GEORG-AUGUST-UNIVERSITÄT GÖTTINGEN IN PUBLICA COMMODA SEIT 1737

HPS

Seminar Report

OneAPI for heterogeneous computing

Vincenz Dumann

MatrNr: 22308641

Supervisor: Vanessa End

Georg-August-Universität Göttingen
Institute of Computer Science

October 2, 2022

Contents

List of Tables

List of Figures

Listings

List of Abbreviations

HPC High-Performance Computing

DPC++ Data Parallel C++

API Application Programming Interface

XPU Any ('X') Processing Unit

1 Introduction

A typical question in an Exam about high performance computing can be the following: "When creating Software, should the used hardware be considered?". The expected answer to this would be something like "It depends, whether the software should be used on exactly one machine, then yes. If it should be used on multiple machines, the hardware should play a smaller role." This means, that developers have to choose between two optimization approaches: Best performance on one system, or generally reasonable performance on multiple devices. This is of course a problem, in the worst case, both approaches are needed and developers need to develop and maintain code bases for two projects, which do essentially the same. A solution to this problem was presented by the US-American company Intel in 2020: With oneApi they defined a new standard as unified application programming interface to get the best out of multiple hardware resources even with one general code base. In this report, which was done during the Seminar "Newest Trends in High Performance Data Analytics" at the University of Göttingen, oneApi is firstly described more in depth (chapter 2), then an experiment is described which tested oneApi in a real live use case (chapter 4). In addition to this, the programming language "Data Parallel C++" is introduced, as is is heavily linked to oneApi (chapter 2. The report closes with a review about the gained knowledge (chapter 5 and a prospect about the future of oneApi (chapter 6. The next chapter will give an overview about related work for further reading.

1.1 Related Work

As oneApi is a relatively new topic, there is not that much scientific work available. In "Applying Intel's oneApi to a machine learning case study"[Mar22], written in 2022 by Pablo Antonio Martínez, Biagio Peccerillo, Sandro Bartolini, José Manuel García and Gregorio Bernabe, the authors analysed oneApi in the context of machine learning. Yong Wang; Yongfa Zhou and Qi Scott Wang analysed in their work "Developing medical ultrasound beamforming application on GPU and FPGA using oneApi"[Wan21] (2021) oneApi in a medical context. Both of this work came to the conclusion, that oneApi is a good solution when it comes to the development of new systems. An other approach was done in "Porting a Legacy CUDA Stencil Code to oneApi"[CS21] (2020), done by Steffen Christgau and Thomas Steinke from the Zuse Institute of the University Berlin. The authors used an existing project and ported it to a oneApi-based application, then they analysed the performance and the results in comparison to the original implementation. Their work is described in chapter 4 more in-depth.

In addition to those scientific papers, there is a lot of literature from the industry open available. Especially intel, the company behind oneApi provides a lot of open information and very detailed guidelines[INT20].

2 oneApi

oneApi is an open, free programming concept, based on standards. It was developed by Intel, to provide cross-platform portability and performance for diverse accelerators and CPUs from a wide range of vendors with different hardware architectures and generations.

A single application interface for a wide variety of hardware - and still well performing code. Intel wants to achieve these two goals, which seem contradictory at first glance, with the oneApi Toolkit, which reached product maturity on 11 November 2020 after two years of intensive development.[Rei21] As a consistent further development of Intel Parallel Studio XE, Intel oneApi 1.0 builds on the proven compilers and libraries. The specification of oneApi is based primarily on the new programming language DPC++ (Data Parallel C++), whose compiler is in turn supplemented by further API library specifications. The problem that oneApi adresses (or, regarding intel as the vendor: it solves)[Pre21]: In order to get the maximum performance out of the most diverse hardware in the long term, a fine balance is necessary - between exploiting all the possibilities of a special hardware and resorting to the specifics of programming this hardware. On the one hand, the source code should use all these hardware functions as efficiently as possible, but on the other hand it should also be portable, easy to maintain and energy-efficient. Otherwise, the long-term use of this code would be difficult - such a balance is anything but easy to achieve, because software development goals are also highly contradictory[Pre21].

Until now, developers have only escaped this dilemma by prioritising goals according to user requirements. In addition, it is anything but trivial for software developers to maintain several separate code bases of an application for different computer architectures. This is because as data-intensive workloads become more diverse, there are also increasingly innovative architectures to process the data in an optimised way for different purposes. The range of these target architectures from Intel and other chip manufacturers includes CPUs (so-called scalar devices), but also integrated or discrete GPUs (vector devices), deep learning hardware (matrix devices) and FPGAs (spatial devices). Only a successful mix of scalar, vector, matrix and spatial architectures (SVMS), delivered via CPUs, GPUs, FPGAs and other specialised accelerators, will provide good overall performance for all workloads[Rei21]. For high-performance connection of the computers in the cluster, oneApi transparently supports many technologies, such as Intel's OmniPath architecture, but also InfiniBand and Ethernet.

A simplified standard programming model that can run on SVMS architectures makes code maintenance much easier for users. It also increases the productivity of developers, because their code can be reused more often and is usable for longer due to the higher abstraction. Incidentally, the training effort is also reduced, because one no longer needs to be familiar with the specifics of GPU Co. and can still program the hardware in a performant way. Figure 1 shows how this works, the parts of it are described with the example of a program doing matrix multiplication more in depths now.

In this example, the consumer application uses in it's business logic a matrix multiplication. This could be done manually (direct programming), or the oneApi library can be used. In this case, the developer does not have to implement the multiplication on his own, instead, the implementations provided from the hardware producers are used. This can be compared with an interface in the object oriented software development - oneApi defines, which functions are needed to implement, and the producers deliver the best implementation for their specific hardware. One goal is for hardware vendors across the industry to develop their own compatible implementations for their CPUs and accelerators. Then programmers could use a single language and a single set of library APIs to program multiple architectures and devices from different vendors in the same operation.

This is precisely the goal of the industry initiative behind oneApi. Based on standards and open specifications, the new cross-architecture language "Data Parallel C++[1]" -

[1]More information about DPC++ in chapter 3

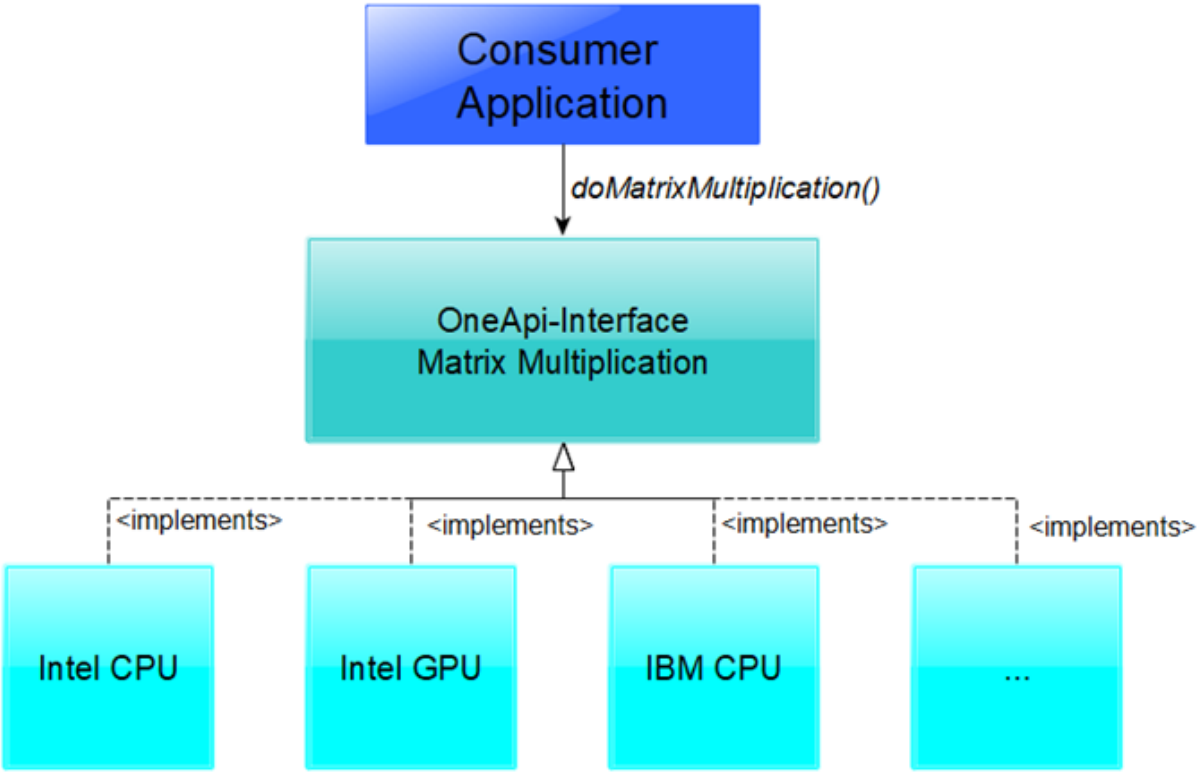


Figure 1: oneApi Concept: Matrix multiplication

DPC++ for short - was defined. In addition, there is a whole series of domain libraries, and even porogramming in Python is also possible.

Parallelism across architectures is achieved through the new programming language DPC++, which in turn is based on the Khronos Group's open language SYCL (More about SYCL in the next chapter). As an abstraction layer, SYCL makes uniform programming of diverse processors possible in the first place[Ben20]. Parallel execution, even across architectural boundaries, is made possible by Intel through the further development of the proven Parallel Studio toolkit, which flows as an integral component into Intel's oneApi implementation. In order to facilitate the heterogeneous programming of parallel processes, oneApi additionally requires a comprehensive software model, which the compilers then fall back on. The result is a single binary for all architectures. This is why the compile and link procedures also differ from usual methods of binary code generation.

2.1 Software Model

The software model is based on the SYCL specification and describes the interaction between host and special hardware in terms of code execution and memory usage. This model consists of four parts, for a sketch see Figure 2:

- A platform model that specifies the host and the device.
- An execution model that specifies the command queues and the commands to be executed on the device.
- A memory model that specifies memory usage between the host and the device

- A kernel model that aligns computational kernels with devices.

Intel has already implemented this open software model and is constantly improving it. Intel's own implementation with the product name "Intel oneApi Toolkit" already has various offshoots and sub-versions.

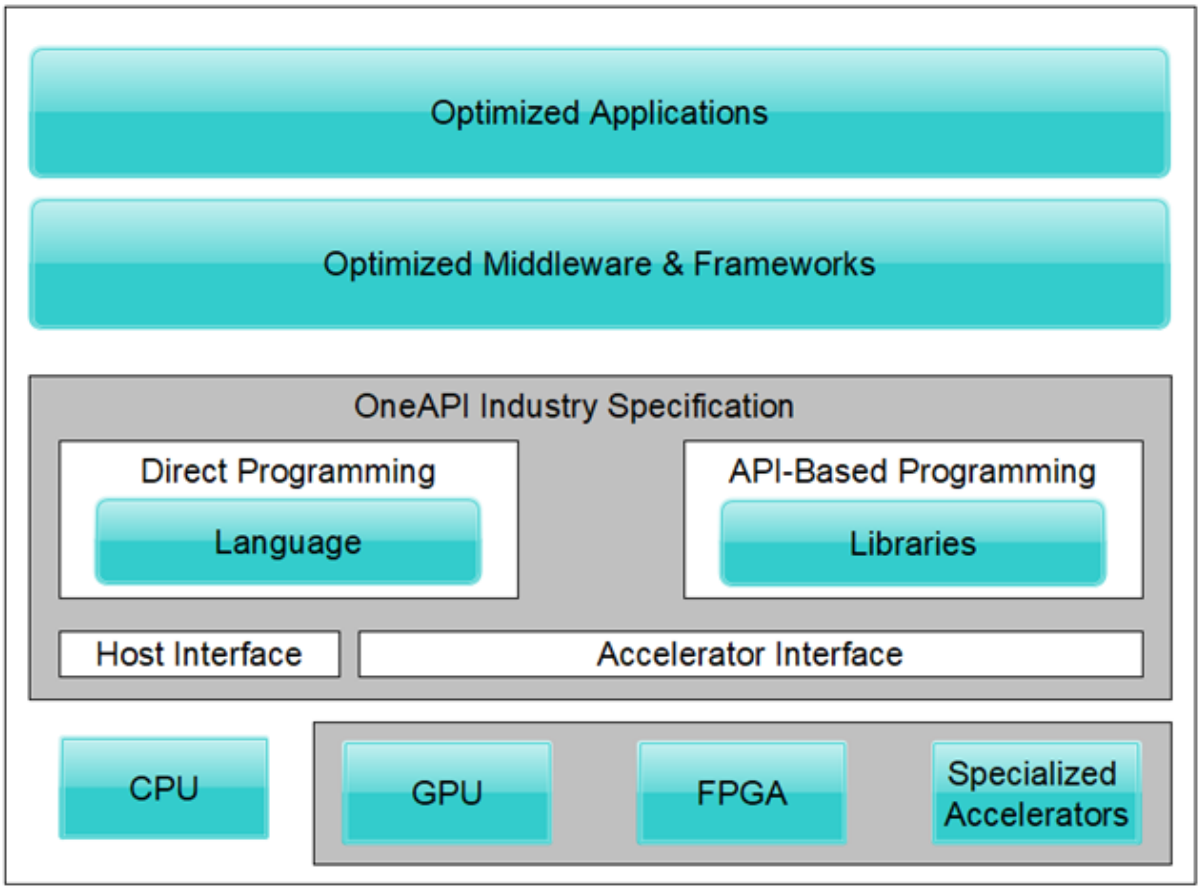


Figure 2: Layer architecture of oneApi

- Optimized Application
  - This module describes the application that the user is developing. Here, for example, the input parameters are defined and the interface to the other components is specified.
- Optimised Middleware and Frameworks
  - oneApi offeres a lot of libraries and frameworks, that can be included into the programming. These include general libraries, for example for high performance computing or interfaces in the field of "internet of things", but also very specific ones such as libraries for high-performance rendering of videos.
- oneApi Industry Specification
  - Here, the actual programming takes place. See the next chapter for more detailed information

  - This can be done with direct programming or Api-based programming

- XPUs
  - The XPU Layer contains all XPUs, on which the code is executed later.
  - This XPUs need to support the oneApi Interface!

Figure 2 shows the various components that are part of the base toolkit (Intel oneApi Base Tookit), especially the compiler DPC++ and the compatibility tool for DPC++, various analysis and debugging tools as well as several optimised libraries, for example for analytics, neural networks/deep learning or video processing.

The Intel oneApi Base Toolkit can be supplemented at with the aforementioned domain-specific toolkits, including the HPC Toolkit, the IoT Toolkit, the DL Framework Developer Toolkit and the Rendering Toolkit. All these toolkits are tailored to different users and different application fields.

The Intel oneApi HPC Toolkit, for example, is especially designed for distributed computing of supercomputing applications. In addition to DPC++, it also supports OpenMP (Open Multi-Processing) version 5.0, a standardised API for shared memory programming in C++, C and Fortran on multiprocessor computers that has been jointly developed by various hardware and compiler manufacturers since 1997. Thanks to OpenMP support, existing code for HPC applications can also be offloaded from the host to a discrete GPU. Users can therefore either switch to using DPC++ or use these offload functions for proven C/C++/Fortran code.

3 Data Parallel C++

Data Parallel C++, short DPC++, is a modern, parallel C++ for heterogeneous architectures. DPC++ combines conventional C++ with SYCL, a cross-platform abstraction layer, and adds further features, like ordered queues or ndrange subgroups (see Figure 3).

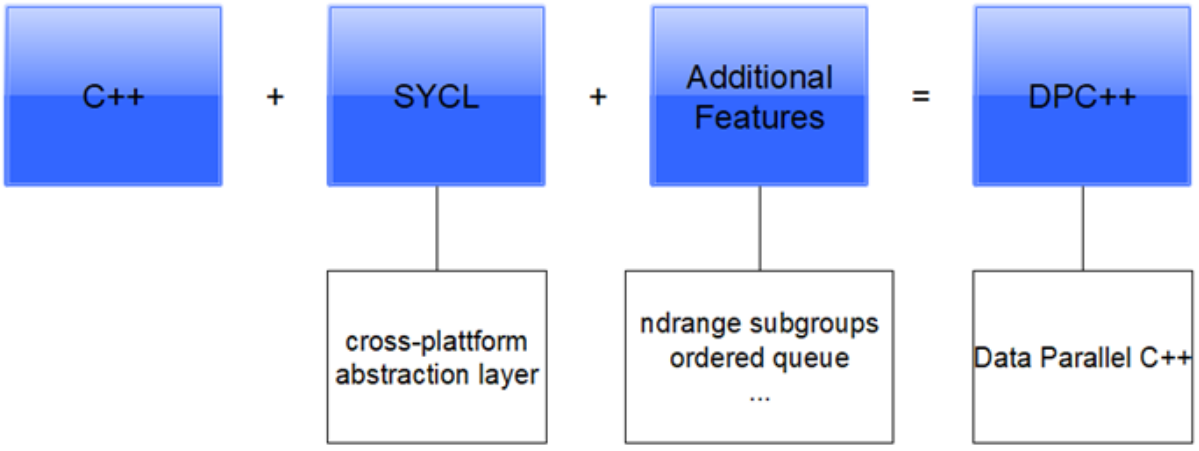


Figure 3: Concept of Data Parallel c++

SYCL (pronounced "sickle") is a free, cross-platform abstraction layer designed on the OpenCL concepts of portability and efficiency, which allows code from heterogeneous platforms to be written in a "single source" style, fully utilizing the C++ standard[Ben20].

SYCL allows the development of a single source in which C++ model functions can contain both host (cpu) and device (gpu/fpga/arm) code in order to build complex algorithms using OpenCL acceleration and thus be able to reuse all of their source code on different types of hardware and data[Aks20]. For better understanding, see this example, taken from the intel training program, which is available online[Ham20]

To add up Vectors in C++, the following code can be used:

```
for (size_t i = 0; i < length; ++i) {
    Z[i] += A * X[i] + Y[i];
}
```

SYCL enables the developer to make this code executed parallel:

```
h.parallel_for<class saxpy> (sycl::range1{length},[=] (sycl::id<1> it) {
    const int i = it[0];
    Z[i] += A * X[i] + Y[i];
});
```

The result of the calculation will be the same, but due to the parallelism way faster, especially when doing bigger calculations.

3.1 Usage in oneApi

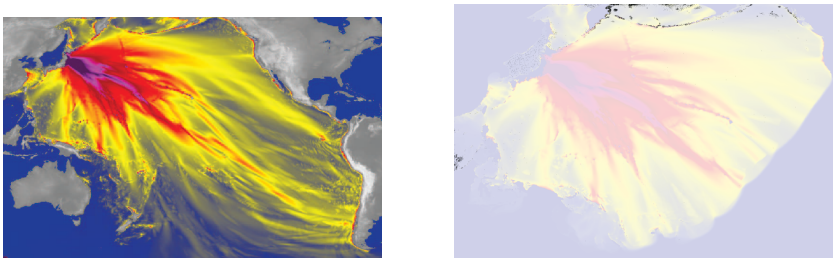
oneApi supports two programming paradigms: Direct Programming and Api Based Programming. When using Direct Programming, the developer does not use additional libraries, instead, all the functionality is written using the basic functions of DPC++. With this approach, the developer has better control about the functionality, but the amount of code (and therefore normaly the amount of time needed) is very high. A commented example of direct programming (matrix multiplication) can be found in Figure 6 in the appendix.

When doing API-Based Programming, the functions are defined in the oneApi-Interface, for example the matrix multiplication, but also more complex libraries (as example, there is a Video Cutting Library for oneApi, which contains functions for rendering, cutting etc.). This approach is normally faster, but the developer has less control about the program. In case of the performance, it should not be worse then direct programming, as all the functionalities are implemented specifically for the existing hardware - which is the main idea behind oneApi. The commented example of this approach one can find in Figure 7, also in the appendix.

Now that the basic concepts behind oneApi have been discussed, the question is of course how well it works in a practical environment. This question was also posed by professors x and y from the Zuse Institute in Berlin, who conducted a study on the subject. This study and its results are presented in the next chapter.

4 Practical Experiment

In their paper "Porting a Legacy CUDA Stencil Code to oneApi", Steffen Christgau and Thomas Steinke from the Supercomputing Department of the Zuse Institute Berlin, the authors "present early experiences when using both the compatibility tool and oneApi



(a) Original Easywave (b) Converted Results

Figure 4: Easywave: Example of a tsunami prediction. Black dots mark the difference between the original calculation and the converted tool.
[CS21]

as well the employed extension to the SYCL programming standard for the tsunami simulation code easyWave"[CS21]. To explore the power and performance of oneApi, the authors first tested the conversion tool from the oneApi toolkit to convert the CUDA-based software "easyWave". Then, as second part of the experiment, they compared performance on different hardware systems, as well as in the cloud-based oneApi Toolkit. The authors described their contributions as following:

- Comparison of the performance on different hardware architectures addressed by OpenMP, CUDA and DPC++/oneApi along with a validation of the computed results.
- Analysis of the DPC++ code's performance portability.
- Summary of challenges with oneApi's compatbility tool dpct, the DPC++ language, and its compiler.

[CS21]

4.1 Conversion tool

EasyWave is an open-source software that can be used to predict the impact of tsunamis. One can get two versions of it, a single threaded version and a Cuda-Based, multithreaded version. An example screenshot of one of those calculations can be seen in figure 4: One can see the center of the wave is near Japan, but the impact reaches the Chinese mainland, the Philippines and even the American west coast and the Cape Horn in South America.

The conversion was done with the "Comnpatibility tool" of the oneApi Dev Toolbox. It only changes code related to CUDA and leaves other parts unmodified. The created code only needed very few manual modifications to be executable (Here it must be mentioned, that the conversion tool is still in the Beta version), and the code was still human readable. As input dataset, e2r4Pacific grid and the z.Tohoku11 seismic event data were used, both can be found at the source repository. The grid has a size of 2851 $\times$ 1801, and represent 10 hours of wave propagation, using 5 minute timesteps. The calculated results were very close (see Figure 5). The only differences appeared at the edges of the waves, and they are very small. In the figure, they are marked with black dots. An example of the code converted is in the Appendix, see Figure 8. At this point, the authors of the study once again explicitly point out that the conversion tool is still in the alpha version and that such errors are therefore to be expected. The fact that such an accurate result could

Parameter	HLRN-IV	DevCloud
CPU	Intel Xeon 6148 (Skylake SP)	Intel Xeon E-2176G (Coffee Lake E)
Frequency	2.4 GHz (base)	3.7 GHz (base)
	3.7 GHz (boost)	4.7 GHz (boost)
Cores	2×20	1×6
Peak Perf. (SP)	1.0 TFLOP/s	259.2 GFLOP/s
Memory BW (th.)	256 GB/s	41.6 GB/s
GPU	Nvidia Tesla V100 (Volta)	Intel UHD Graphics 630 (Gen 9.5/GT2)
Frequency	1246MHz (base)	350MHz (base)
	1380MHz (boost)	1200MHz (boost)
Cores	5120 FP32 Cores	24 Executions Units
Peak Perf. (SP)	14 TFLOP/s	441.6 GFLOP/s
Memory BW (th.)	900 GB/s	41.6 GB/s

Table 1: Technical parameters of the evaluation platforms
[CS21]

already be achieved came as a surprise to the authors, and the inaccuracies were sent to Intel as a bug report so that further optimizations can be made.

4.2 Experiment Setup

The first part of the research, the study of the conversion tool, is successfully completed. Now, the main part of interest will be researched, the performance difference between oneApi and the basic version. For this, the creators of the study created a new version of EasyWave, based on OpenMP, to get a baseline for the performance analysis, before generating the version for Data Parallel C++, so in total, 4 versions on single nodes of the HLRN-IV system in Göttingen as well as the oneApi Dev-Cloud were executed:

- The original, single threaded version of EasyWave (called 'unopt. AoS code' from now on).
- An OpenMP-based, multithreaded version with small changes for better vectorization ('OpenMP SIMD')
- The original CUDA-version ('CUDA')
- The Generated oneApi-Version, using Data Parallel C++ ('DPC++')

The OpenMP SIMD and the DPC++ version were additionally executed under on different hardware systems for further research. Those are listed with the detailed settings in Table 1. Since easyWave is not NUMA-aware, the authors ensured that it runs on the HLRN-IV platform only on one of the processors. Further, this platform node is also equipped with Nvidia Tesla V100 (Volta) GPUs of which a single one was used for GPU runs of easyWave[CS21].

4.3 Experiment Results

The results are listed in table 2, and visualised in Figure 5. The results clearly show that the oneApi versions are nowhere near the perfectly optimized versions of OpenMP SIMD

Environment	Hardware	Program Variant	t in s
HLRN-IV	Skylake Xeon, 1T	unopt. AoS code	90,2
HLRN-IV	Skylake Xeon, 1T	OpenMP SIMD	28,2
HLRN-IV	Skylake Xeon, 20T	OpenMP SIMD	3,8
HLRN-IV	Tesla V100	CUDA	1,5
DevCloud	Lake Xeon 1T	OpenMP SIMD	25,6
DevCloud	Coffee Lake Xeon 6T	OpenMP SIMD	17,7
DevCloud	Coffee Lake Xeon 12T	OpenMP SIMD	18,9
DevCloud	Coffee Lake Xeon	DPC++	22,6
DevCloud	Gen 9.5 Graphics	DPC++	47,1

Table 2: Runtimes on different hardwares
[CS21]

and CUDA. The CUDA version in particular is unsurpassed. On the other hand, the (still unoptimized) DPC++ version on the Coffee Lake Xenon is faster than the corresponding OpenMP version, and only minimally slower than the corresponding OpenMP SIMD version.

Another interesting observation is that the OpenMP SIMD version on the Coffee Lake Xenon with 6 threads is faster than the one on the same environment with 12 threads - there seems to be a problem in the distribution of tasks or the communication between the threads, which negatively affects the performance.

The results of the study also show impressively how much performance can be achieved with, according to the developers, "minimal adjustments" - a speedup of 3 from the unoptimized AoS code to the OpenMP version with one thread, or from 18 to an environment with parallel running calculations speaks a clear language. The speedup of the optimized OpenMP SIMD-versioen from one thread to twenty turns out however smaller: Instead of the optimal speedup of 20, only a speedup of 7 can be achieved. On the server side, a speedup of 1.6 is achieved with the OpenMP SIMD versions from one thread to six, and a speedup of 1.4 from one to twelve.

4.4 Performance Analysis

Based on the results, three main findings can be derived:

1. The software, which was optimised for the specific hardware is still the fastest, especially when not executed on the cloud.

   On the other hand, The generated Version of the software, running on the Coffee Lake Xenon, was faster then single threaded OpenMP version, and way faster then the original, unoptimised version

   The performance of OpenMP with 20 Threads and the CUDA-Version are not reached by a lot by all other settings and versions

2. The difference between the performance of the OpenMP version, and the generated version were very small.

3. The performance differences between the native environments were much bigger then those on the cloud-software.

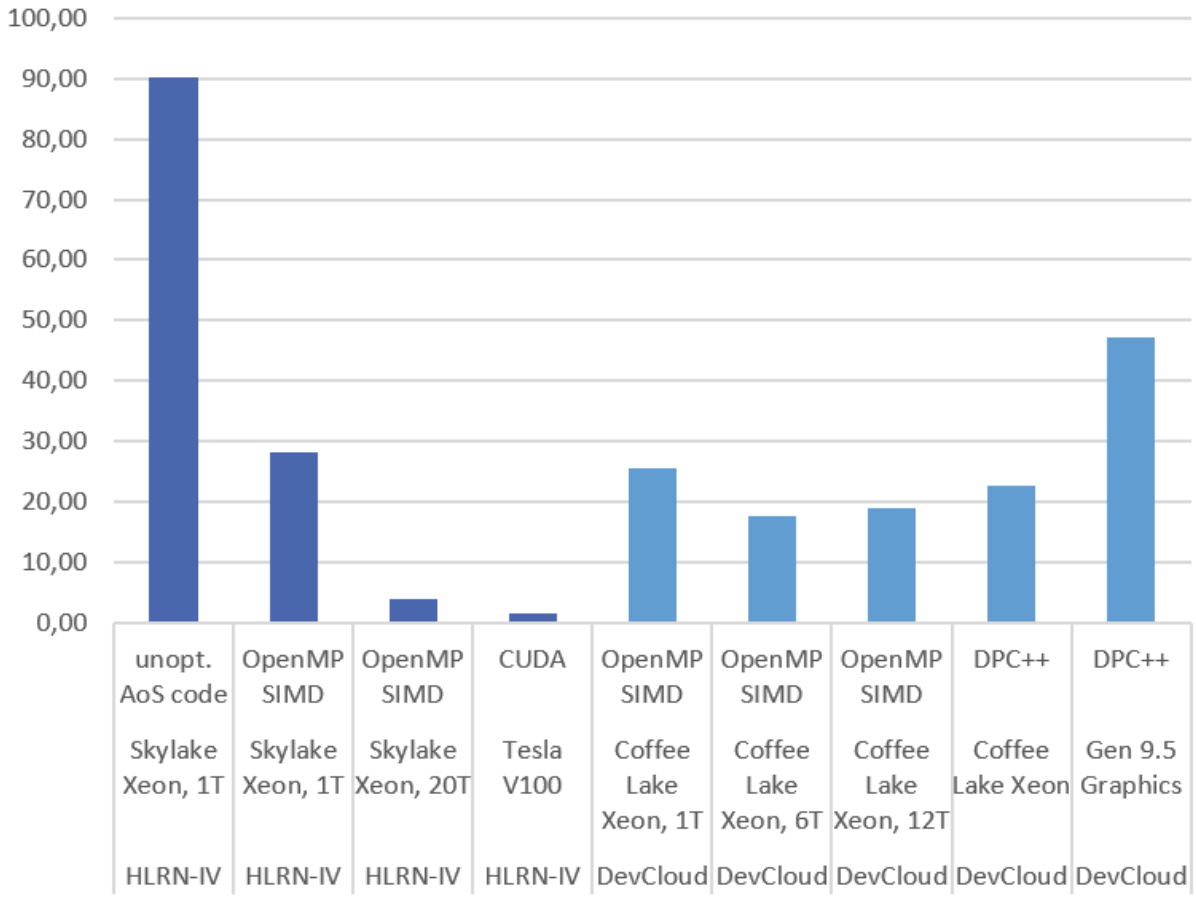


Figure 5: Visualised Results
[CS21]

The performance of OpenMP on the Cloud increases up to 6 Threads, whereas the 12 thread version took a little bit longer.

In general, the report shows, that the versions using DPC++ are not up to the existing environments - but the authors of the report argued, that the converted version can be improved, and that oneApi was still in an early alpha-version, when the research took place. They made the thesis, that the performance will be increased for software, that was developed for the use with oneApi from the beginning on. Compared with the effort, that was needed for the transformation, the performance loss is acceptable.

5 Summary

In this report, the technology oneApi was introduced. oneApi is an interface for hardware-independend programming, provided by intel. In a programm, it is used in three layers, to abstract the XPUs to special middle ware. The basic idead of oneApi is, that the producers of hardware implement the oneApi-interface with specific implementations, which are the optimised use for their specific parts. When using oneApi, the program is using those implementations to optimise the program. The Interface is implemented in Data Parallel C++, a special language which combines C++ with SYCL and some specific functions. In an experiment, experts from the Zuse institute in Berlin researched

the oneApi conversion tool, which can be used to transform CUDA-Applications into oneApi-Software. The transformation was done without bigger problems. Following this, the performance was analysed: The overall performance was not as good as the optimised program versions on native hardware, but overall, the results were satisfying.

6 Prospect

All in all, oneApi is on a good path into the future. Maintaining multiple hardware resources is already very important, and will be even more important in the future. oneApi delivers a solution for this. The test results are also good, and oneApi is still improving, as the measurements were done with the alpha version of oneApi. Here, the support of Intel will be the most important factor, as the company is one of the most important players on the market. On the other hand, oneApi is like other middleware systems: The introduction of another layer in a software architecture always comes with a price, be it additional complexity or poorer performance, or simply additional licensing costs and additional, required know-how of the developers. Whether oneApi can justify this is difficult to foresee, this will show the development on the market under real conditions. Intel seems to be very optimistic in this regard, though - however, it is not foreseeable how the competition will react to Intel's initiative here - the market for technology is very competitive and it is almost impossible to maintain a unique selling proposition for a long time.

References

[Aks20] Vincent Heuveline Aksel Alpay. "SYCL beyond OpenCL: The architecture, current state and future direction of hipSYCL". In: (2020).

[Ben20] Alexey Bader Ben Ashbaugh. "Data Parallel C++: Enhancing SYCL Through Extensions for Productivity and Performance". In: (2020).

[CS21] Steffen Christgau and Thomas Steinke. "Porting a Legacy CUDA Stencil Code to oneAPI". In: (2021).

[Ham20] Jeff Hammond. *A Tutorial for Developing SYCL Kernels*. 2020. URL: `https://www.intel.com/content/www/us/en/developer/articles/training/programming-data-parallel-c.html#gs.9f2132` (visited on 10/30/2022).

[INT20] INTEL. *oneApi.io*. 2020. URL: `https://oneapi.io` (visited on 09/30/2022).

[Mar22] Pablo Antonio Martínez. "Applying Intel's oneAPI to a machine learning case study". In: (2022).

[Pre21] Edmund Preiss. *oneApi.io*. 2021. URL: `https://www.sigs-datacom.de/trendletter/2021-20/4-was-ist-oneapi` (visited on 09/30/2022).

[Rei21] James Reinders. *Data Parallel C++. Mastering DPC++ for Programming of Heterogeneous Systems using C++ and SYCL*. Apress Open, 2021.

[Wan21] Yong Wang; Yongfa Zhou; Qi Scott Wang. "Developing medical ultrasound beamforming application on GPU and FPGA using oneAPI". In: (2021).

A Additional Information

B Code samples

```
using namespace cl:sycl;

// declare host arrays
double *Ahost = new double(M*N);
double *Bhost = new double(N*P);
double *Chost = new double(M*P);

{
    // Init the device queue with a gpu_selector
    // So this code will be executed on a GPU, if one is available
    queue q(gpu_selector());

    // Creating 2D buffers for matrices which are bound to host arrays
    buffer<double, 2> a{Ahost, range<2>{M;N}}
    buffer<double, 2> b{Bhost, range<2>{N;P}}
    buffer<double, 2> c{Chost, range<2>{M;P}}

    // Submitting command group to queue to compute matrix c=a*b
    q.submit([&](handler &h) {
        auto A = a.get_access<access::mode::read>(h);
        auto B = b.get_access<access::mode::read>(h);
        auto C = c.get_access<access::mode::write>(h);

        int WidthA = a.get_range([1]);

        // Execution Kernel, using SYCL
        h.parallel_for<class MatrixMult>(range<2>{M, P}, [=](id<2> index){
            int row = index[0];
            int col = index[1];

            // compute the result of one element in c
            double sum = 0.0;
            for (int i = 0; i y WidthA, i++) {
                sum += A[row][i] ? B[i][col];
            }
            C[index] = sum;
        });
    }
}

// When we exit the block, the buffer destructor will write result back to C
```

Figure 6: Code-Sample: Direct Programming

```
44  using namespace cl:sycl;
45
46  // declare host arrays
47  double *Ahost = new double(M*N);
48  double *Bhost = new double(N*P);
49  double *Chost = new double(M*P);
50
51  {
52      // Init the device queue with a gpu_selector
53      // So this code will be executed on a GPU, if one is available
54      queue q(gpu_selector());
55
56      // Creating 2D buffers for matrices which are bound to host arrays
57      buffer<double, 2> a{Ahost, range<2>{M;N}}
58      buffer<double, 2> b{Bhost, range<2>{N;P}}
59      buffer<double, 2> c{Chost, range<2>{M;P}}
60
61      // Using OneAPI Functionality
62      mkl::transpose nT = mkl::transpose::nontrans;
63      mkl::blas::gemm(q, nT, nT, M, P, N, 1.0, a, M, b, N, 0.0, c, M);
64
65      // What this does:
66      //  void gemm   (queue &exec_queue, transpose transa, transpose transb,
67      //              int64_t m, ing64_t n, int64_t k, T alpha,
68      //              buffer>T, 1  &a, int64_t lda,
69      //              buffer<T, 1> &b, int64t ldb, T beta,
70      //              buffer<T,1> &c int64_t ldc);
71      // call gemm
72  }
73
74  // When we exit the block, the buffer destructor will write result back to C
```

Figure 7: Code-Sample: Api-Based Programming

```
int blocks = N / 1024;
{
    /* CUDA */
    __global__ void kernel(float *v)
    {
        int idx = blockIdx.x * blockSize.x + threadIdx.x;
        v[idx] = sqrt(idx * 1.0);
    }

    float* dev_v;
    cudaMalloc(&dev_v, N * sizeof(*dev_v));
    kernel<<<blocks, 1024>>>(dev_v);
    cudaFree(dev_v);

    /* DPC++ */
    void kernel(float *v, cl::sycl::nd_item<3> item_ct1)
    {
        int idx = item_ct1.get_group(0) *
        item_ct1.get_local_range().get(0) +
        item_ct1.get_local_id(0);

        v[idx] = cl::sycl::sqrt(idx * 1.0);
    }

    *((void **)&dev_v) = cl::sycl::malloc_device(...);
    dpct::get_default_queue_wait().submit(
        [&](cl::sycl::handler &cgh) {
        auto global_rng = cl::sycl::range<3>(blocks,1,1)
        * cl::sycl::range<3>(1024, 1, 1);
        auto local_rng = cl::sycl::range<3>(1024, 1, 1);

        cgh.parallel_for<dpct_kernel_name<
        class kernel_e321fab>>(
            cl::sycl::nd_range<3>(...),
            [=](cl::sycl::nd_item<3> item_ct1) {
            kernel(dev_v, item_ct1);
        });
    }
}
cl::sycl::free(dev_v, ...);
```

Figure 8: Code-Sample: Transformation of Cuda to OneAPI DPC++