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

April 18–20, 2023 | University of Cambridge, UK
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 notstandardize 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++\textsuperscript{23} \texttt{STD::COMPLEX}

Why not standardize C++\textsuperscript{23} \texttt{std::complex} for SYCL?

- \texttt{std::complex<T>} requirements on T relaxed
  - Trivially copyable type
  - Literal type
    - Useable as a \texttt{constexpr} variable
  - Numeric type
    - Default constructible, copyable, destructible
    - No operations throw exceptions
  - Still unspecified if T is not a floating point type
- \texttt{std::complex<T>} itself is a trivially copyable, literal, numeric type
C++23 std::complex

Why not standardize C++23 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

- Initial Issue: 7-Feb-2022
- Initial Commit SYCL Complex: 14-Apr-2022
- Codeplay Involvement: 5-May-2022
- PR Stated to SYCL Spec: 28-Jun-2022
- Open To Intel LLVM: 20-Jan-2023

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](https://www.justsoftwaresolutions.co.uk/cplusplus/hidden-friends.html)
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)

```cpp
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));
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>`
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

```cpp
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

```cpp
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”
- **Undefined Behavior!**
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](https://gist.github.com/shafik)
28 Numerics library

28.4 Complex numbers

28.4.1 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:

- the expression `reinterpret_cast<cv T(&)[2]>(z)` is well-formed,
- `reinterpret_cast<cv T(&)[2]>(z)[0]` designates the real part of `z`, and
- `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:

- `reinterpret_cast<cv T*>(a)[2*i]` designates the real part of `a[i]`, and
- `reinterpret_cast<cv T*>(a)[2*i + 1]` designates the imaginary part of `a[i]`.

https://eel.is/c++draft/complex.numbers.general#4
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?
static_assert(sizeof(long) == sizeof(long long));

long foo(longs l1, long longs ll) {
  l1 = 1;
  ll = 0;
  return l1;
}

int main() {
  long n = 0;
  std::printf("%d", n);
  n = foo(n, reinterpret_cast<long long*>(n));
  std::printf("%d", n);
}

x86-64 clang 16.0.0 -O0 -std=c++11

Program returned: 0
Program stdout

x86-64 clang 16.0.0 -O1

Program returned: 0
Program stdout

https://godbolt.org/z/Gnoq7hPf7
BIT_CAST — SAFE TYPE PUNNING

C++23

```cpp
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)

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:

```cpp
T* t1p;
T* t2p;

// provided that `t2p` points to an initialized object ...
std::memcopy(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]
MEMCPY — SAFE TYPE PUNNING (IF CAREFUL)

C++20 (or earlier with appropriate changes)

```cpp
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

SYCL implementation of complex numbers and associated math functions

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

AIDEN BELTON-SCHURE, JEFFERSON LE QUELLEC & THOMAS APPLENCOURT
VALIDATION

- Based on LLVM (libc++) std::complex
  - Robust on CPUs
- SYCL CPLX 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](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
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:

```cpp
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

- $\text{SYCL CPLX}$ & 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 $\text{SYCL CPLX}$ 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
}

- SYCL CPLX 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 SYCL CPLX
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

GENE

- GENE (Gyrokinetic Electromagnetic Numerical Experiment)
  - [https://genecode.org](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

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

- 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`
MILC-DLASH

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