The 11th International workshop on OpenCL and SYCL

IWOCL & SYCLcon 2023

The Complexity in Standardizing Complex Numbers in SYCL

Nevin “:-)” Liber, Argonne National Laboratory

with Yasaman Ghadar

Thomas Applencourt, Aiden Belton-Schure, Jefferson le Quellec, Amanda Dufek, Bryce Allen, Brice Videau, Kevin Harms

U.S. DEPARTMENT OF ENERGY

Argonne NATIONAL LABORATORY

WHO AM I?

WHO AM I?

Nevin ":-)" Liber

- Argonne National Laboratory
  - Computer Scientist
    - Argonne Leadership Computing Facility
  - C++, SYCL, Kokkos
  - Aurora
  - WG21 - ISO C++ Committee
    - Vice Chair, Library Evolution Working Group Incubator (LEWGI / SG18)
  - INCITS/C++ - US C++ Committee
    - Vice Chair
    - Admin Chair
  - Khronos SYCL Committee Member

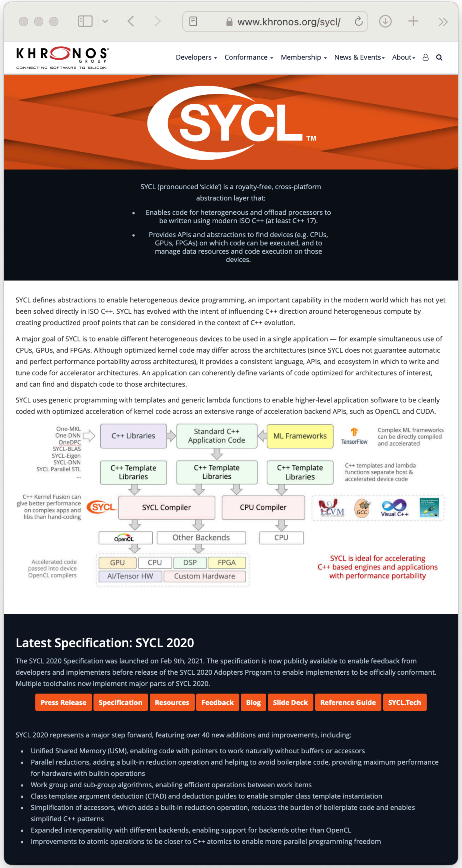
CURRENT COMPLEX NUMBER SUPPORT

CPU & GPU MODELS WITH COMPLEX NUMBER SUPPORT

- C
  - `floating_point_type _Complex`
- C++
  - `std::complex<T>`
- CUDA
  - `cuComplex` (`float`)
- CUDA, ROCm
  - `thrust::complex<T>`
- OpenMP
  - Relies on underlying C++ and C implementations
- Kokkos
  - `Kokkos::complex<T>`

SYCL

- Historically based on OpenCL
  - Does not support complex numbers
- Fragmentation in the SYCL ecosystem
  - Some implementations provide it as an extension
    - Intel used (C++17) `std::complex` for some GPUs



C++20 STD::COMPLEX

Why not standardize C++17/C++20 `std::complex` for SYCL?

- Implementations not guaranteed to be trivially copyable / device copyable
  - *In practice, very likely they are trivially copyable (more on this later)*
- SYCL not guaranteed to have same representation, endianness, padding, etc. between host and device
  - *We'd like to address this in SYCL-Next*
- Implementation not guaranteed to avoid host-only language features
  - Virtual functions, exceptions, etc.
    - *In practice, very unlikely to be implemented with these features*

C++20 STD::COMPLEX

Why not standardize C++17/C++20 `std::complex` for SYCL?

- Not guaranteed `std::complex<sycl::half>` compiles or works
  - `std::complex<T>` was implemented as three specializations
    - `float`, `double`, `long double`
    - Implicit vs. `explicit` conversions
  - Not future-proof

C++23 STD::COMPLEX

Why not standardize C++23 `std::complex` for SYCL?

- `std::complex<T>` requirements on T relaxed
  - Trivially copyable type
  - Literal type
    - Useable as a `constexpr` variable
  - Numeric type
    - Default constructible, copyable, destructible
    - No operations throw exceptions
  - Still unspecified if T is not a floating point type
- `std::complex<T>` itself is a trivially copyable, literal, numeric type

C++<u>23</u> STD::COMPLEX

Why not standardize C++<u>23</u> `std::complex` for SYCL?

- "No operations throw exceptions" is about specification, not implementation
  - Implementation might have internal `try`/`throw`/`catch` block
    - *In practice, not likely*
- Only guaranteed to work on C++23 (extended) floating point types
  - `sycl::half` is not a C++ floating point type, extended or otherwise
    - C++23 has a requirement that all C++ floating point types have an overload for `abs`, `floor`, trigonometry functions, etc.
      - And for the math functions, must be in `<cmath>` & `<math.h>`

C++23 STD::COMPLEX

Why not standardize C++23 `std::complex` for SYCL?

- Required `complex<long double>` unclear
  - Host
    - Yes
  - Device
    - Not all devices support it
    - Don’t want to require emulation
  - Compile time
    - User-defined floating-point literals require `long double` parameter
      - `complex<long double> operator""_Z(long double);`

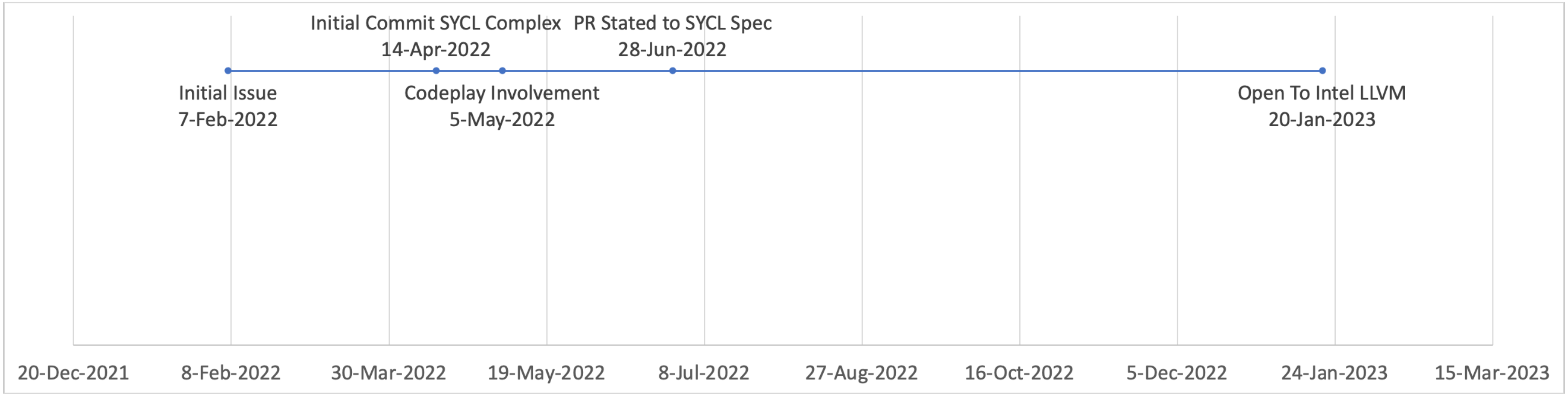
SYCL::COMPLEX PROPOSAL

SYCL::COMPLEX PROPOSAL

- `sycl::complex` specification for the SYCL-Next standard
- SyclCLPX
  - Reference implementation
  - Validation

SYCL::COMPLEX PROPOSAL

Timeline



SYCL::COMPLEX VS STD::COMPLEX

SYCL::COMPLEX VS STD::COMPLEX

- Can instantiate it with `sycl::half`
  - Extend in the future, if necessary
- No `long double` support
  - Some platforms won't support it
- Can convert to/from `std::complex`
- Cannot `reinterpret_cast<cv T(&)[2]>` an object of type `cv sycl::complex<T>`
- Operators are hidden `friends`
- Support fast-math

- Proposed extension for `sycl::marray<sycl::complex<T>, N>`
  - Math operators, comparison operators, math functions
- Possible future extension for `sycl::vec`

SYCL::COMPLEX

Free functions

- Support for SYCL-2020 math functions
  - `abs`, `acos`, `asin`, `atan`, `acosh`, `asinh`, `atanh`, `cos`, `cosh`, `exp`, `log`, `log10`, `pow`, `sin`, `sinh`, `sqrt`, `tan`, `tanh`
- New functions
  - `real(z)` — real component
  - `imag(z)` — imaginary component
  - `arg(z)` — phase angle in radians
  - `norm(z)` — squared magnitude
  - `conj(z)` — conjugate
  - `proj(z)` — projection
  - `polar(rho, theta)` — complex number from polar coordinates

OPERATORS AS HIDDEN FRIENDS OF COMPLEX TYPE

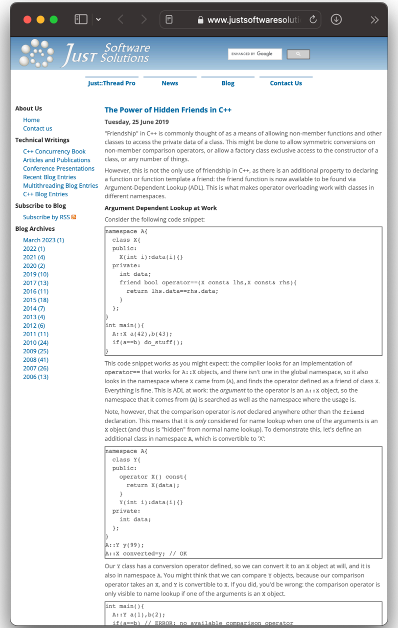
OPERATORS AS HIDDEN FRIENDS OF COMPLEX TYPE

Hidden `friend` function

- Only considered if one of the parameters is `sycl::complex<T>`
  - Not hidden from Argument Dependent Lookup (ADL)
  - Hidden from unqualified name lookup
  - Hidden from qualified name lookup

- Avoids accidental implicit conversions
- Smaller overload set
- Speeds up compilation

- The Power of Hidden Friends in C++ — *Anthony Williams*
  - https://www.justsoftwaresolutions.co.uk/cplusplus/hidden-friends.html



www.justsoftwaresoluti

Just Software Solutions

Just::Thread Pro | News | Blog | Contact Us

About Us
Home
Contact us

Technical Writings
C++ Concurrency Book
Articles and Publications
Conference Presentations
Recent Blog Entries
Multithreading Blog Entries
C++ Blog Entries

Subscribe to Blog
Subscribe by RSS

Blog Archives
March 2023 (1)
2022 (1)
2021 (4)
2020 (2)
2019 (10)
2017 (13)
2016 (11)
2015 (18)
2014 (7)
2013 (4)
2012 (6)
2011 (11)
2010 (24)
2009 (25)
2008 (41)
2007 (26)
2006 (13)

The Power of Hidden Friends in C++

Tuesday, 25 June 2019

"Friendship" in C++ is commonly thought of as a means of allowing non-member functions and other classes to access the private data of a class. This might be done to allow symmetric conversions on non-member comparison operators, or allow a factory class exclusive access to the constructor of a class, or any number of things.

However, this is not the only use of friendship in C++, as there is an additional property to declaring a function or function template a friend: the friend function is now available to be found via Argument-Dependent Lookup (ADL). This is what makes operator overloading work with classes in different namespaces.

Argument Dependent Lookup at Work

Consider the following code snippet:

```
namespace A{
  class X{
  public:
    X(int i):data(i){}
  private:
    int data;
    friend bool operator==(X const& lhs,X const& rhs){
      return lhs.data==rhs.data;
    }
  };
}
int main(){
  A::X a(42),b(43);
  if(a==b) do_stuff();
}
```

This code snippet works as you might expect: the compiler looks for an implementation of `operator==` that works for `A::X` objects, and there isn't one in the global namespace, so it also looks in the namespace where `X` came from (`A`), and finds the operator defined as a friend of class `X`. Everything is fine. This is ADL at work: the *argument* to the operator is an `A::X` object, so the namespace that it comes from (`A`) is searched as well as the namespace where the usage is.

Note, however, that the comparison operator is *not* declared anywhere other than the `friend` declaration. This means that it is *only* considered for name lookup when one of the arguments is an `X` object (and thus is "hidden" from normal name lookup). To demonstrate this, let's define an additional class in namespace `A`, which is convertible to 'X':

```
namespace A{
  class Y{
  public:
    operator X() const{
      return X(data);
    }
    Y(int i):data(i){}
  private:
    int data;
  };
}
A::Y y(99);
A::X converted=y; // OK
```

Our `Y` class has a conversion operator defined, so we can convert it to an `X` object at will, and it is also in namespace `A`. You might think that we can compare `Y` objects, because our comparison operator takes an `X`, and `Y` is convertible to `X`. If you did, you'd be wrong: the comparison operator is only visible to name lookup if one of the arguments is an `X` object.

```
int main(){
  A::Y a(1),b(2);
  if(a==b) // ERROR: no available comparison operator
```

OPERATORS AS HIDDEN FRIENDS OF COMPLEX TYPE

`std::complex`

- `std::complex` doesn’t use hidden `friends`
  - `std::complex` originally part of C++98
  - Breaking change

- The C++ Standard Library is using hidden `friends` for new libraries

OPERATORS AS HIDDEN FRIENDS OF COMPLEX TYPE

Hidden `friend`

- First declaration of a function (not just `operators`)
- Declaration must contain the definition (implementation)

```
class X {
    // hidden friend
    friend bool operator==(X const&, X const&) { /* ... */ }
};
// Not hidden friend
bool operator!=(X const&, X const&);
```

OPERATORS AS HIDDEN FRIENDS OF COMPLEX TYPE

Avoid accidental implicit conversions

```
class X {
    // hidden friend
    friend bool operator==(X const&, X const&) { return true; }
};
// Not hidden friend
inline bool operator!=(X const& lhs, X const& rhs) { return !(lhs == rhs); }

struct Y {
    operator X() const { return X(); }  // implicit conversion to X
};

X x;
Y y;
assert(  x == x );
assert(!(x != x));
assert(  x == y );
assert(!(x != y));
assert(  y == x );
assert(!(y != x));
assert(  y == y );
assert(!(y != y));
```

OPERATORS AS HIDDEN FRIENDS OF COMPLEX TYPE

Avoid accidental implicit conversions

```
class X {
    // hidden friend
    friend bool operator==(X const&, X const&) { return true; }
};
// Not hidden friend
inline bool operator!=(X const& lhs, X const& rhs) { return !(lhs == rhs); }

struct Y {
    operator X() const { return X(); }  // implicit conversion to X
};

X x;
Y y;
assert(  x == x );
assert(!(x != x));
assert(  x == y );
assert(!(x != y));
assert(  y == x );
assert(!(y != x));
assert(  y == y );  // error: no match for 'operator==' (operand types are 'Y' and 'Y')
assert(!(y != y));
```

(The line `assert(  y == y );` is struck through on the slide.)

OPERATORS AS HIDDEN FRIENDS OF COMPLEX TYPE

Argument Dependent Lookup (ADL) not triggered

```
class X {
    // hidden friend
    friend bool operator==(X const&, X const&) { return true; }
};
// Not hidden friend
inline bool operator!=(X const& lhs, X const& rhs) { return !(lhs == rhs); }

struct Y {
    operator X() const { return X(); }  // implicit conversion to X
};

X x;
Y y;
  operator==(x, x);
::operator==(x, x);
  operator!=(x, x);
::operator!=(x, x);
```

OPERATORS AS HIDDEN FRIENDS OF COMPLEX TYPE

Argument Dependent Lookup (ADL) not triggered

```
class X {
    // hidden friend
    friend bool operator==(X const&, X const&) { return true; }
};
// Not hidden friend
inline bool operator!=(X const& lhs, X const& rhs) { return !(lhs == rhs); }

struct Y {
    operator X() const { return X(); }  // implicit conversion to X
};

X x;
Y y;
  operator==(x, x);
::operator==(x, x); // error: no member named 'operator==' in the global namespace
  operator!=(x, x);
::operator!=(x, x);
```

IMPLICIT CONVERSIONS AND LIMITATIONS

IMPLICIT CONVERSIONS AND LIMITATIONS

- std::complex<T> is implicitly convertible to std::complex<U>

- sycl::complex<T> implicitly convertible with sycl::complex<U>
- sycl::complex<T> implicitly convertible with std::complex<U>

IMPLICIT CONVERSIONS AND LIMITATIONS

- std::complex<T> is ~~implicitly~~ convertible to std::complex<U>
  - explicitly when T is a narrowing conversion to U (loses information)
    - e.g., double to float
  - Implicitly otherwise

- Do the same from sycl::complex<T> to sycl::complex<U>
- Do the same from sycl::complex<T> to std::complex<U>
- Do the same from std::complex<T> to sycl::complex<U>

IMPLICIT CONVERSIONS AND LIMITATIONS

```
namespace sycl {
template<typename T>
struct complex {
    constexpr complex(complex const&) = default;

    template<typename U>
    explicit(/* see below */) constexpr complex(complex<U> const&);

    template<typename U>
    explicit(/* see below */) constexpr complex(std::complex<U> const&);

    template<typename U>
    explicit(/* see below */) operator std::complex<U>() const;

    //...
};
```

IMPLICIT CONVERSIONS AND LIMITATIONS

Assignment

```
sycl::complex<float> cf;
std::complex<double> cd;

cd = cf; // cf implicitly converted to sycl::complex<double>
cf = cd; // error
```

IMPLICIT CONVERSIONS AND LIMITATIONS

- Conversions change types
  - (Small) run-time cost, usually optimized away
- Converting `const std::complex<T>&` and `const sycl::complex<U>&`
  - Temporary can bind to a const reference
- No conversions between `std::complex<T>*` and `sycl::complex<U>*`
  - They really are different types

TYPE PUNNING

TYPE PUNNING SYCL::COMPLEX TO STD::COMPLEX

```
template<typename T>
void bar(std::complex<T>& stc) { /* ... */ }

template<typename T>
void baz(sycl::complex<T> syc) {
    bar(reinterpret_cast<std::complex<T>&>(syc));
}
```

- “This works because both types have the same in-memory layout”

TYPE PUNNING SYCL::COMPLEX TO STD::COMPLEX

```
template<typename T>
void bar(std::complex<T>& stc) { /* ... */ }

template<typename T>
void baz(sycl::complex<T> syc) {
    bar(reinterpret_cast<std::complex<T>&>(syc));
}
```

- ~~“This works because both types have the same in-memory layout”~~
  - ***<u>Undefined Behavior!</u>***

TYPE PUNNING & STRICT ALIASING

C++23 [basic.lval]p11

- Type punning via `reinterpret_cast` or a `union` is ***undefined behavior*** if a type is not *similar* to:
  - The dynamic type of the object
  - A type that is the signed or unsigned type corresponding to the object's dynamic type
  - A `char`, `unsigned char` or `std::byte` type
- Compiler assumes objects of dissimilar types are not aliased (for optimizations)
  - If non-empty, do not occupy the same memory
- Special dispensation for punning `std::complex<T>` and `T[2]` (`_Complex` harmony)
- What is the Strict Aliasing Rule and Why do we care? — *Shafik Yaghmour*
  - https://gist.github.com/shafik

Not Secure — eel.is/c++draft/complex.numbers.general#4

28 Numerics library [numerics]

28.4 Complex numbers [complex.numbers]

28.4.1 General [complex.numbers.general]

1 The header `<complex>` defines a class template, and numerous functions for representing and manipulating complex numbers.

2 The effect of instantiating the template `complex` for any type that is not a cv-unqualified floating-point type ([basic.-fundamental]) is unspecified. Specializations of `complex` for cv-unqualified floating-point types are trivially-copyable literal types ([basic.types.general]).

3 If the result of a function is not mathematically defined or not in the range of representable values for its type, the behavior is undefined.

4 If `z` is an lvalue of type *cv* `complex<T>` then:

(4.1) — the expression `reinterpret_cast<cv T(&)[2]>(z)` is well-formed,

(4.2) — `reinterpret_cast<cv T(&)[2]>(z)[0]` designates the real part of `z`, and

(4.3) — `reinterpret_cast<cv T(&)[2]>(z)[1]` designates the imaginary part of `z`.

Moreover, if `a` is an expression of type *cv* `complex<T>*` and the expression `a[i]` is well-defined for an integer expression `i`, then:

(4.4) — `reinterpret_cast<cv T*>(a)[2*i]` designates the real part of `a[i]`, and

(4.5) — `reinterpret_cast<cv T*>(a)[2*i + 1]` designates the imaginary part of `a[i]`.

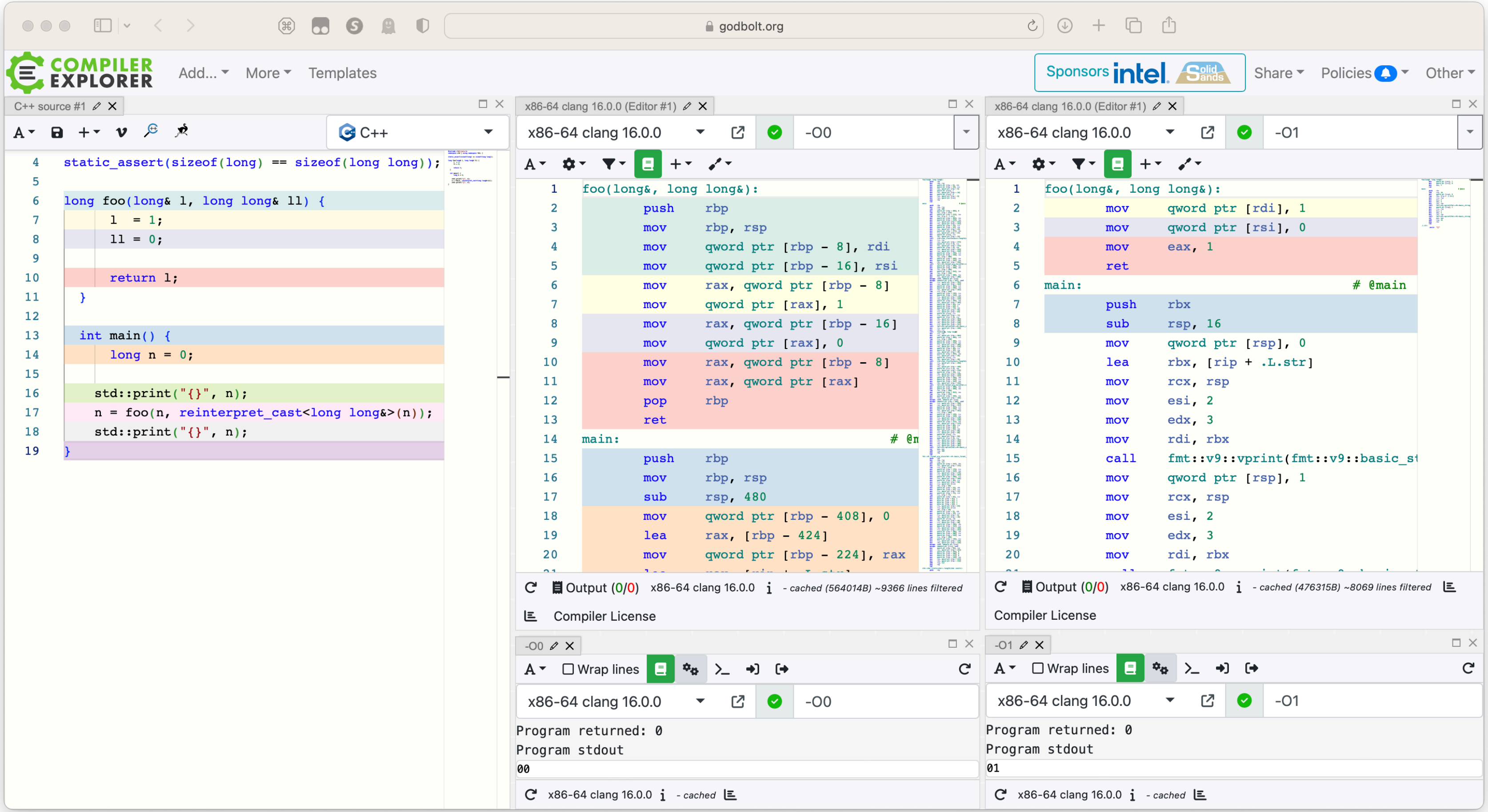
https://eel.is/c++draft/complex.numbers.general#4

TYPE PUNNING & STRICT ALIASING

reinterpret_cast between long and long long

```
static_assert(sizeof(long) == sizeof(long long));

long foo(long& l, long long& ll) {
      l  = 1;
      ll = 0;

      return l;
  }

  int main() {
      long n = 0;

    std::print("{}", n);
    n = foo(n, reinterpret_cast<long long&>(n));
    std::print("{}", n);
}
```

- What is the expected output?



https://godbolt.org/z/Gnoq7hPf7

BIT_CAST — SAFE TYPE PUNNING

C++23

```
template<typename To, typename From>
constexpr To bit_cast(const From& from) noexcept;
```

- `sizeof(To) == sizeof(From)`
- `is_trivially_copyable_v<To>`
- `is_trivially_copyable_v<From>`

- Note: One can `bit_cast` between `sycl::complex*` & `std::complex*`
  - But dereferencing the type punned pointer is still ***undefined behavior!***

MEMCPY — SAFE TYPE PUNNING (IF CAREFUL)

Not Secure — eel.is/c++draft/basic.types.general#3

3 For two distinct objects `obj1` and `obj2` of trivially copyable type `T`, where neither `obj1` nor `obj2` is a potentially-overlapping subobject, if the underlying bytes ([intro.memory]) making up `obj1` are copied into `obj2`,[30] `obj2` shall subsequently hold the same value as `obj1`.

[*Example 2*:

```
T* t1p;
T* t2p;
    // provided that t2p points to an initialized object ...
std::memcpy(t1p, t2p, sizeof(T));
    // at this point, every subobject of trivially copyable type in *t1p contains
    // the same value as the corresponding subobject in *t2p
```

— *end example*]

http://eel.is/c++draft/basic.types.general#3

MEMCPY — SAFE TYPE PUNNING (IF CAREFUL)

C++20 (or earlier with appropriate changes)

```
template <typename To, typename From>
requires(sizeof(To) == sizeof(From) &&
         is_trivially_copyable_v<To> &&
         is_trivially_copyable_v<From>)
constexpr To my_bit_cast(const From& from) noexcept {
    To to;
    memcpy(addressof(to), addressof(from), sizeof from);
    return to;
}
```

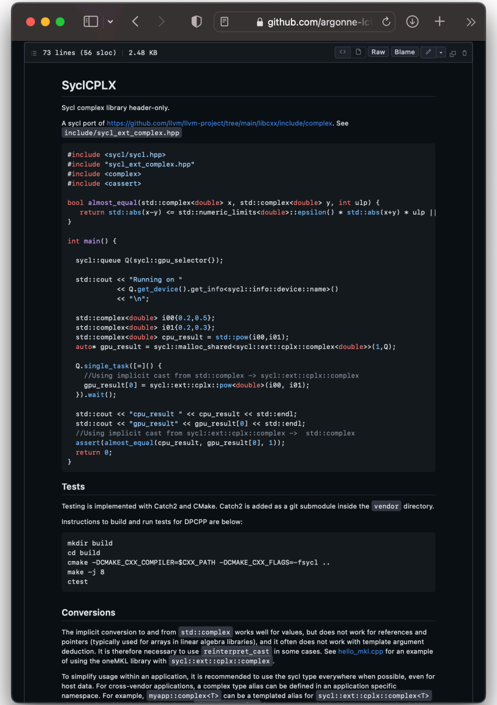
SyclCPLX

AIDEN BELTON-SCHURE & JEFFERSON LE QUELLEC

SyclCPLX

SYCL implementation of complex numbers and associated math functions

- https://github.com/argonne-lcf/SyclCPLX
  - Open-source
  - IEEE compliant
  - Header-only



github.com/argonne-lcf

73 lines (56 sloc) | 2.48 KB

SyclCPLX

Sycl complex library header-only.

A sycl port of https://github.com/llvm/llvm-project/tree/main/libcxx/include/complex. See `include/sycl_ext_complex.hpp`

```
#include <sycl/sycl.hpp>
#include "sycl_ext_complex.hpp"
#include <complex>
#include <cassert>

bool almost_equal(std::complex<double> x, std::complex<double> y, int ulp) {
   return std::abs(x-y) <= std::numeric_limits<double>::epsilon() * std::abs(x+y) * ulp ||
}

int main() {

  sycl::queue Q(sycl::gpu_selector{});

  std::cout << "Running on "
            << Q.get_device().get_info<sycl::info::device::name>()
            << "\n";

  std::complex<double> i00{0.2,0.5};
  std::complex<double> i01{0.2,0.3};
  std::complex<double> cpu_result = std::pow(i00,i01);
  auto* gpu_result = sycl::malloc_shared<sycl::ext::cplx::complex<double>>(1,Q);

  Q.single_task([=]() {
    //Using implicit cast from std::complex -> sycl::ext::cplx::complex
    gpu_result[0] = sycl::ext::cplx::pow<double>(i00, i01);
  }).wait();

  std::cout << "cpu_result " << cpu_result << std::endl;
  std::cout << "gpu_result" << gpu_result[0] << std::endl;
  //Using implicit cast from sycl::ext::cplx::complex ->  std::complex
  assert(almost_equal(cpu_result, gpu_result[0], 1));
  return 0;
}
```

Tests

Testing is implemented with Catch2 and CMake. Catch2 is added as a git submodule inside the `vendor` directory.

Instructions to build and run tests for DPCPP are below:

```
mkdir build
cd build
cmake -DCMAKE_CXX_COMPILER=$CXX_PATH -DCMAKE_CXX_FLAGS=-fsycl ..
make -j 8
ctest
```

Conversions

The implicit conversion to and from `std::complex` works well for values, but does not work for references and pointers (typically used for arrays in linear algebra libraries), and it often does not work with template argument deduction. It is therefore necessary to use `reinterpret_cast` in some cases. See hello_mkl.cpp for an example of using the oneMKL library with `sycl::ext::cplx::complex`.

To simplify usage within an application, it is recommended to use the sycl type everywhere when possible, even for host data. For cross-vendor applications, a complex type alias can be defined in an application specific namespace. For example, `myapp::complex<T>` can be a templated alias for `sycl::ext::cplx::complex<T>`

VALIDATION OF IMPLEMENTATION

AIDEN BELTON-SCHURE, JEFFERSON LE QUELLEC & THOMAS APPLENCOURT

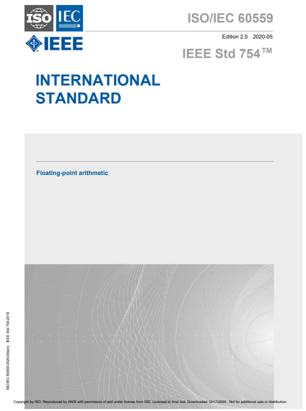
VALIDATION

- Based on LLVM (libc++) `std::complex`
  - Robust on CPUs
- SyclCPLX uses SYCL built-ins
  - Run on both CPUs & GPUs
- Test suite
  - Valid mathematical outputs
  - Valid error code handling
  - Drop-in replacement for `std::complex`
  - Supports `double`, `float` and `sycl::half`
  - API matches proposed API for SYCL Standard

VALIDATION

Complex math standard

- Test suite compares results to std::complex
- Have to understand complex math
  - Have to understand floating point math
- ISO/IEC 60559 / IEEE Std 754™
  - https://www.iso.org/standard/80985.html
  - “Operations not specified by this standard, such as *complex arithmetic*…”
    - Signaling of (floating point) exceptions is language-defined
      - C++ punts that to C


ISO/IEC 60559
Edition 2.0 2020-05
IEEE Std 754™
INTERNATIONAL STANDARD
Floating-point arithmetic

VALIDATION

Inverse trigonometric functions

- Trig functions on complex numbers can be multi-valued
  - Mathematically, a “branch cut” is picked by convention
    - $\sqrt{1} = \pm 1$
  - C++ does not specify which cut is used
  - Specifically, the CPU and GPU can pick different values
  - We check that $f^{-1}(f(x)) == x$
    - Floating point is not exact
    - Error handling (NaN, Inf, etc.) might be different as well
    - Metamorphic relation / metamorphic testing

VALIDATION

Half emulation

- C++ is not required to support half precision
  - Note: C++23 introduces optional `float16_t`
- Emulation
  - All calculations are done in (32-bit) `float`
  - Initial input and final output are truncated to 16-bits
  - Note: underlying operations higher precision than true 16-bit floating point

VALIDATION

SYCL Implementations

- Need to run across multiple SYCL implementations
  - Initial development under Intel oneAPI
  - Tested under hipSYCL
    - Needed workaround for a different alias to `namespace sycl`
    - SYCL-2020 `isinf`, `isfinite`, `ldexp`, `signbit` not yet supported
      - Need an implementation for each Open SYCL backend

MICRO BENCHMARKING WITH GOOGLE BENCHMARK ON ICPX FROM INTEL ONEAPI

BRYCE ALLEN & THOMAS APPLENCOURT

MICRO BENCHMARKING WITH GOOGLE BENCHMARK ON ICPX FROM INTEL ONEAPI

Setup

- Intel oneAPI already supports `std::complex` in device code
- Benchmarks were added to SyclCPLX to compare with oneAPI `std::complex`
- Large device arrays with random data and ran in all threads:

```
Q.parallel_for(sycl::range<1>(N), [=](sycl::id<1> i) {
      c[i] = op(a[i], b[I]);
}).wait();
```

- *op* is one of addition, subtraction, multiplication, division
- Tested on Intel hardware at Argonne ALCF with $N=16\times2^{20}$

MICRO BENCHMARKING WITH GOOGLE BENCHMARK ON ICPX FROM INTEL ONEAPI

Results

- SYCLCPLX & oneAPI similar results for addition, subtraction, multiplication
- oneAPI 50% faster for single precision division
  - Uses fast-math
    - Allows reordering (associativity)
    - Assumes all math is finite
  - Compiler didn't optimize away SYCLCPLX non-finite checks

MICRO BENCHMARKING WITH GOOGLE BENCHMARK ON ICPX FROM INTEL ONEAPI

Results

```
template <typename T>
_SYCL_EXT_CPLX_INLINE_VISIBILITY constexpr bool isnan(const T a) {
#ifdef _SYCL_EXT_CPLX_FAST_MATH
    return false;
#else
    return sycl::isnan(a);
#endif
}
```

- SYCLCPLX still 20% slower on division by a complex number
  - oneAPI doesn't scale the values
    - Technique for reducing floating point error
    - Add non-scaling to SYCLCPLX

MICRO BENCHMARKING WITH GOOGLE BENCHMARK ON ICPX FROM INTEL ONEAPI

Results

```
template <class _Tp>
friend complex<_Tp> operator/(const complex<_Tp>& __z,
                              const complex<_Tp>& __w) {
#ifdef _SYCL_EXT_CPLX_FAST_MATH
    _Tp __a = __z.__re_;
    _Tp __b = __z.__im_;
    _Tp __c = __w.__re_;
    _Tp __d = __w.__im_;
    _Tp __r = __a * __c + __b * __d;
    _Tp __n = __b * __b + __d * __d;
    _Tp __x = __r / __n;
    _Tp __y = (__b * __c - __a * __d) / __n;
    return complex<_Tp>(__x, __y);
#else
    // full implementation
#endif
}
```

APPLICATION INTEGRATION

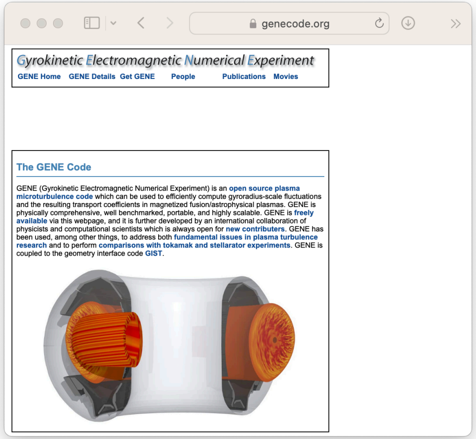
GENE AND GTENSOR

BRYCE ALLEN

GENE AND GTENSOR

GENE

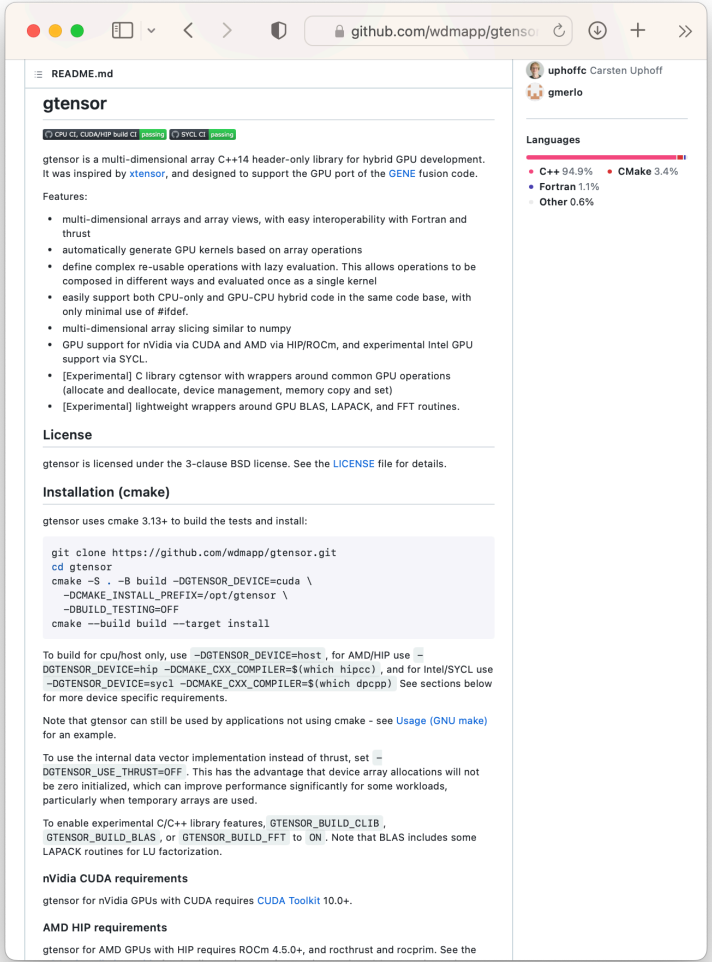
- GENE (Gyrokinetic Electromagnetic Numerical Experiment)
  - https://genecode.org
  - Initially written in modern Fortran
  - Ported to GPUs via gtensor
  - Advanced linear algebra and other math libraries
    - LU solvers, sparse solvers, matrix operations, FFT



GENE AND GTENSOR

gtensor

- gtensor
  - https://github.com/wdmapp/gtensor
  - C++ performance portability library
  - `gt::complex` alias
    - oneAPI: `std::complex`
    - CUDA & ROCm: `thrust::complex`



github.com/wdmapp/gtenso

README.md

uphoffc Carsten Uphoff
gmerlo

Languages

C++ 94.9% CMake 3.4%
Fortran 1.1%
Other 0.6%

gtensor

CPU CI, CUDA/HIP build CI passing | SYCL CI passing

gtensor is a multi-dimensional array C++14 header-only library for hybrid GPU development. It was inspired by xtensor, and designed to support the GPU port of the GENE fusion code.

Features:

- multi-dimensional arrays and array views, with easy interoperability with Fortran and thrust
- automatically generate GPU kernels based on array operations
- define complex re-usable operations with lazy evaluation. This allows operations to be composed in different ways and evaluated once as a single kernel
- easily support both CPU-only and GPU-CPU hybrid code in the same code base, with only minimal use of #ifdef.
- multi-dimensional array slicing similar to numpy
- GPU support for nVidia via CUDA and AMD via HIP/ROCm, and experimental Intel GPU support via SYCL.
- [Experimental] C library cgtensor with wrappers around common GPU operations (allocate and deallocate, device management, memory copy and set)
- [Experimental] lightweight wrappers around GPU BLAS, LAPACK, and FFT routines.

License

gtensor is licensed under the 3-clause BSD license. See the LICENSE file for details.

Installation (cmake)

gtensor uses cmake 3.13+ to build the tests and install:

```
git clone https://github.com/wdmapp/gtensor.git
cd gtensor
cmake -S . -B build -DGTENSOR_DEVICE=cuda \
  -DCMAKE_INSTALL_PREFIX=/opt/gtensor \
  -DBUILD_TESTING=OFF
cmake --build build --target install
```

To build for cpu/host only, use `-DGTENSOR_DEVICE=host`, for AMD/HIP use `-DGTENSOR_DEVICE=hip -DCMAKE_CXX_COMPILER=$(which hipcc)`, and for Intel/SYCL use `-DGTENSOR_DEVICE=sycl -DCMAKE_CXX_COMPILER=$(which dpcpp)` See sections below for more device specific requirements.

Note that gtensor can still be used by applications not using cmake - see Usage (GNU make) for an example.

To use the internal data vector implementation instead of thrust, set `-DGTENSOR_USE_THRUST=OFF`. This has the advantage that device array allocations will not be zero initialized, which can improve performance significantly for some workloads, particularly when temporary arrays are used.

To enable experimental C/C++ library features, `GTENSOR_BUILD_CLIB`, `GTENSOR_BUILD_BLAS`, or `GTENSOR_BUILD_FFT` to `ON`. Note that BLAS includes some LAPACK routines for LU factorization.

nVidia CUDA requirements

gtensor for nVidia GPUs with CUDA requires CUDA Toolkit 10.0+.

AMD HIP requirements

gtensor for AMD GPUs with HIP requires ROCm 4.5.0+, and rocthrust and rocprim. See the

GENE AND GTENSOR

GPU support

- gt-blas, gt-fft, gt-solver
- CUDA & ROCm
    - CUDA: cuBLAS, cuSPARSE, cuFFT
    - ROCm: rocBLAS, rocSolver, rocSPARSE, rocFFT

    - C `_Complex` for the above
    - Required `reinterpret_cast` from `thrust::complex`
        - ***SIGH***

GENE AND GTENSOR

- Alignment issues
  - Fortran aligns complex numbers on component types (each float aligned)
  - GPUs typically align on the whole type (complex type aligned)
  - Solution:
    - Copy one to the other for single values
    - Assert alignment for arrays passed by pointer

GENE AND GTENSOR

Steps

1. Add the sycl_ext_complex.hpp header to the gtensor repository
2. Change the `gt::complex` type for SYCL backend from `std::complex` to `sycl::complex`
3. Run tests and benchmarks and insure CI is passing
4. Update the SYCL backend for gt-* library wrappers to properly handle arguments of `sycl::complex`
5. Update GENE to use the new version of gtensor with the above changes
6. Run GENE CI and unit tests

GENE AND GTENSOR

- `namespace` issues
  - Ambiguity with `gt::backend::sycl` and `sycl`
  - Added support to SYCLCPLX for custom namespace via `_SYCL_CPLX_NAMESPACE`
- gt-* library wrappers involve pointers to complex numbers

GENE AND GTENSOR

- `namespace` issues
  - Ambiguity with `gt::backend::sycl` and `sycl`
  - Added support to SYCLCPLX for custom namespace via `_SYCL_CPLX_NAMESPACE`
- gt-* library wrappers involve pointers to complex numbers
  - `reinterpret_cast`

GENE AND GTENSOR

- `namespace` issues
  - Ambiguity with `gt::backend::sycl` and `sycl`
  - Added support to SYCLCPLX for custom namespace via `_SYCL_CPLX_NAMESPACE`
- gt-* library wrappers involve pointers to complex numbers
  - ~~`reinterpret_cast`~~`^H^H^H^H^H^H^H^H^H^H^H^H^H^H^H^H^H`
  - *I won't bore you with the details on how it was solved*
    - *Coincidentally, I have some work to do when I get back*

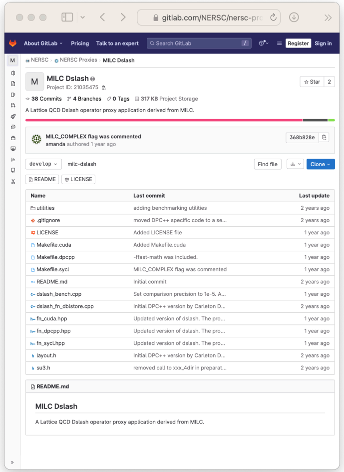
- Results
  - All tests passed! Performance and build times unaffected!

MILC-DSLASH

AMANDA DUFEK

MILC-DSLASH

- Benchmark simulates 4-dimension SU(3) lattice-gauge theory
  - https://gitlab.com/NERSC/nersc-proxies/milc-dslash
  - Multiple matrix vector products of double precision complex numbers
- Sequential version
  - 5 nested `for` loop structures
- Parallel version
  - Loop level parallelism



MILC-DSLASH

- Baseline implementation
  - `struct { double r; double i; };` to represent complex numbers
- Second implementation
  - SYCLCPLX
- Third implementation
  - SYCLCPLX disabling invalid entry (NaN, infinity, etc.) checking

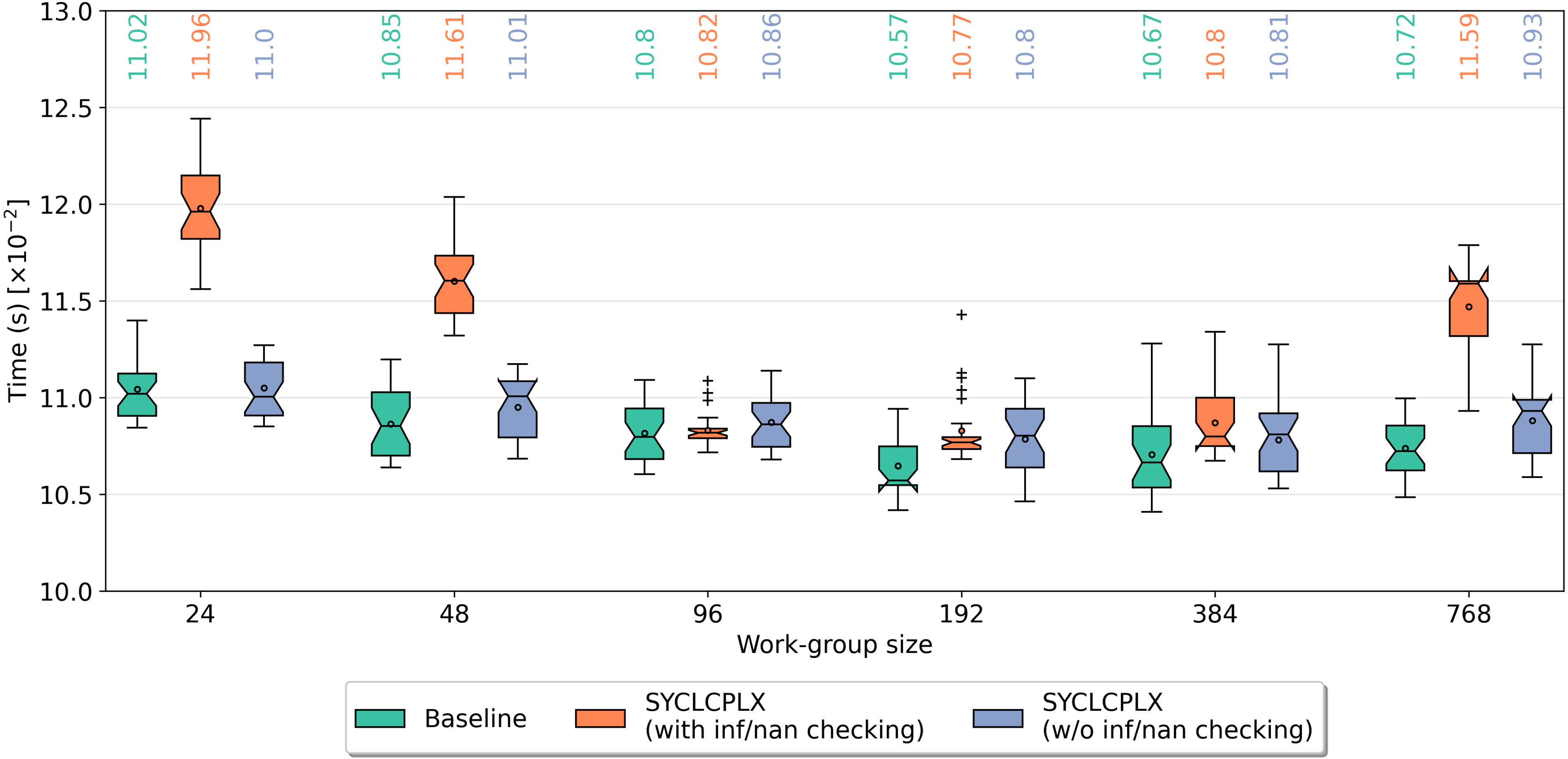
MILC-DSLASH

- Single NVIDIA A100 GPU

- Various work-group sizes
  - 24, 48, 96, 192, 384, 768

- For each implementation
  - 30 runs
    - 1 warmup iteration
    - 100 kernel iterations

- Mean/median kernel runtimes
  - `std::chrono::system_clock`



11.02
11.96
11.0
10.85
11.61
11.01
10.8
10.82
10.86
10.57
10.77
10.8
10.67
10.8
10.81
10.72
11.59
10.93
13.0
12.5
12.0
11.5
11.0
10.5
10.0
Time (s) [×10⁻²]
24
48
96
192
384
768
Work-group size
Baseline
SYCLCPLX (with inf/nan checking)
SYCLCPLX (w/o inf/nan checking)

MILC-DSLASH

Results

- Checking for invalid elements negatively affects performance (sometimes)
  - 5-8% for work-group sizes of 24, 48, 768
  - Not detectible for work-group sizes of 96, 192, 384
- Baseline < 2% better than SYCLCPLX
  - May not be reproducible or generalizable to other systems
- Performance varies little with work-group size
  - Peak performance with a work-group size of 192

ACKNOWLEDGEMENTS

SPECIAL THANKS

- Gordon Brown
  - Managing this project on the Codeplay side
- Brandon Cook
  - Letting us work on this project
- The SYCLcon / IWOCL Program Committee
  - Their comments helped shape this presentation
- The Khronos SYCL Committee
  - Invaluable discussions & comments while developing this proposal

- This research was supported by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy (DOE) Office of Science and the National Nuclear Security Administration
- This research used resources of the Argonne Leadership Computing Facility, which is a DOE Office of Science User Facility supported under Contract DE-AC02-06CH11357
- This research also used resources of the National Energy Research Scientific Computing Center (NERSC), a DOE Office of Science User Facility located at Lawrence Berkeley National Laboratory, operated under Contract DE-AC02-05CH11231

https://github.com/argonne-lcf/SyclCPLX