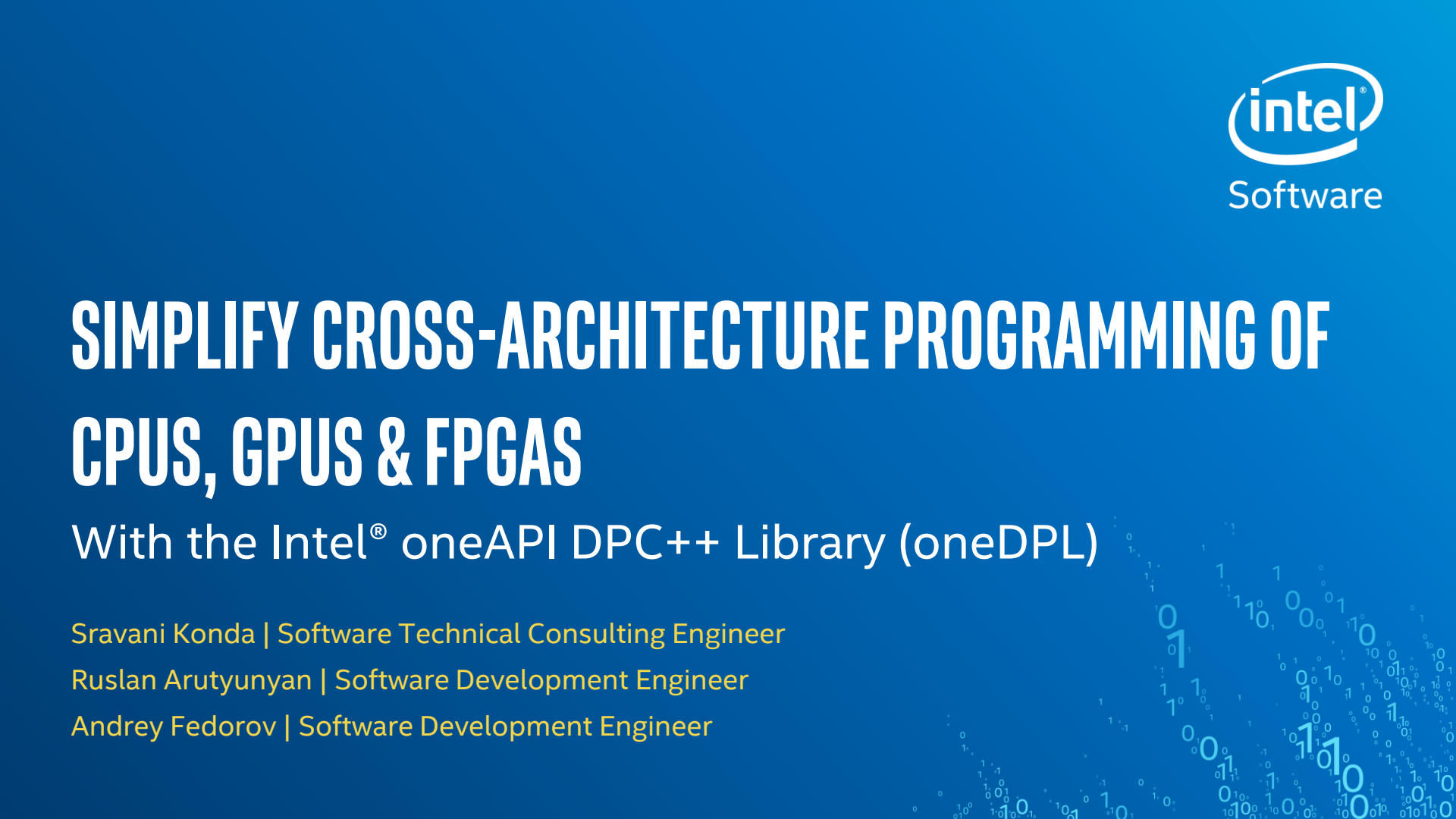


SIMPLIFY CROSS-ARCHITECTURE PROGRAMMING OF CPUS, GPUS & FPGAS

With the Intel® oneAPI DPC++ Library (oneDPL)

Sravani Konda | Software Technical Consulting Engineer
Ruslan Arutyunyan | Software Development Engineer
Andrey Fedorov | Software Development Engineer

Agenda

- Intel® oneAPI DPC++ Library
- C++ Standard APIs for Device Programming
- Parallel STL and its Support for DPC++
- Extension API
- Examples
- Summary

Intel® oneAPI DPC++ Library (oneDPL)

High-Productivity APIs for Heterogeneous Computing - CPUs, GPUs, and FPGAs

APIs are based on familiar standards and libraries - C++ STL, SYCL and Parallel STL, Boost.Compute

Optimized C++ Standard Algorithms - 84 parallelized C++17 algorithms and utilities

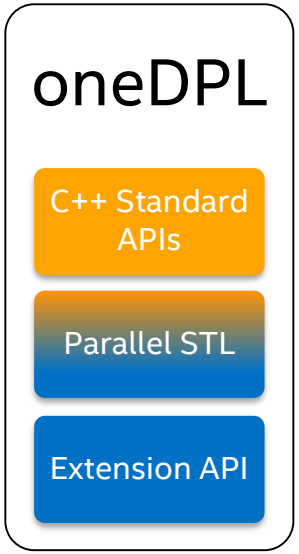
Integrated with Intel® DPC++ Compatibility Tool - Migration of Thrust* APIs

Available on Open Source - https://github.com/oneapi-src/oneDPL

oneDPL Modules

oneDPL consists of 3 Modules

1. C++ STL API functionality
   - Tested C++ Standard APIs that work in DPC++ kernels
2. Parallel STL for DPC++
   - Parallelized C++17 algorithms with DPC++ backend support
3. DPC++ Extension API*
   - Additional library classes and functions



*Extension API is still in beta, the names and semantics might change in future

C++ STD FOR DEVICE PROGRAMMING

C++ Standard Library in DPC++

Host

- Code running on CPU can use everything from STD

Device

- SYCL spec restrictions on C++ features used in device code prevents some std classes/functions from working in device code
  - items using exception
  - dynamic memory allocation
  - virtual functions…

Tested Standard C++ APIs

80+ standard C++ APIs have been tested

- Usage in DPC++ kernels
- Data Transfer from/to host and device

Across 3 major implementations:

1. libstdc++ (GNU): deployed in most Linux distributions
2. libc++ (LLVM): MacOS and FreeBSD
3. MSVC (Microsoft): Windows, shipped with Microsoft Visual Studio

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

Example – std::swap()

```
#include <CL/sycl.hpp>
#include <utility>
#include <iostream>
using namespace cl::sycl;
void kernel_test() {
    sycl::queue deviceQueue;
    sycl::range<1> numOfItems{2};
    sycl::cl_int swap_num[2] = {4, 5};
    std::cout << swap_num[0] << ", " << swap_num[1] << std::endl;
    {
        sycl::buffer<sycl::cl_int, 1> swap_buffer(swap_num, numOfItems);
        deviceQueue.submit([&](sycl::handler& cgh) {
            auto swap_accessor = swap_buffer.get_access<sycl::access::mode::read_write>(cgh);
            cgh.single_task<class KernelSwap>([=] {
                int& num1 = swap_accessor[0];
                int& num2 = swap_accessor[1];
                std::swap(num1, num2);
            });
        });
    }
    std::cout << swap_num[0] << ", " << swap_num[1] << std::endl;
}
int main() {
    kernel_test();
}
```

PARALLEL STL OVERVIEW

Parallel STL Brief Overview

C++17/20 parallel extensions

- Execution policies define how/where to run standard algorithms
- CPU execution: TBB for threading, OpenMP for SIMD
- GPU & accelerators: DPC++ backend support for all algorithms that are part of C++17

Open-source upstream implementation (LLVM) derived from Intel's implementation for CPU

Interface with Execution Policies

- **standard sequential** sort
  sort(v.begin(), v.end());
- **explicitly sequential** sort
  sort(**execution::seq**,v.begin(), v.end());
- permitting **parallel execution**
  sort(**execution::par**,v.begin(), v.end());
- permitting **parallel execution** and **vectorization** as well
  sort(**execution::par_unseq**,v.begin(), v.end());
- permitting **vectorization (C++20)**
  sort(**execution::unseq**,v.begin(), v.end());

Parallel STL in C++

```
#include <algorithm>
#include <execution>
void increment( float *in, float *out, int N ) {
    using namespace std;
    using namespace std::execution;
    transform( par, in, in + N, out, []( float f ) {
        return f+1;
    });
}

template <class ExecutionPolicy, class InputIt, class OutputIt, class
UnaryOp>
OutputIt transform(ExecutionPolicy&& exec, …);
```

Proper overloaded **“transform”** version is resolved by *ExecutionPolicy* type

DPC++ SUPPORT IN PARALLEL STL

Example: Version 0

```
#include <execution>
#include <algorithm>
int main()
{
    std::vector<int> buf(1000);
    std::fill(policy, buf.begin(), buf.end(), 42);
}
```

DPC++ Execution Policies

DPC++ execution policy encapsulates a SYCL queue

The queue defines where the Parallel STL algorithm executes

Policy Usage:

- Include <dpstd/execution> in your code
- Create a policy object by providing a policy type, an (optional) class type for unique kernel name and one of the following constructor arguments
  - A SYCL queue
  - A SYCL device
  - A SYCL device selector
  - An existing policy object
- Pass the created policy object to a Parallel STL algorithm

DPC++ default policy

dpstd::execution::default_policy*

- Predefined object of the device_policy class with default kernel name and default queue.
- Pass directly when invoking an algorithm.

```
std::for_each(default_policy, …);
```

- Create customized policy objects

```
auto policy_a = make_device_policy<class PolicyA>(default_policy);

std::for_each(policy_a, …);
```

* Beta API, the names and semantics might change in future

Example: Version 1

```
#include <CL/sycl.hpp>
#include <dpstd/execution>
#include <dpstd/algorithm>

int main()
{
    std::vector<int> buf(1000);
    std::fill(dpstd::execution::default_policy,
        buf.begin(), buf.end(), 42);
}

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

Example: Version 2

```
#include <CL/sycl.hpp>
#include <dpstd/execution>
#include <dpstd/algorithm>
#include <dpstd/iterator>
int main()
{
     cl::sycl::buffer<int> buf { 1000 };
     std::fill(dpstd::execution::default_policy,
         dpstd::begin(buf), dpstd::end(buf), 42);

     auto result = std::find(dpstd::execution::default_policy,
         dpstd::begin(buf), dpstd::end(buf), 42);
}
```

'result' type guarantees:

- Is CopyConstructible, CopyAssignable, comparable with operators == and !=
- The following expressions are valid: a + n, a - n, a – b
- get_buffer() method. Returns sycl::buffer

Passing Access Modes to the Backend

```
auto buf_begin = dpstd::begin(buf);
std::fill(policy, buf_begin, buf_begin + 1000, 42);

auto buf_begin =
    dpstd::begin<cl::sycl::access::mode::write>(buf);
std::fill(policy, buf_begin, buf_begin + 1000, 42);
```

Unified Shared Memory in DPC++

- An extension to the standard SYCL memory model
- C++ pointer-based alternative to the buffer programming model
- Enables sharing of memory between the host and device
- Provides explicit and implicit models for managing memory

Unified Shared Memory (USM) Support

```
#include <CL/sycl.hpp>
#include <dpstd/execution>
#include <dpstd/algorithm>
int main()
{
    cl::sycl::queue q;
    const int n = 1000;
    int* d_head = static_cast<int*>(cl::sycl::malloc_device(n*
    sizeof(int), q.get_device(), q.get_context()));
    std::fill(dpstd::execution::make_device_policy(q), d_head,
        d_head + n, 42);
    q.wait();
    cl::sycl::free(d_head, q.get_context());
}
```

USM Support (using USM allocator)

```
#include <CL/sycl.hpp>
#include <dpstd/execution>
#include <dpstd/algorithm>
int main()
{
     cl::sycl::queue q;
     const int n = 1000;
     cl::sycl::usm_allocator<int, cl::sycl::usm::alloc::shared>
         alloc(q.get_context(), q.get_device());


     std::vector<int, decltype(alloc)> vec(n, alloc);
     std::fill(dpstd::execution::default_policy, vec.begin(), vec.end(), 42);

      q.wait();
}
```

EXTENSION API

Extension API

Fancy Iterators:

- counting_iterator
- zip_iterator
- transform_iterator

Utilities:

- identity
- minimum
- maximum

Algorithms:

- reduce_by_segment
- inclusive_scan_by_segment
- exclusive_scan_by_segment
- binary_search
- lower_bound
- upper_bound

EXAMPLES

Gamma correction

Example stages:

- Processing image using standard algorithms
- Processing image using standard algorithms with execution policies
- Processing image using standard algorithms with DPC++

Gamma Correction

```
int main() {
  int width = 2560, height = 1600; // Image size is width x height
  Img<ImgFormat::BMP> image{width, height};
  ImgFractal fractal{width, height};
  // Some code with filling image with created fractal
  // ...









}
```

Gamma Correction

```
int main() {
  int width = 2560, height = 1600; // Image size is width x height
  Img<ImgFormat::BMP> image{width, height};
  ImgFractal fractal{width, height};
  // Some code with filling image with created fractal
  // ...

  // Lambda to process image with gamma = 2
  auto gamma_f = [](ImgPixel &pixel) {
    float v = (0.3f * pixel.r + 0.59f * pixel.g + 0.11f * pixel.b) / 255.0f;
    std::uint8_t gamma_pixel = static_cast<std::uint8_t>(255 * v * v);
    if (gamma_pixel > 255)
      gamma_pixel = 255;
    pixel.set(gamma_pixel, gamma_pixel, gamma_pixel, gamma_pixel);
  };









}
```

Gamma Correction

```
int main() {
  int width = 2560, height = 1600; // Image size is width x height
  Img<ImgFormat::BMP> image{width, height};
  ImgFractal fractal{width, height};
  // Some code with filling image with created fractal
  // ...

  // Lambda to process image with gamma = 2
  auto gamma_f = [](ImgPixel &pixel) {
    float v = (0.3f * pixel.r + 0.59f * pixel.g + 0.11f * pixel.b) / 255.0f;
    std::uint8_t gamma_pixel = static_cast<std::uint8_t>(255 * v * v);
    if (gamma_pixel > 255)
      gamma_pixel = 255;
    pixel.set(gamma_pixel, gamma_pixel, gamma_pixel, gamma_pixel);
  };




    // how to handle the image using gamma_f with help of the algorithms?



}
```

Gamma Correction (std algorithm)

```
int main() {
  int width = 2560, height = 1600; // Image size is width x height
  Img<ImgFormat::BMP> image{width, height};
  ImgFractal fractal{width, height};
  // Some code with filling image with created fractal
  // ...

  // Lambda to process image with gamma = 2
  auto gamma_f = [](ImgPixel &pixel) {
    float v = (0.3f * pixel.r + 0.59f * pixel.g + 0.11f * pixel.b) / 255.0f;
    std::uint8_t gamma_pixel = static_cast<std::uint8_t>(255 * v * v);
    if (gamma_pixel > 255)
      gamma_pixel = 255;
    pixel.set(gamma_pixel, gamma_pixel, gamma_pixel, gamma_pixel);
  };



    auto buffer_begin = image.data(); // iterator to pass image to the algorithm
    auto buffer_end = image.data() + image.width() * image.height(); // iterator to pass image to the algorithm



}
```

Gamma Correction (std algorithm)

```
int main() {
  int width = 2560, height = 1600; // Image size is width x height
  Img<ImgFormat::BMP> image{width, height};
  ImgFractal fractal{width, height};
  // Some code with filling image with created fractal
  // ...

  // Lambda to process image with gamma = 2
  auto gamma_f = [](ImgPixel &pixel) {
    float v = (0.3f * pixel.r + 0.59f * pixel.g + 0.11f * pixel.b) / 255.0f;
    std::uint8_t gamma_pixel = static_cast<std::uint8_t>(255 * v * v);
    if (gamma_pixel > 255)
      gamma_pixel = 255;
    pixel.set(gamma_pixel, gamma_pixel, gamma_pixel, gamma_pixel);
  };



    auto buffer_begin = image.data(); // iterator to pass image to the algorithm
    auto buffer_end = image.data() + image.width() * image.height(); // iterator to pass image to the algorithm
    // call std::for_each
    std::for_each(buffer_begin, buffer_end, gamma_f);

}
```

Gamma Correction (std algorithm with policy)

```
int main() {
  int width = 2560, height = 1600; // Image size is width x height
  Img<ImgFormat::BMP> image{width, height};
  ImgFractal fractal{width, height};
  // Some code with filling image with created fractal
  // ...

  // Lambda to process image with gamma = 2
  auto gamma_f = [](ImgPixel &pixel) {
    float v = (0.3f * pixel.r + 0.59f * pixel.g + 0.11f * pixel.b) / 255.0f;
    std::uint8_t gamma_pixel = static_cast<std::uint8_t>(255 * v * v);
    if (gamma_pixel > 255)
      gamma_pixel = 255;
    pixel.set(gamma_pixel, gamma_pixel, gamma_pixel, gamma_pixel);
  };



    auto buffer_begin = image.data(); // iterator to pass image to the algorithm
    auto buffer_end = image.data() + image.width() * image.height(); // iterator to pass image to the algorithm
    // call std::for_each with par_unseq policy
    std::for_each(std::execution::par_unseq, buffer_begin, buffer_end, gamma_f);

}
```

Gamma Correction (std algorithm with DPC++)

```
int main() {
  int width = 2560, height = 1600; // Image size is width x height
  Img<ImgFormat::BMP> image{width, height};
  ImgFractal fractal{width, height};
  // Some code with filling image with created fractal
  // ...

  // Lambda to process image with gamma = 2
  auto gamma_f = [](ImgPixel &pixel) {
    float v = (0.3f * pixel.r + 0.59f * pixel.g + 0.11f * pixel.b) / 255.0f;
    std::uint8_t gamma_pixel = static_cast<std::uint8_t>(255 * v * v);
    if (gamma_pixel > 255)
      gamma_pixel = 255;
    pixel.set(gamma_pixel, gamma_pixel, gamma_pixel, gamma_pixel);
  };




    // call std::for_each with par_unseq policy
    // std::for_each(std::execution::par_unseq, buffer_begin, buffer_end, gamma_f);

}
```

Gamma Correction (std algorithm with DPC++)

```
int main() {
  int width = 2560, height = 1600; // Image size is width x height
  Img<ImgFormat::BMP> image{width, height};
  ImgFractal fractal{width, height};
  // Some code with filling image with created fractal
  // ...

  // Lambda to process image with gamma = 2
  auto gamma_f = [](ImgPixel &pixel) {
    float v = (0.3f * pixel.r + 0.59f * pixel.g + 0.11f * pixel.b) / 255.0f;
    std::uint8_t gamma_pixel = static_cast<std::uint8_t>(255 * v * v);
    if (gamma_pixel > 255)
      gamma_pixel = 255;
    pixel.set(gamma_pixel, gamma_pixel, gamma_pixel, gamma_pixel);
  };


    buffer<ImgPixel, 1> buf(image.data(), image.width() * image.height()); // buffer to move data and count dependencies


    // call std::for_each with par_unseq policy
    // std::for_each(std::execution::par_unseq, buffer_begin, buffer_end, gamma_f);

}
```

Gamma Correction (std algorithm with DPC++)

```
int main() {
  int width = 2560, height = 1600; // Image size is width x height
  Img<ImgFormat::BMP> image{width, height};
  ImgFractal fractal{width, height};
  // Some code with filling image with created fractal
  // ...

  // Lambda to process image with gamma = 2
  auto gamma_f = [](ImgPixel &pixel) {
    float v = (0.3f * pixel.r + 0.59f * pixel.g + 0.11f * pixel.b) / 255.0f;
    std::uint8_t gamma_pixel = static_cast<std::uint8_t>(255 * v * v);
    if (gamma_pixel > 255)
      gamma_pixel = 255;
    pixel.set(gamma_pixel, gamma_pixel, gamma_pixel, gamma_pixel);
  };


    buffer<ImgPixel, 1> buf(image.data(), image.width() * image.height()); // buffer to move data and count dependencies
    auto buffer_begin = dpstd::begin(buf); // iterator to pass buffer to the algorithm
    auto buffer_end = dpstd::end(buf); // iterator to pass buffer to the algorithm
    // call std::for_each with par_unseq policy
    // std::for_each(std::execution::par_unseq, buffer_begin, buffer_end, gamma_f);

}
```

Gamma Correction (std algorithm with DPC++)

```
int main() {
  int width = 2560, height = 1600; // Image size is width x height
  Img<ImgFormat::BMP> image{width, height};
  ImgFractal fractal{width, height};
  // Some code with filling image with created fractal
  // ...

  // Lambda to process image with gamma = 2
  auto gamma_f = [](ImgPixel &pixel) {
    float v = (0.3f * pixel.r + 0.59f * pixel.g + 0.11f * pixel.b) / 255.0f;
    std::uint8_t gamma_pixel = static_cast<std::uint8_t>(255 * v * v);
    if (gamma_pixel > 255)
      gamma_pixel = 255;
    pixel.set(gamma_pixel, gamma_pixel, gamma_pixel, gamma_pixel);
  };


    buffer<ImgPixel, 1> buf(image.data(), image.width() * image.height()); // buffer to move data and count dependencies
    auto buffer_begin = dpstd::begin(buf); // iterator to pass buffer to the algorithm
    auto buffer_end = dpstd::end(buf); // iterator to pass buffer to the algorithm
    // call std::for_each with DPC++
    std::for_each(dpstd::execution::default_policy, buffer_begin, buffer_end, gamma_f);

}
```

Gamma Correction (std algorithm with DPC++)

```
int main() {
  int width = 2560, height = 1600; // Image size is width x height
  Img<ImgFormat::BMP> image{width, height};
  ImgFractal fractal{width, height};
  // Some code with filling image with created fractal
  // ...

  // Lambda to process image with gamma = 2
  auto gamma_f = [](ImgPixel &pixel) {
    float v = (0.3f * pixel.r + 0.59f * pixel.g + 0.11f * pixel.b) / 255.0f;
    std::uint8_t gamma_pixel = static_cast<std::uint8_t>(255 * v * v);
    if (gamma_pixel > 255)
      gamma_pixel = 255;
    pixel.set(gamma_pixel, gamma_pixel, gamma_pixel, gamma_pixel);
  };

  { // We need to have the scope to transfer data in image after buffer's destruction
    buffer<ImgPixel, 1> buf(image.data(), image.width() * image.height()); // buffer to move data and count dependencies
    auto buffer_begin = dpstd::begin(buf); // iterator to pass buffer to the algorithm
    auto buffer_end = dpstd::end(buf); // iterator to pass buffer to the algorithm
    // call std::for_each with DPC++
    std::for_each(dpstd::execution::default_policy, buffer_begin, buffer_end, gamma_f);
  }
}
```

Extension API

Fancy Iterators:

- counting_iterator
- zip_iterator
- transform_iterator

Utilities:

- identity
- minimum
- maximum

Algorithms:

- reduce_by_segment
- inclusive_scan_by_segment
- exclusive_scan_by_segment
- binary_search
- lower_bound
- upper_bound

Counting Iterator

Boost and Thrust also have counting iterator

This represents current index of sequence we process

Counting Iterator. Example

```
#include<dpstd/iterator>
#include<dpstd/execution>
#include<dpstd/algorithm>

int main() {
    const int n = 10;
    sycl::buffer<int> buf(n);
    auto first_counting = dpstd::counting_iterator<int>(1);

    // fill the sequence with 1, ..., n
    std::copy(dpstd::execution::default_policy, first_counting, first_counting + n, dpstd::begin(buf));

    // check result
    auto host_accessor = buf.get_access<sycl::access::mode::read>();
    std::cout << "Output is: " << std::endl;
    for(int i = 0; i < n; ++i) {
        std::cout << host_accessor[i] << " ";
    }
    return 0;
}
```

Zip Iterator

Boost and Thrust also have zip iterator and make_zip_iterator function

This iterator allows you to process some sequence as one

Zip Iterator. Example

```
#include<dpstd/iterator>
#include<dpstd/execution>
#include<dpstd/algorithm>

int main() {
    const int n = 10;
    sycl::buffer<int> x_buf(n);
    sycl::buffer<int> y_buf(n);
    sycl::buffer<int> z_buf(n);
    auto first_counting = dpstd::counting_iterator<int>(1);

    auto first_zip = dpstd::make_zip_iterator(
        dpstd::begin(x_buf), dpstd::begin(y_buf), dpstd::begin(z_buf));

    auto first_counting_zip = dpstd::make_zip_iterator(
        first_counting, first_counting, first_counting);

    // fill x, y, z sequences with 1, ..., n
    std::copy(dpstd::execution::default_policy, first_counting_zip, first_counting_zip + n, first_zip);
    return 0;
}
```

Summary

- oneDPL consists of 3 Modules: C++ STL API functionality with tested STD APIs; Parallel STL for DPC++; DPC++ Extension API*
- Common APIs that can be used on CPU, GPU, FPGAs
- Simplify parallel programming on heterogeneous architectures
- Integrated with other Intel® oneAPI components - DPC++ Compiler and DPC++ Compatibility Tool
- Available on Open Source - https://github.com/oneapi-src/oneDPL

Legal Disclaimer & 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. For more complete information visit www.intel.com/benchmarks.

INFORMATION IN THIS DOCUMENT IS PROVIDED "AS IS". NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT. INTEL ASSUMES NO LIABILITY WHATSOEVER AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO THIS INFORMATION INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT.

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

NOTICES AND DISCLAIMERS

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.

All product plans and roadmaps are subject to change without notice.

Intel technologies may require enabled hardware, software or service activation.

Results have been estimated or simulated.

No product or component can be absolutely secure.

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

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 and the Intel logo are trademarks of Intel Corporation in the U.S. and/or other countries. Other names and brands may be claimed as the property of others. © Intel Corporation.

TECH.
DECODED

intel
Software