DPC++ Programming Model:
Best Practices

Additional Slides

Anoop Madhusoodhanan Prabha, Software Engineer, Intel Corporation

COMMON CODE INCLUDED IN ALL EXAMPLES (IN THIS SLIDE DECK)

```
//common_code.hpp
#include<CL/sycl.hpp>
#include<vector>
#include<iostream>
constexpr auto dp_r = cl::sycl::access::mode::read;
constexpr auto dp_rw = cl::sycl::access::mode::read_write;
constexpr auto dp_w = cl::sycl::access::mode::write;
```

USM ALLOCATION TYPES

Type	Description	Accessibly By		Migratable To	
Device	Device allocation	Host	✗	Host	✗
		Device	✓	Device	✗
		Other device	?	Other device	✗
Host	Host allocation	Host	✓	Host	✗
		Any device	✓	Device	✗
Shared	Potentially migrating allocation	Host	✓	Host	✓*
		Device	✓	Device	✓*
		Other device	?	Other device	?

USM: SHARED MEMORY

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

malloc_shared() allocates memory accessible by both the host and the device.

USM: SHARED MEMORY

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

Same pointer used on the host side can be used on the device side.

USM: SHARED MEMORY

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

Free the memory allocated

INVOKING DPC++ KERNEL MULTIPLE TIMES

```
//dpcpp -fsycl-unnamed-lambda multi_kernel.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);
                for(int i = 0; i < 3; i++){
                        h.parallel_for(R, [=](id<1> i) {
                                a[i] -= 2;
                        });
                }
        });
        q.wait();
        for (int i = 0; i < N; i++)
                std::cout<<v[i]<<"\n";
        return 0;
}
```

For loop inside the command group scope – NOT ALLOWED

INVOKING DPC++ KERNEL MULTIPLE TIMES

```
//dpcpp -fsycl-unnamed-lambda dpc_queue.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);
        for(int i = 0; i < 3; i++){
                q.submit([&](handler &h) {
                        auto a = buf.get_access<dp_rw>(h);
                                h.parallel_for(R, [=](id<1> i) {
                                        a[i] -= 2;
                                });
                        }
                });
        }
        q.wait();
        for (int i = 0; i < N; i++)
                std::cout<<v[i]<<"\n";
        return 0;
}
```

For loop should be outside the command group scope and enqueues the command group multiple times in queue

INVOKING MATH FUNCTIONS FROM DPC++ KERNEL

```
//dpcpp -fsycl-unnamed-lambda math_function.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 gpuQ(gpu_selector{});
      buffer<double, 1> buf(v.data(), R);
      gpuQ.submit([&](handler &h) {
            auto a = buf.get_access<dp_rw>(h);
                  h.parallel_for(R, [=](id<1> i) {
                        a[i] = cl::sycl::exp(a[i]);
                  });
      });
      auto b = buf.get_access<dp_r>();
      for (int i = 0; i < N; i++)
            std::cout<<v[i]<<"\n";
      return 0;
}
```

INVOKING MATH FUNCTIONS FROM DPC++ KERNEL

```
//dpcpp -fsycl-unnamed-lambda math_function.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 gpuQ(gpu_selector{});
      buffer<double, 1> buf(v.data(), R);
      gpuQ.submit([&](handler &h) {
            auto a = buf.get_access<dp_rw>(h);
                  h.parallel_for(R, [=](id<1> i) {
                        a[i] = cl::sycl::exp(a[i]);
                  });
      });
      auto b = buf.get_access<dp_r>();
      for (int i = 0; i < N; i++)
            std::cout<<v[i]<<"\n";
      return 0;
}
```

This example uses GPU device

INVOKING MATH FUNCTIONS FROM DPC++ KERNEL

```
//dpcpp -fsycl-unnamed-lambda math_function.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 gpuQ(gpu_selector{});
      buffer<double, 1> buf(v.data(), R);
      gpuQ.submit([&](handler &h) {
            auto a = buf.get_access<dp_rw>(h);
                  h.parallel_for(R, [=](id<1> i) {
                        a[i] = cl::sycl::exp(a[i]);
                  });
      });
      auto b = buf.get_access<dp_r>();
      for (int i = 0; i < N; i++)
            std::cout<<v[i]<<"\n";
      return 0;
}
```

Compiler may take exp() from standard C headers, global/anonymous namespace rather than cl::sycl.

SYCL_EXTERNAL MACRO

```
//dpcpp -fsycl-unnamed-lambda math_function.cpp
func.cpp

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

```
// func.cpp

#include "common_code.hpp“
SYCL_EXTERNAL double func(double a){
return (cl::sycl::exp(a));
}
```

Functions called from DPC++ kernel and not residing in the same compilation unit should be annotated with SYCL_EXTERNAL macro.

DPC++ ZIP ITERATORS

```
//dpcpp -fsycl-unnamed-lambda zip_iterator.cpp –std=c++14
#include "common_code.hpp“
#include<dpstd/execution>
#include<dpstd/algorithm>
#include<dpstd/iterators.h>
using namespace cl::sycl;
using namespace dpstd::execution;
int main() {
   queue q;
   constexpr int n = 100;
   std::vector<int> v1(n, 1), v2(n, 2), v3(n, 0);
   auto start = dpstd::make_zip_iterator(v1.begin(), v2.begin(), v3.begin());
   auto end = dpstd::make_zip_iterator(v1.end(), v2.end(), v3.end());
   auto exec_policy = make_sycl_policy(q);
   std::for_each(exec_policy, start, end, [](auto t) {
      using std::get;
      get<2>(t) = get<1>(t) + get<0>(t);
   });
   for (auto it = v3.begin(); it < v3.end(); it++)
      std::cout << (*it) << "\n";
   std::cout << std::endl;
   return 0;
}
```

- **STL algorithms has limitation on the number of data sources it can operate on. This limitation comes the number of iterators we can provide as argument to a STL algorithm.**
- **Zip iterators enables relax this limitation.**

DPC++ ZIP ITERATORS

```
//dpcpp -fsycl-unnamed-lambda zip_iterator.cpp –std=c++14
#include "common_code.hpp“
#include<dpstd/execution>
#include<dpstd/algorithm>
#include<dpstd/iterators.h>
using namespace cl::sycl;
using namespace dpstd::execution;
int main() {
   queue q;
   constexpr int n = 100;
   std::vector<int> v1(n, 1), v2(n, 2), v3(n, 0);
   auto start = dpstd::make_zip_iterator(v1.begin(), v2.begin(), v3.begin());
   auto end = dpstd::make_zip_iterator(v1.end(), v2.end(), v3.end());
   auto exec_policy = make_sycl_policy(q);
   std::for_each(exec_policy, start, end, [](auto t) {
      using std::get;
      get<2>(t) = get<1>(t) + get<0>(t);
   });
   for (auto it = v3.begin(); it < v3.end(); it++)
      std::cout << (*it) << "\n";
   std::cout << std::endl;
   return 0;
}
```

This header should be included to use zip iterators

DPC++ ZIP ITERATORS

```
//dpcpp -fsycl-unnamed-lambda zip_iterator.cpp –std=c++14
#include "common_code.hpp“
#include<dpstd/execution>
#include<dpstd/algorithm>
#include<dpstd/iterators.h>
using namespace cl::sycl;
using namespace dpstd::execution;
int main() {
   queue q;
   constexpr int n = 100;
   std::vector<int> v1(n, 1), v2(n, 2), v3(n, 0);
   auto start = dpstd::make_zip_iterator(v1.begin(), v2.begin(), v3.begin());
   auto end = dpstd::make_zip_iterator(v1.end(), v2.end(), v3.end());
   auto exec_policy = make_sycl_policy(q);
   std::for_each(exec_policy, start, end, [](auto t) {
      using std::get;
      get<2>(t) = get<1>(t) + get<0>(t);
   });
   for (auto it = v3.begin(); it < v3.end(); it++)
      std::cout << (*it) << "\n";
   std::cout << std::endl;
   return 0;
}
```

Zip Iterator zips up the iterators of individual containers of interest.

DPC++ ZIP ITERATORS

```
//dpcpp -fsycl-unnamed-lambda zip_iterator.cpp –std=c++14
#include "common_code.hpp“
#include<dpstd/execution>
#include<dpstd/algorithm>
#include<dpstd/iterators.h>
using namespace cl::sycl;
using namespace dpstd::execution;
int main() {
   queue q;
   constexpr int n = 100;
   std::vector<int> v1(n, 1), v2(n, 2), v3(n, 0);
   auto start = dpstd::make_zip_iterator(v1.begin(), v2.begin(), v3.begin());
   auto end = dpstd::make_zip_iterator(v1.end(), v2.end(), v3.end());
   auto exec_policy = make_sycl_policy(q);
   std::for_each(exec_policy, start, end, [](auto t) {
      using std::get;
      get<2>(t) = get<1>(t) + get<0>(t);
   });
   for (auto it = v3.begin(); it < v3.end(); it++)
      std::cout << (*it) << "\n";
   std::cout << std::endl;
   return 0;
}
```

The zip iterator is used for expressing bounds in PSTL algorithms.

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


intel