Intel® oneAPI DPC++ Library (oneDPL)

Reduce Cross-platform Programming Efforts & Write Performant Parallel Code with oneDPL

Alexey Kukanov

Principal Engineer, Intel Corporation

intel®

Agenda

- Parallel Algorithms API
  - Overview
  - Support for Data Parallel C++ (DPC++)
- C++ Standard APIs for Device Programming
- Custom Iterators
- Outlook: Experimental Features
  - Ranges API
  - Async API
- Summary

oneAPI

One Programming Model for Multiple Architectures and Vendors

Freedom to Make Your Best Choice

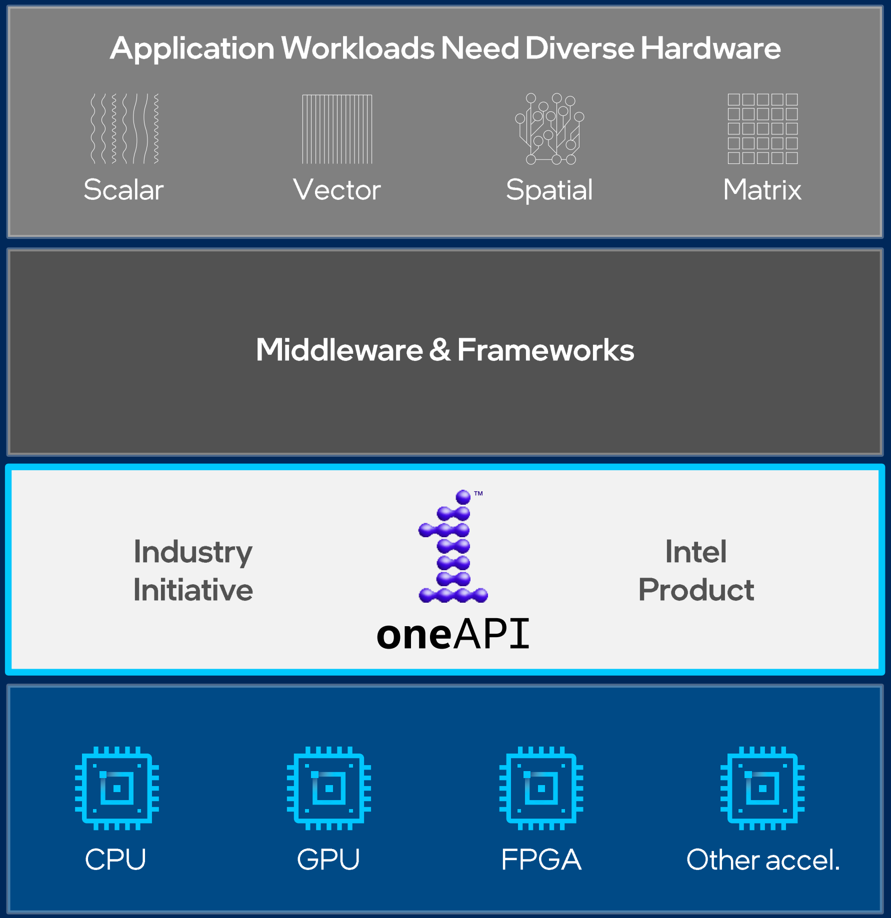
- Choose the best accelerated technology; the software doesn't decide for you

Realize all the Hardware Value

- Performance across CPU, GPUs, FPGAs, and other accelerators

Develop & Deploy Software with Peace of Mind

- Open industry standards provide a safe, clear path to the future
- Compatible with existing languages and programming models including C++, Python*, SYCL*, OpenMP*, Fortran, and MPI



oneAPI Industry Initiative

Break the Chains of Proprietary Lock-in

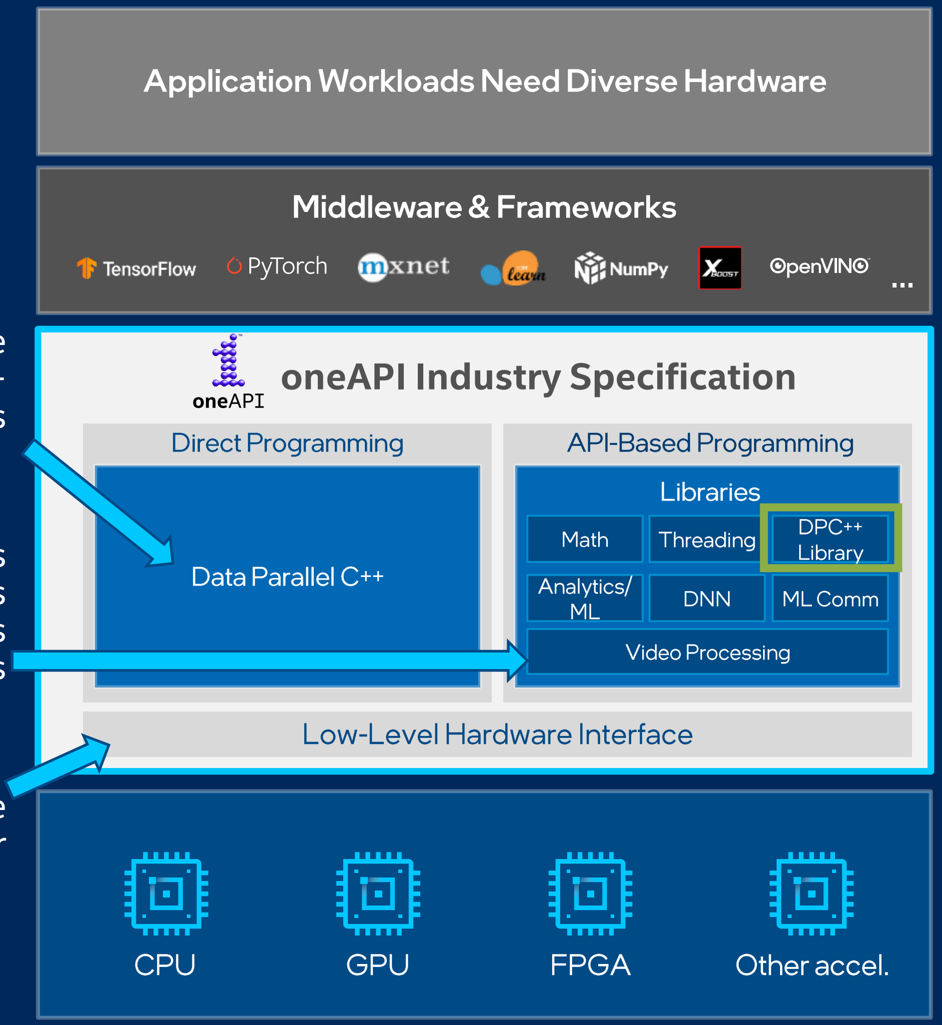
Open to promote community and industry collaboration

Enables code reuse across architectures and vendors

A cross-architecture language based on C++ and SYCL standards

Set of libraries minimizes programming efforts across different devices and domains

Low-level hardware abstraction layer



oneAPI

The productive, smart path to freedom for accelerated computing from the economic and technical burdens of proprietary programming models

Intel® oneAPI DPC++ Library (oneDPL)

- **High productivity and portable performance for heterogeneous computing** – CPUs, GPUs, and FPGAs
- **APIs based on standards and familiar extensions** – C++ STL, SYCL, Boost.Compute
- **Optimized C++ standard algorithms** implemented on top of SYCL, OpenMP, oneTBB
- **Interoperable with DPC++ and other oneAPI libraries**
- **Integrated with Intel® DPC++ Compatibility Tool** to simplify migration of CUDA* applications using Thrust* API to DPC++ code

oneDPL Specification & Reference Implementation

Spec: oneDPL section covers:

- Parallelized C++ algorithms
- Tested C++ standard APIs that work in DPC++ kernels
- Execution Policies
- Custom Utilities and Algorithms

oneAPI

https://spec.oneapi.io/versions/latest/elements/oneDPL/source/index.html#onedpl-section

oneDPL implementation

Source Code

- Available as part of the Intel® oneAPI Base Toolkit
- Supports multiple backends: oneTBB, OpenMP, DPC++
- Header-only library (relies on runtime libraries of the respective backends)

oneAPI

https://github.com/oneapi-src/oneDPL

Basic oneDPL Usage Guidelines

- C++17 is the minimal supported version of the C++ standard
- Header names start with oneapi/dpl:
  `#include <oneapi/dpl/algorithm>`
- All functionality is provided in `namespace oneapi::dpl`
  - short alias: `namespace dpl = oneapi::dpl;`

Parallel Algorithms API

Parallel Algorithms API: Brief Overview

- C++ parallel algorithms
- Execution policies define how to run an algorithm

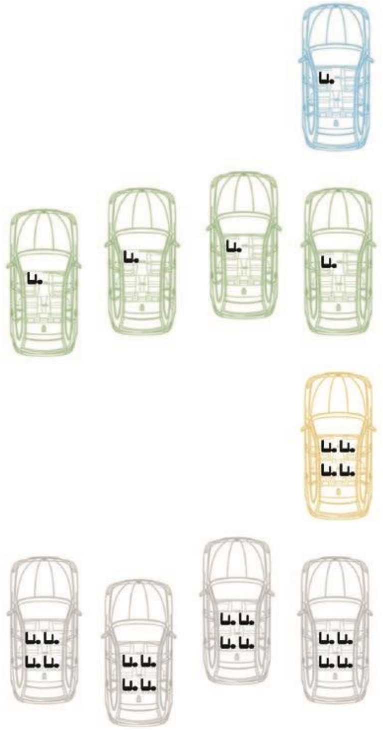
Expressing Parallelism in C++17/20

```
/* a serial range-based for loop */
for(auto &i : v) {i++;}

------------
#include <algorithm>
/* a serial for-each algorithm w/o execution policy  */
std::for_each(v.begin(), v.end(), [](auto &i){i++;});

------------
#include <algorithm>
#include <execution>
/* a for-each algorithm with parallel execution policy */
std::for_each(std::execution::par, v.begin(), v.end(), [](auto &i){i++;});
```

Semantics of Standard Execution Policies

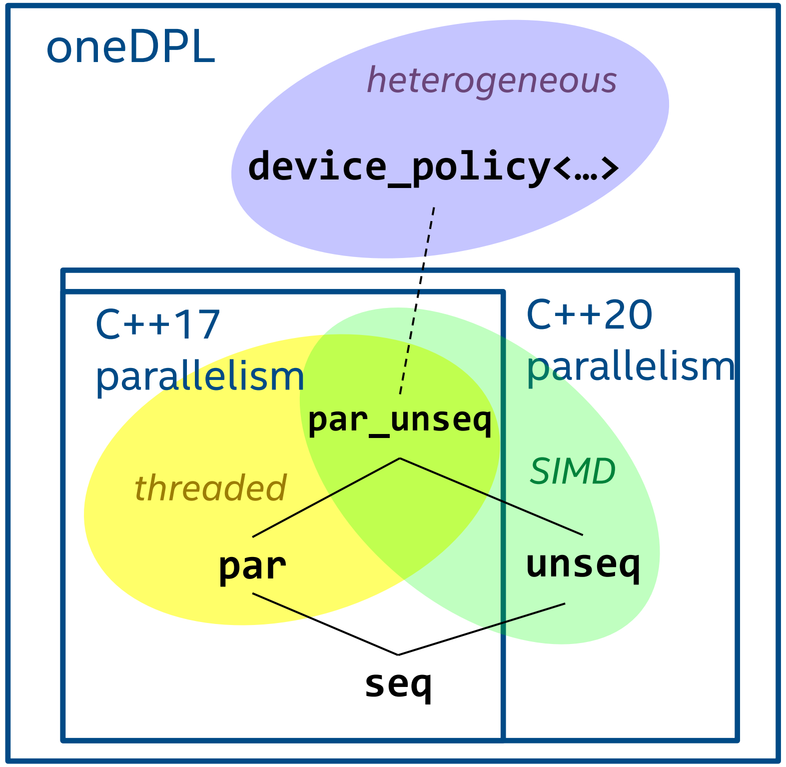
- **standard sequential** sort
  `std::sort(v.begin(), v.end());`
- **explicitly sequential** sort
  `std::sort(std::`**`execution::seq`**`, v.begin(), v.end());`
- permitting **parallel execution**
  `std::sort(std::`**`execution::par`**`, v.begin(), v.end());`
- permitting **vectorization only (no parallel execution)**
  `std::sort(std::`**`execution::unseq`**`, v.begin(), v.end());`
- permitting **parallel execution** and **vectorization**
  `std::sort(std::`**`execution::par_unseq`**`, v.begin(), v.end());`

multilane highway analogy from ProTBB book, M. Voss et al.



Parallel Algorithms API: Brief Overview (cont.)

- C++ parallel algorithms
- oneDPL extension algorithms
  - Segmented reduction & scan, binary search of multiple values
- Execution policies define how/**where** to run an algorithm
  - CPU: oneTBB or OpenMP for threading (par), OpenMP for SIMD (unseq)
  - GPU & accelerators: DPC++ supported devices



DPC++ Support in Parallel API

DPC++ Execution Policies

- `oneapi::dpl::execution::device_policy<…>` class template
  - The policy type selects the SYCL-based implementation of an algorithm
  - A policy object encapsulates a SYCL queue that defines the device to run on
  - Implicitly convertible to `sycl::queue` for better interoperability
- Policy usage:
  - Construct a policy object using a SYCL queue, device, device selector, or an existing policy object
  - Pass the created policy object to a oneDPL algorithm
- `oneapi::dpl::execution::dpcpp_default`
  - Predefined object of the device_policy class, uses the default SYCL queue

How to Target a Device:

First simple example (to be continued …)

```
#include <oneapi/dpl/execution>
#include <oneapi/dpl/algorithm>
int main()
{
      std::vector<int> vec(1000);
      dpl::fill(dpl::execution::dpcpp_default,
                vec.begin(), vec.end(), 42);
      return 0;
}

// To build: dpcpp <example_name>.cpp -o test
```

Buffer Wrappers

`oneapi::dpl::begin` and `oneapi::dpl::end` are helper functions for passing SYCL buffers to oneDPL algorithms

Applied to a SYCL buffer, these functions return an object of an unspecified type with some properties of a random access iterator:

- Can be copy-constructed and copy-assigned
- Can be compared for equality (== and !=)
- Can be used in expressions a + n, a – n, a – b, where **a** and **b** are objects of the type and **n** is an integer value

But most importantly, it can be passed to oneDPL algorithms

Simple Example using SYCL Buffer

Keeping data on the device with consecutive usage

```
#include <oneapi/dpl/execution>
#include <oneapi/dpl/algorithm>
#include <oneapi/dpl/iterator>
#include <sycl/sycl.hpp> // or <CL/sycl.hpp>
int main()
{
    sycl::buffer<int> buf{ 1000 };
    dpl::fill(dpl::execution::dpcpp_default, dpl::begin(buf),
              dpl::end(buf), 42);
    auto result = dpl::find(dpl::execution::dpcpp_default,
                            dpl::begin(buf), dpl::end(buf), 42);
    return 0;
}
```

Advanced Techniques:

Passing access modes to the backend

```
auto buf_begin = dpl::begin(buf);
dpl::fill(my_policy, buf_begin, buf_begin + 1000, 42);
```

```
auto buf_begin = dpl::begin(buf, sycl::write_only, sycl::noinit);
dpl::fill(my_policy, buf_begin, buf_begin + 1000, 42);
```

Example with SYCL Unified Shared Memory (USM)

Directly pass USM pointers to parallel algorithms

```
#include <oneapi/dpl/execution>
#include <oneapi/dpl/algorithm>
#include <sycl/sycl.hpp>
int main()
{
        using oneapi::dpl::execution::make_device_policy;
        sycl::queue q;
        const int n = 1000;
        int* data = sycl::malloc_shared<int>(n, q);

        auto pol = make_device_policy<class Fill>(q);
        dpl::fill(pol, data, data + n, 42);
        auto result = dpl::find(make_device_policy<class Find>(q),
                                data, data + n, 42);
        // q.wait();

        sycl::free(data, q);
        return 0;
}
```

C++ for Device Programming

Using the C++ Standard Library in DPC++

- Host
  - Code running on the host CPU can use everything
- Device
  - Some std classes/functions cannot work in device kernel code due to SYCL/DPC++ restrictions
    - Functions/methods that use exception
    - Dynamic memory allocation
    - Virtual functions

Tested Standard C++ APIs

- 120 standard C++ APIs have been tested
  - Usage in DPC++ kernels
  - Data transfer between host and device (only for specific C++ types)
- Across 3 major implementations:
  - libstdc++(GNU): deployed in most Linux* distributions
  - libc++(LLVM): macOS* and FreeBSD*
  - Microsoft STL*: Windows*; shipped with Microsoft Visual Studio*
- Available in both `namespace std` and `namespace oneapi::dpl`

Visit Intel® oneAPI DPC++ Library Guide for a detailed list

Example: Complex Numbers

use of std components on the device

```
#include <oneapi/dpl/execution>
#include <oneapi/dpl/algorithm>
#include <oneapi/dpl/iterator>
#include <sycl/sycl.hpp>
#include <complex>

int main() {
    const size_t n = 100;
    sycl::buffer<std::complex<double>> c_buf{n};
    auto idx = dpl::counting_iterator<int>(0);

    dpl::transform(dpl::execution::dpcpp_default, idx, idx+n, dpl::begin(c_buf),
    [=](auto i) {
        const double v = fabs(static_cast<double>(n)/2.0 - static_cast<double>(i));
        return std::complex<double>{v, v};
    });
    return 0;
}
```

Custom Iterators

Expanding Applicability of Parallel Algorithms with Custom Iterators

Custom Iterators:

- counting_iterator
- zip_iterator
- transform_iterator
- permutation_iterator
- discard_iterator

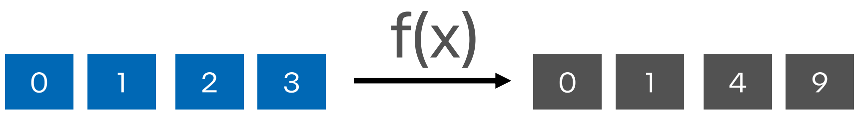
Counting Iterator

[0,4) → 0 | 1 | 2 | 3

- Represents a linear, increasing sequence of integer values
- Commonly used as an index: `[](auto i){a[i] = b[i];}`
- Advantage: Not in memory

Increase Adaptability with Custom Iterators

Transform Iterator

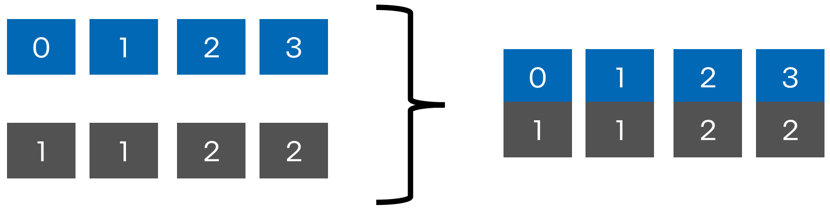


- Applies transformation to dereferenced values of another iterator
- Example: Square numbers
  - Input iterator

    Here: counting iterator
  - Unary function

    Here: `[](auto& x){return x*x;}`

Zip Iterator



- Combines multiple iterators into a single iteration space
  - Combinations of custom iterators are possible (permutation, counting, …)
- Make function accepts an argument pack

How to Use a Zip Iterator

TriAdd example with oneDPL

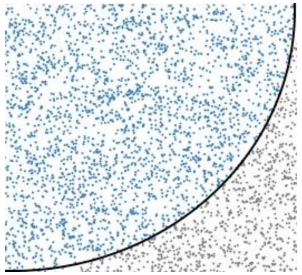
Adding two vectors **B** and **C** multiplied by a scalar **s** to vector **A**:

A += B + s * C

```
/* ... */
double *d_A = sycl::malloc_device<double>(length, q);
double *d_B = sycl::malloc_device<double>(length, q);
double *d_C = sycl::malloc_device<double>(length, q);
/* ... */
double scalar(3);
  {
    for (int iter = 0; iter <= iterations; iter++) {
        auto begin = dpl::make_zip_iterator(d_A, d_B, d_C);
        dpl::transform(dpl::execution::make_device_policy<class TriAdd>(q),
                       begin, begin + length, d_A, [=](const auto &t) {
                           auto [a, b, c] = t;
                           return a + b + scalar * c;
                       });
    }
  }
/* ... */
```

https://github.com/ParRes/Kernels/blob/default/Cxx11/nstream-onedpl.cc

Random sampling example: Monte Carlo Pi



Making full use of oneDPL

```
int main() {
    sycl::queue q(sycl::gpu_selector{});
    auto my_policy = dpl::execution::make_device_policy(q);
    auto sum = dpl::transform_reduce( my_policy, dpl::counting_iterator<int>(0),
                                      dpl::counting_iterator<int>(N), 0, std::plus<float>{},
                                      [=](auto n){
                                          float local_sum = 0.0f;
                                          // Get random coords
                                          dpl::minstd_rand engine(SEED, 2*n*LOCAL_N);
                                          dpl::uniform_real_distribution<float> distr(-1.0f,1.0f);
                                          for(int i = 0; i < LOCAL_N; ++i) {
                                            float x = distr(engine), y = distr(engine);
                                            if (x*x + y*y <= 1.0)
                                              local_sum += 1.0;
                                          }
                                          return local_sum / (float)LOCAL_N;
                                      });
    estimated_pi = 4.0*(float)sum / N;
    return 0;
}
```

Experimental Features: Ranges API

Range-based API for Algorithms

- C++20 adds Ranges into the C++ standard library
  - Very powerful and expressive functional API
  - But does not yet support execution policies
- oneDPL provides Range support as an experimental feature
  - A subset (~40) of the standard algorithms with execution policies
  - A dozen custom range views
  - Not fully standard-compliant (not based on concepts, no projections, …)
  - Only for DPC++ execution policies
  - sycl::buffer can be passed to algorithms as a range
- Available since Intel® oneAPI DPC++ Library (oneDPL) version 2021.1
  - As of today: not a part of oneAPI spec

SYCL buffer as a range

Pass a buffer to a parallel algorithm

```
#include <oneapi/dpl/execution>
#include <oneapi/dpl/algorithm>
#include <oneapi/dpl/ranges>
#include <sycl/sycl.hpp>
int main()
{
    using namespace oneapi::dpl;
    sycl::buffer<int> buf{ 1000 };
    experimental::ranges::for_each(execution::dpcpp_default,
                                   buf, [](auto& i){ i=42; });
    auto result = experimental::ranges::find(execution::dpcpp_default,
                                             buf, 42);

    return 0;
}
```

Ranges: Programmability and Performance

- A pipeline of 3 algorithms (each using a DPC++ kernel)

```
using namespace oneapi::dpl;
reverse( policy, begin(data), end(data));
transform( policy, begin(data), end(data), begin(result), [](auto i){ return i*i; });
auto res = find_if( policy, begin(result), end(result), pred );
```

- With custom iterators (only 1 kernel)

```
using namespace oneapi::dpl;
auto iter =
    make_transform_iterator(make_reverse_iterator(end(data)), [](auto i){return i*i;});
auto res = find_if( policy, iter, iter + data.size(), pred );
```

- With ranges (only 1 kernel)

```
using namespace oneapi::dpl::experimental::ranges;
auto res = find_if( policy, views::all(data) | views::reverse |
                            views::transform([](auto i){return i*i;}), pred );
```

Experimental Features: Async API

Async API

- oneDPL algorithms with DPC++ execution policies are blocking
  - C++ standard compliant: return when execution completes (on the device)
  - In some cases, may transfer data back to the host
- Experimental explicitly asynchronous API
  - Algorithms never wait, instead returning a future-like object
  - Simultaneous use of multiple devices
  - Exploits DPC++ asynchronous capabilities (hiding latencies, etc.)
  - Composing oneDPL algorithms into static data flow graphs

Async Example

Avoids blocking between chained algorithms

```
#include <oneapi/dpl/execution>
#include <oneapi/dpl/iterator>
#include <oneapi/dpl/async>
#include <sycl/sycl.hpp>

int main() {
    sycl::buffer<int> buf { 1000 };
    auto policy = dpl::execution::dpcpp_default;
    auto first = dpl::begin(buf);
    auto last = dpl::end(buf);
    // returns a future-like object encapsulating a SYCL event
    auto future_1 = dpl::experimental::fill_async(policy, first, last, 42);
    auto future_2 = dpl::experimental::transform_async(
        policy, first, last, first, [](int x){ return x + 1; }, future_1);
    auto reduced_value = dpl::experimental::reduce_async( policy, first, last, future_2).get();
    // .get() method blocks and returns a value
    return 0;
}
```

Summary

And materials for further learning

Summary

oneDPL is a productivity library for heterogeneous computing

- Use C++ standard API in kernels
- Express higher-level parallel patterns with Parallel API
- Target compute devices with custom policies
- Improve expressiveness with custom iterators
- Combine programmability and optimizations with Ranges API
- Control non-blocking behavior with Async API

oneDPL resources

oneDPL specification

https://spec.oneapi.io/versions/latest/elements/oneDPL/source/index.html

oneDPL Library Guide

https://docs.oneapi.io/versions/latest/onedpl/index.html

oneDPL source code

https://github.com/oneapi-src/oneDPL

Build & optimize your code in the Intel® DevCloud for oneAPI

Intel® oneAPI preinstalled and ready to go: https://devcloud.intel.com/oneapi/

oneAPI Resources
software.intel.com/oneapi

Learn and Get Started

- software.intel.com/oneapi
- Training
- Documentation
- Code Samples

oneAPI

Industry Initiative

- oneAPI.com
- oneAPI Industry Specification
- Open Source Implementations

oneAPI

Ecosystem

- Community Forums
- Academic Program
- Intel® DevMesh Innovator Projects

oneAPI Available on

Intel® DevCloud

A development sandbox to develop, test and run workloads across a range of Intel CPUs, GPUs, and FPGAs using Intel's oneAPI software.

Get Up & Running In Seconds!

software.intel.com/devcloud/oneapi

intel. DevCloud

- 1 Minute to Code
- No Hardware Acquisition
- No Download, Install or Configuration
- Easy Access to Samples & Tutorials
- Support for Jupyter Notebooks, Visual Studio Code

Q & A session

Notices & Disclaimers

Intel technologies may require enabled hardware, software or service activation. Learn more at intel.com or from the OEM or retailer.

Your costs and results may vary.

Intel does not control or audit third-party data. You should consult other sources to evaluate accuracy.

Optimization Notice: Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice. Notice Revision #20110804. https://software.intel.com/en-us/articles/optimization-notice

Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors.

Performance tests, such as SYSmark and MobileMark, are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products. See backup for configuration details. For more complete information about performance and benchmark results, visit www.intel.com/benchmarks.

Performance results are based on testing as of dates shown in configurations and may not reflect all publicly available updates. See configuration disclosure for details. No product or component can be absolutely secure.

No license (express or implied, by estoppel or otherwise) to any intellectual property rights is granted by this document.

Intel disclaims all express and implied warranties, including without limitation, the implied warranties of merchantability, fitness for a particular purpose, and non-infringement, as well as any warranty arising from course of performance, course of dealing, or usage in trade.

intel®