8.6

And also on a *AMD Ryzen 7 5800H* with:

- Clock Rate: 3200 - 4400 MHz
- Level 1 Cache: 512 KB
- Level 2 Cache: 4 MB
- Level 3 Cache: 16 MB
- Number of Cores/threads: 8/16
- Socket: FP6
- Features: XFR, FMA3, SSE 4.2, AVX2, SMT

The SYCL source codes have been compiled with the hipSYCL 0.9.2 version, using the -O3 optimization flag. While the CuBLAS program has been compiled with the version 11.4.48 of the nvcc compiler and the MKL program with Intel(R) oneAPI DPC++/C++ Compiler 2022.0.0, using for both also the -O3 flag.
All the versions, with the best tuning parameters previously found, have been launched 5 times collecting data about the execution time using the clock library from C++. For our goal, we used the average execution time in ms and we also considered the standard deviation.

Results. In these experiments, our aim is to understand whether and how well the versions of Matrix multiplication, implemented by us on SYCL, are portable in terms of performance.
Therefore, we tested the various versions on different input sizes, depending also on the platform. On GPU we considered matrices from 1024x1024 to 8192x8192 of floating point elements, while on CPU matrices from 1024x1024 to 4096x4096. We considered inputs of different sizes mainly to analyze the behavior of our solutions as it varied.
Below, however, we will present only the results collected for the larger input sizes for both platforms since the results collected for the remaining sizes are more or less in line. The only note we can add is that on CPU for smaller input sizes we achieved times closer to the baseline and this probably because the sizes were such that they did not over-saturate the memory and thus they had better performance.
However, all the data collected from the experiments are still available to the previously cited repository [9].

NxM	cuBLAS	MKL
1024x1024	3	11
2048x2048	14.4	57.4
4096x4096	65.88	388.28
8192x8192	388.576	-

TABLE III: Average execution time in ms of cuBLAS and MKL

TABLE IV: 8192x8192 on GPU

Version	Avg. Time (ms)	Standard deviation	Efficiency from baseline	% from baseline
Naive	1809.6	3.85	0.21	21.47
Naive + unroll	1567	2.35	0.25	24.80
Naive + coarsening	609.2	2.68	0.64	63.78
Naive + unroll + coarsening	529	3.08	0.73	73.45
Tiling	1629	4.85	0.24	23.85
Tiling + unroll	1627.8	4.60	0.24	23.87
Tiling + thread coarsening	554.2	2.77	0.70	70.11
Tiling + unroll + thread coarsening	554.8	3.60	0.70	70.04

Version	Avg. Time (ms)	Standard deviation	Efficiency from baseline	% from baseline
Naive	35788.2	382.44	0.01	1.08
Naive + unroll	33833.2	485.05	0.01	1.15
Naive + coarsening	5070.2	724.14	0.08	7.66
Naive + unroll + coarsening	3952.2	25.14	0.10	9.82
Tiling	1399	41.34	0.28	27.75
Tiling + unroll	1381.8	56.25	0.28	28.10
Tiling + thread coarsening	925.2	3.90	0.42	41.97
Tiling + unroll + thread coarsening	809.2	8.90	0.48	47.98

TABLE V: 4096x4096 on CPU

The previous tables show the average execution time of the various version on both the platforms and the average execution time of the baseline. For our implementation the tables show the efficiency of them too. The efficiency how described before is the simple ratio between the execution time of the baseline and the execution time of our implementation. Therefore, as we can see some of our implementations on GPU reached 70% of the baseline (cuBLAS [11]), while on CPU we had worse performance. In fact, our implementations reached only about 50% of the baseline (MKL [12]). Here a possible cause of these results on CPU may be due to the used implementation of SYCL, since we used only *hipSYCL* in our tests.

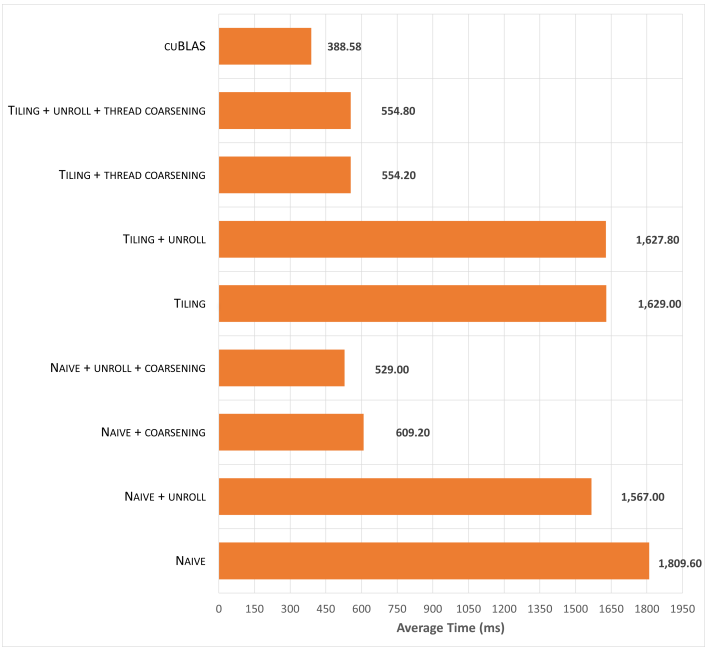

Fig. 2: Average execution time on GPU with size 8192x8192

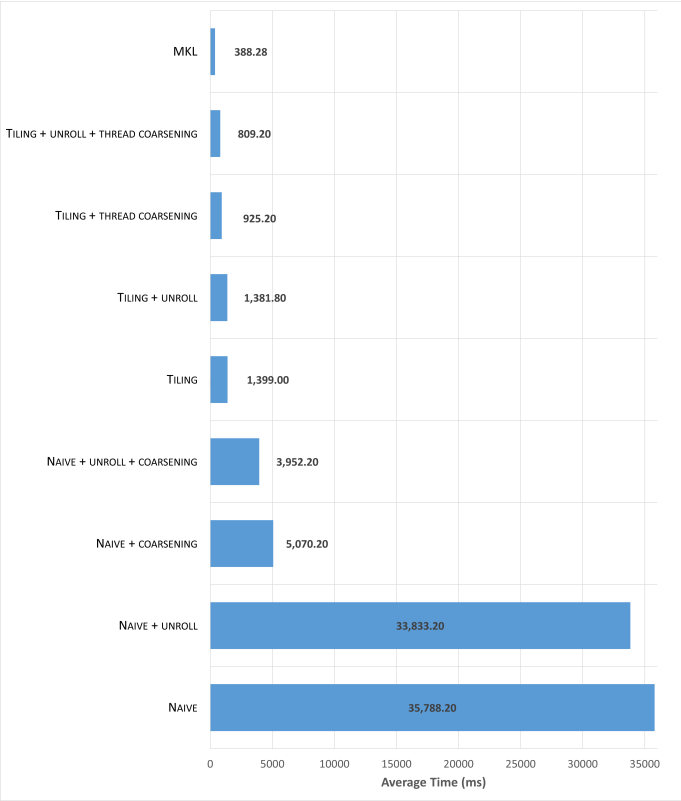

Fig. 3: Average execution time on CPU with size 4096x4096

The charts in Figure 2 and 3 illustrate the execution time of the 8 version and of the baseline respectively on GPU and CPU and on input of size 8192x8192 and 4096x4096. By observing the chart of the Figure 3 and by looking only at the execution times we can notice that on CPU the various optimizations applied always improve the performance. Indeed, the best version on CPU is the last one where all the optimizations are applied together. On the other hand on GPU by observing the chart of the Figure 2, we have a different situation. In fact, the various optimizations don't always improve the performances. Despite this aspect on both the platform the last version with all the three techniques applied tends to be the best and the closest to the baseline.

From the charts we can also see that the two optimizations that have most improved performance are tiling and thread coarsening. Especially the former on CPU and the latter on GPU.

In addition we can make an observation about the loop unrolling technique when it is applied together with tiling. In fact, its application does not cause much of a change in performance, and the cause probably lies in the optimizations made by the compiler, which probably by default performs the loop unrolling of the inner loop used by the tiling.

Recalling the PP metric defined earlier to evaluate the performance portability of an application on a set of platforms, based on the work of Pennycook [5], these are the values of PP obtained for the various versions implemented based on the results of our experiments on GPU and CPU:

TABLE VI: PP on CPU and GPU with size 4096x4096

Version	PP	PP in %
Naive	0.02	2.08
Naive + unroll	0.02	2.21
Naive + coarsening	0.14	13.68
Naive + unroll + coarsening	0.17	17.17
Tiling	0.28	27.88
Tiling + unroll	0.28	28.17
Tiling + thread coarsening	0.51	50.80
Tiling + unroll + thread coarsening	0.56	56.09

As we expected from the results, the best version overall on the two platforms is the last one, in which all the three optimizations described before are applied together. It reaches a value of portability of about 56%, and in particular, as seen before, reaches an efficiency of about 70% on GPU and 50% on CPU.

V. Conclusions

As we have seen, our versions of Matrix multiplication implemented on SYCL managed to obtain results not too far away from those of the corresponding baselines, and all this using only three main optimization techniques: blocking/tiling, thread coarsening and loop unrolling.

However, one reason for these positive results on CPU and GPU is also due to the optimization techniques used, which more or less tend to produce similar improvements on both GPU and CPU, reducing the overall overhead and improving memory accesses.

Despite the seemingly positive results, however, we must point out that the tests were run on only two platforms, an AMD CPU and an NVIDIA GPU. So to have more truthful and meaningful results we should extend the experiment to additional platforms from other vendors or even from the same ones already considered. It would also be important to consider other examples of applications other than Matrix multiplication, such as more compute-bounded problems.

Finally, the last observation we can make, as mentioned earlier, is to compare how the results may vary according to the implementation of SYCL used and thus observe the performance portability of both applications and implementations of SYCL for the various platforms.

REFERENCES

[1] Portability Across DOE Office of Science HPC Facilities, https://performanceportability.org/perfport/definition/

[2] N. Anchev, M. Gusev, S. Ristov and B. Atanasovski, *Some optimization techniques of the matrix multiplication algorithm*, Proceedings of the ITI 2013 35th International Conference on Information Technology Interfaces, 2013

[3] J. Lawson, M. Goli, D. McBain, D. Soutar, L. Sugy, *Cross-Platform Performance Portability Using Highly Parametrized SYCL Kernels*, 2019

[4] Johnston, Beau Vetter, Jeffrey Milthorpe, Josh. (2020), *Evaluating the Performance and Portability of Contemporary SYCL Implementations*, 2020

[5] S. J. Pennycook, J. D. Sewall and V. W. Lee, *A Metric for Performance Portability*, 2016

[6] hipSYCL - a SYCL implementation for CPUs and GPUs, https://github.com/illuhad/hipSYCL

[7] N. Anchev, M. Gusev, S. Ristov, and B. Atanasovski, *Some Optimization Techniques of the Matrix Multiplication Algorithm*, 2013

[8] https://www.khronos.org/sycl/

[9] https://github.com/Rob11001/SYCL_Performance_Portability

[10] Nardi Luigi, David Koeplinger, and Kunle Olukotun. "Practical Design Space Exploration", IEEE MASCOTS, 2019

[11] NVIDIA cuBLAS: Dense Linear Algebra on GPUs. https://developer.nvidia.com/cublas. Accessed: 2019-04-09.

[12] Intel oneAPI Math Kernel Library, https://www.intel.com/content/www/us/en/develop/documentation/oneapi-programming-guide/top/api-based-programming/intel-oneapi-math-kernel-library-onemkl.html