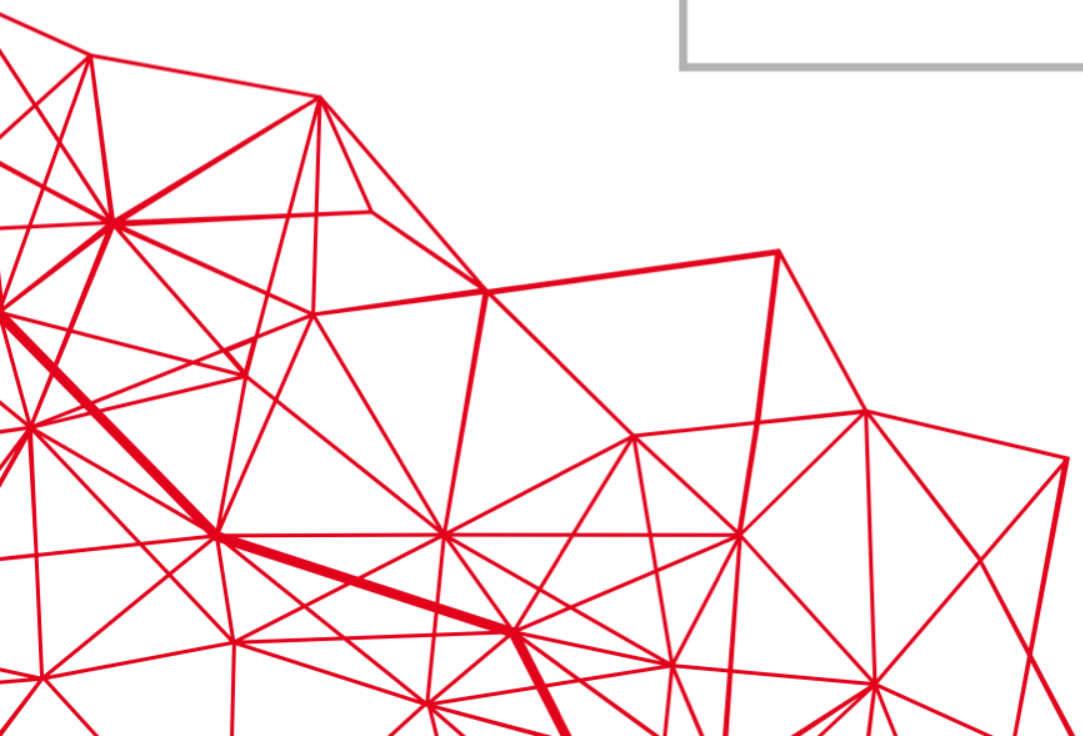


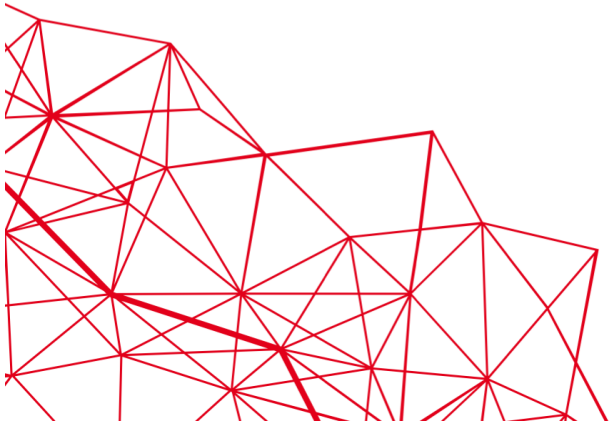
ISC High Performance

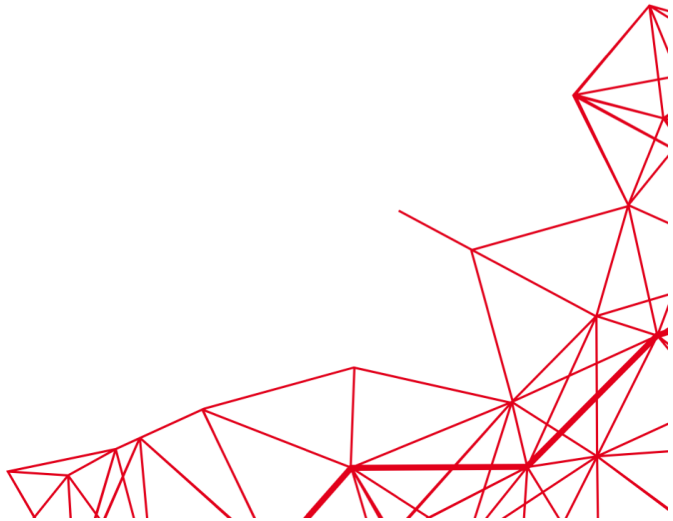
IMAGINE TOMORROW

MAY 21 – 25, 2023 | HAMBURG, GERMANY

Porting numerical integration codes from CUDA to oneAPI: a case study

ISC 2023
MAY 21 – 25
#ISC23





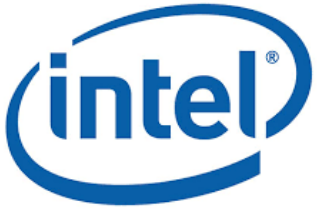
Ioannis Sakiotis (isaki001@odu.edu)
Dr. Kamesh Arumugam
Dr. Marc Paterno
Dr. Desh Ranjan
Dr. Balsa Terzic
Dr. Mohammad Zubair

Outline

- Motivation
- Use-case
- Code migration
- Performance issues
- Observations

Execution Platform Portability

- Devices with different architectures
- No widely-adopted common standards
- Multiple implementations to target multiple architectures

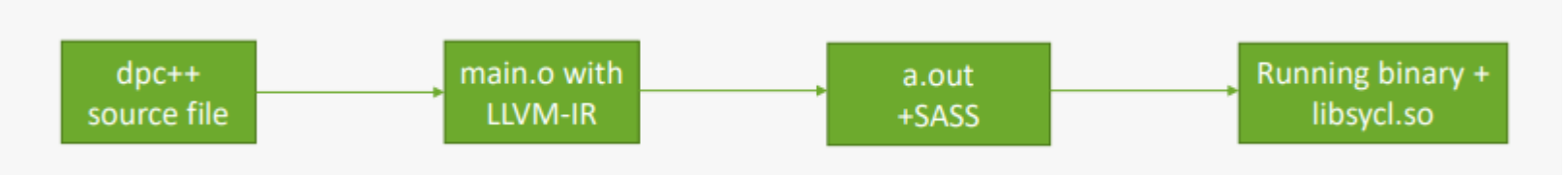


Platform Agnostic Programming Models

- Historically most popular: NVIDIA GPUs
- Aurora: Intel GPUs
- Frontier: AMD GPUs
- Kokkos, Raja, SYCL

Execution Platform Portability

- oneAPI platform towards portability
- DPC++: conformant to C++ and SYCL standards
- Growing oneAPI ecosystem
- Intel DPC++ Compatibility Tool
- NVIDIA GPUs through the CUDA backend for SYCL
- Wide range of performance penalties



https://sycl.tech/assets/files/Michel_Migdal_Codeplay_Porting_Tips_CDUA_To_SYCL.pdf

Numerical Integration in Scientific Computing

- Multi-dimensional non-smooth integrands
- GPU accelerated libraries
- **PAGANI**: deterministic quadrature
- **m-Cubes**: Monte Carlo quadrature
- Optimized in CUDA for NVIDIA V100
- Execute on upcoming clusters
- Not all problems justify overhead of GPU execution

```
class F_1_6D {
public:
  __host__ __device__ double
  operator()(double s,
             double t,
             double u,
             double v,
             double w,
             double x)
  {
    return cos(s + 2. * t + 3. * u + 4. * v + 5. * w + 6. * x);
  }
};
```

Parallel Integration Algorithms

PAGANI

- Big kernel computing integral
- Small kernels for post-processing
- NVIDIA Thrust library
- Used C++ Classes for device-data
- C++ classes for initialization on the host & execution on the device

m-Cubes

- Two big kernels
- Serial post-processing on host

CODE MIGRATION

Two Stage Porting Process

- Manual
  - Intel Developer Cloud
  - Straightforward kernel conversion
  - Processing library options was difficult
  - Only small kernel example in sample codes
- Intel DPC++ Compatibility Tool
  - Easy on Intel Developer Cloud
  - Difficult on CUDA-backend

```
template<size_t ndim>
struct Sub_regions{
  Sub_regions() {}
  Sub_regions(sycl::queue& q, const size_t partitions_per_axis);
  ~Sub_regions();

  void host_device_init(sycl::queue& q, const size_t numRegions);
  void print_bounds();
  double compute_region_volume(size_t const regionID);
  double compute_total_volume();
  void uniform_split(sycl::queue& q, size_t numOfDivisionPerRegionPerDimension);
  quad::Volume<double, ndim> extract_region(size_t const regionID);

  double* dLeftCoord = nullptr;
  double* dLength = nullptr;
  size_t size = 0;
  sycl::queue* _q;

};
```

Code Migration Issues

- The type of accessors for shared memory

```
template <typename T>
__device__ void
blockReduceMinMax(T& min, T& max)
{
  static __shared__ T shared_max[32];
  static __shared__ T shared_min[32];
```

```
q_ct1.submit([&](sycl::handler& cgh) {
  sycl::accessor<dpct_placeholder /*Fix the type mannually*/,
                 1,
                 sycl::access_mode::read_write,
                 sycl::access::target::local>
    shared_max_acc_ct1(sycl::range(32), cgh);
```

Code Migration Issues

- CUDA: atomicAdd
- Didn’t work: dpct::atomic_fetch_add
  - ptxas fatal   : Unresolved extern function
- Worked: sycl::atomic_ref

Code Migration Issues

- NVIDIA Thrust min_max function
- Intel DPC++ Compatibility tool did not support migration

```
thrust::device_ptr<T> d_ptrE = thrust::device_pointer_cast(arr);
auto __tuple = thrust::minmax_element(d_ptrE, d_ptrE + size);
range.low = *__tuple.first;
range.high = *__tuple.second;
```

Code Migration Issues

Undefined reference on CUDA backend

```
quad::Range<T> range;
auto q = dpct::get_default_queue();
double* min = sycl::malloc_shared<double>(1, q);
double* max = sycl::malloc_shared<double>(1, q);

oneapi::mkl::stats::dataset<oneapi::mkl::stats::layout::row_major, T*>
  wrapper(1, size, arr);

auto this_event =
  oneapi::mkl::stats::min_max<oneapi::mkl::stats::method::fast,
                              double,
                              oneapi::mkl::stats::layout::row_major>(
        q, wrapper, min, max);
this_event.wait();

range.low = min[0];
range.high = max[0];
free(min, q);
free(max, q);
```

worked

```
quad::Range<T> range;
if constexpr (use_custom == true && cuda_backend == true) {
  auto q = dpct::get_default_queue();
  int64_t* min = sycl::malloc_shared<int64_t>(1, q);
  int64_t* max = sycl::malloc_shared<int64_t>(1, q);
  const int stride = 1;

  sycl::event est_ev =
    oneapi::mkl::blas::column_major::iamax(q, size, arr, stride, max);

  sycl::event est_ev2 =
    oneapi::mkl::blas::column_major::iamin(q, size, arr, stride, min);

  est_ev.wait();
  est_ev2.wait();

  quad::cuda_memcpy_to_host<T>(&range.low, &arr[min[0]], 1);
  quad::cuda_memcpy_to_host<T>(&range.high, &arr[max[0]], 1);
  free(min, q);
```

Code Migration Issues

- No matching call for call to ‘dpct::inner_product’
- Remove use of different types T1 and T2
- Oneapi::mkl::blas::column_major_dot

```
thrust::device_ptr<T1> wrapped_mask_1 = thrust::device_pointer_cast(arr1);
thrust::device_ptr<T2> wrapped_mask_2 = thrust::device_pointer_cast(arr2);
double res = thrust::inner_product(thrust::device,
                                   wrapped_mask_2,
                                   wrapped_mask_2 + size,
                                   wrapped_mask_1,
                                   0.);

return res;
```

```
T1* res = sycl::malloc_shared<T1>(1, q);
auto est_ev =
  oneapi::mkl::blas::column_major::dot(q, size, arr1, 1, arr2, 1, res);
est_ev.wait();
double result = res[0];
sycl::free(res, q);
```

Code Migration Issues: Passing arguments to kernels

- CUDA: Encapsulate device arrays in objects
- **Sub_regions** allocates and deallocates device-data
- **Sub_regions** is not trivially copyable
- In SYCL kernel is replaced by a lambda expression
- SYCL error when lambda captures regions

```
void cuda_wrapper(const Sub_regions& regions) {
  const size_t nBlocks = regions.size;
  const size_t nThreads = 64;
  kernel<<<nBlocks, nThreads>>>(regions.leftcoord);
  cudaDeviceSynchronize();
}
```

Code Migration Issues

- Can pass pointer
- **regions** is allocated host
- cannot access pointer on device

```
void bad_sycl_wrapper(Sub_regions* regions) {
  sycl::queue q(sycl::gpu_selector());
  const size_t nBlocks = regions->size;
  const size_t nThreads = 64;

  q.submit([&](sycl::handler& h) {
    using range = sycl::range<1>;
    using ndrange = sycl::nd_range<1>;
    using nditem = sycl::nd_item<1>;
    auto total_size = range(nBlocks) * range(nThreads);
    auto group_size = range(nThreads);
    auto kernel = [=](nditem item_ct1){
      // accessing captured wrapper function's argument
      // yields a run-time error.
      double x = regions->leftcoord[0];
      ...
    };
    h.parallel_for(nd_range(total_size, group_size),
                   kernel);
  });
  q.wait_and_throw();
}
```

Code Migration Issues

- Create local variable and access that instead of pointer
- Suggestion: place parallel code in wrapper function

```
void sycl_wrapper(Sub_regions* regions) {
  sycl::queue q(sycl::gpu_selector());
  const size_t nBlocks = regions->size;
  const size_t nThreads = 64;
  const double* leftcoord = regions->leftcoord;

  q.submit([&](sycl::handler& h) {
    using range = sycl::range<1>;
    using ndrange = sycl::nd_range<1>;
    using nditem = sycl::nd_item<1>;
    auto total_size = range(nBlocks) * range(nThreads);
    auto group_size = range(nThreads);
    auto kernel = [=](nditem item_ct1){
      // accessing captured local variable is fine.
      double x = leftcoord[0];
      ...
    };
    h.parallel_for(nd_range(total_size, group_size),
                   kernel);
  });
  q.wait_and_throw();
}
```

Code Migration Issues

- Catch2 testing framework
- Compilation errors on test code
- **DPL** and **Catch2** headers
- Intel DPC++ Conversion Tool placed dpct/dpl_utils.hpp header causing the same error

```
#define CATCH_CONFIG_MAIN
#include "catch2/catch.hpp"
#include <oneapi/dpl/execution>
#include <oneapi/dpl/algorithm>

//error: ranges/nanorange.hpp:3303:46: error: reference to '
    match_results' is ambiguous

TEST_CASE("TEST HEADER INCLUSION")
{
  sycl::queue q;
}
```

Code Migration Issues

- CMake to build tests and demos
- Manual compilation: OK
- CMake configuration on Intel Developer Cloud: OK
- CMake configuration on CUDA backend: Challenging
- Initially, separate build for CUDA backend
- Needed one CMake configuration for the repository

Code Migration Issues

- Specify architecture for CUDA and oneAPI with different strings
- Pass CMake variable to signal usage of CUDA backend
- set(CMAKE_CXX_COMPILER clang++)
- set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -fsycl -fsycl-targets=nvptx64-nvidia-cuda -Xsycl-target-backend --cuda-gpu-arch=${ONEAPI_TARGET_ARCH}")
- target_compile_options(exec PRIVATE "-lonemkl" "-mllvm" "-inline-threshold=10000")

PERFORMANCE

Performance Issues

- Compared CUDA and SYCL on V100
- Initial PAGANI performance: ~5x slower
- Initial m-Cubes performance:~80 times slower

Methodology

- Execute main kernels with different integrands
- Vary computational intensity
- Export detailed metrics through **nvprof –print-gpu-trace**
- Use .csv files in R script to compute statistical data and visualize metrics
- Manual inspection of Nsight Compute reports

Experimentation

- Combined shared memory accessors
- Removed templates
- Manually set loop-unrolling
- Kernels to invoke integrands on the device hundreds of times
- Experimented with simpler integrands
- Different execution times with mathematical functions (pow, trigonometric functions)

Compare integrand invocations

- 5 – 8 dimensions
- Generate 1 million points
- Mean of 10 kernel executions
- cos, pow, exp
- No fast-math flags
- Worse: case 4% slowdown

Table 1. mean (μ) and standard deviation (σ) of execution times for invoking $5-8D$ benchmark integrands

id	μ CUDA (ms)	μ oneAPI (ms)	σ CUDA	σ oneAPI	$\frac{\mu\ oneAPI}{\mu\ CUDA}$
f1	1866.4	1952.4	13.3	21.4	1.04
f2	8413.9	8487.3	5012.5	5042.9	1.009
f3	1812.4	1828.3	18.5	27.1	1.009
f4	11416.1	11410.1	2184.9	2148.1	0.99
f5	634.3	654.4	73.5	67.3	1.03
f6	300.4	300.8	32.05	32.6	1.001

m-Cubes: main kernel execution

- Various thread block configurations
- 100 million to 1 billion samples
- Mean of 100 calls per kernel
- 10% penalty on f2 and f4

Table 2. m-Cubes: mean (μ) and standard deviation (σ) of execution times for 8D benchmark integrands in CUDA and oneAPI

id	μ CUDA (ms)	μ oneAPI (ms)	σ CUDA	σ oneAPI	$\frac{\mu\ oneAPI}{\mu\ CUDA}$
f1	286.7	286.7	2.1	0.9	1.0
f2	402.1	443.1	2.6	0.9	1.1
f3	284.5	285.8	1.6	1.4	1.0
f4	385.7	423.5	2.4	0.5	1.1
f5	284.3	285.9	2.1	1.7	1.0
f6	283.8	285.4	1.9	1.6	1.0

PAGANI: main kernel execution

- Various thread block configurations
- 1 block per region
- Different number of regions
- Different resolution splits on the unit hypercube $(0,1)^d$
- Mean of 100 calls per kernel
- Highest penalties: f2 and f4

Table 3. PAGANI: mean (μ) and standard deviation (σ) of execution times for 8D benchmark integrands in CUDA and oneAPI

id	μ CUDA (ms)	μ oneAPI (ms)	σ CUDA	σ oneAPI	$\frac{\mu\ oneAPI}{\mu\ CUDA}$
f1	172.3	177.5	0.9	1.2	1.02
f2	1500.4	1651.0	0.3	2.1	1.1
f3	286.4	290.7	0.8	0.4	1.01
f4	1434.7	1524.9	0.4	1.9	1.06
f5	166.5	170.7	0.6	0.4	1.03
f6	136.8	139.4	0.4	0.2	1.02

Compare simple integrands

- Observed different performance on mathematical functions
- Eliminate any cause of potential deviation in the integrands
- Addition integrands
- $\sum_{i=1}^{d} x_i$
- 5 – 8 dimensions

Table 4. m-Cubes: mean (μ) and standard deviation (σ) of execution times for addition integrands ($\sum_{i=1}^{d} x_i$) in CUDA and oneAPI

id	μ CUDA (ms)	μ oneAPI (ms)	σ CUDA	σ oneAPI	$\frac{\mu\ oneAPI}{\mu\ CUDA}$
5D	206.1	214.5	2.1	1.7	1.04
6D	214.1	217.2	2.2	1.0	1.01
7D	234.1	235.2	1.8	0.9	1.005
8D	284.7	285.7	1.9	1.9	1.005

Table 5. PAGANI: mean (μ) and standard deviation (σ) of execution times for addition integrands ($\sum_{i=1}^{d} x_i$) in CUDA and oneAPI

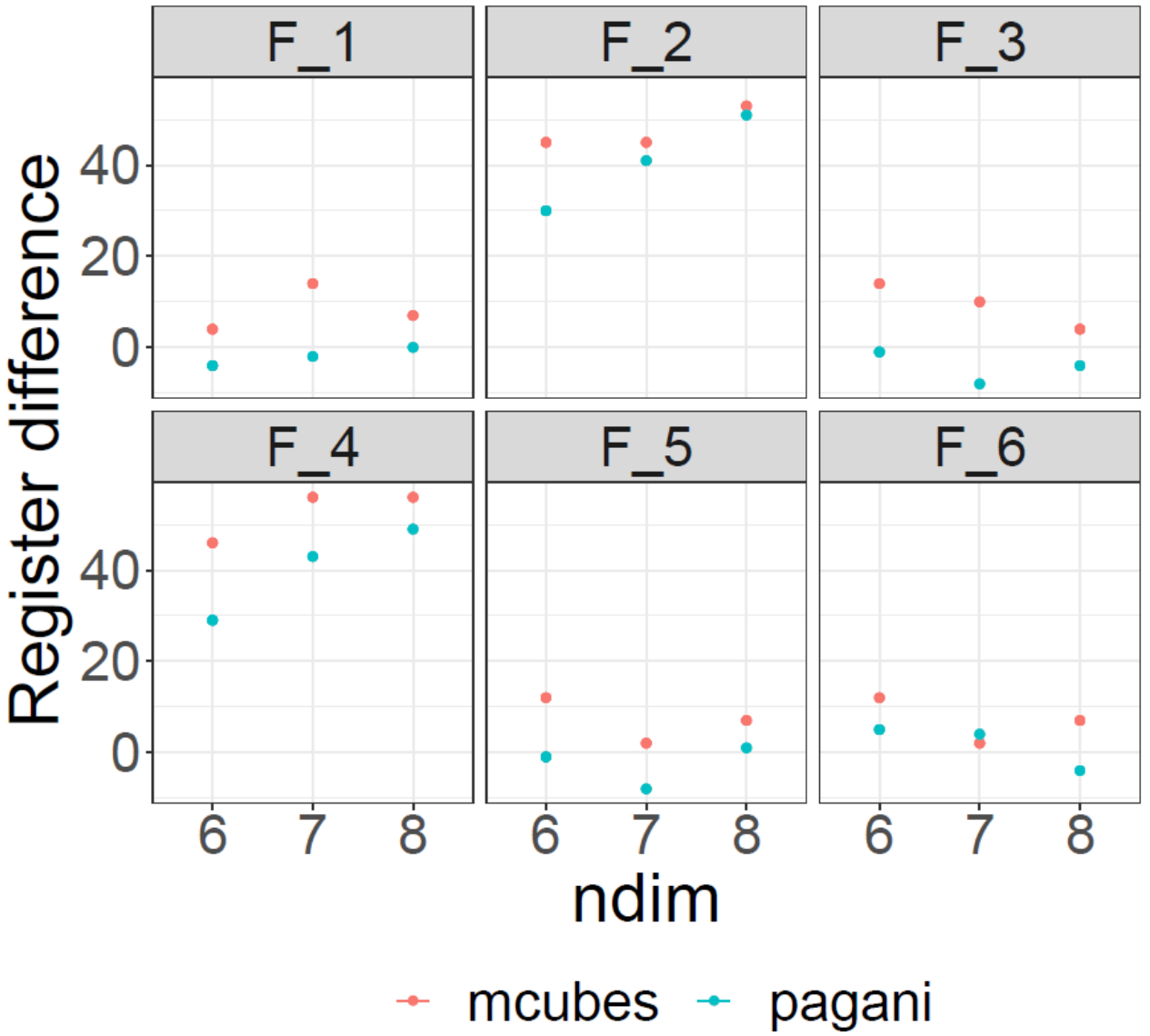
id	CUDA (ms)	oneAPI (ms)	Std. CUDA	Std. oneAPI	$\frac{oneAPI}{CUDA}$
5D	1.5	1.7	0.05	0.06	1.1
6D	24.8	26.7	0.3	1.4	1.1
7D	129.8	131.6	0.7	0.2	1.01
8D	137.4	137.6	1.3	1.0	1.001

Observations

- Bad performance of atomics due to not including device architecture flag
- Setting inline-threshold to 10,000 allowed us to achieve the 10% slowdown
- Using nd_item<1> yielded slight improvement over nd_item<3>
- Reduce_over_group was sensitive to inline threshold
- Register count difference

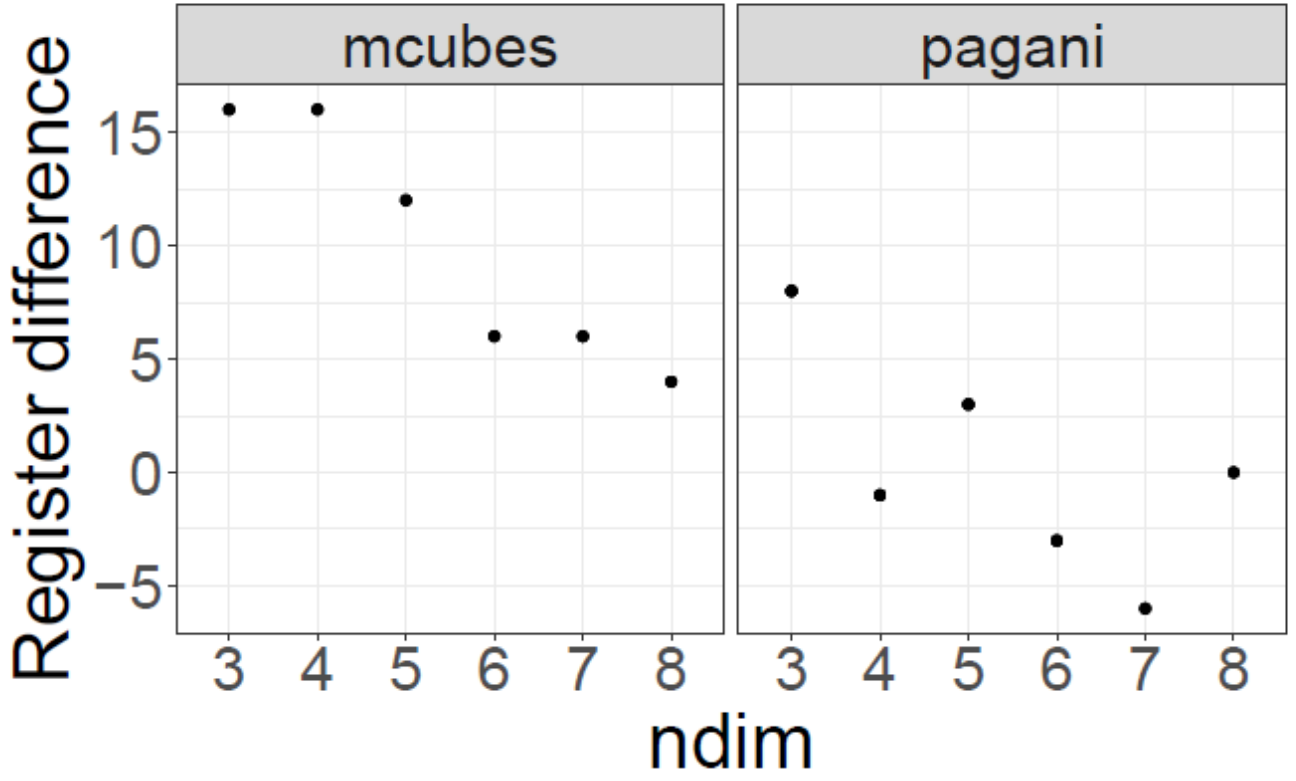
Achieved Performance with oneAPI

- Numerous performance tests
- CUDA -> oneAPI: 10% performance penalty
- Variance in performance penalty
- Increased register usage



Achieved Performance with oneAPI

- Numerous performance tests
- CUDA -> oneAPI: 10% performance penalty
- Variance in performance penalty
- Increased register usage



Conclusion

- Intel DPC++ Compatibility tool is great
- Mapping library calls may be challenging
- Clang and NVCC optimizations
- Loop-unrolling and inline-thresholds are critical
- Register usage can degrade performance
- 0-10% performance penalty on NVIDIA GPUs