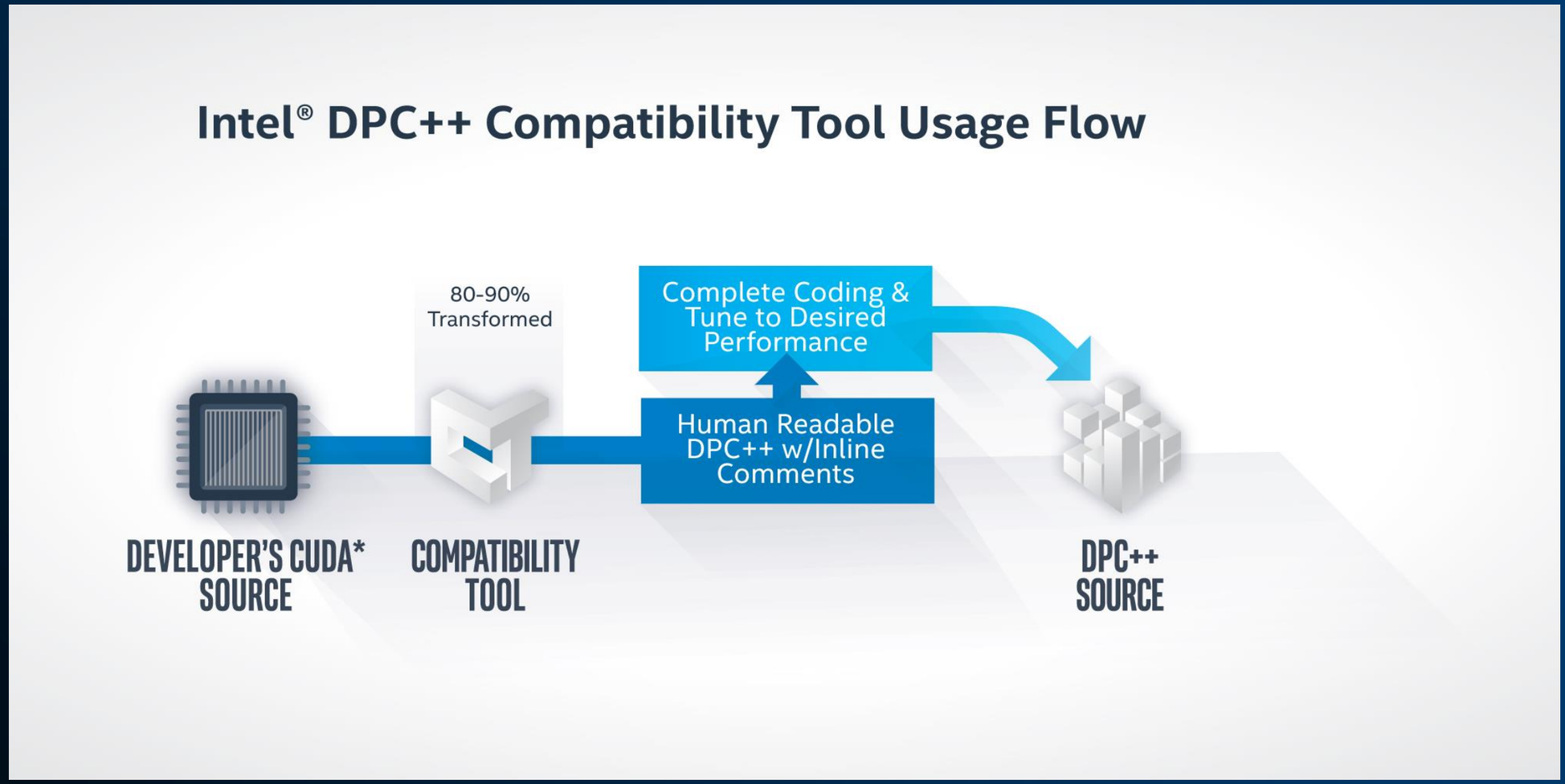
DPC++ Programming Model:
Best Practices

Anoop Madhusoodhanan Prabha, Software Engineer, Intel Corporation

AGENDA

- Migration to DPC++
  - Intel® DPC++ Compatibility tool
  - Offload Advisor Analysis
- Writing DPC++
  - Fundamental Building Blocks of DPC++
  - Synchronization
  - Custom Device Selector
  - Error handling
  - Unified Shared Memory
  - oneAPI DPC++ Library (oneDPL)

MIGRATE DEVELOPER'S CUDA* SOURCE TO DPC++ SOURCE

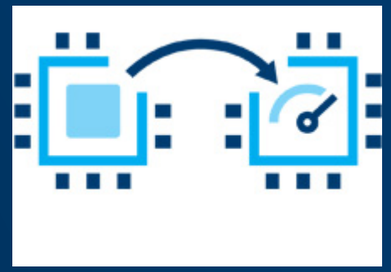


*Other names and brands may be claimed as the property of others.

INTEL® ADVISOR BETA OFFLOAD ADVISOR

oneAPI

- Profiling capabilities from Intel Advisor:
  - Survey Analysis
  - Trip count and FLOPs Analysis
  - Dependency Analysis



```
advixe-python collect.py --config=gen9  <advisor_proj_dir> -- <executable>
```

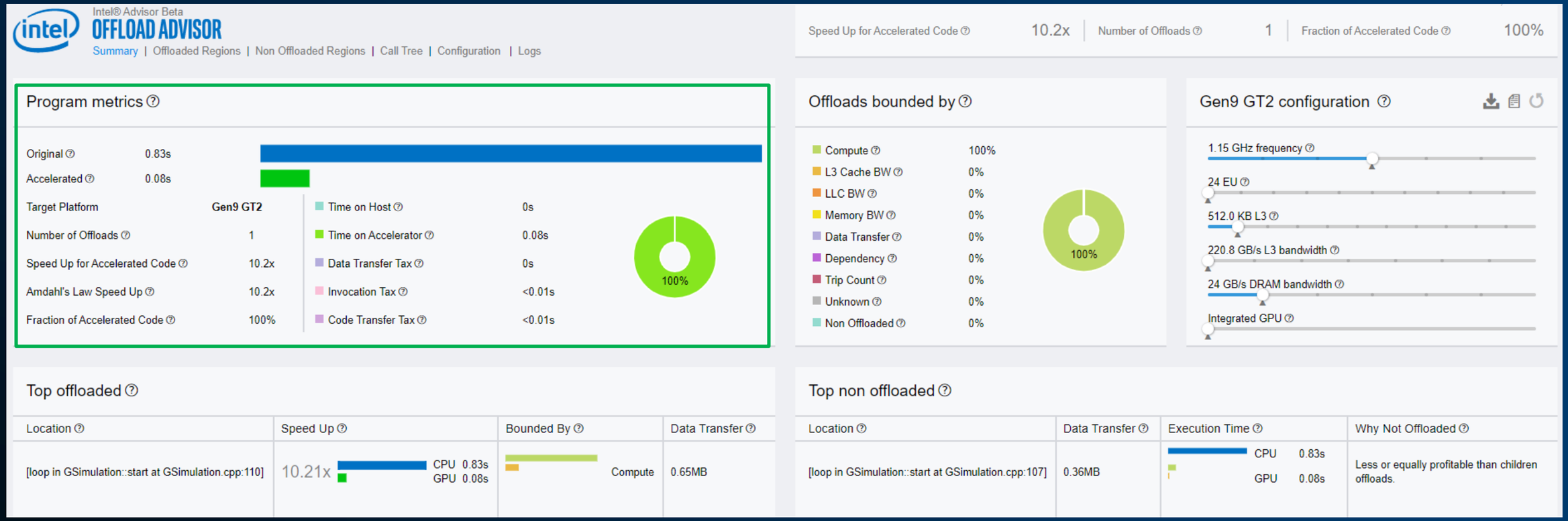
- Performance Modeling
  - Performance prediction for selected DPC++ accelerator device.

```
advixe-python analyze.py --config=gen9 <advisor_proj_dir> -o <advisor_result_dir>
```

  - Open <advisor_result_dir>/report.html to view the analysis results.

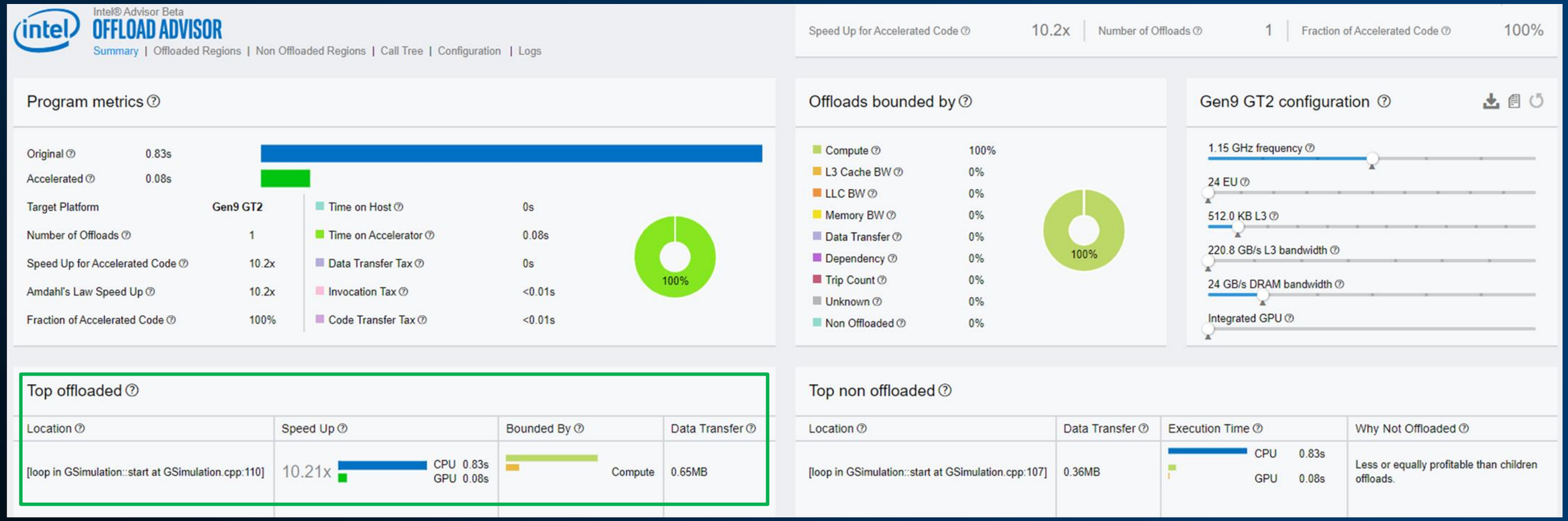
intel

INTEL® ADVISOR BETA OFFLOAD ADVISOR REPORT



Highlights the original runtime of the profiled application, predicted runtime if the identified offload region is run on Gen9 GT2 and the predicted speedup resulting from offload.

INTEL® ADVISOR BETA OFFLOAD ADVISOR REPORT



- **List of identified offload region, predicted speedup when offloaded, characterizes the region as compute/memory bound and predicted data transfer associated with the offload.**
- **These identified loops can be ported to DPC++.**

DPC++ HELLOWORLD

```
//dpcpp -fsycl-unnamed-lambda helloworld.cpp

#include “common_code.hpp”
using namespace cl::sycl;
int main(){
      constexpr int N = 100;
      auto R = range<1>(N);
      std::vector<double> v(N,10);
      queue q;
      {
            buffer<double, 1> buf(v.data(), R);
            q.submit([&](handler &h) {
                  auto a = buf.get_access<dp_rw>(h);
                  cgh.parallel_for(R, [=](id<1> i) {
                        a[i] -= 2;
                  });
            });
      }
      for (int i = 0; i < N; i++)
            std::cout << v[i] << "\n";
      return 0;
}
```

This header captures some common include files and constexpr variables used in all code samples in this slide deck. The contents of this header is available in the backup slide for reference.

DPC++ HELLOWORLD

```
//dpcpp -fsycl-unnamed-lambda helloworld.cpp

#include “common_code.hpp“
using namespace cl::sycl;
int main(){
        constexpr int N = 100;
        auto R = range<1>(N);
        std::vector<double> v(N,10);
        queue q;
        {
                buffer<double, 1> buf(v.data(), R);
                q.submit([&](handler &h) {
                        auto a = buf.get_access<dp_rw>(h);
                        h.parallel_for(R, [=](id<1> i) {
                                a[i] -= 2;
                        });
                });
        }
        for (int i = 0; i < N; i++)
                std::cout << v[i] << "\n";
        return 0;
}
```

Fundamental building blocks of DPC++ like queue, context, device, buffer, accessor etc. are defined under cl::sycl namespace.

DPC++ HELLOWORLD

```
//dpcpp -fsycl-unnamed-lambda helloworld.cpp

#include “common_code.hpp“
using namespace cl::sycl;
int main(){
      constexpr int N = 100;
      auto R = range<1>(N);
      std::vector<double> v(N,10);
      queue q;
      {
            buffer<double, 1> buf(v.data(), R);
            q.submit([&](handler &h) {
                  auto a = buf.get_access<dp_rw>(h);
                  h.parallel_for(R, [=](id<1> i) {
                        a[i] -= 2;
                  });
            });
      }
      for (int i = 0; i < N; i++)
            std::cout << v[i] << "\n";
      return 0;
}
```

range is a templatized class offered by DPC++ which takes the number of dimensions as template parameter and number of arguments matches the dimension specifying the extent in each dimension.

DPC++ HELLOWORLD

```
//dpcpp -fsycl-unnamed-lambda helloworld.cpp

#include “common_code.hpp“
using namespace cl::sycl;
int main(){
      constexpr int N = 100;
      auto R = range<1>(N);
      std::vector<double> v(N,10);
      queue q;
      {
            buffer<double, 1> buf(v.data(), R);
            q.submit([&](handler &h) {
                  auto a = buf.get_access<dp_rw>(h);
                  h.parallel_for(R, [=](id<1> i) {
                        a[i] -= 2;
                  });
            });
      }
      for (int i = 0; i < N; i++)
            std::cout << v[i] << "\n";
      return 0;
}
```

Default constructor of queue chooses the default DPC++ device on the machine and creates a DPC++ queue to connect to this device.

DPC++ HELLOWORLD

```
//dpcpp -fsycl-unnamed-lambda helloworld.cpp

#include “common_code.hpp“
using namespace cl::sycl;
int main(){
      constexpr int N = 100;
      auto R = range<1>(N);
      std::vector<double> v(N,10);
      queue q;
      {
            buffer<double, 1> buf(v.data(), R);
            q.submit([&](handler &h) {
                  auto a = buf.get_access<dp_rw>(h);
                  h.parallel_for(R, [=](id<1> i) {
                        a[i] -= 2;
                  });
            });
      }
      for (int i = 0; i < N; i++)
            std::cout << v[i] << "\n";
      return 0;
}
```

DPC++ Buffer defines a shared array which can accessed using accessors from either host code or device kernel.
Template Parameters:
1. **Data type**
2. **Dimensionality of the data stored**

Arguments:
1. **Pointer to the host data**
2. **Range object specifying the extent of the data in every dimension.**

DPC++ HELLOWORLD

```
//dpcpp -fsycl-unnamed-lambda helloworld.cpp

#include “common_code.hpp“
using namespace cl::sycl;
int main(){
      constexpr int N = 100;
      auto R = range<1>(N);
      std::vector<double> v(N,10);
      queue q;
      {
            buffer<double, 1> buf(v.data(), R);
            q.submit([&](handler &h) {
                  auto a = buf.get_access<dp_rw>(h);
                  h.parallel_for(R, [=](id<1> i) {
                        a[i] -= 2;
                  });
            });
      }
      for (int i = 0; i < N; i++)
            std::cout << v[i] << "\n";
      return 0;
}
```

Submit a command group function object to the DPC++ queue and it takes as argument a command group handler. Command group function object encapsulates:

1. **Accessors to Buffers**
2. **DPC++ kernel**

DPC++ HELLOWORLD

```
//dpcpp -fsycl-unnamed-lambda helloworld.cpp

#include “common_code.hpp“
using namespace cl::sycl;
int main(){
      constexpr int N = 100;
      auto R = range<1>(N);
      std::vector<double> v(N,10);
      queue q;
      {
            buffer<double, 1> buf(v.data(), R);
            q.submit([&](handler &h) {
                  auto a = buf.get_access<dp_rw>(h);
                  h.parallel_for(R, [=](id<1> i) {
                        a[i] -= 2;
                  });
            });
      }
      for (int i = 0; i < N; i++)
            std::cout << v[i] << "\n";
      return 0;
}
```

- **Create an accessor to the buffer to access the data managed by the buffer from DPC++ kernel function.**
- **Accessor describes which buffers are required by the kernel and expresses data dependencies between command groups.**
- **The access mode requested here is read_write. This is passed as a template parameter.**

DPC++ HELLOWORLD

```
//dpcpp -fsycl-unnamed-lambda helloworld.cpp

#include “common_code.hpp“
using namespace cl::sycl;
int main(){
      constexpr int N = 100;
      auto R = range<1>(N);
      std::vector<double> v(N,10);
      queue q;
      {
            buffer<double, 1> buf(v.data(), R);
            q.submit([&](handler &h) {
                  auto a = buf.get_access<dp_rw>(h);
                  h.parallel_for(R, [=](id<1> i) {
                        a[i] -= 2;
                  });
            });
      }
      for (int i = 0; i < N; i++)
            std::cout << v[i] << "\n";
      return 0;
}
```

The DPC++ device kernel function instance is created by parallel_for member function of command group handler. parallel_for takes two parameters:

1. **Range object which specifies the extent of the data to be operated on.**
2. **DPC++ kernel function object which executes the kernel body for every value in the range specified.**

SYNCHRONIZATION

- Synchronization in DPC++ Application
  - Synchronization between host and devices using:
    - Buffer Destruction
    - Host Accessors
    - Wait on SYCL Event
    - Wait on Queue
- Synchronization within DPC++ Kernel
  - Synchronization between work items within a work-group using work-group barriers.
  - No mechanism of synchronization between work-groups.

SYNCHRONIZATION – BUFFER DESTRUCTION

```
//dpcpp -fsycl-unnamed-lambda buffer_destruction.cpp

#include “common_code.hpp“
using namespace cl::sycl;
int main(){
	constexpr int N = 100;
	auto R = range<1>(N);
	std::vector<double> v(N,10);
	queue q;
	{
		buffer<double, 1> buf(v.data(), R);
		q.submit([&](handler &h) {
			auto a = buf.get_access<dp_rw>(h);
			h.parallel_for(R, [=](id<1> i) {
				a[i] -= 2;
			});
		});
	}
	for (int i = 0; i < N; i++)
		std::cout << v[i] << "\n";
	return 0;
}
```

SYNCHRONIZATION – BUFFER DESTRUCTION

```
//dpcpp -fsycl-unnamed-lambda buffer_destruction.cpp

#include “common_code.hpp“
using namespace cl::sycl;
int main(){
      constexpr int N = 100;
      auto R = range<1>(N);
      std::vector<double> v(N,10);
      queue q;
      {
            buffer<double, 1> buf(v.data(), R);
            q.submit([&](handler &h) {
                  auto a = buf.get_access<dp_rw>(h);
                  h.parallel_for(R, [=](id<1> i) {
                        a[i] -= 2;
                  });
            });
      }
      for (int i = 0; i < N; i++)
            std::cout << v[i] << "\n";
      return 0;
}
```

- **Buffer creation happens within a separate C++ scope.**
- **Buffer takes ownership of the data stored in vector.**
- **When execution advances beyond this scope, buffer destructor is invoked which relinquishes the ownership of data and copies back the data to the host memory.**

SYNCHRONIZATION – HOST ACCESSOR

```
//dpcpp -fsycl-unnamed-lambda host_accessor.cpp

#include “common_code.hpp“
using namespace cl::sycl;
int main(){
	constexpr int N = 100;
	auto R = range<1>(N);
	std::vector<double> v(N,10);
	queue q;
	buffer<double, 1> buf(v.data(), R);
	q.submit([&](handler &h) {
		auto a = buf.get_access<dp_rw>(h);
		h.parallel_for(R, [=](id<1> i) {
			a[i] -= 2;
		});
	});
	auto b = buf.get_access<dp_r>();
	for (int i = 0; i < N; i++)
		std::cout<<b[i]<<"\n";
	return 0;
}
```

SYNCHRONIZATION – HOST ACCESSOR

```
//dpcpp -fsycl-unnamed-lambda host_accessor.cpp

#include “common_code.hpp“
using namespace cl::sycl;
int main(){
      constexpr int N = 100;
      auto R = range<1>(N);
      std::vector<double> v(N,10);
      queue q;
      buffer<double, 1> buf(v.data(), R);
      q.submit([&](handler &h) {
            auto a = buf.get_access<dp_rw>(h);
            h.parallel_for(R, [=](id<1> i) {
                  a[i] -= 2;
            });
      });
      auto b = buf.get_access<dp_r>();
      for (int i = 0; i < N; i++)
            std::cout<<b[i]<<"\n";
      return 0;
}
```

Buffer takes ownership of the data stored in vector.

SYNCHRONIZATION – HOST ACCESSOR

```
//dpcpp -fsycl-unnamed-lambda host_accessor.cpp

#include “common_code.hpp“
using namespace cl::sycl;
int main(){
        constexpr int N = 100;
        auto R = range<1>(N);
        std::vector<double> v(N,10);
        queue q;
        buffer<double, 1> buf(v.data(), R);
        q.submit([&](handler &h) {
                auto a = buf.get_access<dp_rw>(h);
                h.parallel_for(R, [=](id<1> i) {
                        a[i] -= 2;
                });
        });
        auto b = buf.get_access<dp_r>();
        for (int i = 0; i < N; i++)
                std::cout<<b[i]<<"\n";
        return 0;
}
```

Creating host accessor is a blocking call and will only return after all enqueued DPC++ kernels that modify the same buffer in any queue completes execution and the data is available to the host via this host accessor.

USING MULTIPLE DPC++ DEVICES

```
//dpcpp –fsycl-unnamed-lambda cpu_gpu_compute.cpp
int main(){
  constexpr int N = 100;
  auto R = range<1>(N/2);
  std::vector<double> v(N,10);
  queue cpuQ(cpu_selector{});
  queue gpuQ(gpu_selector{});
  buffer<int,1> bufgpu(v.data(), R);
  buffer<int,1> bufcpu(v.data()+(N/2), R);
  gpuQ.submit([&](handler &h){
        auto agpu = bufgpu.get_access<dp_rw>(h);
        h.parallel_for (R, [=](id<1> i){
              agpu[i]+=2;
        });
  });
  cpuQ.submit([&](handler &h){
        auto acpu = bufcpu.get_access<dp_rw>(h);
        h.parallel_for(R, [=](id<1> i){
              acpu[i]-=2;
        });
  });
  auto cpu = bufcpu.get_access<dp_r>() ;
  auto gpu = bufgpu.get_access<dp_r>() ;
  for(int i = 0; i < N/2; i++)
        std::cout<<gpu[i]<<"\t"<<cpu[i+(N/2)]<<"\n";
  return 0;
}
```

USING MULTIPLE DPC++ DEVICES

```
//dpcpp –fsycl-unnamed-lambda cpu_gpu_compute.cpp
int main(){
  constexpr int N = 100;
  auto R = range<1>(N/2);
  std::vector<double> v(N,10);
  queue cpuQ(cpu_selector{});
  queue gpuQ(gpu_selector{});
  buffer<int,1> bufgpu(v.data(), R);
  buffer<int,1> bufcpu(v.data()+(N/2), R);
  gpuQ.submit([&](handler &h){
        auto agpu = bufgpu.get_access<dp_rw>(h);
        h.parallel_for (R, [=](id<1> i){
              agpu[i]+=2;
        });
  });
  cpuQ.submit([&](handler &h){
        auto acpu = bufcpu.get_access<dp_rw>(h);
        h.parallel_for(R, [=](id<1> i){
              acpu[i]-=2;
        });
  });
  auto cpu = bufcpu.get_access<dp_r>() ;
  auto gpu = bufgpu.get_access<dp_r>() ;
  for(int i = 0; i < N/2; i++)
        std::cout<<gpu[i]<<"\t"<<cpu[i+(N/2)]<<"\n";
  return 0;
}
```

Create separate queue for CPU and GPU device

USING MULTIPLE DPC++ DEVICES

```
//dpcpp –fsycl-unnamed-lambda cpu_gpu_compute.cpp
int main(){
  constexpr int N = 100;
  auto R = range<1>(N/2);
  std::vector<double> v(N,10);
  queue cpuQ(cpu_selector{});
  queue gpuQ(gpu_selector{});
  buffer<int,1> bufgpu(v.data(), R);
  buffer<int,1> bufcpu(v.data()+(N/2), R);
  gpuQ.submit([&](handler &h){
        auto agpu = bufgpu.get_access<dp_rw>(h);
        h.parallel_for (R, [=](id<1> i){
              agpu[i]+=2;
        });
  });
  cpuQ.submit([&](handler &h){
        auto acpu = bufcpu.get_access<dp_rw>(h);
        h.parallel_for(R, [=](id<1> i){
              acpu[i]-=2;
        });
  });
  auto cpu = bufcpu.get_access<dp_r>() ;
  auto gpu = bufgpu.get_access<dp_r>() ;
  for(int i = 0; i < N/2; i++)
        std::cout<<gpu[i]<<"\t"<<cpu[i+(N/2)]<<"\n";
  return 0;
}
```

Split the data across two buffers which will be passed for compute to CPU and GPU devices

USING MULTIPLE DPC++ DEVICES

```
//dpcpp –fsycl-unnamed-lambda cpu_gpu_compute.cpp
int main(){
  constexpr int N = 100;
  auto R = range<1>(N/2);
  std::vector<double> v(N,10);
  queue cpuQ(cpu_selector{});
  queue gpuQ(gpu_selector{});
  buffer<int,1> bufgpu(v.data(), R);
  buffer<int,1> bufcpu(v.data()+(N/2), R);
  gpuQ.submit([&](handler &h){
        auto agpu = bufgpu.get_access<dp_rw>(h);
        h.parallel_for (R, [=](id<1> i){
              agpu[i]+=2;
        });
  });
  cpuQ.submit([&](handler &h){
        auto acpu = bufcpu.get_access<dp_rw>(h);
        h.parallel_for(R, [=](id<1> i){
              acpu[i]-=2;
        });
  });
  auto cpu = bufcpu.get_access<dp_r>() ;
  auto gpu = bufgpu.get_access<dp_r>() ;
  for(int i = 0; i < N/2; i++)
        std::cout<<gpu[i]<<"\t"<<cpu[i+(N/2)]<<"\n";
  return 0;
}
```

Command group function objects are submitted to CPU and GPU queues and the host execution continues (Non-blocking).

USING MULTIPLE DPC++ DEVICES

```
//dpcpp –fsycl-unnamed-lambda cpu_gpu_compute.cpp
int main(){
  constexpr int N = 100;
  auto R = range<1>(N/2);
  std::vector<double> v(N,10);
  queue cpuQ(cpu_selector{});
  queue gpuQ(gpu_selector{});
  buffer<int,1> bufgpu(v.data(), R);
  buffer<int,1> bufcpu(v.data()+(N/2), R);
  gpuQ.submit([&](handler &h){
        auto agpu = bufgpu.get_access<dp_rw>(h);
        h.parallel_for (R, [=](id<1> i){
              agpu[i]+=2;
        });
  });
  cpuQ.submit([&](handler &h){
        auto acpu = bufcpu.get_access<dp_rw>(h);
        h.parallel_for(R, [=](id<1> i){
              acpu[i]-=2;
        });
  });
  auto cpu = bufcpu.get_access<dp_r>() ;
  auto gpu = bufgpu.get_access<dp_r>() ;
  for(int i = 0; i < N/2; i++)
        std::cout<<gpu[i]<<"\t"<<cpu[i+(N/2)]<<"\n";
  return 0;
}
```

Creation of host accessors will wait for all command groups which operate on the respective buffers to complete execution before copying back the data to the host memory

CUSTOM DEVICE SELECTOR FOR VENDOR SPECIFIC GPU

```
// vendor_gpu_selector.hpp

class vendor_gpu_selector : public device_selector {
public:
        virtual int operator()(const device &d) const {
                 int vendorID = d.get_info<info::device::vendor_id>();
                if ((vendorID == 0x...) && (d.is_gpu()))
                        return 1;
                else
                        return -1;
        }
};
```

- **Machine might have GPUs from multiple vendors and you are interested in offloading to a specific vendor's GPU. Writing custom device selector will pick the interested device.**
- **Custom device selector class can be defined by:**
  - **Derived from device_selector class**
  - **Provides the device ranking logic operator() function.**
  - **The device with highest value returned will be selected.**

ERROR HANDLING IN DPC++

- Synchronous Errors
  - Errors caught when invoking APIs on host side.
  - Can be handled using try/catch block.
- Asynchronous Errors
  - Errors generated during execution of DPC++ Kernel on the device.
  - Not reported immediately as they occur.
  - DPC++ queue/context should be instantiated with asynchronous error handler.
  - Following scenarios invoke the asynchronous exception handler:
    - wait_and_throw()
    - throw_asynchronous()
    - Destruction of queue or context object.

ASYNCHRONOUS ERROR HANDLER

```
// async_exception.hpp

#include “common_code.hpp”
using namespace cl::sycl;
auto exception_handler = [](exception_list list){
  for(auto &excep_ptr : list){
        try{
              std::rethrow_exception(excep_ptr);
        } catch (exception &e){
              std::cout<<"Asynchronous Error caught: "<<e.what()<<"\n";
        }
  }
};
```

Highlighted section of code is the asynchronous error handler. It parses through all the errors generated asynchronously and handles them one by one.

USING ASYNCHRONOUS ERROR HANDLER

```
//dpcpp -fsycl-unnamed-lambda work_group_size.cpp
#include “common_code.hpp“
#include "async_exception.hpp“
#include “vendor_gpu_selector.hpp“
using namespace cl::sycl;
int main(){
	constexpr int N=1024;
	constexpr int WG=512;
	std::vector<double> v(N,10);
	auto R = range<1>(N);
	buffer<int, 1> buf(v.data(), R);
	queue q(vendor_gpu_selector{}, exception_handler);
	q.submit([&](handler &h){
		auto a = buf.get_access<dp_rw>(h);
		h.parallel_for(nd_range<1>(R, range<1>(WG)), [=](nd_item<1> it){
			auto i = it.get_global_id();
			a[i] = i[0];
		});
	}).wait_and_throw();
	auto b = buf.get_access<dp_r>();
	for(int i = 0; i < N; i++)
		std::cout<<b[i]<<"\n";
	return 0;
}
```

USING ASYNCHRONOUS ERROR HANDLER

```
//dpcpp -fsycl-unnamed-lambda work_group_size.cpp
#include “common_code.hpp“
#include "async_exception.hpp“
#include “vendor_gpu_selector.hpp“
using namespace cl::sycl;
int main(){
      constexpr int N=1024;
      constexpr int WG=512;
      std::vector<double> v(N,10);
      auto R = range<1>(N);
      buffer<int, 1> buf(v.data(), R);
      queue q(vendor_gpu_selector{}, exception_handler);
      q.submit([&](handler &h){
            auto a = buf.get_access<dp_rw>(h);
            h.parallel_for(nd_range<1>(R, range<1>(WG)), [=](nd_item<1> it){
                  auto i = it.get_global_id();
                  a[i] = i[0];
            });
      }).wait_and_throw();
      auto b = buf.get_access<dp_r>();
      for(int i = 0; i < N; i++)
            std::cout<<b[i]<<"\n";
      return 0;
}
```

Includes the asynchronous error handler

USING ASYNCHRONOUS ERROR HANDLER

```
//dpcpp -fsycl-unnamed-lambda work_group_size.cpp
#include “common_code.hpp“
#include "async_exception.hpp“
#include “vendor_gpu_selector.hpp“
using namespace cl::sycl;
int main(){
      constexpr int N=1024;
      constexpr int WG=512;
      std::vector<double> v(N,10);
      auto R = range<1>(N);
      buffer<int, 1> buf(v.data(), R);
      queue q(vendor_gpu_selector{}, exception_handler);
      q.submit([&](handler &h){
            auto a = buf.get_access<dp_rw>(h);
            h.parallel_for(nd_range<1>(R, range<1>(WG)), [=](nd_item<1> it){
                  auto i = it.get_global_id();
                  a[i] = i[0];
            });
      }).wait_and_throw();
      auto b = buf.get_access<dp_r>();
      for(int i = 0; i < N; i++)
            std::cout<<b[i]<<"\n";
      return 0;
}
```

Includes the custom vendor specific GPU selector class. This program assumes the vendor as “Intel” and Graphics SKU as Gen9.

USING ASYNCHRONOUS ERROR HANDLER

```
//dpcpp -fsycl-unnamed-lambda work_group_size.cpp
#include “common_code.hpp“
#include "async_exception.hpp“
#include “vendor_gpu_selector.hpp“
using namespace cl::sycl;
int main(){
      constexpr int N=1024;
      constexpr int WG=512;
      std::vector<double> v(N,10);
      auto R = range<1>(N);
      buffer<int, 1> buf(v.data(), R);
      queue q(vendor_gpu_selector{}, exception_handler);
      q.submit([&](handler &h){
            auto a = buf.get_access<dp_rw>(h);
            h.parallel_for(nd_range<1>(R, range<1>(WG)), [=](nd_item<1> it){
                  auto i = it.get_global_id();
                  a[i] = i[0];
            });
      }).wait_and_throw();
      auto b = buf.get_access<dp_r>();
      for(int i = 0; i < N; i++)
            std::cout<<b[i]<<"\n";
      return 0;
}
```

- **Maximum work group size supported by Gen9 graphics is 256.**
- **For demonstration of asynchronous error handling, max work group size is set to 512**

USING ASYNCHRONOUS ERROR HANDLER

```
//dpcpp -fsycl-unnamed-lambda work_group_size.cpp
#include “common_code.hpp“
#include "async_exception.hpp“
#include “vendor_gpu_selector.hpp“
using namespace cl::sycl;
int main(){
      constexpr int N=1024;
      constexpr int WG=512;
      std::vector<double> v(N,10);
      auto R = range<1>(N);
      buffer<int, 1> buf(v.data(), R);
      queue q(vendor_gpu_selector{}, exception_handler);
      q.submit([&](handler &h){
            auto a = buf.get_access<dp_rw>(h);
            h.parallel_for(nd_range<1>(R, range<1>(WG)), [=](nd_item<1> it){
                  auto i = it.get_global_id();
                  a[i] = i[0];
            });
      }).wait_and_throw();
      auto b = buf.get_access<dp_r>();
      for(int i = 0; i < N; i++)
            std::cout<<b[i]<<"\n";
      return 0;
}
```

Queue constructor takes two arguments:

- **Device selector class instance**
- **Asynchronous error handler**

USING ASYNCHRONOUS ERROR HANDLER

```
//dpcpp -fsycl-unnamed-lambda work_group_size.cpp
#include “common_code.hpp“
#include "async_exception.hpp“
#include “vendor_gpu_selector.hpp“
using namespace cl::sycl;
int main(){
      constexpr int N=1024;
      constexpr int WG=512;
      std::vector<double> v(N,10);
      auto R = range<1>(N);
      buffer<int, 1> buf(v.data(), R);
      queue q(vendor_gpu_selector{}, exception_handler);
      q.submit([&](handler &h){
            auto a = buf.get_access<dp_rw>(h);
            h.parallel_for(nd_range<1>(R, range<1>(WG)), [=](nd_item<1> it){
                  auto i = it.get_global_id();
                  a[i] = i[0];
            });
      }).wait_and_throw();
      auto b = buf.get_access<dp_r>();
      for(int i = 0; i < N; i++)
            std::cout<<b[i]<<"\n";
      return 0;
}
```

Invoking parallel_for member function which takes nd_range as argument.

USING ASYNCHRONOUS ERROR HANDLER

```
//dpcpp -fsycl-unnamed-lambda work_group_size.cpp
#include “common_code.hpp“
#include "async_exception.hpp“
#include “vendor_gpu_selector.hpp“
using namespace cl::sycl;
int main(){
      constexpr int N=1024;
      constexpr int WG=512;
      std::vector<double> v(N,10);
      auto R = range<1>(N);
      buffer<int, 1> buf(v.data(), R);
      queue q(vendor_gpu_selector{}, exception_handler);
      q.submit([&](handler &h){
            auto a = buf.get_access<dp_rw>(h);
            h.parallel_for(nd_range<1>(R, range<1>(WG)), [=](nd_item<1> it){
                  auto i = it.get_global_id();
                  a[i] = i[0];
            });
      }).wait_and_throw();
      auto b = buf.get_access<dp_r>();
      for(int i = 0; i < N; i++)
            std::cout<<b[i]<<"\n";
      return 0;
}
```

nd_range constructor takes two arguments:

- **Global range of the data**
- **Work group range of the data.**

USING ASYNCHRONOUS ERROR HANDLER

```
//dpcpp -fsycl-unnamed-lambda work_group_size.cpp
#include “common_code.hpp“
#include "async_exception.hpp“
#include “vendor_gpu_selector.hpp“
using namespace cl::sycl;
int main(){
      constexpr int N=1024;
      constexpr int WG=512;
      std::vector<double> v(N,10);
      auto R = range<1>(N);
      buffer<int, 1> buf(v.data(), R);
      queue q(vendor_gpu_selector{}, exception_handler);
      q.submit([&](handler &h){
            auto a = buf.get_access<dp_rw>(h);
            h.parallel_for(nd_range<1>(R, range<1>(WG)), [=](nd_item<1> it){
                  auto i = it.get_global_id();
                  a[i] = i[0];
            });
      }).wait_and_throw();
      auto b = buf.get_access<dp_r>();
      for(int i = 0; i < N; i++)
            std::cout<<b[i]<<"\n";
      return 0;
}
```

Every item in nd_range is an instance of nd_item. The template parameter of nd_item is the dimensionality of the data.

USING ASYNCHRONOUS ERROR HANDLER

```
//dpcpp -fsycl-unnamed-lambda work_group_size.cpp
#include “common_code.hpp“
#include "async_exception.hpp“
#include “vendor_gpu_selector.hpp“
using namespace cl::sycl;
int main(){
      constexpr int N=1024;
      constexpr int WG=512;
      std::vector<double> v(N,10);
      auto R = range<1>(N);
      buffer<int, 1> buf(v.data(), R);
      queue q(vendor_gpu_selector{}, exception_handler);
      q.submit([&](handler &h){
            auto a = buf.get_access<dp_rw>(h);
            h.parallel_for(nd_range<1>(R, range<1>(WG)), [=](nd_item<1> it){
                  auto i = it.get_global_id();
                  a[i] = i[0];
            });
      }).wait_and_throw();
      auto b = buf.get_access<dp_r>();
      for(int i = 0; i < N; i++)
            std::cout<<b[i]<<"\n";
      return 0;
}
```

Invokes wait_and_throw() is a blocking call which will wait for the command group associated with this DPC++ event to complete execution and if there is any uncaught asynchronous errors, it will call the asynchronous error handler with those errors.

USING ASYNCHRONOUS ERROR HANDLER

```
//dpcpp -fsycl-unnamed-lambda work_group_size.cpp
#include “common_code.hpp“
#include "async_exception.hpp“
#include “vendor_gpu_selector.hpp“
using namespace cl::sycl;
int main(){
      constexpr int N=1024;
      constexpr int WG=512;
      std::vector<double> v(N,10);
      auto R = range<1>(N);
      buffer<int, 1> buf(v.data(), R);
      queue q(vendor_gpu_selector{}, exception_handler);
      q.submit([&](handler &h){
            auto a = buf.get_access<dp_rw>(h);
            h.parallel_for(nd_range<1>(R, range<1>(WG)), [=](nd_item<1> it){
                  auto i = it.get_global_id();
                  a[i] = i[0];
            });
      }).wait_and_throw();
      auto b = buf.get_access<dp_r>();
      for(int i = 0; i < N; i++)
            std::cout<<b[i]<<"\n";
      return 0;
}
```

This program when build/run will catch the following asynchronous error:
OpenCL API failed. OpenCL API returns: -54 (CL_INVALID_WORK_GROUP_SIZE) -54 (CL_INVALID_WORK_GROUP_SIZE)

UNIFIED SHARED MEMORY

oneAPI

- SYCL 1.2.1 specification offers:
  - Buffer/Accessor: For tracking and managing memory transfer and guarantee data consistency across host and DPC++ devices.
- Many HPC and Enterprise applications use pointers to manage data.
- DPC++ Extension for Pointer Based programming:
  - Unified Shared Memory (USM): Device Kernels can access the data using pointers

intel

USM: EXPLICIT DATA TRANSFER

```
//dpcpp -fsycl-unnamed-lambda explicit.cpp
#include “common_code.hpp”
using namespace cl::sycl;
int main(){
        int *a, *d_a;
        queue q;
        auto ctx = q.get_context();
        constexpr int N = 100;
        a = static_cast<int *>(malloc(sizeof(int)*N));
        d_a = static_cast<int *>(malloc_device(sizeof(int)*N, q.get_device(), ctx));
        for(int i = 0; i < N; i++)
                a[i] = i;
        auto e1 = q.memcpy(d_a, a, sizeof(int)*N);
        auto e2 = q.submit([&](handler &h) {
                h.depends_on(e1);
                h.parallel_for(range<1>(N), [=](id<1> i){
                        d_a[i]++;
                });
        });
        q.submit([&](handler &h){
                h.depends_on({e1,e2});
                h.memcpy(a, d_a, sizeof(int)*N);
        }).wait();
        for(int i = 0; i < N; i++)
                std::cout<<a[i]<<"\n";
        free(a);
        free(d_a,ctx);
        return 0;
}
```

- **System malloc is used for allocating array on the host. This memory can’t be accessed from device kernel.**
- **C Style API for USM allocation: malloc_device() allocates memory on the device memory.**

USM: EXPLICIT DATA TRANSFER

```
//dpcpp -fsycl-unnamed-lambda explicit.cpp
#include “common_code.hpp”
using namespace cl::sycl;
int main(){
        int *a, *d_a;
        queue q;
        auto ctx = q.get_context();
        constexpr int N = 100;
        a = static_cast<int *>(malloc(sizeof(int)*N));
        d_a = static_cast<int *>(malloc_device(sizeof(int)*N, q.get_device(), ctx));
        for(int i = 0; i < N; i++)
                a[i] = i;
        auto e1 = q.memcpy(d_a, a, sizeof(int)*N);
        auto e2 = q.submit([&](handler &h) {
                h.depends_on(e1);
                h.parallel_for(range<1>(N), [=](id<1> i){
                        d_a[i]++;
                });
        });
        q.submit([&](handler &h){
                h.depends_on({e1,e2});
                h.memcpy(a, d_a, sizeof(int)*N);
        }).wait();
        for(int i = 0; i < N; i++)
                std::cout<<a[i]<<"\n";
        free(a);
        free(d_a,ctx);
        return 0;
}
```

- **Transfers data from host to device memory.**
- **Captures the DPC++ event associated with this memcpy.**

USM: EXPLICIT DATA TRANSFER

```
//dpcpp -fsycl-unnamed-lambda explicit.cpp
#include "common_code.hpp"
using namespace cl::sycl;
int main(){
        int *a, *d_a;
        queue q;
        auto ctx = q.get_context();
        constexpr int N = 100;
        a = static_cast<int *>(malloc(sizeof(int)*N));
        d_a = static_cast<int *>(malloc_device(sizeof(int)*N, q.get_device(), ctx));
        for(int i = 0; i < N; i++)
                a[i] = i;
        auto e1 = q.memcpy(d_a, a, sizeof(int)*N);
        auto e2 = q.submit([&](handler &h) {
                h.depends_on(e1);
                h.parallel_for(range<1>(N), [=](id<1> i){
                        d_a[i]++;
                });
        });
        q.submit([&](handler &h){
                h.depends_on({e1,e2});
                h.memcpy(a, d_a, sizeof(int)*N);
        }).wait();
        for(int i = 0; i < N; i++)
                std::cout<<a[i]<<"\n";
        free(a);
        free(d_a,ctx);
        return 0;
}
```

- **The second command group waits on the memcpy to return before executing the device kernel by invoking depends_on() member function of the handler.**
- **The device pointer is used inside the DPC++ kernel for compute**
- **The DPC++ event associated with this kernel compute is captured.**

USM: EXPLICIT DATA TRANSFER

```
//dpcpp -fsycl-unnamed-lambda explicit.cpp
#include “common_code.hpp”
using namespace cl::sycl;
int main(){
        int *a, *d_a;
        queue q;
        auto ctx = q.get_context();
        constexpr int N = 100;
        a = static_cast<int *>(malloc(sizeof(int)*N));
        d_a = static_cast<int *>(malloc_device(sizeof(int)*N, q.get_device(), ctx));
        for(int i = 0; i < N; i++)
                a[i] = i;
        auto e1 = q.memcpy(d_a, a, sizeof(int)*N);
        auto e2 = q.submit([&](handler &h) {
                h.depends_on(e1);
                h.parallel_for(range<1>(N), [=](id<1> i){
                        d_a[i]++;
                });
        });
        q.submit([&](handler &h){
                h.depends_on({e1,e2});
                h.memcpy(a, d_a, sizeof(int)*N);
        }).wait();
        for(int i = 0; i < N; i++)
                std::cout<<a[i]<<"\n";
        free(a);
        free(d_a,ctx);
        return 0;
}
```

- **The memcpy operation will wait on events e1 and e2.**
- **Transfers data back from device to host memory.**
- **Invoking wait() on the queue will update USM arrays.**

USM: EXPLICIT DATA TRANSFER

```
//dpcpp -fsycl-unnamed-lambda explicit.cpp
#include “common_code.hpp”
using namespace cl::sycl;
int main(){
        int *a, *d_a;
        queue q;
        auto ctx = q.get_context();
        constexpr int N = 100;
        a = static_cast<int *>(malloc(sizeof(int)*N));
        d_a = static_cast<int *>(malloc_device(sizeof(int)*N, q.get_device(), ctx));
        for(int i = 0; i < N; i++)
                a[i] = i;
        auto e1 = q.memcpy(d_a, a, sizeof(int)*N);
        auto e2 = q.submit([&](handler &h) {
                h.depends_on(e1);
                h.parallel_for(range<1>(N), [=](id<1> i){
                        d_a[i]++;
                });
        });
        q.submit([&](handler &h){
                h.depends_on({e1,e2});
                h.memcpy(a, d_a, sizeof(int)*N);
        }).wait();
        for(int i = 0; i < N; i++)
                std::cout<<a[i]<<"\n";
        free(a);
        free(d_a,ctx);
        return 0;
}
```

No extra C++ scoping around the DPC++ kernel or host accessor creation is needed to access the data unlike the use of buffers.

USM: EXPLICIT DATA TRANSFER

```
//dpcpp -fsycl-unnamed-lambda explicit.cpp
#include “common_code.hpp”
using namespace cl::sycl;
int main(){
        int *a, *d_a;
        queue q;
        auto ctx = q.get_context();
        constexpr int N = 100;
        a = static_cast<int *>(malloc(sizeof(int)*N));
        d_a = static_cast<int *>(malloc_device(sizeof(int)*N, q.get_device(), ctx));
        for(int i = 0; i < N; i++)
                a[i] = i;
        auto e1 = q.memcpy(d_a, a, sizeof(int)*N);
        auto e2 = q.submit([&](handler &h) {
                h.depends_on(e1);
                h.parallel_for(range<1>(N), [=](id<1> i){
                        d_a[i]++;
                });
        });
        q.submit([&](handler &h){
                h.depends_on({e1,e2});
                h.memcpy(a, d_a, sizeof(int)*N);
        }).wait();
        for(int i = 0; i < N; i++)
                std::cout<<a[i]<<"\n";
        free(a);
        free(d_a,ctx);
        return 0;
}
```

Free the allocated memory. The function takes as argument both the pointer and context in which it was allocated.

USM ALLOCATOR CLASS

```
//dpcpp -fsycl-unnamed-lambda usmallocator.cpp
#include “common_code.hpp”
using namespace cl::sycl;
int main(){
      queue q;
      constexpr int N = 100;
      usm_allocator<int, usm::alloc::shared> alloc(q.get_context(), q.get_device());
      std::vector<int, decltype(alloc)> v(N, 10, alloc);
      int *ptr = &v[0];
      q.submit([&](handler &h) {
            h.parallel_for(range<1>(N), [=](id<1> i) {
                  ptr[i[0]]++;
            });
      }).wait();
      std::cout<<v[99]<<"\n";
      return 0;
}
```

1. **usm_allocator is a C++ allocator class for USM. Takes the data type and kind of allocation as template parameter**
2. **This allocator is passed to std::vector constructor.**

USM ALLOCATOR CLASS

```
//dpcpp -fsycl-unnamed-lambda usmallocator.cpp
#include “common_code.hpp”
using namespace cl::sycl;
int main(){
      queue q;
      constexpr int N = 100;
      usm_allocator<int, usm::alloc::shared> alloc(q.get_context(), q.get_device());
      std::vector<int, decltype(alloc)> v(N, 10, alloc);
      int *ptr = &v[0];
      q.submit([&](handler &h) {
            h.parallel_for(range<1>(N), [=](id<1> i) {
                  ptr[i[0]]++;
            });
      }).wait();
      std::cout<<v[99]<<"\n";
      return 0;
}
```

Get the pointer from the allocated vector

USM ALLOCATOR CLASS

```
//dpcpp -fsycl-unnamed-lambda usmallocator.cpp
#include “common_code.hpp”
using namespace cl::sycl;
int main(){
        queue q;
        constexpr int N = 100;
        usm_allocator<int, usm::alloc::shared> alloc(q.get_context(), q.get_device());
        std::vector<int, decltype(alloc)> v(N, 10, alloc);
        int *ptr = &v[0];
        q.submit([&](handler &h) {
            h.parallel_for(range<1>(N), [=](id<1> i) {
                ptr[i[0]]++;
            });
        }).wait();
        std::cout<<v[99]<<"\n";
        return 0;
}
```

The same host pointer can be used on the device side inside the DPC++ kernel

USM ALLOCATOR CLASS

```
//dpcpp -fsycl-unnamed-lambda usmallocator.cpp
#include “common_code.hpp”
using namespace cl::sycl;
int main(){
      queue q;
      constexpr int N = 100;
      usm_allocator<int, usm::alloc::shared> alloc(q.get_context(), q.get_device());
      std::vector<int, decltype(alloc)> v(N, 10, alloc);
      int *ptr = &v[0];
      q.submit([&](handler &h) {
            h.parallel_for(range<1>(N), [=](id<1> i) {
                  ptr[i[0]]++;
            });
      }).wait();
      std::cout<<v[99]<<"\n";
      return 0;
}
```

- **Submitting command group function object to a queue will return DPC++ event.**
- **Invoking wait() on the DPC++ event is a blocking call and will only return when the DPC++ kernel finishes execution**

ONEAPI DPC++ LIBRARY

oneAPI

- OneDPL consists of 3 modules:
  - STL functionality
  - Parallel STL with DPC++ backend
    - 84 parallelized C++17 algorithms
  - API extensions
- Increase the successful application of parallel algorithms with custom iterators.

intel

PARALLEL STL (PSTL)

- C++17 standard

  Parallel implementation of STL algorithms

  – Standard Sequential Transform

  ```
  transform(v.begin(), v.end(), v.begin(), [](){ });
  ```

  – Explicit Sequential Transform (Host Only)

  ```
  transform(execution::seq, v.begin(), v.end(), v.begin(), [](){ });
  ```

  – Parallel Execution Transform (Host Only)

  ```
  transform(execution::par, v.begin(), v.end(), v.begin(), [](){ });
  ```

  – Parallel and Vectorized Transform (Host Only)

  ```
  transform(execution::par_unseq, v.begin(), v.end(), v.begin(), [](){ });
  ```

  – Vectorized Transform (Host Only, C++20 standard)

  ```
  transform(execution::unseq, v.begin(), v.end(), v.begin(), [](){ });
  ```

PSTL TRANSFORM WITH DPC++ EXECUTION POLICY

```
//dpcpp -fsycl-unnamed-lambda dpc_transform.cpp
#include “common_code.hpp”
#include<dpstd/execution>
#include<dpstd/algorithm>
using namespace cl::sycl;
using namespace dpstd::execution;
int main(){
        queue q;
        constexpr int N = 100;
        std::vector<int> v(N,10);
        auto exec_policy = make_sycl_policy(q);
        std::transform(exec_policy, v.begin(), v.end(), v.begin(), [](int &a) -> int { return ++a;
});
        for(auto it = v.begin(); it < v.end(); it++)
                std::cout<<(*it)<<"\n";
        return 0;
}
```

Use of oneAPI DPC++ Library APIs demands inclusion of these headers

PSTL TRANSFORM WITH DPC++ EXECUTION POLICY

```
//dpcpp -fsycl-unnamed-lambda dpc_transform.cpp
#include “common_code.hpp”
#include<dpstd/execution>
#include<dpstd/algorithm>
using namespace cl::sycl;
using namespace dpstd::execution;
int main(){
        queue q;
        constexpr int N = 100;
        std::vector<int> v(N,10);
        auto exec_policy = make_sycl_policy(q);
        std::transform(exec_policy, v.begin(), v.end(), v.begin(), [](int &a) -> int { return ++a;
});
        for(auto it = v.begin(); it < v.end(); it++)
                std::cout<<(*it)<<"\n";
        return 0;
}
```

The execution policy and other DPC++ library APIs are defined under this namespace.

PSTL TRANSFORM WITH DPC++ EXECUTION POLICY

```
//dpcpp -fsycl-unnamed-lambda dpc_transform.cpp
#include “common_code.hpp”
#include<dpstd/execution>
#include<dpstd/algorithm>
using namespace cl::sycl;
using namespace dpstd::execution;
int main(){
      queue q;
      constexpr int N = 100;
      std::vector<int> v(N,10);
      auto exec_policy = make_sycl_policy(q);
      std::transform(exec_policy, v.begin(), v.end(), v.begin(), [](int &a) -> int { return ++a;
});
      for(auto it = v.begin(); it < v.end(); it++)
            std::cout<<(*it)<<"\n";
      return 0;
}
```

make_sycl_policy() creates a DPC++ execution policy and applies this to the DPC++ device to which the queue is connected to.

PSTL TRANSFORM WITH DPC++ EXECUTION POLICY

```
//dpcpp -fsycl-unnamed-lambda dpc_transform.cpp
#include “common_code.hpp”
#include<dpstd/execution>
#include<dpstd/algorithm>
using namespace cl::sycl;
using namespace dpstd::execution;
int main(){
        queue q;
        constexpr int N = 100;
        std::vector<int> v(N,10);
        auto exec_policy = make_sycl_policy(q);
        std::transform(exec_policy, v.begin(), v.end(), v.begin(), [](int &a) -> int { return ++a;
});
        for(auto it = v.begin(); it < v.end(); it++)
                std::cout<<(*it)<<"\n";
        return 0;
}
```

- **Pass the execution policy as first argument to the std::transform.**
- **PSTL API will handle both the data transfer and the compute.**

DPC++ BUFFER ITERATORS

```
//dpcpp -fsycl-unnamed-lambda buffer_iterator.cpp
#include “common_code.hpp”
#include<dpstd/execution>
#include<dpstd/algorithm>
using namespace cl::sycl;
using namespace dpstd::execution;
int main(){
      queue q;
      constexpr int N = 100;
      std::vector<int> v(N,10);
      {
            buffer<int,1> buf(v.data(), range<1>(N));
            auto exec_policy = make_sycl_policy(q);
            auto start = dpstd::begin(buf);
            auto end = dpstd::end(buf);
            std::transform(exec_policy, start, end, start, [](int a) -> int { return ++a; });
      }
      for(int i = 0; i < N; i++)
            std::cout<<v[i]<<"\n";
      return 0;
}
```

DPC++ Library provides begin() and end() interfaces for getting buffer iterators

Start with **oneAPI today** for an accelerated xPU future

Get the oneAPI specification at
oneAPI.com

Use Intel's oneAPI Beta

Test code and workloads across a range of Intel® data-centric architectures at
Intel® DevCloud for oneAPI

software.intel.com/devcloud/oneAPI

Learn more and download the **beta toolkits** at
software.intel.com/oneapi

oneAPI

oneAPI AVAILABLE NOW ON INTEL DEVCLOUD

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

software.intel.com/en-us/devcloud/oneapi

- Learn about oneAPI Toolkits
- Learn Data Parallel C++
- Evaluate Workloads
- Build Heterogenous Applications
- Prototype your project

NO DOWNLOADS | NO HARDWARE ACQUISITION | NO INSTALLATION | NO SET-UP AND CONFIGURATION

GET UP AND RUNNING IN SECONDS!

intel

NOTICES & DISCLAIMERS

oneAPI

This document contains information on products, services and/or processes in development. All information provided here is subject to change without notice.

Intel technologies' features and benefits depend on system configuration and may require enabled hardware, software or service activation. Learn more at intel.com, or from the OEM or retailer.

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

intel

QUESTIONS & ANSWERS


intel