Argonne
NATIONAL LABORATORY

ANL/ALCF-19/5

A Case Study with the HACCmk Kernel in SYCL

Argonne Leadership Computing Facility

About Argonne National Laboratory
Argonne is a U.S. Department of Energy laboratory managed by UChicago Argonne, LLC under contract DE-AC02-06CH11357. The Laboratory's main facility is outside Chicago, at 9700 South Cass Avenue, Argonne, Illinois 60439. For information about Argonne and its pioneering science and technology programs, see www.anl.gov.

DOCUMENT AVAILABILITY

Online Access: U.S. Department of Energy (DOE) reports produced after 1991 and a growing number of pre-1991 documents are available free via DOE's SciTech Connect (http://www.osti.gov/scitech/)

Reports not in digital format may be purchased by the public from the National Technical Information Service (NTIS):

U.S. Department of Commerce
National Technical Information Service
5301 Shawnee Rd
Alexandria, VA 22312
www.ntis.gov
Phone: (800) 553-NTIS (6847) or (703) 605-6000
Fax: (703) 605-6900
Email: **orders@ntis.gov**

Reports not in digital format are available to DOE and DOE contractors from the Office of Scientific and Technical Information (OSTI):

U.S. Department of Energy
Office of Scientific and Technical Information
P.O. Box 62
Oak Ridge, TN 37831-0062
www.osti.gov
Phone: (865) 576-8401
Fax: (865) 576-5728
Email: **reports@osti.gov**

Disclaimer
This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor any agency thereof, nor UChicago Argonne, LLC, nor any of their employees or officers, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof. The views and opinions of document authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof, Argonne National Laboratory, or UChicago Argonne, LLC.

Argonne NATIONAL LABORATORY

ANL/ALCF-19/5

A Case Study with the HACCmk Kernel in SYCL

prepared by Zheming Jin

Argonne Leadership Computing Facility, Argonne National Laboratory

December 1, 2019

A Case Study with the HACCmk Kernel in SYCL

I. Introduction

The SYCL standard specifies a cross-platform abstraction layer that enables programming of heterogeneous computing systems using standard C++ [1]. In the Open Computing Language (OpenCL) programming model, host and device codes are written in different languages [2]. To improve development productivity and performance portability, the SYCL programming model combines host and device codes for an application in a type-safe way.

In this report, we are interested in applying the SYCL programming model to a computationally intensive routine derived from the Hardware Accelerated Cosmology Code (HACC) framework for a study on performance portability on a heterogeneous computing device. We use Intel® OneAPI toolkit [3] to build the OpenCL and SYCL programs, and evaluate the performance of both implementations on Intel® integrated GPUs.

We find that the SYCL implementation can achieve the same performance as the OpenCL implementation after we specify a target backend for the compiler to build the SYCL program in offline compilation. Without the specification of a target backend, the runtime will have to compile intermediate device codes for a target platform. Such runtime overhead may become significant compared to the execution of a kernel on a target device. As the kernel routine is compute-bound, we evaluate the impact of the number of compute units upon the kernel's raw performance using two GPUs. When the hardware resource is not underutilized, we can obtain almost linear performance speedup from 48 compute units to 72 compute units in offline compilation, which indicates that the number of compute units on a GPU are important to improving the raw performance of a compute-bound kernel. The experimental results show that SYCL is a promising programming model for heterogeneous computing.

The remainder of the report is organized as follows: In Section II, we describe the SYCL programming model, the steps to map an OpenCL program to a SYCL program, and the architecture of an integrated GPU. Section III introduces the kernel routine, and presents the OpenCL and SYCL implementations of the kernel. In Section IV, we evaluate the performance of the implementations on the GPUs. Section V concludes the report.

II. Background

A. SYCL

Many C++ programming models for hardware accelerators, such as C++ AMP, CUDA, Thrust C++, are single source [4]. Such languages are easy to use and can be type-checked as everything is in one source file. They facilitate offline compilation so that the binary can be checked at compile time. A SYCL program, which is based on a single-source C++ model as shown in Figure 1, can be compiled for a host while kernel(s) are extracted from the source and compiled for a device. A SYCL device compiler parses a SYCL application and generates intermediate representations (IR). A standard host compiler parses the same application to generate native host code. The SYCL runtime will load IR at runtime, enabling other compilers to parse it into native device code. Hence, the flow allows people to continue to use existing toolchains for a host platform, and choose preferred device compilers for a target platform.

The design of SYCL allows for the combination of the performance and portability features of OpenCL and the flexibility of using high-level C++ abstractions. Most of the abstraction features of C++, such as templates, classes, and operator overloading, are available for a kernel function in SYCL. Some C++ language features, such as virtual functions, virtual inheritance, throwing/catching exceptions, and run-time type-information, are not allowed inside kernels due to the capabilities of the underlying OpenCL standard. These features are available outside kernel scope.

A SYCL application is logically structured in three scopes: application scope, command-group scope, and kernel scope. The kernel scope specifies a single-kernel function that will be executed on a device after compilation. The command-group scope specifies a unit of work that will comprise of a kernel function and buffer accessors. The application scope specifies all other codes outside of a command-group scope. A SYCL kernel function may be defined by the body of a lambda function, by a function object or by the binary generated from an OpenCL kernel string. Although an OpenCL kernel is interoperable in the SYCL programming model, in this report we focus on the implementation of kernel functions using lambda functions.

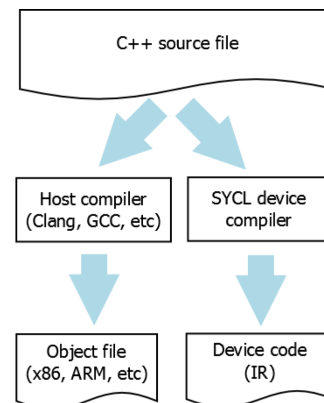

Figure 1. SYCL is a single-source programming model

TABLE I. MAPPING FROM OPENCL TO SYCL

Step	OpenCL	SYCL
1	Platform query	Device selector class
2	Device query of a platform	
3	Create context for devices	
4	Create command queue for context	Queue class
5	Create memory objects	Buffer class
6	Create program object	Lambda expressions
7	Build a program	
8	Create kernel(s)	
9	Set kernel arguments	
10	Enqueue a kernel object for execution	Submit a SYCL kernel to a queue
11	Transfer data from device to host	Implicit via accessors
12	Event handling	Event class
13	Release resources	Implicit via destructor

Table I lists the steps of creating an OpenCL application and their corresponding steps in SYCL. The first three steps in OpenCL are reduced to the instantiation of a device selector class in SYCL. A selector searches a device of a user's provided preference (e.g., GPU) at runtime. The SYCL queue class is an out-of-order queue that encapsulates a queue for scheduling kernels on a device. A kernel function in SYCL can be invoked as a lambda function. The function is grouped into a command group object, and then it is submitted to execution via command queue. Hence, steps 6 to 10 in OpenCL are mapped to the definition of a lambda function and submission of its command group to a SYCL queue. Data transfer between host and device can be implicitly realized by SYCL accessors, and the event handling can be handled by SYCL event class. Releasing the allocated sources of queue, program, kernel, and memory objects in SYCL is handled by the runtime which implicitly calls destructors inside scopes. Compared to the number of OpenCL programming steps, the SYCL programming model reduces the number of programming steps by half with higher abstractions, offloading the burden of managing OpenCL devices, program objects, kernels, and memory objects to a compiler.

B. *Integrated GPU*

While mainstream integrated GPUs are produced by ARM, Intel®, and NVIDIA, we give a summary of the architecture of an Intel® integrated GPU on which we obtain our experimental results. Such class of GPUs, with a central processing unit (CPU) and a GPU integrated on the same chip, is commonly used in laptops, desktop computers, and low-cost servers. While they are not designed to outperform discrete GPUs due to the power, area, and thermal constrains, there is a need to have a good knowledge of a processor with an integrated GPU.

This kind of GPU connects to CPU cores via a ring interconnect, and they share main memory with CPU cores. To reduce data access latency from main memory, an integrated GPU maintains a memory hierarchy comprised of register files, instruction and data caches. Some products include an embedded DRAM (eDRAM) behind a last-level cache to further reduce latency to system memory for higher effective bandwidth. The building block of the graphics compute architecture is the execution unit (EU). It is a combination of simultaneous multi-threading and fine-grained interleaved multi-threading. Each EU can run seven threads concurrently to hide memory access latency. Each thread has 128 SIMD-8 32-bit registers. Each EU can co-issue to four instruction processing units (IPUs) including two floating-point units (FPUs), a branch unit for branch instructions, and a message unit for memory operations, sampler operations, and other longer-latency system communications. Each FPU supports both floating-point and integer operations, and can execute up to four 32-bit floating-point or integer operations. However, only one FPU provides support for transcendental math functions and double-precision floating-point operations.

From the perspective of an OpenCL kernel, multiple kernel instances, which are equivalent to OpenCL work-items, are executed simultaneously within a hardware thread. For a SIMD-16 compile of a kernel, 112 kernel instances can be executing concurrently on an EU. If there is a divergent branch in one or more kernel instances, an EU's branch unit keeps track of such divergence and generates masks to control which instances need to execute the branch.

Arrays of EUs are organized as a subslice. The number of EUs per subslice depend on the generation of compute architecture. Each subslice contains a thread dispatcher unit and supporting instruction caches. In addition, it includes a sampler unit for sampling texture and image surfaces, and a data port memory management unit for a variety of general-purpose buffer accesses, scatter/gather operations, as well as shared memory accesses. Subslices are grouped into slices. In general, a slice has three subslices, but the number of slices depend on products and their generations. A slice integrates additional logics for thread dispatch routing, banked L3 data cache, banked shared memory, and fixed function logic for atomics and barriers.

III. IMPLEMENTATIONS OF HACCMK KERNEL

The HACCmk kernel is publicly available in the CORAL benchmark suite [5]. The size of the kernel is approximately 250 lines of code. The kernel consists of two loops: the outer loop, which is parallelized with the OpenMP directive `omp parallel for` [6] iterates over the particles. Each iteration contains a function call to the major computational kernel. The function contains a loop over particles for computing the short-forces between the particles. The forces are computed with single-precision floating-point operators. As the kernel has two nested loops over all the particles, the complexity of the algorithm is $O(n^2)$. A summary of the kernel is available in [7]. In the following sections, we will focus on the implementation of the HACCmk kernel using SYCL and OpenCL.

A. *OpenCL Kernel Program*

As shown in Listing 1, the implementation of the OpenCL kernel is a parallel design in which each work-item corresponds to the workload of an iteration of the outer loop.

```
kernel void haccmk(const int count,
                   const float fsrrmax2,
                   const float mp_rsm2,
                   const float fcoeff,
                   global float* restrict xx1,
                   global float* restrict yy1,
                   global float* restrict zz1,
                   global float* restrict mass1,
                   global float * restrict vx2,
                   global float * restrict vy2,
                   global float * restrict vz2 )
{
  const float ma0 = 0.269327f;
  const float ma1 = -0.0750978f;
  const float ma2 = 0.0114808f;
  const float ma3 = -0.00109313f;
  const float ma4 = 0.0000605491f;
  const float ma5 = -0.00000147177f;

  int i  = get_global_id(0);
  float dxc, dyc, dzc, m, r2, f, xi, yi, zi;
  xi = 0.f;
  yi = 0.f;
  zi = 0.f;
  float xxi = xx1[i];
  float yyi = yy1[i];
  float zzi = zz1[i];
  for ( int j = 0; j < count; j++ ) {
    dxc = xx1[j] - xxi;
    dyc = yy1[j] - yyi;
    dzc = zz1[j] - zzi;
    r2 = dxc * dxc + dyc * dyc + dzc * dzc;
    if ( r2 < fsrrmax2 ) m = mass1[j];
    else m = 0.f;
    f = r2 + mp_rsm2;
    f = m * ( 1.f / ( f * sqrt( f ) ) -
        ( ma0 + r2*(ma1 + r2*(ma2 + r2*(ma3 +
          r2*(ma4 + r2*ma5))))));
    xi = xi + f * dxc;
    yi = yi + f * dyc;
    zi = zi + f * dzc;
  }
  vx2[i] = vx2[i] + xi * fcoeff;
  vy2[i] = vy2[i] + yi * fcoeff;
  vz2[i] = vz2[i] + zi * fcoeff;
}
```

Listing 1. The HACCmk kernel in OpenCL

Hence, it is an N-dimensional range kernel whose global work size (i.e., the number of work-items) is equal to the number of particles specified in the outer loop. We refer to this value as "nParticles". These work-items in the global work space can execute independently with each other.

Compared to the original kernel written in C, the function signature is modified to be compatible with OpenCL kernel syntax; `xx1`, `yy1`, `zz1`, `mass1`, `vx2`, `vy2`, `vz2` arrays are specified in the global memory address space with `__global`. The `restrict` keyword is added for each global memory address pointer to prevent the compiler from creating unnecessary memory dependencies between non-conflict memory load and store operations. Each unique work-item is identified by querying the OpenCL API function. The "Step10" subroutine in the original kernel is flattened into the OpenCL kernel.

```
1  q.submit([&](cl::sycl::handler& cgh) {
2    auto acc_m      =
       buf_m.get_access<sycl_read>(cgh);
3    auto acc_fsrmax =
       buf_fsrmax.get_access<sycl_read>(cgh);
4    auto acc_mp_rsm =
       buf_mp_rsm.get_access<sycl_read>(cgh);
5    auto acc_fcoeff =
       buf_fcoeff.get_access<sycl_read>(cgh);
6    auto acc_xx     =
       buf_xx.get_access<sycl_read>(cgh);
7    auto acc_yy     =
       buf_yy.get_access<sycl_read>(cgh);
8    auto acc_zz     =
       buf_zz.get_access<sycl_read>(cgh);
9    auto acc_mass   =
       buf_mass.get_access<sycl_read>(cgh);
10   auto acc_vx2    =
       buf_vx2.get_access<sycl_read_write>(cgh);
11   auto acc_vy2    =
       buf_vy2.get_access<sycl_read_write>(cgh);
12   auto acc_vz2    =
       buf_vz2.get_access<sycl_read_write>(cgh);
13   cgh.parallel_for<class HACCmk>(nParticles,
                           [=](cl::sycl::id<1> i) {
14     const float ma0 = 0.269327f;
15     const float ma1 = -0.0750978f;
16     const float ma2 = 0.0114808f;
17     const float ma3 = -0.00109313f;
18     const float ma4 = 0.0000605491f;
19     const float ma5 = -0.00000147177f;
20     float dxc, dyc, dzc, m, r2, f, xi, yi, zi;
21     xi = 0.f; yi = 0.f; zi = 0.f;
22     float xxi = acc_xx[i];
23     float yyi = acc_yy[i];
24     float zzi = acc_zz[i];
25     float fsrrmax2 = acc_fsrmax[0];
26     float mp_rsm2 = acc_mp_rsm[0];
27     float fcoeff2 = acc_fcoeff[0];
28     int   count = acc_m[0];
29     for ( int j = 0; j < count; j++ ) {
30       dxc = acc_xx[j] - xxi;
31       dyc = acc_yy[j] - yyi;
32       dzc = acc_zz[j] - zzi;
33       r2 = dxc * dxc + dyc * dyc + dzc * dzc;
34       if ( r2 < fsrrmax2 ) m = acc_mass[j];
35       else m = 0.f;
36       f = r2 + mp_rsm2;
37       f = m * ( 1.f / ( f * cl::sycl::sqrt( f ) )
           - ( ma0 + r2*(ma1 + r2*(ma2 +
           r2*(ma3 + r2*(ma4 + r2*ma5))))));
38       xi = xi + f * dxc;
39       yi = yi + f * dyc;
40       zi = zi + f * dzc;
41     }
42     acc_vx2[i] = acc_vx2[i] + xi * fcoeff2;
43     acc_vy2[i] = acc_vy2[i] + yi * fcoeff2;
44     acc_vz2[i] = acc_vz2[i] + zi * fcoeff2;
45   });
46 });
```

Listing 2. Implementation of the HACCmk kernel routine in SYCL

B. SYCL Kernel Program

Listing 2 shows the SYCL implementation of the HACCmk kernel under the scope of command group. The function object is given a command group handler object (`cgh`) to perform all the necessary work required to process data on a device using a kernel. The group of commands for data transferring and processing is enqueued as a command

group on a device. A command group is submitted to a SYCL command queue for execution.

A SYCL buffer differs from an OpenCL buffer in that it can handle both storage and ownership of data. A device accessor in SYCL allows a kernel, which is defined in a lambda function, to access data stored in a device buffer. Accesses to the buffers are controlled via device accessors constructed through the `get_access` method of the buffers (lines 2-12). For brevity, we use `sycl_read` and `sycl_read_write` to represent the access mode `cl::sycl::access::mode::read` and `cl::sycl::access::mode::read_write`, respectively. In the kernel function, the identifiers of global work-items are retrieved with the member functions of the `id` class (line 13). The floating-point math function (line 36) needs to be qualified in the SYCL namespace to tell a SYCL compiler that the math function, which is not confused with the math function called on a host, will be executed on a device. Overall, it is relatively straightforward to port an OpenCL application to a SYCL application by following the steps listed in Table I.

IV. EXPERIMENT

A. Setup

We choose two server platforms in our experiment. One server has an Intel® Xeon® E3-1284L v4 CPU running at 2.9 GHz. The CPU has four cores and each core supports two threads. The integrated GPU is Broadwell GT3e, Generation 8.0. It contains 48 execution units, and each execution unit corresponds to a compute unit in the OpenCL programming model. The maximum dynamic frequency of the GPU is 1.15 GHz. The other server has an Intel® Xeon® E3-1585 v5 CPU running at 3.5 GHz. The CPU also has four cores and each core supports two threads. The integrated GPU is Skylake GT3e, Generation 9.0. It contains 72 execution units. The maximum dynamic frequency of the GPU is 1.15 GHz. A few details of the two GPUs are listed in Table II.

TABLE II. SUMMARY OF THE TWO GPUS

Parameter	Iris™ Pro Graphics P580	Iris™ Pro Graphics P6300
Generation	Gen9 (Skylake)	Gen8 (Broadwell)
Technology	14 nm	14 nm
Base Freq.	0.35 GHz	0.3 GHz
Max Dynamic Freq.	1.15 GHz	1.15 GHz
Embedded DRAM	128 MB	128 MB
Slices/Subslices	3/9	2/6
Execution Units	72	48
Max GFLOPS	1325	883

For the GPU compute runtime, the device version is OpenCL 2.1 NEO and the driver version 19.43.14583. The maximum work-group size on the GPUs is 256. Empirical results show that the runtime can select an appropriate work-group size; therefore we have the OpenCL implementation determine how to break the global work-items into appropriate work-group instances. In our test, the number of particles iterated in the inner loop is not fixed at 15,000. We will evaluate the impact of the inner loop count upon the application performance.

We build the OpenCL and SYCL programs using the Intel® OneAPI toolkit [3] released recently. The SYCL compiler supports two compilation modes. It can compile device code into a device-agnostic form that can run on any compatible devices. This is known as online compilation as the device-agnostic code is compiled into a device-specific form at runtime. In addition, the compiler allows production of device-specific code at compile time. This process is known as offline compilation. We compile both programs using the optimization option "-O3". Besides the OpenCL just-in-time (online) compilation, we use an OpenCL offline compiler to generate an intermediate representation from the kernel for offline compilation.

We measure the elapsed time of executing the host application as our performance metric. The host application includes the initialization of OpenCL/SYCL runtime, the construction of device buffers, data transfers from the host to device, kernel execution on the device, and the return of results from the device to host. The construction and initialization of host buffers, and checking the status after invoking each OpenCL built-in function are not included in timing. We do not just consider kernel execution time on a device as the performance metric due to the potentially significant offloading overhead.

B. Experimental Results

We are interested in evaluating the impact of inner loop count upon the performance of the application on the two GPUs. Hence, in our test the global work size is 256 and the number of particles iterated in the inner loop ranges from 1,875 to 60,000.

Table III shows the execution time in millisecond of the OpenCL and SYCL applications after the program is

TABLE III. THE EXECUTION TIME IN MILLISECOND ON THE P580 GPU. THE GLOBAL WORK SIZE IS 256.

ILP	OpenCL (online)	OpenCL (offline)	SYCL (online)	SYCL (offline)
1.875K	217	1.3	81	1.2
3.75K	218	2.2	82	2.1
7.5K	221	3.7	84	3.8
15K	223	7.2	87	7.1
30K	231	14	94	13.6
60K	244	27	107	26

TABLE IV. THE EXECUTION TIME IN MILLISECOND ON THE P6300 GPU. THE GLOBAL WORK SIZE IS 256.

ILP	OpenCL (online)	OpenCL (offline)	SYCL (online)	SYCL (offline)
1.875K	264	1.1	85	1.1
3.75K	264	1.8	88	1.9
7.5K	268	3.5	89	3.5
15K	269	6.7	92	6.8
30K	275	13.2	98	13.3
60K	289	26.3	112	26.3

TABLE V. COMPARISON OF THE EXECUTION TIME ON THE TWO GPUS. THE GLOBAL WORK SIZE IS 8192 AND THE ILP IS 15,000.

GPU	OpenCL (online)	OpenCL (offline)	SYCL (online)	SYCL (offline)
P6300	275	12.7	97	12.4
P580	225	8.9	89	8.5
Speedup	1.2	1.4	1.1	1.5

compiled using the offline and online modes on the P580 GPU. While the execution time of both programs is almost the same in offline compilation, the SYCL program is on average 2.5X faster than the OpenCL program in online compilation. On the other hand, the execution time is approximately linear with the inner loop count (ILP) when the overhead of compiling IR for the target GPU is eliminated in offline compilation.

Table IV shows the execution time in millisecond of the OpenCL and SYCL applications after the program is compiled using the offline and online modes on the P6300 GPU. The execution time of both programs is almost the same in offline compilation. The SYCL program is on average 2.9X faster than the OpenCL program in online compilation. The execution time of the OpenCL and SYCL programs in online compilation is on average 20% and 5.5% longer than that on the P580 GPU, respectively. The execution time is also approximately linear with the ILP when the compilation overhead is eliminated in offline compilation.

The results show that the overhead of online compilation can become significant compared to the offline compilation and/or kernel execution. Comparing the execution time in offline compilation on the two GPUs, we do not observe any performance improvement though the P580 GPU has 1.5X more execution units. This indicates that executing 256 work-items does not fully utilize 72 execution units on the GPU.

Table V shows the execution time in millisecond of the OpenCL and SYCL applications when the global work size is 8,192 and the ILP is fixed at 15,000. Due to the significant overhead of online compilation, the speedup is approximately 1.2. However, the performance speedup is close to 1.5 in offline compilation. Assuming the execution units are fully utilized, the number of compute units on a GPU are important to improving the raw performance of a compute-bound kernel.

V. CONCLUSION

Unlike OpenCL, SYCL is a single-source programming model that allows kernel codes to be embedded in host codes. In this report, we describe the SYCL programming model and list the migration steps from OpenCL to SYCL. While the transformation is relatively straightforward given that the SYCL programming model is an extension to OpenCL, understanding buffer accessors, kernel execution model, and program scopes is important for the smooth transformation of the applications. When comparing the offline and online compilations, we find that the overhead of online compilation may become significant compared to offline compilation and kernel execution on a device. However, the SYCL implementations are as fast as the OpenCL implementations in offline compilation on the GPUs. In addition, the number of execution units is important to the performance of a compute-bound kernel when the kernel can fully utilize hardware resources on a GPU.

SYCL is a promising programing model in terms of performance portability and programming productivity. The maturing SYCL compiler will continue to promote performance, portability, and productivity.

ACKNOWLEDGMENT

Results presented were obtained using the Chameleon testbed supported by the National Science Foundation. This research used resources of the Argonne Leadership Computing Facility, which is a DOE Office of Science User Facility supported under Contract DE-AC02-06CH11357.

REFERENCES

[1] https://www.khronos.org/registry/SYCL/specs/sycl-1.2.1.pdf

[2] Stone, J.E., Gohara, D. and Shi, G., 2010. OpenCL: A parallel programming standard for heterogeneous computing systems. Computing in science & engineering, 12(3), p.66.

[3] https://software.intel.com/sites/default/files/oneAPIProgrammingGuide_5.pdf

[4] Wong, M., Richards, A., Rovatsou, M. and Reyes, R., 2016. Khronos's OpenCL SYCL to support heterogeneous devices for C++.

[5] https://asc.llnl.gov/CORAL-benchmarks/

[6] Chapman, B., Jost, G. and Van Der Pas, R., 2008. Using OpenMP: portable shared memory parallel programming (Vol. 10). MIT press.

[7] https://asc.llnl.gov/CORAL-benchmarks/Summaries/HACCmk_Summary_v1.0.pdf

Argonne Leadership Computing Facility
Argonne National Laboratory
9700 South Cass Avenue, Bldg. 240
Argonne, IL 60439

www.anl.gov

Argonne National Laboratory is a U.S. Department of Energy
laboratory managed by UChicago Argonne, LLC