UnoAPI: Balancing Performance, Portability, and Productivity (P3) in HPC Education

Konstantin Läufer and George K. Thiruvathukal
Loyola University Chicago
Software and Systems Laboratory
Department of Computer Science
{gkt,laufer}@cs.luc.edu

***Abstract*—oneAPI is a major initiative by Intel aimed at making it easier to program heterogeneous architectures used in high-performance computing using a unified application programming interface (API). While raising the abstraction level via a unified API represents a promising step for the current generation of students and practitioners to embrace high-performance computing, we argue that a curriculum of well-developed software engineering methods and well-crafted exemplars will be necessary to ensure interest by this audience and those who teach them. We aim to bridge the gap by developing a curriculum—codenamed UnoAPI—that takes a more holistic approach by looking beyond language and framework to include the broader development ecosystem, similar to the experience found in popular HPC languages such as Python. We hope to make parallel programming a more attractive option by making it look more like general application development in modern languages being used by most students and educators today. Our curriculum emanates from the perspective of well-crafted exemplars from the foundations of computer systems—given that most HPC architectures of interest begin from the systems tradition—with an integrated treatment of essential principles of distributed systems, programming languages, and software engineering. We argue that a curriculum should cover the essence of these topics to attract students to HPC and enable them to confidently solve computational problems using oneAPI. By the time of this submission, we have shared our materials with a small group of undergraduate sophomores, and their responses have been encouraging in terms of self-reported comprehension and ability to reproduce the compilation and execution of exemplars on their personal systems. We plan a follow-up study with a larger cohort by incorporating some of our materials in our existing course on High-Performance Computing.**

I. Introduction

In recent years, a number of frameworks and languages have emerged to improve the level of abstraction in parallel computing. Although parallel computing itself has reached a high level of maturity, as we move toward exascale and beyond computing, the challenges that plagued the earliest days of parallel and distributed computing are beginning to resurface, in particular, how to leverage—and manage—heterogeneity. Heterogeneity manifests itself in an altogether new way in today's systems, where one or more architectures may be present within a single system. That is, it is not uncommon to find systems that have conventional CPU cores combined with accelerators such as GPUs and FPGAs.

One of the key frameworks to emerge in response to the heterogeneity challenge is oneAPI, which promises to be a single application programming interface (API) that can be used to program all available architectures. Programming is done using a modern dialect of C/C++, known as Data-Parallel C++ and SYCL, a standard for higher-level abstractions for parallelism and concurrency. While oneAPI is not the only alternative, the ability to write largely device-independent programs is a promising direction with great potential to improve the way we teach the current and future generations of students how to exploit parallelism as the world moves toward more "on chip" heterogeneity and clusters thereof.

While this development is promising, we contend that raising the level of abstraction within a programming language or accompanying set of libraries is likely going to prove insufficient for reaching a new generation of developers, especially if the goal is to create *applications*. Anecdotally, students today—and their instructors, including the co-authors—are attracted to higher-level languages that are actively being taught in universities and in demand by the general computing industry. Fewer and fewer of them are learning low-level languages and have come to expect much more when it comes to a language. Virtually all modern languages have mature ecosystems with easy access to many libraries via repositories such as Maven (Java and other JVM-based languages), PyPI (Python), NPM (Node/JavaScript), and Go (with its integrated source-based management).

On the C++ side, it is important to understand that the language has undergone substantial modernization in recent decades. The ISO/ANSI standard was first completed in 1998 and amended in 2003. However, in 2011, the C++-11 standard was developed, which incorporated many innovations associated with modern languages (including C# and Java):

- New *foreach* loop syntax to visit elements of a collection in natural order
- Improved initialization syntax, especially for unions and arrays
- Automatic variable type inference (`auto` keyword)
- Variadic templates for increased parametric code reuse
- Improved libraries for time, atomics, regex, etc.
- Threading library (allowing code to be written without direct use of Pthreads)

Recent language changes have introduced ideas from *functional programming*, including lambda expressions. While

these features may be unfamiliar to established practitioners of C++, we introduce them to all of our students enrolled in Java-based data structures courses—COMP 271 and 272 prerequisites—so they can make the transition to C++. Reminiscent of the days of Oldsmobile advertising, "This is not your father's (parent's) Oldsmobile," it can safely be said that modern C++ is not your parent's C++ programming language or the one we used while we were graduate students in the 1990s. Learning modern C++ requires directing significant curricular effort toward making sure students and educators alike understand how the concepts associated with modern languages are crucial to understanding and writing oneAPI programs.

Apart from the language, however, learning how to incorporate third-party libraries—also written in C++—is of great practical importance. While this may appear to be just an implementation detail, the joy of working with modern languages lies in being able to take advantage of external dependencies when creating applications. Perhaps no other language represents this ethos better than Python with its *batteries included* approach [1], which contributes greatly to its popularity among students, educators, and HPC researchers. Until relatively recently, working with third-party libraries in C++ required a great deal of effort. And given that many students will find themselves needing to take advantage of a computing cluster (e.g. Intel DevCloud) where they will not have root access, being able to work with external dependencies is a must. The days of from-scratch Makefiles can and should be replaced with more modern build systems, e.g., CMake and Bezel. In line with modern build systems from other programming languages, CMake is able to support fetching and building of external libraries/dependencies without having to get a sysadmin to set up additional libraries on the system. We leverage CMake's ability in this regard to include capabilities such as performance timing, logging, command-line argument parsing, and unit testing to our curricular examples. While the addition of CMake and dependency management might appear at first glance to introduce complexity, the end result suggests otherwise. When making use of external C++ libraries, students can leverage important capabilities with clear, comprehensible code and break free of writing *ad hoc* solutions to common problems from scratch. This allows more effort to be put into focusing on the scientific/mathematical problems to be solved instead of already solved problems.

Lastly, we maintain and release all of our exemplars—current and emerging—on GitHub, including the integration of these examples into an online book. We use continuous integration (CI) to ensure that all examples build with the major releases of oneAPI and pass all tests. The intent is to create both a living and lasting work that can evolve not only through our efforts but with students, researchers/practitioners, and other educators alike.

Our hope is to create a curriculum that will not only teach oneAPI and its components but also result in the least amount of frustration for students and their educators, who also may need to come up to speed on the latest advancements in C++ and modern software tools that support it.

II. Background and Related Work

A growing challenge for the HPC community has been *performance portability* [2], [3], i.e., the ability to maintain consistent application performance in the context of increasing complexity and heterogeneity of hardware, as well as the additional need to maintain developer productivity and computational precision [4]. The intersection of performance, portability, and productivity goals (P3) has also been the focus of an annual workshop at the Supercomputing Conference (SC) for several years.

While earlier approaches to heterogeneous computing focused more on task parallelism [5], the wider availability of general-purpose accelerators, such as GPUs and FPGAs, has lead to a strong focus on data parallelism [6]–[8]. The computer science education community has responded by incorporating data parallelism into a variety of recent curricular offerings [9]–[13]. On the language side, the community has also incorporated modern C++ into the curriculum [14], [15], as well as other performance-oriented modern languages, e.g., Rust [16]. Additional curricular innovation has occurred in applications of data-parallel computing [17], [18].

The institutional context for our UnoAPI proposal is our existing undergraduate elective course COMP 364: High-Performance Computing, which broadly corresponds to the Parallel Programming (ParProg) course from the TCPP Core Topics report [19]. We have offered this course almost every fall semester since 2015 to approximately 15 students per year. The course uses Eijkhout's introduction to HPC [20] and emphasizes scientific computing with the following learning objectives. Figure 1 shows the course in the context of its prerequisites with relevant learning objectives.

- Learn how to analyze the scalability and efficiency of parallel algorithms and applications.
- Implement a distributed-memory parallel program using MPI and/or other messaging middleware.
- Implement a shared-memory parallel program using threads.
- Learn the hardware components and taxonomy of modern parallel computing systems, e.g. GPGPU and FPGA accelerators
- Learn to measure single-processor performance and apply optimization techniques.

III. UnoAPI Curricular Goals

UnoAPI's main goal is to help address the P3 challenge by making HPC attractive to a broader student audience. By taking a holistic approach to constructing the curriculum, we aim to make the curriculum transformative and engage a broad population of emerging researchers and professionals to take up parallel computing. Our recurring theme is to maintain a connection with concepts and programming techniques students have already seen in other, familiar ecosystems, which thereby lead them to appreciate HPC as similarly compelling and intriguing. Our hope is to help build a pipeline of

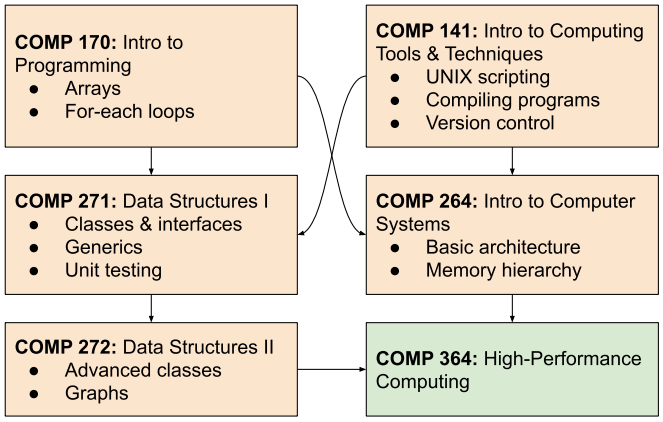

Fig. 1: COMP 364 with prerequisites including learning objectives relevant to high-performance computing with DPC++ and oneAPI

students who come to HPC research with proven, state-of-the-art software engineering and modern systems programming habits, and are thereby prepared to contribute productively to performant, portable, maintainable, and reproducible software.

While most curricular innovation in the area of heterogeneous HPC has focused on performance—broadly speaking—we aim to make additional improvements along the portability and productivity dimensions. In particular, this view is consistent with the HPC education community's increased emphasis on reproducibility [21].

In addition to key core parallel programming topics from the TCPP report [19], we have incorporated the following complementary learning outcomes from the ACM/IEEE Computer Society Software Engineering 2014 Curriculum Guidelines [22]. Bloom's cognitive skill levels (know, comprehend, and apply) and relevance to the core curriculum (essential versus desirable) are shown in parentheses:

- Portability
  - Build automation (PRO.cm.4, A, E)
  - External dependency management (PRO.cm.3, C, E)
  - Rootless package management (PRO.cm.4, A, E)
  - Cross-platform standards (PRO.imp.7, K, E)
- Reproducibility
  - Automated testing (VAV.tst.11, A, E)
  - Continuous integration (PRO.cm.4, A, E)
- Productivity
  - Version control (PRO.cm.1, A, E)
  - Modern language usage (CMP.cf.8, A, E)
  - Separation of concerns principle (DES.ev.1, K, E)
  - Software reuse (CMP.ct.2, A, E)
  - Software composition (PRO.pp.1, A, E)

While the codes shown in this list may appear a bit cryptic, they are readily used by academic computing-related departments to determine what knowledge units are addressed by courses within the curriculum. For example, `PRO.cm.4` is addressing the topic of Build Automation (topic 4) within the major curricular unit of Process (`PRO`) and Configuration Management (`cm`). We have listed the leaf topic (e.g. Build automation) in the above outline and where to find it within the ACM/IEEE SE curricular hierarchy.

In terms of leveraging accepted software engineering and software architecture practices [22] in HPC education, the UnoAPI approach builds on our prior contribution to the Curriculum Development and Educational Resources (CDER) book project to teach concurrent application development using Android and Java [23]. While we believe that many of our colleagues already follow these practices, the HPC community's growing interest in P3 issues suggests that these practices are important enough to become explicit learning objectives in their own right.

We hope that our curricular offering will be compelling not just for those interested in scientific programming and other applications of HPC, but also software engineering majors interested in contributing to well-constructed HPC solutions.

IV. Methods

To address the learning objectives listed above, the proposed UnoAPI approach combines the following key ingredients. Figure 2 shows the resulting color-coded course outline.

- Platform: oneAPI with Data-Parallel C++
- Language and included libraries: modern C++ and the C++ Standard Library
- Third-party libraries to solve common recurring problems
- Tools for various aspects of software development
- Techniques to help achieve the P3 goals
- Pedagogy based on well-crafted exemplars

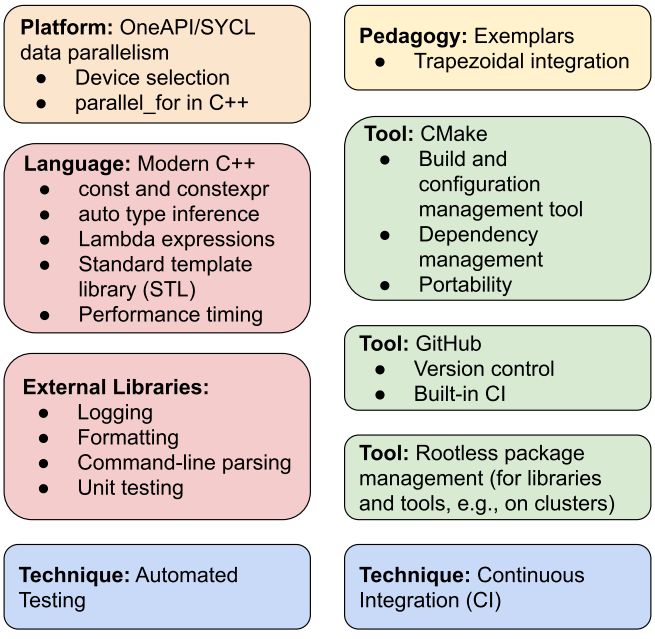

Fig. 2: COMP 364 proposed course outline

We now describe our approach in detail based on our data-parallel trapezoidal integration exemplar. We choose to

present this exemplar from our growing collection—including many examples distributed by Intel as part of their own documentation—here as it is easy to comprehend yet allows us to demonstrate various key aspects of data-parallel C++ programming using Intel's oneAPI platform [7], [8], which is based on the SYCL cross-platform standard for heterogeneous accelerator-based computing [6]. Although the example is embarrassingly parallel, it nevertheless exhibits numerous non-trivial pedagogical challenges that we will discuss in detail.

The exemplar's sequential version illustrates the underlying fused-loop map-reduce algorithm, which maps each adjacent pair of function values to a trapezoid area and, in the same loop body, reduces (adds) the area to the cumulative result.

```
if (run_sequentially) {
  std::vector values(size, 0.0);
  double result{0.0};

  values[0] = f(x_min);
  for (auto i{0UL}; i < number_of_trapezoids; i++) {
    values[i + 1] = f(x_min + i * dx);
    result +=
      trapezoid(values[i], values[i + 1], half_dx);
  }

  fmt::print("result = {}\n", result);
}
```

We'll see shortly how to write the data-parallel version of this algorithm in DPC++.

A. The Platform: Intel oneAPI/Khronos SYCL

The SYCL standard [6] defines high-level abstractions for parallel computing using modern C++, in an effort by Khronos to influence the ISO C++ standard to support heterogeneous computing. Data-Parallel C++ (DPC++) [8], a part of Intel's oneAPI standard, is an implementation of SYCL. Intel provides remote access to various types of accelerator hardware, including GPUs and FPGAs, through its DevCloud program.

1) Device Selection and Task Queues

A typical DPC++ program starts with the selection of one or more accelerator devices based on criteria of varying specificity. In our exemplar, the user can choose between running the code on the host CPU and an available accelerator:

```
const sycl::device_selector & device_selector{
  run_cpuonly ?
    static_cast<const sycl::device_selector &>(
      sycl::cpu_selector{}) :
    static_cast<const sycl::device_selector &>(
      sycl::default_selector{})
};
```

The interface between the programmer and the chosen device is a *queue*, to which we can later submit *commands* for execution on the device.

```
sycl::queue q{
  device_selector,
  dpc_common::exception_handler,
  sycl::property::queue::in_order()
};
spdlog::info("Device: {}", q.get_device().
  get_info<sycl::info::device::name>());
```

If we do not explicitly specify a device when creating our queue, the queue will automatically select the most suitable available device on the current hardware. Also, we can choose between a simple in-order queue, as we have done here, or we can have the queue figure out the best order for executing the submitted commands without deadlocking.

2) Buffers Shared Between Host and Device

Data-parallel computing typically involves some form of data sharing between the host and the device. SYCL allows several choices for this with varying degrees of control where the data should reside and how it is shared between host and device. E.g., we could allocate a standard vector on the host and use universal shared memory (USM) to share it with the device executing the data-parallel instructions. This might require copying substantial amounts of data between host and device and thereby impair performance.

Instead, a *buffer* is a higher-level data container that allows SYCL to determine where best to allocate the corresponding memory; a *range* represents a 1, 2, or 3-dimensional index range for a buffer. By not explicitly backing a buffer by a host-allocated standard vector, the data can remain on the device for faster access during kernel execution—until we may need to access it on the host later.

```
sycl::buffer<double> v_buf{sycl::range<1>{size}};
sycl::buffer<double> r_buf{sycl::range<1>{1}};
```

3) parallel_for() Construct

At the heart of SYCL's support for data parallelism lies the `parallel_for()` construct, which allows us to express the instructions that should execute in parallel. While also providing varying degrees of control over splitting up the workload and assigning it to the accelerator device, SYCL is able to come up with a suitable assignment that maximizes parallelism based on the capabilities of the device.

```
q.submit([&](auto & h) {
  const sycl::accessor v{v_buf, h};
  h.parallel_for(size, [=](const auto & index) {
    v[index] = f(x_min + index * dx);
  }); // end inner lambda/parallel_for
}); // end of command group
```

In this example, `f()` represents the computation we perform in parallel on each data item. As shown in §IV-B7, separating `f` into its own compilation unit enables us to unit-test it, as well as choose a specific implementation of `f` at build time.

We will also discuss below how the `parallel_for()` construct relates to other C++ constructs.

B. The Language: Modern C++

"How many C++ programmers does it take to change a lightbulb? Just one, and when they're done, the refrigerator and kitchen sink no longer work."

Since its initial standardization in 1998, followed by the 2003 update mostly to address some shortcomings in the original standard, C++ has undergone a long period of stability. Starting with C++-11, however, the first version associated with

"modern C++ " and continuing with subsequent standards every three years, there has been substantial innovation in terms of language features and best practices. These have largely brought the language in line with other languages used widely in computer science education, such as Java and Python, and arguably make C++'s erstwhile complexity, as witnessed by the joke above, more manageable. We currently follow the C++-17 standard because it forms the basis for SYCL and is well-supported by the Data-Parallel C++ compiler.

1) const and constexpr

This language feature falls within the broader category of static type safety. By declaring variables as `const` or `constexpr`, we enable the C++ compiler to prohibit unintended attempts to modify those variables and optimize the generated code accordingly. This C++ concept derives from modern functional programming languages such as Scala `val` and ML and related languages with its `let` expressions. In the following declarations, we use C++-11's simplified, uniform, and safe initializer syntax.

```
constexpr size_t DEFAULT_NUM_TRAPEZOIDS{10};
size_t number_of_trapezoids{DEFAULT_NUM_TRAPEZOIDS};
double x_min{0.0};
double x_max{1.0};
bool show_function_values{false};
bool run_sequentially{false};
bool run_cpuonly{false};
uint x_precision{1};
uint y_precision{1};
```

2) auto Variable Type

In the presence of nested type constructors, such as parameterized container types, explicit variable types can quickly become complex and unwieldy. The `auto` variable type provides a convenient and concise way to let the compiler figure out such types automatically—known as type inference—instead of the programmer having to spell them out. Nevertheless, these variables are still statically typed, and the compiler will prohibit any incorrect attempts to interact with them. While the notion of type inference originated in functional languages, it has found its way into C#, C++, and eventually Java.

```
const auto size{num_of_trapezoids + 1};
const auto dx{(x_max - x_min) / numb_of_trapezoids};
const auto half_dx{0.5 * dx}; // needed later
```

Although these uses of `auto` are trivial, others—including those in the `parallel_for()` example above—avoid highly complex, error-prone explicit types. E.g., the type of the iterator `t` for iterating over a nested container takes up well over a dozen lines.

In sum, minimizing the complexity arising from explicit type declaration not only reduces complexity but also brings statically-typed language such as C++ in line with the allure of dynamically-typed languages (e.g. Python) when it comes to clarity and conciseness of expression but offer the safety and performance advantages of statically typed-languages.

3) Lambda Expressions

Lambda expressions are a syntactic mechanism for expressing a (possibly parameterized) unit of computation without having to define it explicitly as a function. This conveniently allows us to specify the computation where it is used, can capture and access variables in the current scope instead of having to pass them as additional parameters, and avoids the need to come up with a name that is used only once.

The `parallel_for()` example shown above actually consists of two nested lambda expressions, an outer one for submitting a command to a queue that is parameterized by a handler, and an inner one for the loop body of the `parallel_for()` that is parameterized by an abstract index. The capture clause `[&]` for the outer lambda indicates access to the captured variable(s) (buffer `v_buf`) *by reference* and thereby allows modification, such as assigning values to `v_buf` through the locally declared accessor `v`. By contrast, the capture clause `[=]` for the inner lambda indicates access to the captured variables (scalars `x_min` and `dx`) *by copy* and thereby prevents modification.

This example illustrates the nature of `parallel_for()` as a higher-order method, whose parameter is an (itself parameterized) lambda expression that gets applied to each data element in the given range in a way that maximizes the degree of parallelism based on the available hardware. In this sense, `parallel_for()` is closely related to the `map` function commonly available in functional languages, whose effectful version is often called `foreach`. The purpose of these functions is the uniform transformation of each element in a container without specifying a particular order, unlike an imperative `for` loop, and thereby allowing parallelism as an optimization if available.

A closely related concept is the reduction of the elements in a container to a single combined result, which corresponds to the second stage of a map-reduce algorithm. While functional languages usually call this `reduce` and/or `fold`, oneAPI/SYCL's `parallel_for` handles it in conjunction with a *reduction variable*:

```
const auto sum_reduction{
  sycl::reduction(r_buf, h, sycl::plus<>())};
h.parallel_for(
  sycl::range<1>{number_of_trapezoids},
  sum_reduction,
  [=](const auto & index, auto & sum) {
    sum.combine(
      trapezoid(v[index], v[index + 1], half_dx));
});
```

4) C++ Standard Library

The C++ Standard Library, based on the earlier Standard Template Library (STL), is a comprehensive, mature, highly optimized library included with C++ toolchains. It provides building blocks for solving common programming problems, such as generic containers, I/O streams, algorithms to operate on containers and streams, threads, and various auxiliary types and functions.

In particular, the `std::vector` class is a highly performant implementation of a resizable linear container that adds a thin layer around native C++ arrays and optionally supports custom memory allocators.

To facilitate working with vectors and other containers, C++ now supports *foreach* loops for iterating over each element of the container exactly once (see also §IV-B3):

```
forward_list<string> lines;
// ...insert elements into lines...
for (const auto & line: lines) {
  // visit each line once
}
```

Beyond explicit loops, which can be verbose and error-prone, there is a trend toward combining higher-order constructs, which can have parameters that are themselves functions. Despite their high level of abstraction, recent versions of these constructs are highly optimized.

E.g., the following code reads successive words from standard input but adds only those to the vector `words` that have the specified minimum length. (The minimum length predicate is a lambda expression.)

```
std::vector<std::string> words;
std::remove_copy_if(
  std::istream_iterator<std::string>(std::cin), {},
  std::back_inserter(words),
  [=](const auto & word) {
    return word.length() < min_word_length;
  }
);
```

5) C++ Namespaces

C++ supports namespaces for grouping related type and function definitions and declarations together. In our exemplar(s), we make explicit reference to the namespace of a type or function when making use of various features of SYCL and oneAPI. There is a *tradeoff* involved, much like there is for other programming languages that support namespaces. For example, Python programmers readily use *modules as objects*, e.g. `sys.argv`. C++, on the other hand, uses the double colon, `::`, to achieve the same thing, which looks a bit unsightly compared to the Python equivalent. We acknowledge that the `::` can impact readability; however, C++ provides for more concise expression as long as the symbols are imported. If one writes `using sys::queue` then the code can reference `queue` instead of `using sys::queue`. The concept of namespaces is eminently *teachable*. We opt for explicit namespace references so those learning oneAPI and SYCL know, specifically, where to find more information about the definition or declaration.

6) Variadic Templates

Modern C++ supports more powerful, flexible syntax for variadic templates beyond simple expansion of type parameters. Similar to duck typing in dynamic languages but statically typed, this helps us make the code "DRY" (don't repeat yourself) because `values` can be any indexed container (`Indexable`), such as a standard vector or a SYCL buffer.

```
template <class Indexable>
void print_function_values(
  const Indexable & values,
  const double x_min, const double dx,
  const uint x_precision, const uint y_precision)
{
  for (auto i{0UL}; i < values.size(); i++) {
    fmt::print("{}: f({:.{}f}) = {:.{}f}\n", i,
      x_min + i * dx, x_precision,
      values[i], y_precision
    );
  }
}
```

7) Separate Compilation and External Functions

Separate compilation of source files helps us decompose a software system into smaller modules. This has multiple benefits, including separation of concerns, unit testing, easier collaborative development, and the ability to defer certain decisions (e.g., what we're integrating) until build time.

In our exemplar, we'll want to unit-test the function to be integrated and defer choosing a specific implementation of that function until build time. To be able to separately compile the function and call it inside a DPC++ kernel, we declare it in this SYCL-specific way:

```
#include <CL/sycl.hpp>

SYCL_EXTERNAL double f(double x);
```

To observe a speedup when using `parallel_for`, we define `f` as an intentionally inefficient way to compute the unit value:

```
double f(const double x) {
    return cos(x) * cos(x) + sin(x) * sin(x);
}
```

8) Performance Timing

We can use the `chrono` section of the C++ Standard Library for this purpose.

```
void mark_time(
  ts_vector & timestamps,
  const std::string_view label)
{
  timestamps.push_back(std::pair(label.data(),
    std::chrono::steady_clock::now()));
}
```

Every time we want to add a timestamp, we invoke `mark_time` with a suitable string label for the phase whose performance we are measuring. At the end, we use the `print_timestamps` function to print the collected measurements in comma-separated-values (CSV) format. (For readability, we have replaced the actual device name, "Intel(R) UHD Graphics P630 [0x3e96]", with "gen9.")

```
TIME,DELTA,UNIT,DEVICE,PHASE
68719429381281,0,ns,gen9,Start
68719429386913,5632,ns,gen9,Memory allocation
68719466567445,37180532,ns,gen9,Queue creation
68719769668012,303100567,ns,gen9,Integration
68719773241994,3573982,ns,gen9,DONE
68719773241994,343860713,ns,gen9,TOTAL
```

These measurements lead to various insights on what is going "under the hood" during program execution, to name a few:

- Initial allocation of a SYCL buffer takes very little time compared to allocating an standard vector.
- Queue creation introduces significant overhead.
- To achieve an overall speedup in light of this overhead, a high degree of parallelism is required (between about 10 and 20 million trapezoids on an Intel DevCloud gen9 node).
- Compared to the sequential version, there is an overall speedup even when using SYCL on the host CPU rather than the accelerator.

C. Best-of-breed External Libraries

As we have seen above, the "included batteries", i.e., the C++ standard library, addresses many common, general programming concerns. Nevertheless, there are various other common concerns that are too specific for the C++ Standard Library to address, or which it doesn't address *yet*, e.g.:

- Logging
- Formatting
- Command-line option parsing
- Unit testing

One major learning objective of UnoAPI is to avoid the "not invented here" syndrome, which refers to a strong bias against outside ideas (see also Wikipedia) and thereby hampers productivity. To this end, we strive to impart on our students a culture of aggressive software reuse in the form of proven, best-of-breed, open-source external libraries, in addition to the C++ and DPC++ ones on the oneAPI platform. We start with curated lists of libraries and other pertinent resources, such as Awesome C++, which exist for most languages and platforms. By "standing on the shoulders of giants" who have professionally engineered these building blocks, our students will not only benefit from using these in their solutions, but also become familiar with good design and documentation of such components, which they can then imitate in their own work. The key selection criteria are intended to minimize software composition risks: correctness/security, stability/active maintenance, usability/good documentation, and performance.

This approach requires the support of a suitable build management system, such as CMake, and userland package management, such as Homebrew, discussed below in detail. We have designed our exemplars to work with recent versions of these tools.

1) Logging: spdlog

This example shows the complementary role of logging relative to actual output using spdlog.

```
if (show_function_values) {
  spdlog::info("preparing function values");
  const sycl::host_accessor values{v_buf};
  spdlog::info("showing function values");
  print_function_values(values, x_min, dx,
    x_precision, y_precision);
}
```

This type of logging works only outside of kernel code running on an accelerator. oneAPI supports limited I/O in the kernel for logging and debugging purposes but does not allow file I/O (see also Doing IO in the Kernel).

2) Formatting: {fmt}

Most modern languages support some form of string interpolation, where one can reference variables—and even expressions—from the current scope directly in a formatting string. While the familiar `printf()` format syntax does not provide full-fledged string interpolation, it is familiar and arguably more convenient than `std::ostream::operator<<`.

The {fmt} library brings back fast formatting using an enhanced `printf` format syntax and is being incorporated into C++-20. We can even make decimal precision a dynamic parameter by using nested placeholders (pairs of curly braces).

```
fmt::print("{}: f({:.{}f}) = {:.{}f}\n",
  i, x_min + i * dx, x_precision,
  values[i], y_precision
);
```

3) Command-line Option Parsing: CLI11

Command-line option parsing enables us to defer certain decisions about the way our exemplars execute all the way until invocation time without the need to recompile the code. This ability is crucial for making our code parametric *at the application level*. E.g., we can write shell commands to carry out a detailed performance study over the Cartesian product of possible parameter values.

State-of-the-art command-line parsing libraries, such as CLI11, are feature-rich and may support dynamic validation of provided values. The following shows how we support CLI for our trapezoidal integration example:

```
CLI::App app{"Trapezoidal integration"};

app.option_defaults()->always_capture_default(true);
app.add_option(
  "-n,--trapezoids", number_of_trapezoids,
  "number of trapezoids")->
  check(CLI::PositiveNumber.description(" >= 1"));
app.add_option(
  "-l,--lower,--xmin", x_min, "x min value");
// ...
app.add_option(
  "-y,--y-format-precision", y_precision,
  "decimal precision for y (function) values")->
  check(CLI::PositiveNumber.description(" >= 1"));

CLI11_PARSE(app, argc, argv);
```

The resulting executable is self-documenting, using the `-h` or `--help` flags following the UNIX tradition:

```
$ ./cmake-build-release/bin/integration -h
Trapezoidal integration
Usage: ./cmake-build-release/bin/integration [OPTS]

Options:
  -h,--help     Print this help message and exit
  -n,--trapezoids UINT: >= 1=10 number of trapezoids
  -l,--lower,--xmin FLOAT=0     x min value
  ...
```

Note how the CLI allows one to think parametrically about the application with intuitively named arguments—and concise

ones—in the UNIX tradition. E.g., we can compute the integral $\int_{1/2}^{2} f(x)\,dx$ using 100 trapezoids on the CPU:

```
$ ./cmake-build-release/bin/integration -n 100
    -l 0.5 -u 2.0 -c
[info] integrating function from 0.5 to 2
       using 100 trapezoid(s), dx = 0.015
[info] preparing for vectorized integration
...
result = 1.4999999999999973
```

The CMake build tool allows us to address the concern of release builds (optimized, suitable for performance analysis) versus debug builds (instrumented for possible debugging) without listing specific compiler options. (We'll discuss CMake in more detail in §IV-D1.)

```
cmake -DCMAKE_BUILD_TYPE={Debug|Release} ...
```

4) Unit Testing: GoogleTest

The main value proposition of automated unit testing is that it encourages frequent regression testing by making it painless [24]. During the last two decades, this "test-infected" mindset has gradually entered the mainstream including introductory computer science courses. We argue that it can benefit and integrate seamlessly with HPC education.

Support for unit testing in C/C++ has improved considerably, and we prefer GoogleTest for this purpose. A typical floating-point correctness test looks like this:

```
TEST_F(IntegrationTest, Simple3) {
  EXPECT_NEAR(trapezoid(-1, 1, 0.5), 0, EPS);
}
```

D. Software Engineering Practices: Techniques and Tools

We argue that following modern software engineering practices, especially build and configuration management and build automation (continuous integration) can help greatly with portability and reproducibility in the software supply chain.

1) Build and Configuration Management: CMake

A key portability challenge results from differences across users' development and production environments, such as different versions of operating systems, compilers, libraries, and other tools. Among several efforts to abstract away these differences and support building a project on any environment meeting certain minimum criteria, (modern) CMake has emerged as the most painless choice, especially for C/C++-based projects.

CMake enables us to use C/C++ similar to other languages, such as Java, Scala, Python, JavaScript/Node, by managing external library dependencies declaratively and fetching them dynamically. This encourages parametric thinking and makes it possible to develop adaptable HPC codes.

The following settings consistently ensure a specific language standard:

```
set(CMAKE_CXX_STANDARD 17)
set(CMAKE_CXX_STANDARD_REQUIRED ON)
set(CMAKE_CXX_EXTENSIONS OFF)
```

This section is an example of declaring an external dependency that gets included into the build process at source level.

```
FetchContent_Declare(
  fmt
  GIT_REPOSITORY https://github.com/fmtlib/fmt.git
  GIT_TAG        8.1.1
)
FetchContent_MakeAvailable(fmt)
```

The external dependencies declared in this way then become available for linking into the executable(s).

```
add_executable(integration
  main.cpp f.cpp trapezoid.cpp timestamps.cpp
)
target_link_libraries(integration
  fmt::fmt spdlog::spdlog CLI11::CLI11
)
```

2) Version Control: git/GitHub

Version control, especially distributed, hosted services such as GitHub, GitLab, and Bitbucket, are one of the foundations of modern software engineering practice. Version control allows a development team to keep their code in a secure place and enables collaboration following numerous different models and cultures.

3) Continuous Integration: GitHub Actions

There are various choices for adding continuous integration to a software project, such as setting up a dedicated server or connecting to an external provider. Even more conveniently, major hosted version control services already include support for continuous integration. In practice, automated builds (workflow runs) are triggered every time a project contributor commits changes to the code base. Such a build automates the steps one would typically perform manually on one's workstation:

- starting with a vanilla configuration,
- installing prerequisites to the build,
- checking out the project source code,
- building executable artifacts, and
- running/testing them.

Starting with a vanilla (default) system configuration ensures that all project dependencies (tools, external libraries, etc.) are fully understood and explicit. In addition, build automation results in almost immediate feedback on an incorrect commit that "breaks the build" (see Appendix A for the actual scripts).

We argue that continuous integration can make us reproducibility-aware: A researcher wishing to reproduce the work represented by a particular project can "fork" the project into their own account on, say, GitHub. The CI workflow then runs on that user's fork of the project. This gives us some degree of reproducibility "for free," subject to limitations in devices available through the actual CI container (i.e., not usually accelerators).

4) Automated Testing

Automated testing usually takes place as part of continuous integration. Indeed, the last section of the CI workflow shown above invokes all discovered executable tests.

5) Rootless Package Management for Libraries and Tools

On publicly accessible HPC clusters, such as Intel's DevCloud and those run by some national laboratories, users don't typically have root access (administrative privileges). This precludes them from using native package management, such as `apt` on Ubuntu, to install missing packages or newer versions of outdated packages.

During the last decade or so, userland package management tools have emerged as a complement to native package management. These tools, such as Homebrew on MacOS and Linux and Chocolatey on Windows, allow users to install additional packages for their personal use without requiring root access, e.g., a current version of CMake.

V. Evaluation and Future Plans

The UnoAPI curricular modules are available at unoapi.cs.luc.edu, and the source code of the exemplars can be found at github.com/LoyolaChicagoCode/unoapi-dpcpp-examples. While our trapezoidal integration exemplar used throughout this paper is complete, the curricular modules are still a work in progress.

By the time of this submission, we have shared these materials with a small group of undergraduate sophomores, and their responses have been encouraging in terms of self-reported comprehension and ability to reproduce the compilation and execution of exemplars on their personal systems.

During the second half of the 2022/23 academic year, we are planning to conduct a broader study with a larger cohort by incorporating some of our materials in the existing course COMP 364: High-Performance Computing.

We also plan to study the role of internationalization (i18n) and localization (l10n) in software reproducibility and enhance our written curricular materials accordingly.

VI. Conclusion

While we have noticed growing awareness among our colleagues of the practices we have described as part of the UnoAPI curriculum, the HPC community's growing interest in P3 issues suggests that these additional programming language and software engineering practices are important enough to become explicit learning objectives in their own right. The various curricular components we have described above are widely available, well documented, and increasingly used in general software development practice. However, what we have not seen in prior work is a coherent, holistic pedagogical vision that combines them as part of the intermediate undergraduate curriculum. We maintain that this combination of materials is eminently teachable and have shared our vision in the hope of encouraging other educators to incorporate these techniques and tools into their teaching to enhance early interest about HPC among emerging student talent.

In sum, we have argued that systems and HPC programming can be effective and engaging using proper software engineering techniques, and we can teach students how to write high-quality, reproducible code in this domain. While a small group of undergraduate sophomores has responded positively to our materials, we plan a broader study with a larger cohort.

References

[1] P. F. Dubois, "Guest editor's introduction: Python: Batteries included," *Computing in Science & Engineering*, vol. 9, no. 03, pp. 7–9, May 2007, ISSN: 1558-366X. DOI: 10.1109/MCSE.2007.51.

[2] S. J. Pennycook, J. D. Sewall, and V. W. Lee, "A metric for performance portability," in *Proc. 7th International Workshop in Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems*, 2016. DOI: 10.48550/ARXIV.1611.07409.

[3] *Performance portability*, Department of Energy. [Online]. Available: https://performanceportability.org.

[4] D. Chamont, "Performance vs portability vs productivity vs precision : A trail in the hardware and software jungle," in *Learning to Discover: Advanced Pattern Recognition*, Institute Pascal, Orsay, Paris, Oct. 2019.

[5] P. J. Hatcher and M. J. Quinn, *Data-Parallel Programming on MIMD Computers*. MIT Press, 1991, ISBN: 9780262082051.

[6] *The SYCL 1.2.1 specification*, Khronos SYCL Working Group, 2019. [Online]. Available: https://www.khronos.org/registry/SYCL.

[7] B. Ashbaugh, A. Bader, J. Brodman, *et al.*, "Data parallel C++: Enhancing SYCL through extensions for productivity and performance," in *Proceedings of the International Workshop on OpenCL*, ser. IWOCL '20, Munich, Germany: Association for Computing Machinery, 2020, ISBN: 9781450375313. DOI: 10.1145/3388333.3388653.

[8] J. Reinders, B. Ashbaugh, J. Brodman, M. Kinsner, J. Pennycook, and X. Tian, *Data Parallel C++: Mastering DPC++ for Programming of Heterogeneous Systems using C++ and SYCL*. Apress, 2021. DOI: 10.1007/978-1-4842-5574-2.

[9] Y. Sitsylitsyn, "Methods and tools for teaching parallel and distributed computing in universities: A systematic review of the literature," *SHS Web Conf.*, vol. 75, p. 04 017, 2020. DOI: 10.1051/shsconf/20207504017.

[10] J. Ciesko, D. Poliakoff, D. S. Hollman, C. C. Trott, and D. Lebrun-Grandié, "Towards generic parallel programming in computer science education with Kokkos," in *2020 IEEE/ACM Workshop on Education for High-Performance Computing (EduHPC)*, 2020, pp. 35–42. DOI: 10.1109/EduHPC51895.2020.00010.

[11] A. Qasem, D. Bunde, and P. Schielke, "A module-based introduction to heterogeneous computing in core courses," *Journal of Parallel and Distributed Computing*, vol. 158, Aug. 2021. DOI: 10.1016/j.jpdc.2021.07.011.

[12] J. Dokulil, "Let's put the memory model front and center when teaching parallel programming in C++," in *Proc. 2021 NSF/TCPP Workshop on Parallel and Distributed Computing Education (EduPar-21)*, May 2021. [Online]. Available: https://tcpp.cs.gsu.edu/curriculum/sites/default/files/main.pdf.

[13] J. Fuentes, D. López, and S. González, "Teaching heterogeneous computing using DPC++," in *Proc. 2022 NSF/TCPP workshop on parallel and distributed computing education (EduPar-22)*, May 2022.

[14] D. M. Rao, "Fall-12: Using C++11 to teach concurrency and parallelism concepts," in *Proc. Third NSF/TCPP Workshop on Parallel and Distributed Computing Education (EduPar-13)*, Cambridge, MA, USA, May 2013. [Online]. Available: https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.360.7348&rep=rep1&type=pdf.

[15] N. Dale, C. Weems, and T. Richards, *C++ Plus Data Structures*, 6th. Jones & Bartlett, 2018, ISBN: 9781284089196.

[16] M. Blesel, M. Kuhn, and J. Squar, "Heimdallr: Improving compile time correctness checking for message passing with Rust," in *High Performance Computing*, H. Jagode, H. Anzt, H. Ltaief, and P. Luszczek, Eds., Cham: Springer International Publishing, 2021, pp. 199–211, ISBN: 978-3-030-90539-2.

[17] J. C. Adams and M. C. Wissink, "Hearing program behavior with TSAL," in *2019 IEEE/ACM Workshop on Education for High-Performance Computing, EduHPC@SC 2019, Denver, CO, USA, November 17, 2019*, IEEE, 2019, ISBN: 978-1-7281-5975-1.

[18] A. Danner, T. Newhall, and K. C. Webb, "ParaVis: A library for visualizing and debugging parallel applications," in *2019 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, 2019, pp. 326–333. DOI: 10.1109/IPDPSW.2019.00062.

[19] S. K. Prasad, T. Estrada, S. Ghafoor, *et al.*, "NSF/IEEE-TCPP curriculum initiative on parallel and distributed computing - core topics for undergraduates (version 2-beta)," Nov. 2020. [Online]. Available: https://tcpp.cs.gsu.edu/curriculum.

[20] V. Eijkhout, *Introduction to High Performance Scientific Computing*, 3rd edition. Jul. 2022. [Online]. Available: https://theartofhpc.com/istc.

[21] M. A. Heroux, "Reproducibility in scientific software," Jun. 2018. [Online]. Available: https://www.osti.gov/biblio/1525948.
[22] M. Ardis, D. Budgen, G. W. Hislop, J. Offutt, M. Sebern, and W. Visser, "Se 2014: Curriculum guidelines for undergraduate degree programs in software engineering," *Computer*, vol. 48, no. 11, pp. 106–109, 2015.
[23] K. Läufer and G. K. Thiruvathukal, "Managing concurrency in mobile user interfaces with examples in android," in *Topics in Parallel and Distributed Computing: Enhancing the Undergraduate Curriculum: Performance, Concurrency, and Programming on Modern Platforms*, S. K. Prasad, A. Gupta, A. Rosenberg, A. Sussman, and C. Weems, Eds. Cham: Springer International Publishing, 2018, pp. 243–285. DOI: 10.1007/978-3-319-93109-8_9.
[24] K. Beck, *Test Driven Development: By Example*. Addison-Wesley Professional, 2002.

APPENDIX

A. Artifact Description

The artifact associated with this submission is a CMake-based C++ project along with unit tests in source form. The artifact is publicly available as a GitHub repository under the Apache 2.0 open-source license at github.com/LoyolaChicagoCode/unoapi-dpcpp-examples.

This artifact includes a README file with instructions for building and running on Debian-based Linux systems (including Ubuntu), as well as the scripts needed to run the experiments described in this paper. External input data is not required.

1) Running the artifact on GitHub as a fork of the original repo (browser-based)

The repository is configured with continuous integration (CI) using the following GitHub Actions workflow:

```
steps:
- name: Checkout project code
  uses: actions/checkout@v2

- name: Install Intel oneAPI
  timeout-minutes: 5
  run: ./install-dpcpp.sh

- name: Set oneAPI environment
  run: |
    source /opt/intel/oneapi/setvars.sh
    printenv >> $GITHUB_ENV
    echo /opt/oneapi/compiler/latest/linux/bin \
      >> $GITHUB_PATH

- name: CMake configure
  run: cmake -S . -B $BUILD_DIR

- name: CMake build
  run: cmake --build $BUILD_DIR

- name: CMake test
  run: |
    for test_bin in $BUILD_DIR/bin/*tests; do
      "./$test_bin"
    done
```

It relies on this prerequisite installation script, which one can also invoke manually on a local workstation:

```
echo Add Intel Apt repository

wget -qO- \
  https://apt.repos.intel.com/intel-gpg-keys/\
GPG-PUB-KEY-INTEL-SW-PRODUCTS.PUB | \
  sudo tee /etc/apt/trusted.gpg.d/\
GPG-PUB-KEY-INTEL-SW-PRODUCTS.asc
echo "deb https://apt.repos.intel.com/oneapi" \
     "all main" | \
     sudo tee /etc/apt/sources.list.d/oneAPI.list
sudo apt update

echo Install Intel oneAPI

sudo apt install intel-oneapi-compiler-dpcpp-cpp
```

Every commit or pull request to the repository triggers a build and results in an indication of successful or unsuccessful completion of the build, including execution of the unit test suite. This option is subject to limitations in devices available through the actual CI container (i.e., not usually accelerators).

These are the steps to copy, build, and run the artifact in a web browser without involving a local build environment.

- Visit github.com/LoyolaChicagoCode/unoapi-dpcpp-examples.
- Near the top right corner, look for the Fork button, click on the dropdown, and select "Create a new fork."
- Create the new fork in your GitHub account or organization.
- Visit the Actions tab, where you will see the message "Workflows aren't being run on this forked repository."
- Click on the green button labeled "I understand my workflows, go ahead and enable them."
- Use the web interface to create a file, say "dummy.txt", to trigger a workflow run. This will create an entry under the GitHub actions tab for this workflow run. To observe execution in real time or after completion, one can drill into this entry until the step-by-step execution log appears.

2) Running the artifact locally

This option requires a physical or virtual Debian-based system on Intel hardware, which may include an accelerator, such as a GPU. The specific steps are documented in the top-level README file in the GitHub repository. The artifact's main executable is self-documenting using the `-h` or `--help` CLI option.

3) Running the artifact on the Intel DevCloud

This option requires a (free) account on Intel's DevCloud for working with oneAPI and provides access to various types of accelerators, such as GPUs based on the gen9 architecture we've used for the sample runs shown above; it thereby provides the highest degree of reproducibility. Once access to DevCloud has been established, the specific steps are documented in the top-level README file in the GitHub repository. The artifact's main executable is self-documenting using the `-h` or `--help` CLI option.