CSE237C Final Project Deliverable

Yun Joon Soh(yjsoh@ucsd.edu), FatemehSadat Mireshghallah (fmireshg@eng.ucsd.edu)

1 Introduction

Intel released recently released OneAPI, which provides means to interact with their unified, standards-based programming model. It targets scalar, vector, matrix, and spatial (SVMS) architectures deployed in CPU, GPU, AI accelerator and FPGA.

OneAPI provides two main methods to leverage the unified programming model: Data Parallel C++ (DPC++) and through API-based programming. DPC++ allows the programmer to easily port the code from one target architecture to another. API-based programming is limited by the set of APIs it provides but allows the programmer to deal with high level logic instead of architecture specific optimization.

Furthermore, OneAPI comes with other analysis tool, debugging tools and utility tool. VTune is well-known performance analyzer that has been used for programming on CPU. Intel included a code convertor called DPC++ Compatibility Tool which converts CUDA code into DPC++ code.

In this final project, we will be exploring the capability and performance of OneAPI.

2 Deliverable

- At least one CUDA code converted to DPC++ and performance comparison on GPU. We will try to make at least one of vector addition, matrix multiplication or small DNN work using oneAPI.
- At least one code (from the ones mentioned above) written in DPC++ compiled and executed on at least two architecture. This part is to show understand the complication of the DPC++ syntax and set of optimization available for the programmer.
- At least one code (from the ones mentioned in the first bullet point) written in API-based programming model compiled and executed on at least two architecture. This part will explore the design concept of OneAPI API-based programming model.

With the comment after the in-class presentation, we modified the deliverables.

- Include comparison of Vivado HLS vs Intel oneAPI.
- Remove the API-bsed programming model from mandatory deliverable list.

3 Grade

- A+ : At least one simple and one complicated code for each bullet point.
- A: All deliverable completed.
- A-: Missing one deliverable.
- B+: Missing two deliverable.
- B: Submitted at least something repository.
- F: Didn't do anything.

```
  4 experimental,xeon,clx,ram192gb,net1gbe
 12 fpga_compile,xeon,plat8153,skl,ram384gb,net1gbe
 12 fpga_runtime,xeon,gold6128,skl,ram192gb,net1gbe,fpga,arria10
 78 gen9,xeon,e-2176g,cfl,gpu,ram64gb,net1gbe,6cores,eus0024
121 jupyter,batch,xeon,gold6128,skl,ram192gb,net1gbe
```

Figure 1: List of available nodes from Intel DevCloud

4 Intel devcloud

Intel provides development cloud (devcloud) to accommodate the users who want to try out the Intel hardware and software with less trouble [1]. As described in Figure 2, there are five different types nodes available to the end users. We used the node with the GPU and the fpga-runtime node to run the DPC++ codes. For CUDA code we used local desktop, which has a CPU with 6 cores and a GTX 1080 discrete GPU.

Job submission Intel DevCloud uses `qsub` job manager to submit, run, query the status of the job. As the usage guide mentions, all the environment variables are reset when the `qsub` runs the job. Thus, setting the environment variable correctly is crucial at the very beginning of each job script. Intel provides sample project that includes a script to submit the job correctly. We modified this project for our purpose.

5 Data Parallel Compatibility Tool (DPCT)

In this section, we explored the potential of DPCT tool, which converts CUDA code (`*.cu`) into DPC++ code (`*.cpp`).

We list some of the difficulties we had while using it and compare the code and performance against the original cuda code.

How to use DPCT We found the basic usage on the Intel's website [2].

1. Optional for large code base: intercept-build make
2. `dpct SOURCE_CODE.cu`
3. `cd path/to/output/dir`
4. `dpcpp CONVERTED_CODE.cpp`

Caveats Although the steps needed to use DPCT seems simple, the actual reality was not. We list some of the caveats when using DPCT either locally or on devcloud.

- Intel suggests using their environment variable setter scripts to before using any of their binaries. However, those scripts are tailored for BASH users. ZSH users must manually set the environment variables.
- CUDA headers are needed for DPCT to work properly. This is set with `--cuda-include-path=`.
- Using incorrect CUDA version **header** affects the execution. Properly setting the correct version is crucial.
- One bright side of this dependency is that all you need is just the headers and not any of the CUDA source codes.
- As such, if you want to use DPCT on devcloud, you must upload the correct CUDA version of header files to the server.
- Uploading partial header files results in conversion failure.

5.1 Case study: Vector Add

In this subsection, we study the sample code provided by Intel, `vector_add` [3]. We converted the code in Listing 1 into the code Listing 2.

The overall logic of the CUDA code is simple. It first allocates memory on the GPU and invoke vector add function. Vector add function initialize the two values to add and sum up the two values and store the result into the result array. Then the result array, `d_C`, is copied back to the main memory. Lastly, it prints out the results onto `stdout`.

Memory Model When compared with the DPCPP code, you first observe similar malloc function (`malloc_device` instead of `cudaMalloc`. One thing to note is that the interface specifies which device it should allocate the memory on. This portion of the code naturally leads to the memory model of oneAPI.

As the Intel oneAPI programming guide [4] suggests, their memory model is based upon the SYCL memory model. That memory model specifies two types of memory objects: buffers and images. These memory objects are accesses with an accessor object which specifies where (host or device) and how (read or write) it will access the memory.

SYCL memory model also provides unified shared memory model (not shown in the code). In this model, SYCL handles the data movement allowing a less optimized but much easier to program environment. The code provided uses the explicit memory model where the programmer must specify the allocation, data movement and the garbage collection of the memory resources.

Kernel Intel oneAPI programming guide [4] lists a set of features available from C++ language standard that oneAPI adopts. Intel oneAPI separates the device code from the host code through lambda expression, functor (function object), or kernel class. Furhtermore, the guide

suggests to use the lambda expression when using the kernel code in line with the device code. We will only discuss the development of kernel using the lambda function for this case study.

Lambda function has the 3 main clauses: captures, params, and body. Each clause is distinguished with one another by [], (), and , respectively.

```
[captures](params)body
```

For example, in the Listing 2 line 34, a lambda function is a parameter for `submit()` function. Its capture clause is `[&]` which indicates that the variables are passed by reference. The parameter clause of the lambda function is the `cl::sycl::handler &cgh`. The body of the lambda function has two parameters, `nd_range`, which defines the iteration domain, and lambda function to the device code.

As the converted code suggests, the oneAPI heavily rely on SYCL. As the code attempts to be as generic as possible in order to accommodate any CUDA code, the syntax looks quite complicated with the nested lambda function.

Performance We analyze the performance of the translated code. As the graph suggests, the translated code is not highly optimized. We compile GPU code with the original CUDA code. We further compiled the ported code and ran on two different clusters namely, CPU and FPGA. We could not confirm that the binary we executed was completely executed on CPU or FPGA, as the translated code by default calls generic function (`get_default_device()`) to get the device information. However, the comparison of the results clearly indicates that they were ran on different hardware.

Running the transformed code on CPU was neither fast nor scalable. FPGA was better than CPU because it scaled almost as good as the GPU. However, FPGA was still much slower than GPU. The original GPU code outperformed the two.

We understand that this is not a direct apple to apple comparison as the nvcc compiles with the hardware information (at least the CUDA compute compatibility version of the GPU). However, we found out that just relying on the compatibility tool was not enough for performance.

Conclusion Through our exploration, we found that using the compatibility tool works well once anyone can avoid the small caveats when setting up the environment. However, it is not enough if one's goal is to generate a highly efficient code.

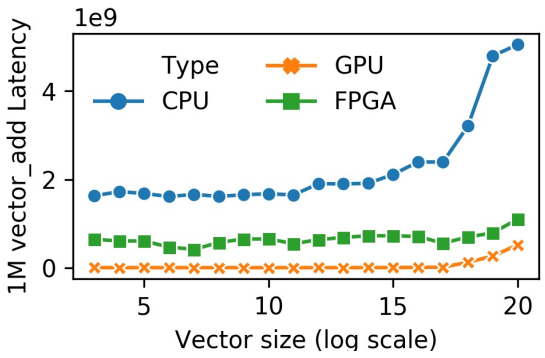

Figure 2: Performance comparison of vector add code on different hardware. Unit of y-axis is nanoseconds.

6 Comparison to Xilinx's Vivado HLS

In this section, we want to compare the Intel OneAPI to Vivado's HLS. A summary of this comparison can be seen in Table 1.

6.1 Purpose

Xilinx's Vivado HLS is a tool that enables C, C++ and SystemC programs to be directly targeted into Xilinx devices without the need to manually create RTL [5]. HLS alleviates the need of using hardware descriptive languages such as Verilog and VHDL, and allows the hardware designers to use C++ syntax to create their designs, and then use a compiler to translate the design and optimize it. Intel OneAPI on the other hand, is purposed to be a unified language for different platforms, to make the job of programmers easier. Intel OneAPI uses a C++ based syntax called data parallel C++ (or DPC++ for short) and can compile one code snippet to execute on different backends (CPU, GPU and FPGA).

6.2 Matrix Multiply and Simple Vector Add Examples

Here, in order to better compare the overall syntax and structure of HLS and DPC++, we go over two code examples. Snippets 3 and 4 show matrix multiplication in HLS and OneAPI, respectively. Snippets 5 and 6 are vector addition, in both languages. For matrix multiplication and vector addition, the core code that does the operations is similar, since it is in C++. For matrix multiplication, the HLS code has three nested for loops, and also some pragmas like pipeline, to increase the throughput. The DPC++ code for OneAPI has two nested for loops, the first one is a parallel for loop that iterates over two iterators at the same time, so overall this one has three loops as well.

```
//==============================================================
// Copyright  2019 Intel Corporation
//
// SPDX-License-Identifier: MIT
// =============================================================

#include <cuda.h>
#include <stdio.h>
#define VECTOR_SIZE 256

__global__ void VectorAddKernel(float* A, float* B, float* C)
{
    A[threadIdx.x] = threadIdx.x + 1.0f;
    B[threadIdx.x] = threadIdx.x + 1.0f;
    C[threadIdx.x] = A[threadIdx.x] + B[threadIdx.x];
}

int main()
{
    float *d_A, *d_B, *d_C;

    cudaMalloc(&d_A, VECTOR_SIZE*sizeof(float));
    cudaMalloc(&d_B, VECTOR_SIZE*sizeof(float));
    cudaMalloc(&d_C, VECTOR_SIZE*sizeof(float));

    VectorAddKernel<<<1, VECTOR_SIZE>>>(d_A, d_B, d_C);

    float Result[VECTOR_SIZE] = { };
    cudaMemcpy(Result, d_C, VECTOR_SIZE*sizeof(float), cudaMemcpyDeviceToHost);

    cudaFree(d_A);
    cudaFree(d_B);
    cudaFree(d_C);

    for (int i = 0; i < VECTOR_SIZE; i++) {
        if (i % 16 == 0) {
            printf("\n");
        }
        printf("%f ", Result[i]);
    }

    return 0;
}
```

Listing 1: Vector add example

For the simple vector addition, the setting is similar. There is a for loop in both cases, and pragmas are used for unrolling.

One eye catching difference between HLS and DPC++ for both examples is the complexity and the lenghth of the codes. If we remove the comments, the DPC++ codes are still longer. This is because OneAPI handles devices and jobs with queues, and jobs should be submitted through queues. Therefore there are always extra lines of codes for that. Also one additional thing that makes the DPC++ code seem vague is the Lambda expression. Lambdas are inline functions that are not named, since they are to be used only once. DPC++ defines the jobs it submits to queues using Lambdas. The syntax used for Lambdas in DPC++ is relatively new, specifically the capture term (the [=] or [&]) which were in C++17. We find this syntax confusing, especially for users who are not advanced in C++.

6.3 Community Size and Supported Backends

Given how Vivado HLS is much older than OneAPI (2012 vs 2019) there is much more community built around it, there are many forums for answering questions, ample documentation and lots of code samples. The OneAPI community however is still very small, there are only a handful of contributors on GitHub, and the documentation is sparse.

In terms of supported backends, Vivado HLS is targeted for Xilinx devices and there is no support for CPUs or GPUs. The OneAPI code however, is supported on

Table 1: Comparison of Xilinx's Vivado HLS and Intel's OneAPI

Framework	Purpose	Lines of Code for Matrix Multiplication	Lines of Code for Simple Vector Add	Code Intelligblity	Community Size	Supported Backends
Xilinx's Vivado HLS	Targeting C/C++ programs for Xilinx devices without the need to manually create RTL.	13	11	High	Large	FPGA
Intel's OneAPI	Unified programming model to eliminates the need for multiple programming languages for different backends	33	32	Low	Small	CPU, GPU, FPGA

CPUs, GPUs and FPGAs.

6.4 Parallel Vector Add

For exploiting more parallelism, there is also an example of a Parallel vector add in the OneAPI tutorials, where parallel for is used for the addition. Parallel for tires to execute the body of the for loop in parallel, therefore there should not be any dependence. This code can be seen in snippet 7.

6.5 Comparison Conclusion

Overall the idea of unifying languages for all backends presented in Intel OneAPI is a really interesting one, however, in terms of practicality, we are not sure whether the chosen syntax and the structure of programming is easy to use for all users. In other words, the main advantage of OneAPI is that one code is executed on all CPU,GPU and FPGA backends. If someone wants to program only an FPGA, they could use HLS which seems simpler. For GPU they could use very high level frameworks such as PyTorch or Tensorflow, that are extremely simpler, and the same goes for CPU. The real question is whether users of high-level langugaes such as these would be willing to program in DPC++?

7 Intel oneAPI Programming Model

Intel oneAPI provides two ways to benefit from their design: DPC++ and API based programming model. In this section, we explore the API Based programming model.

Intel oneAPI provides a variety of libraries. We list a few.

- oneAPI DPC++ Library: Allows inline hardware targeting. General library for using libc++ along with oneAPI.
- oneAPI Math Kernel Library (MKL): Basic math functionality library including linear algebra, FFT, random number generator, vector operations.
- oneAPI Data Analytics Library: Basic data analysis functionalities including correlation, variance-covariance matrix, decision forest, K-means Clustering, K-Neaarest Neighbor, regressions, PCA.
- oneAPI Deep Neural Network Library: Machine learning support with oneAPI.
- oneAPI Collective Comm Library: Integration with Deep Learning Frameworks.

Intel provides code sample for matrix-matrix multiplication [9]. We use a similar code as described in Listing 4 as our baseline. The main portion of Math Kernel Library call is described in Listing 8.

It is easy to note that one using MKL has much simpler code. It initializes the buffers and calls the `gemm` function with lots of parameters.

Performance We further measure the performance on two different computing nodes on Intel DevCloud as described in Section 4. Unfortunately, we were not able to execute the API-based programming model example with `std::bad_alloc` error.

Figure 3 shows the performance of the matrix multiplication. The matrix size is translated into `SIZE / 2 * SIZE / 4` matrix multiplied with `SIZE / 4 * SIZE / 8` matrix. It is not the most interesting graph as due to time limitation and memory related error we were not able to perform large enough matrix to show the real difference.

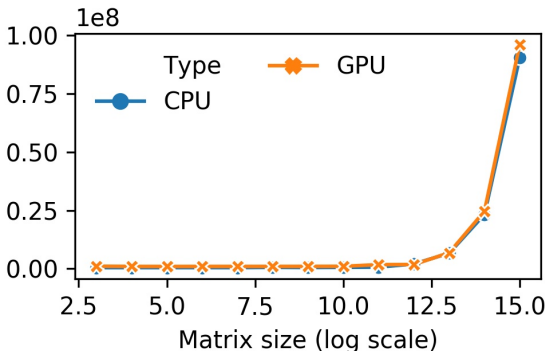

Figure 3: Performance comparison of matrix multiplication code on different hardware. Unit of y-axis is nanoseconds.

8 Conclusion

We conclude that Intel oneAPI is not for everyone. It is neither simple enough for normal users nor fast to use out of the box. However, it works. This could be a great tool for the experts to test and develop their idea as it allows natural migration of code from one hardware to another.

References

[1] "Intel® devcloud," https://software.intel.com/en-us/devcloud.

[2] Dcbenito, "Intel® dpc compatibility tool," https://software.intel.com/en-us/get-started-with-intel-dpcpp-compatibility-tool, Mar 2020.

[3] "Intel® vector_add example," https://software.intel.com/en-us/download/sample-vector-add.

[4] "Intel® oneapi programming guide," https://software.intel.com/en-us/oneapi-programming-guide.

[5] "Xilinx accelerates productivity," https://www.design-reuse.com/news/35626/xilinx-zynq-7000-vivado-design-suite-2014-3.html, accessed March 2020.

[6] R. Kastner, J. Matai, and S. Neuendorffer, "Parallel programming for fpgas," 2018.

[7] "vector_add sample," https://github.com/intel/BaseKit-code-samples/blob/master/DPC%2B%2BCompiler/FPGATutorials/FPGAExtensions/LoopAttributes/loop_unroll/src/loop_unroll.cpp.

[8] "vector_add sample," https://github.com/intel/BaseKit-code-samples/blob/master/DPC%2B%2BCompiler/vector-add/src/vector-add.cpp.

[9] "matrix_mul sample," https://github.com/intel/HPCKit-code-samples/blob/master/MKL/matrix_mul_mkl/src/matrix_mul_mkl.cpp.

```
//==============================================================
// Copyright  2019 Intel Corporation
//
// SPDX-License-Identifier: MIT
// =============================================================

#include <CL/sycl.hpp>
#include <dpct/dpct.hpp>
#include <stdio.h>
#define VECTOR_SIZE 256

void VectorAddKernel(float* A, float* B, float* C, cl::sycl::nd_item<3> item_ct1)
{
    A[item_ct1.get_local_id(2)] = item_ct1.get_local_id(2) + 1.0f;
    B[item_ct1.get_local_id(2)] = item_ct1.get_local_id(2) + 1.0f;
    C[item_ct1.get_local_id(2)] =
        A[item_ct1.get_local_id(2)] + B[item_ct1.get_local_id(2)];
}

int main()
{
    float *d_A, *d_B, *d_C;

    d_A = (float *)cl::sycl::malloc_device(VECTOR_SIZE * sizeof(float),
                                           dpct::get_current_device(),
                                           dpct::get_default_context());
    d_B = (float *)cl::sycl::malloc_device(VECTOR_SIZE * sizeof(float),
                                           dpct::get_current_device(),
                                           dpct::get_default_context());
    d_C = (float *)cl::sycl::malloc_device(VECTOR_SIZE * sizeof(float),
                                           dpct::get_current_device(),
                                           dpct::get_default_context());

    dpct::get_default_queue_wait().submit([&](cl::sycl::handler &cgh) {
        cgh.parallel_for(
            cl::sycl::nd_range<3>(cl::sycl::range<3>(1, 1, 1) *
                                      cl::sycl::range<3>(1, 1, VECTOR_SIZE),
                                  cl::sycl::range<3>(1, 1, VECTOR_SIZE)),
            [=](cl::sycl::nd_item<3> item_ct1) {
            VectorAddKernel(d_A, d_B, d_C, item_ct1);
            });
    });

    float Result[VECTOR_SIZE] = { };
    dpct::get_default_queue_wait()
        .memcpy(Result, d_C, VECTOR_SIZE * sizeof(float))
        .wait();

    cl::sycl::free(d_A, dpct::get_default_context());
    cl::sycl::free(d_B, dpct::get_default_context());
    cl::sycl::free(d_C, dpct::get_default_context());

    for (int i = 0; i < VECTOR_SIZE; i++) {
        if (i % 16 == 0) {
            printf("\n");
        }
        printf("%f ", Result[i]);
    }

    return 0;
}
```

Listing 2: Vector add converted with DPCT

```
void matrixmul(int A[N][M], int B[M][P], int AB[N][P]) {
#pragma HLS ARRAY RESHAPE variable=A complete dim=2
#pragma HLS ARRAY RESHAPE variable=B complete dim=1
// for each row i of A
row: for(int i = 0; i < N; ++i) {
// for each column j of B
col: for(int j = 0; j < P; ++j) {
#pragma HLS PIPELINE II=1
// compute (AB)i,j
int ABij = 0;// = C[i][j];
product: for(int k = 0; k < M; ++k)
ABij += A[i][k]     B[k][j];
AB[i][j] = ABij;
}
}
}
```

Listing 3: HLS code for matrix multiplication [6]

```
void matrixmul( buffer<double, 2> a(range<2>{M, N}), buffer<double, 2> b(range<2>{N, P}),
    buffer<double, 2> c(reinterpret_cast<double*>(c_back), range<2>{M, P}) ) {
  auto property_list =
    cl::sycl::property_list{cl::sycl::property::queue::enable_profiling()};
  event queue_event;
  try{
      #if defined(FPGA_EMULATOR)
    intel::fpga_emulator_selector device_selector;
    #elif defined(CPU_HOST)
    host_selector device_selector;
    #else
    intel::fpga_selector device_selector;
    #endif
    device_queue.submit([&](handler &cgh){
      // Read from a and b, write to c
      auto A = a.get_access<access::mode::read>(cgh);
      auto B = b.get_access<access::mode::read>(cgh);
      auto C = c.get_access<access::mode::write>(cgh);
      int WidthA = a.get_range()[1];
      //Executing kernel
      cgh.parallel_for<class MatrixMult>(range<2>{M, P}, [=](id<2> index){
        //Get global position in Y direction
        int row = index[0];
        //Get global position in X direction
        int col = index[1];

        double sum = 0.0;
        //Compute the result of one element in c
        for (int i = 0; i < WidthA; i++) {
          sum += A[row][i] * B[i][col];
        }

        C[index] = sum;
      });

    });
  }    //End of scope, so we wait for kernel producing result data to host memory c_back to
    complete

  }
```

Listing 4: DPC++ code for matrix multiplication [**?**]

```
void add(data_out_t a[LEN],data_out_t b[LEN],data_out_t c[LEN])
{
  int i;
  data_out_t temp;
  for(i=0;i<LEN;i++)
  #pragma HLS unroll
  {
    temp=a[i];
    c[i]=temp+b[i];
  }
}
```

Listing 5: HLS code for vector addition

```
void vec_add(const std::vector<float>& VA, const std::vector<float>& VB,
    std::vector<float>& VC, int n) {
  auto property_list =
    cl::sycl::property_list{cl::sycl::property::queue::enable_profiling()};
  event queue_event;
  try{
    //Initialize queue with device selector and enabling profiling
    #if defined(FPGA_EMULATOR)
    intel::fpga_emulator_selector device_selector;
    #elif defined(CPU_HOST)
    host_selector device_selector;
    #else
    intel::fpga_selector device_selector;
    #endif
    std::unique_ptr<queue> deviceQueue;
    buffer<float, 1> bufferA(VA.data(), n);
    buffer<float, 1> bufferB(VB.data(), n);
    buffer<float, 1> bufferC(VC.data(), n);

    queue_event = deviceQueue->submit([&](handler& cgh) {
      auto accessorA = bufferA.get_access<sycl_read>(cgh);
      auto accessorB = bufferB.get_access<sycl_read>(cgh);
      auto accessorC = bufferC.get_access<sycl_write>(cgh);
      auto n_items = n;
      cgh.single_task< SimpleVadd<UNROLL_FACTOR> >(
        [=]() {
        #pragma unroll UNROLL_FACTOR
        for(int k = 0; k < n_items; k++){
           accessorC[k] = accessorA[k] + accessorB[k];
        }
      });
    });
    deviceQueue->wait_and_throw();
  }

}
```

Listing 6: DPC++ code for vector addition [7]

```
void VectorAddInDPCPP(const IntArray &addend_1, const IntArray &addend_2,
                      IntArray &sum_parallel) {
  queue q = create_device_queue();

  // print out the device information used for the kernel code
  std::cout << "Device: " << q.get_device().get_info<info::device::name>()
            << std::endl;

  // create the range object for the arrays managed by the buffer
  range<1> num_items{array_size};

  // create buffers that hold the data shared between the host and the devices.
  //    1st parameter: pointer of the data;
  //    2nd parameter: size of the data
  // the buffer destructor is responsible to copy the data back to host when it
  // goes out of scope.
  buffer<int, 1> addend_1_buf(addend_1.data(), num_items);
  buffer<int, 1> addend_2_buf(addend_2.data(), num_items);
  buffer<int, 1> sum_buf(sum_parallel.data(), num_items);

  // submit a command group to the queue by a lambda function that
  // contains the data access permission and device computation (kernel)
  q.submit([&](handler &h) {
    // create an accessor for each buffer with access permission: read, write or
    // read/write the accessor is the only mean to access the memory in the
    // buffer.
    auto addend_1_accessor = addend_1_buf.get_access<dp_read>(h);
    auto addend_2_accessor = addend_2_buf.get_access<dp_read>(h);

    // the sum_accessor is used to store (with write permision) the sum data
    auto sum_accessor = sum_buf.get_access<dp_write>(h);

    // Use parallel_for to run array addition in parallel on device. This
    // executes the kernel.
    //    1st parameter is the number of work items to use
    //    2nd parameter is the kernel, a lambda that specifies what to do per
    //    work item. the parameter of the lambda is the work item id of the
    //    current item.
    // DPC++ supports unnamed lambda kernel by default.
    h.parallel_for(num_items, [=](id<1> i) {
      sum_accessor[i] = addend_1_accessor[i] + addend_2_accessor[i];
    });
  });

  // q.submit() is an asynchronously call. DPC++ runtime enqueues and runs the
  // kernel asynchronously. at the end of the DPC++ scope the buffer's data is
  // copied back to the host.
}
```

Listing 7: DPC++ code for vector addition [8]

```
try {
  // Initializing the devices queue with the default selector
  // The device queue is used to enqueue the kernels and encapsulates
  // all the states needed for execution
  default_selector device_selector;
  queue device_queue(device_selector, asyncHandler);

  std::cout << "Device: " << device_queue.get_device().get_info<info::device::name>() <<
  std::endl;

  // Creating 1D buffers for matrices which are bound to host memory array
  buffer<double, 1> a{A, range<1>{M*N}};
  buffer<double, 1> b{B, range<1>{N*P}};
  buffer<double, 1> c{C, range<1>{M*P}};

  mkl::blas::gemm(device_queue, transA, transB, m, n, k, alpha, a, ldA, b, ldB, beta, c,
  ldC);
}
```

Listing 8: Matrix multiplication using Math Kernel Library (MKL)