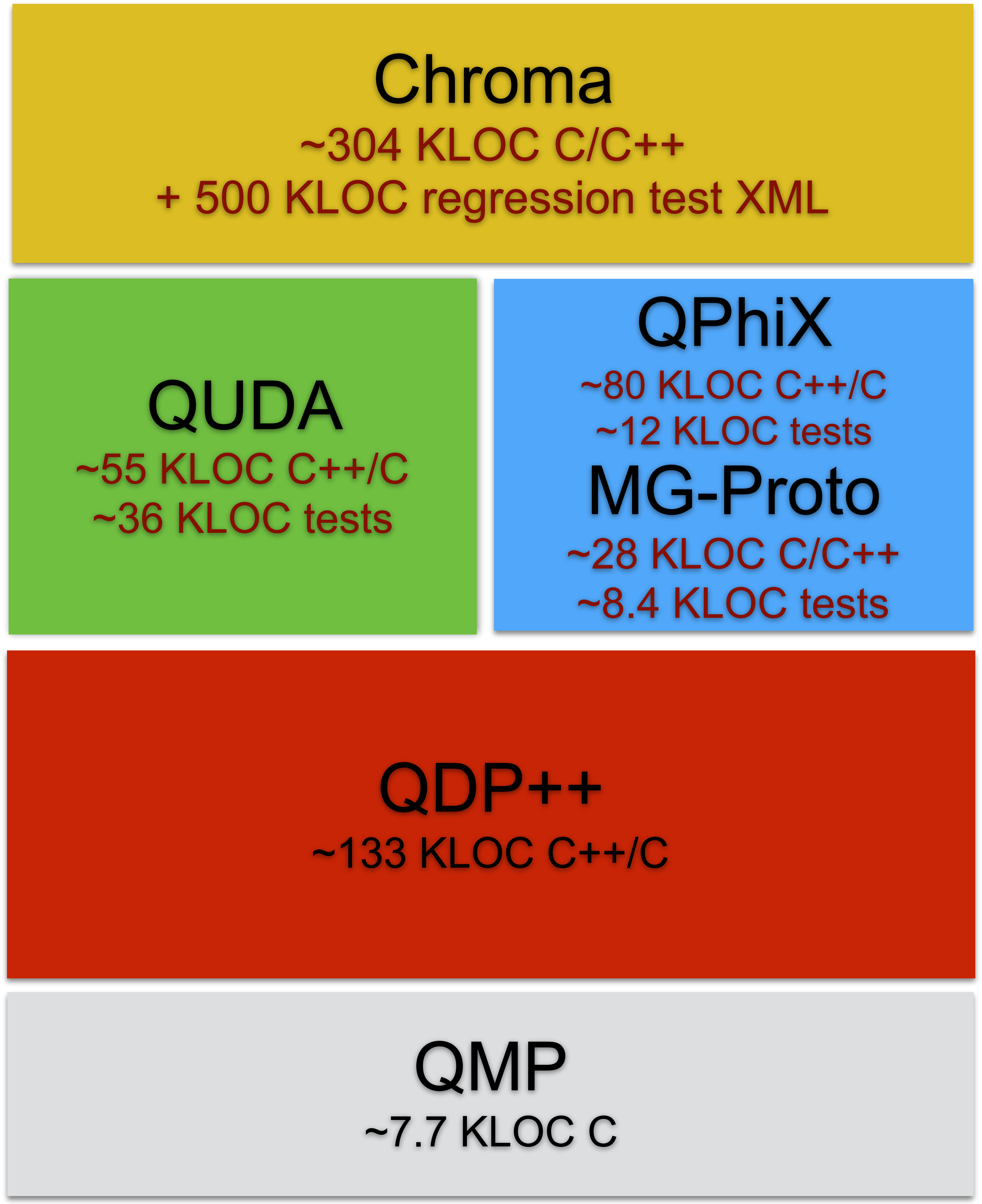
Testing: Strategies When Learning Programming Models and Using High-Performance Libraries

Bálint Joó - Jefferson Lab
IDEAS Best Practices Webinar
March 18, 2020

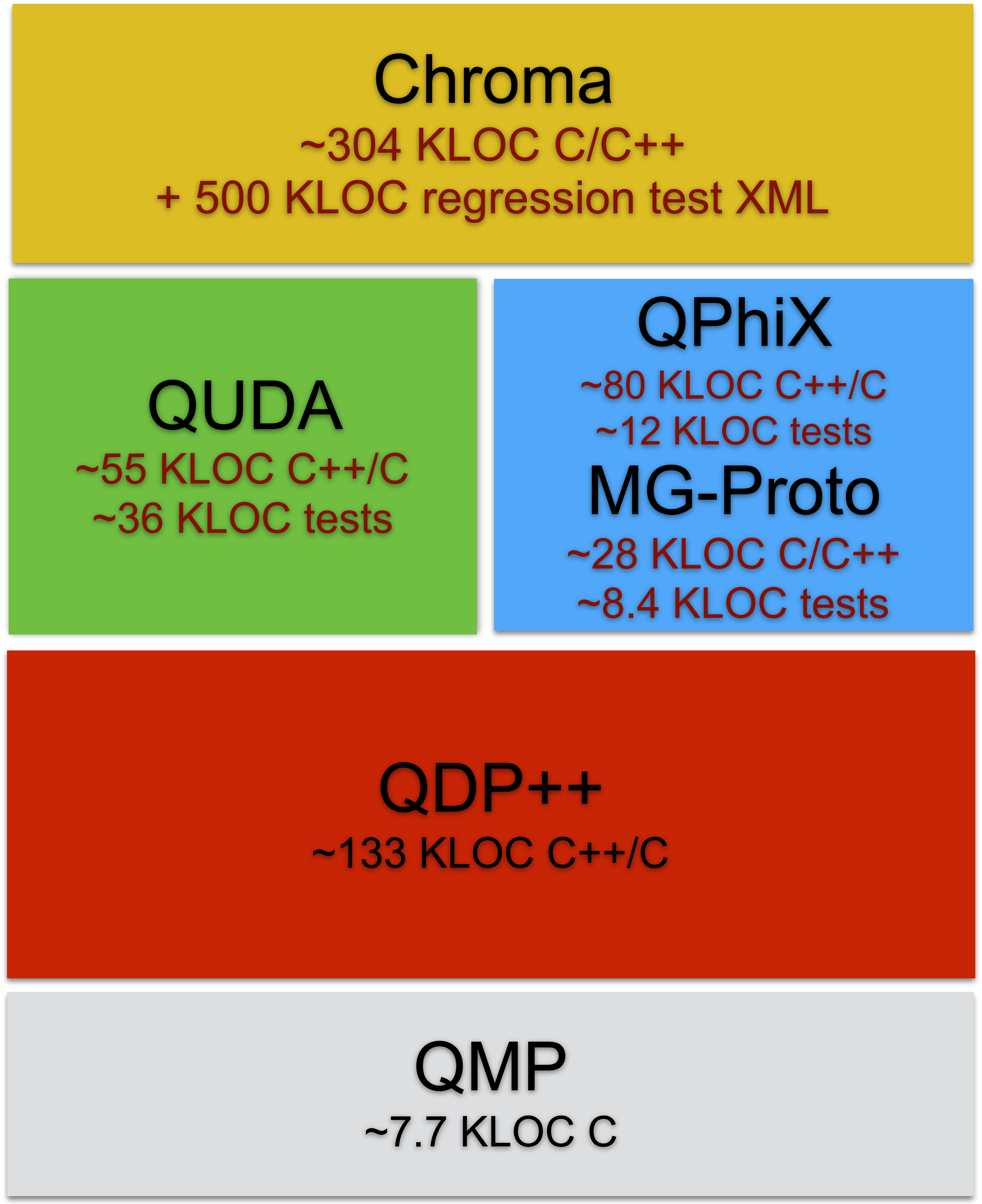
About the Chroma Lattice QCD Code

- I work with an application called Chroma
  - a lattice QCD code, used in Nuclear and High Energy Physics calculations
  - follows the USQCD Layered Software approach developed through iterations of the SciDAC program
  - The code is deployed on NVIDIA GPU based Systems (Summit, Sierra) as well as x86 based systems (Cori, Stampede-2, Frontera)
- The layers encapsulate different responsibilities
  - QMP wraps MPI
  - QDP++ is a data parallel DSL layer which provides QCD types and operations
  - Chroma contains the physics
  - QUDA (for NVIDIA GPUs) and QPhiX & MGProto (for x86 AVX512) are performance libraries with QCD Linear Solvers.



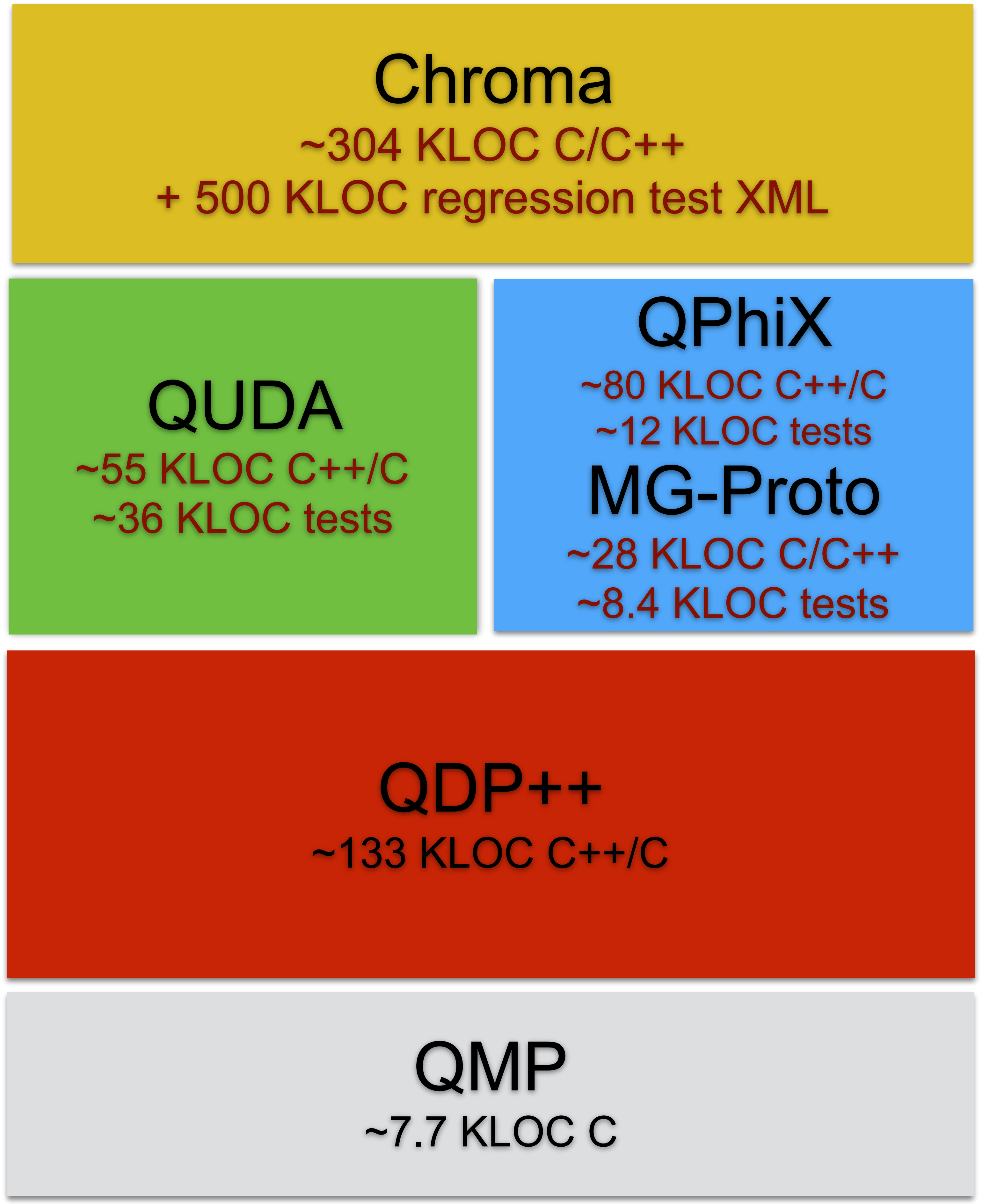
The Chroma stack and testing

- Chroma has over 100 regression tests
  - used to have nightly builds (now defunct?) on CPUs
- QUDA tested independently by the NVIDIA developers
- QPhiX had CI on Travis, but ran afoul of build time limits
  - currently failing - setup has decayed
- MG-Proto has tests, but no CI at this time
- QMP/QDP++ were effectively tested through Chroma regression tests
- BUT: Any testing is better than none! CI for the stack is still an aspiration
- Other LQCD codes with full CI currently: Grid (e.g. Travis CI)
- Will focus on C++ unit testing in this talk…
  - but Fortran users may consider pFUnit



Testing in the development process

- QUDA is a ‘3rd party’ library supplying solvers
  - Lead Developer: Kate Clark from NVIDIA
  - QUDA is maintained by NVIDIA and developed by the LQCD Community
- Chroma classes wrap QUDA calls
  - Test Integration: Compare QUDA output with ‘known less optimized’ output, for linear operators, solvers
- Add new features to QUDA
  - Develop feature in Chroma using less optimized but simpler QDP++
  - Add feature into QUDA
  - Test Integration: Is the QUDA library implementation doing the right thing? If yes, add QUDA internal test too so that the new feature can be verified independently of Chroma.
- When writing new code add tests
  - E.g. when developing Arnoldi process, check resulting vectors for orthonormality etc.
- Bear in mind: Agreement with reference only guarantees bug compatibility in principle



And now, things are about to change…

- Exascale and Pre-Exascale systems in the DOE Complex
  - Perlmutter at NERSC: pre-exascale system powered by NVIDIA GPUs and AMD CPUs
  - Aurora at ALCF: exascale system powered by Intel X^e GPUs and Xeon CPUs
  - Frontier at OLCF (& El Capitan at LLNL): exascale systems powered by AMD GPUs and AMD CPUs
- New programming models
  - Perlmutter: NVIDIA: Phew! Existing CUDA code will be fine
  - Aurora: Uh-oh! No CUDA! Preferred programming model is DPC++/SYCL.
  - Frontier: Uh-oh! No CUDA! Preferred programming model is HIP.
- … or we can use Kokkos with HIP and DPC++ back ends
  - See previous IDEAS talk on Kokkos here
- But how do I learn about these new models? How can I make informed decisions?
  - Develop a MiniApp!!!
  - Explore programming model features through tests!
- This talk is based on C++ based unit testing
  - for Fortran based testing see previous IDEAS talk on PFunit here

The Basics of Unit Tests

- Unit tests verify that a code satisfies some expected behaviour:
  - form an expectation
  - exercise it with code being tested
  - check that the expectation if fulfilled
- Check expectations with assertions
  - ASSERT_TRUE(boolean_result)
  - ASSERT_EQ(val1, val2)
  - ASSERT_LT(val1, val2)
  - …

```
// Test set() and operator() accessors of
// a SIMD Type: SIMDComplex<double,N>
//
TEST(TestVectype, TestLaneAccessorsD4)
{
  SIMDComplex<double,4> v4;

  // Use set() method to set elements
  for(int i=0; i < v4.len(); ++i) {
    v4.set(i, std::complex<double>(i,-i));
  }

  // Use operator() to retrieve the elements
  for(int i=0; i < v4.len(); ++i) {
     double re = v4(i).real();
     double im = v4(i).imag();

      // Assert within a DP epsilon the answer is
      // what one expects
     ASSERT_DOUBLE_EQ( re, static_cast<double>(i) );
     ASSERT_DOUBLE_EQ( im, static_cast<double>(-i) );

  }
}
```

The Basics of Unit Tests

- Unit tests verify that a code satisfies some expected behaviour:
  - form an expectation
  - exercise it with code being tested
  - check that the expectation if fulfilled
- Check expectations with assertions
  - ASSERT_TRUE(boolean_result)
  - ASSERT_EQ(val1, val2)
  - ASSERT_LT(val1, val2)
  - …

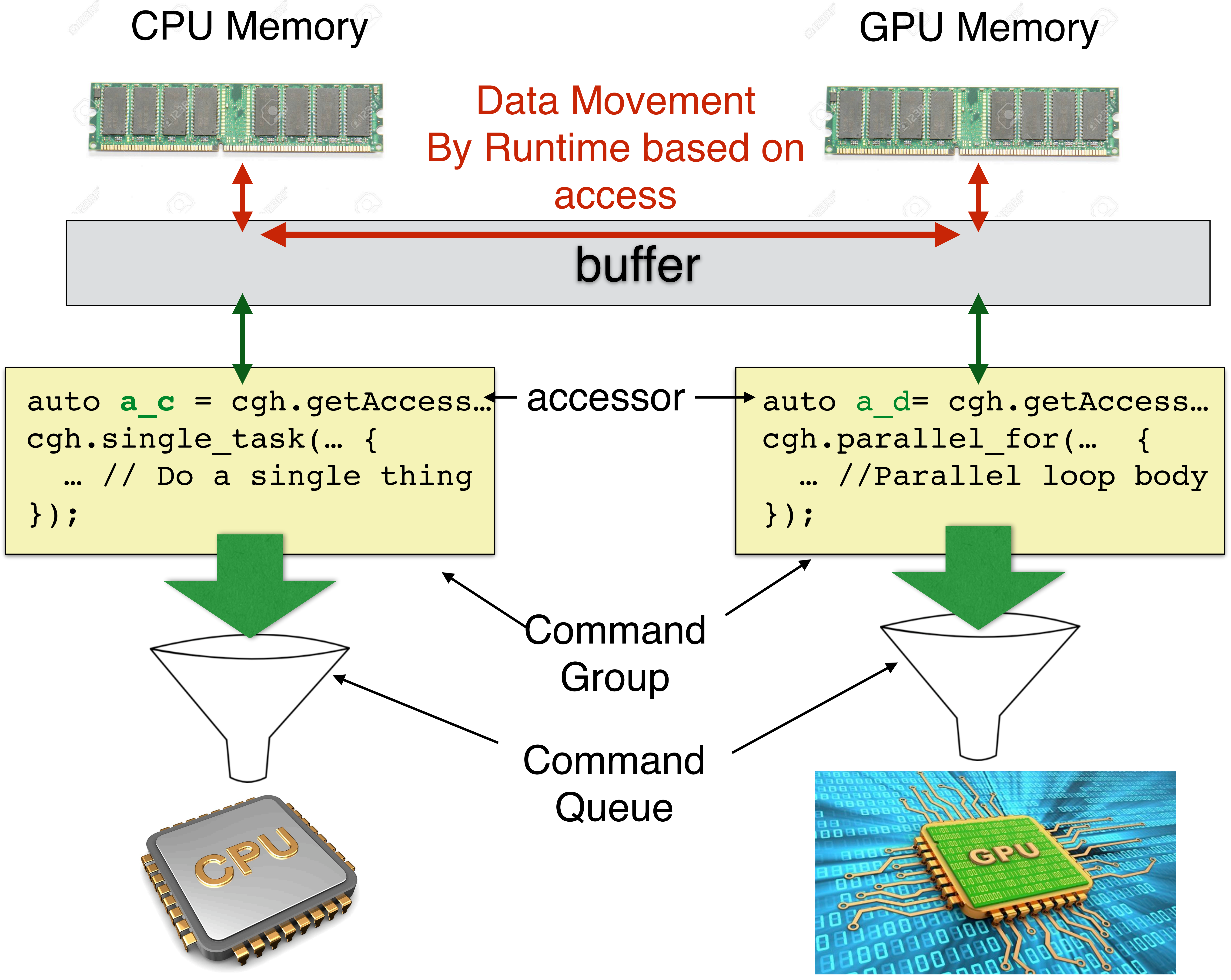
```
// Test set() and operator() accessors of
// a SIMD Type: SIMDComplex<double,N>
//
TEST(TestVectype, TestLaneAccessorsD4)
{
  SIMDComplex<double,4> v4;

  // Use set() method to set elements
  for(int i=0; i < v4.len(); ++i) {
    v4.set(i, std::complex<double>(i,-i));
  }

  // Use operator() to retrieve the elements
  for(int i=0; i < v4.len(); ++i) {
    double re = v4(i).real();
    double im = v4(i).imag();

    // Assert within a DP epsilon the answer is
    // what one expects
    ASSERT_DOUBLE_EQ( re, static_cast<double>(i) );
    ASSERT_DOUBLE_EQ( im, static_cast<double>(-i) );

  }
}
```

Exercise Behavior

Verify Behaviour

Some DPC++ Basics

- DPC++ based on modern C++
- Host - orchestrates work
- Devices (CPU, GPU, FPGA)
  - work is run on devices
  - devices have memories
  - work is organized into **command groups**
  - command groups are submitted into **command queues**
- buffer abstraction
  - manages memory
  - **accessors**: access buffers from command group (& via special host accessor)
  - runtime orchestrates data movement depending on access via accessors
- parallel constructs:
  - **parallel_for**
  - **single_task**
  - reductions (in DPC++)
  - `work' is either a functor or C++ lambda



Understanding API behavior

- Writing tests is a good way to understand a new API.
- In my case I was learning SYCL
  - queues, devices, buffers, accessors, offsets…
  - built in vector type sycl::vector<>
- Approach as before:
  - set up an inital state
  - do something in SYCL
  - assert expectation
- Save set up code between tests:
  - Use a TestFixture!!!!
  - GTest:
    - derive from ::testing::Test
    - override SetUp() and TearDown() methods…
- Examples: Pre-fill f_buf with Vectors of Complex Numbers, each with length N (=4 in this instance)

```
class SyCLVecTypeTest : public ::testing::Test {
public:
    static constexpr size_t num_float_elem() { return 1024; }
    static constexpr size_t num_cmpx_elem() { return num_float_elem()/2; }
    static constexpr size_t N=4;

    sycl::cpu_selector my_cpu;
    sycl::queue MyQueue;
    sycl::buffer<float,1> f_buf;

    SyCLVecTypeTest() : f_buf{sycl::range<1>{num_float_elem()}}, MyQueue{my_cpu} {}

protected:

    void SetUp() override
    {
        {
          std::cout << "Filling" << std::endl;
          sycl::range<1> N_vecs{num_cmpx_elem()/N};

          // Fill the buffers
          MyQueue.submit([&](handler& cgh) {

             auto write_fbuf = f_buf.get_access<sycl::access::mode::write>(cgh);

             cgh.parallel_for<class prefill>(N_vecs, [=](id<1> vec_id) {
                for(size_t lane=0; lane < N; ++lane) {
                   MyComplex<float> fval( vec_id[0]*2*N + 2*lane,
                                          vec_id[0]*2*N + 2*lane + 1 );

                   StoreLane<float,N>(lane,vec_id[0],write_fbuf, fval);
                }
             }); // parallel fo
            }); // queue submit

           MyQueue.wait();
        } // End of scope
    } // SetUp

};
```

Understanding API behavior

- Writing tests is a good way to understand a new API.
- In my case I was learning SYCL
  - queues, devices, buffers, accessors, offsets…
  - built in vector type sycl::vector<>
- Approach as before:
  - set up an inital state
  - do something in SYCL
  - assert expectation
- Save set up code between tests:
  - Use a TestFixture!!!!
  - GTest:
    - derive from ::testing::Test
    - override SetUp() and TearDown() methods…
- Examples: Pre-fill f_buf with Vectors of Complex Numbers, each with length N (=4 in this instance)

Derive from ::testing::Test

Stuff used by all Tests in fixture

override SetUp() & TearDown() as needed

```
class SyCLVecTypeTest : public ::testing::Test {
public:
    static constexpr size_t num_float_elem() { return 1024; }
    static constexpr size_t num_cmpx_elem() { return num_float_elem()/2; }
    static constexpr size_t N=4;

    sycl::cpu_selector my_cpu;
    sycl::queue MyQueue;
    sycl::buffer<float,1> f_buf;

    SyCLVecTypeTest() : f_buf{sycl::range<1>{num_float_elem()}}, MyQueue{my_cpu} {}

protected:

    void SetUp() override
    {
        {
          std::cout << "Filling" << std::endl;
          sycl::range<1> N_vecs{num_cmpx_elem()/N};

          // Fill the buffers
          MyQueue.submit([&](handler& cgh) {

             auto write_fbuf = f_buf.get_access<sycl::access::mode::write>(cgh);

             cgh.parallel_for<class prefill>(N_vecs, [=](id<1> vec_id) {
                for(size_t lane=0; lane < N; ++lane) {
                   MyComplex<float> fval( vec_id[0]*2*N + 2*lane,
                                          vec_id[0]*2*N + 2*lane + 1 );

                   StoreLane<float,N>(lane,vec_id[0],write_fbuf, fval);
                }
             }); // parallel fo
            }); // queue submit

            MyQueue.wait();
        } // End of scope
    } // SetUp

};
```

Working with Test Fixtures

- Use TEST_F(FixtureName, TestName)
- Examples
  - Check the setup is as expected.
    - SetUp(), accessors, parallel_for
  - Check some vector load/store functions
    - offsets, get_pointer(), single_task

```
class SyCLVecTypeTest;  // Defined last slide

// Verify that the TestFixture sets up the f_buf and d_buf arrays
// correctly and that we can reinterpret it as arrays of Complexes

TEST_F(SyCLVecTypeTest, CorrectSetUp)
{
   auto host_access_f = f_buf.get_access<access::mode::read>();
   for(size_t vec=0; vec < num_cmpx_elem()/N; ++vec) {
      for(size_t i=0; i < N ; ++i) {
        size_t j=vec*N + i;  // j-th complex number

        MyComplex<float> f=LoadLane<float,N>(i,vec,host_access_f);
        ASSERT_FLOAT_EQ( f.real(), static_cast<float>(2*j) );
        ASSERT_FLOAT_EQ( f.imag(), static_cast<float>(2*j+1));
      }
    }
}
```

define with TEST_F

```
TEST_F(SyCLVecTypeTest, TestComplexLoad)
{
   // All Vec load/stores need multi-ptr
   // Which are only in kernel scope.
   using T = SIMDComplexSyCL<float,N>;
   {
```

submit to queue

```
     // Single task kernel on device
     MyQueue.submit( [&](handler& cgh) {
```

accessor

```
      auto vecbuf = f_buf.get_access<access::mode::read_write>(cgh);
```

single task

```
      cgh.single_task<class vec_test_load>([=](){

        // Reade elem 0 of the buffer (We know what this is)
         T fc;  Load(fc,0,vecbuf.get_pointer());

         // Write it to element 1
         Store(1,vecbuf.get_pointer(),fc);
      });
     });
    }

   // Check on host
```

host accessor

```
   auto h_f = f_buf.get_access<access::mode::read>();
```

ASSERTIONs on host

```
   for(size_t i=0; i < N ; ++i) {
       float expect_real = 2*i;
       float expect_imag = 2*i+1;
       MyComplex<float> orig=LoadLane<float,N>(i,0,h_f);
       MyComplex<float> res=LoadLane<float,N>(i,1,h_f);
       ASSERT_FLOAT_EQ( orig.real(), res.real() );
       ASSERT_FLOAT_EQ( orig.imag(), res.imag() );
       ASSERT_FLOAT_EQ( res.real(), expect_real );
       ASSERT_FLOAT_EQ( res.imag(), expect_imag );
   }
}
```

Templates & Compile Time Constants

- Save duplication of test (e.g. vector lengths) ?
- Need to pass compile time constants or test templates?
- Use TYPED_TEST
  - templated test class derived from ::testing::Test;
  - ::testing::Types<> typelist with the type instantiations
  - TYPED_TEST_CASE will instantiate for each type
  - TYPED_TEST will let you write the concrete test
  - The concrete type tested is accessed via TypeParam
- std::integral_constant<Type,Value> wraps up a constant as a 'Type'
  - access value via TypeParam::value
- Example generates tests for N=1,2,4 and 8
  - check tests work for all available vector lengths..

Derive template from ::testing::Test

```
template<typename T>
class LaneOpsTester : public ::testing::Test{};
```

List of types with which to instantiate template

```
using test_types = ::testing::Types<
  std::integral_constant<int,1>,
  std::integral_constant<int,2>,
  std::integral_constant<int,4>,
  std::integral_constant<int,8> >;
```

instantiate templates

```
TYPED_TEST_CASE(LaneOpsTester, test_types);
```

define with TYPED_TEST()

```
TYPED_TEST(LaneOpsTester, TestLaneAccess)
{
  static constexpr int N = TypeParam::value;

  SIMDComplexSyCL<double,N> v;
  ComplexZero(v);
  std::array<MyComplex<double>,N> f;

  for(size_t i=0; i < N; ++i ) {
     f[i].real(i+1);
     f[i].imag(3*i + N);
     LaneOps<double,N>::insert(v,f[i],i);
  }

  for(size_t i=0; i < N; ++i ) {
     MyComplex<double> out( LaneOps<double,N>::extract(v,i) );
     ASSERT_FLOAT_EQ( out.real(), f[i].real());
     ASSERT_FLOAT_EQ( out.imag(), f[i].imag());
   }}
```

Run-time Parameterized Test

- Sometimes one needs access to parameterized tests that are not compile time…
- With Google test this gets into a situation needing multiple inheritance.
  - derive from ::testing::Test and
  - from ::testing::WithParamInterface<T>
  - T is the type of the parameter.
- Test written using TEST_P macro
- List of parameters speficied with INSTANTIATE_TEST_CASE_P macro

Derive from ::testing::Test

```
// Base test fixture
class FGMRESDRTests : public ::testing::Test {};
```

```
// Derive a text fixture using testing::WithParamInterface<>
class FGMRESDRTestsFloatParams : public ::FGMRESDRTests,
              public ::testing::WithParamInterface<float> {};
```

also derive from ::testing::WithParamInterface<ParamType>

define with TEST_P()

```
// Write a parameterized Test Case
TEST_P(FGMRESDRTestsFloatParams, testFullSolverDeflate)
{
  // Access the parameter
  float rsd_target_in = GetParam();
  // …
}
```

Access the parameter

```
// Instantiate 2 tests with the parameter values given
INSTANTIATE_TEST_CASE_P(FGMRESDRTests,
                        FGMRESDRTestsFloatParams,
                        testing::Values(1.0e-3,1.0e-9));
```

Define list of parameters

Test Environments

- In your tests, you may need to initialize subsystems, and set things up that you use for all your tests
- This can be done with a TestEnvironment
  - subclass the `::testing::Environment` class
  - override `SetUp()` and `TearDown()` methods
    - if setup calls need argv/argc copy them in environment class constructor
  - add with `::testing::AddGlobalTestEnvironment()`
    - must add before RUN_ALL_TESTS() macro is called in ‘main’
    - if using, it may be best to write your own main() rather than using the supplied gtest_main()
    - SetUp() called in order of addition, TearDown() called in the reverse order. Be aware, in case ordering causes issues.

```
// Set up Chroma
class ChromaEnvironment : public ::testing::Environment {
private:
  int argc_;
  char*** argv_;
  char*** copyArgs(const char*** argv);  // Copy arguments: body not shown
  void freeArgvs();                      // free argv_: body not shown
public:
  ChromaEnvironment(int* argc, char ***argv) : argc_(*argc),
                                               argv_(copyArgs(argv)) {}
```

Constructor can take argv, argc

```
  void SetUp(void) override {
    Chroma::initialize(&argc_, argv_);
    …
  }
```

SetUp() calls framework initializations Chroma, QUDA, Kokkos etc.

```
  void TearDown(void) override {
     Chroma::finalize();
  ]
```

TearDown() calls framework finalizations

```
  virtual ~ChromaEnvironment() {
     freeArgvs();
  }

  bool linkageHack(void) ; // Not shown to save space
};
```

```
// In some other file…
int main(int argc, char *argv[])
{
  ::testing::InitGoogleTest(&argc, argv);
  ::testing::Environment* const chroma_env =
   ::testing::AddGlobalTestEnvironment(new ChromaEnvironment(&argc,&argv));
  return RUN_ALL_TESTS();
}
```

Create & Add Test Environment BEFORE calling RUN_ALL_TESTS()

My Canonical Test Setup

- I typically use the QDP++ framework to write reference code
- I prefer to use CMake to drive the builds and tests
  - CMake makes it easy to use googletest as a sub-module in your project. See e.g. "An Introduction to Modern CMake"
- I generally use an env.sh to set-up compilers, modules, flags etc.
- build_qdpxx.sh builds and includes QDP++
- build_project.sh builds my project and the tests.
- Can have other 'extern' submodules. E.g. Kokkos

```
env.sh
build_qdpxx.sh
build_project.sh

src/
  project/
    CMakeLists.txt
    include/
    lib/
    extern/googletest/
    test/
      qdpxx_reference.cpp
      test_env.h
      CMakeLists.txt
      test_feature1.cpp
      …
  qdpxx/
```

Testing in CMake

- CMake makes adding tests easy
  - include(CTest) in toplevel CMakeLists.txt
  - add_test(NAME name COMMAND com)
- Can wrap in a macro, to build executable and turn it into one test (Introduction to Modern CMake)
- Run tests with:
  - make test
  - ctest
  - run individual executables
    - —help (list gtest options)
    - —gtest_list_tests (list available tests)
    - —gtest_filter=….   (allows filtering of tests)

```
# This should be in the toplevel CMakeLists.txt
include(CTest)

# This can be in the tests/ directory
# Make a library using my reference code, test environmnent main, etc.
add_library( testutils qdpxx_utils.h qdpxx_latticeinit.h qdpxx_latticeinit.cpp
             reunit.cpp test_env.cpp dslashm_w.cpp )

# Link Kokkos (in this case) and gtest and my qdp++ library to the
# test -library above. Kokkos can either be a sub-module built with
# add_subdirectory() or found with find_package()
target_link_libraries( testutils qdp Kokkos::kokkos gtest )

# This macro takes the testname and atts an executable from the argumnets
macro(package_add_test TESTNAME)
    # Make the executable
    add_executable(${TESTNAME} ${ARGN})

   # link libmg (the library I am testing) and my testutils
    target_link_libraries(${TESTNAME} libmg testutils )

    # Add the test to CTest
    add_test(NAME ${TESTNAME} COMMAND ${TESTNAME})
endmacro()

package_add_test(test_kokkos test_kokkos.cpp)
package_add_test(test_kokkos_perf test_kokkos_perf.cpp)
```

Testing in CMake

- CMake makes adding tests easy
  - include(CTest) in toplevel CMakeLists.txt
  - add_test(NAME name COMMAND com)
- Can wrap in a macro, to build executable and turn it into one test (Introduction to Modern CMake)
- Run tests with:
  - make test
  - ctest
  - run individual executables
    - —help (list gtest options)
    - —gtest_list_tests (list available tests)
    - —gtest_filter=….   (allows filtering of tests)

include CMake CTest

```
# This should be in the toplevel CMakeLists.txt
include(CTest)
```

Put reference code, test environment etc into a library: testutils

```
add_library( testutils qdpxx_utils.h qdpxx_latticeinit.h qdpxx_latticeinit.cpp
             reunit.cpp test_env.cpp dslashm_w.cpp )
```

link reference framework, Kokkos etc, and Googletest to the testutils library

```
target_link_libraries( testutils qdp Kokkos::kokkos gtest )
```

Macro to create executables, link testutils and create a test from sources

```
# This macro takes the testname and atts an executable from the argumnets
macro(package_add_test TESTNAME)
    # Make the executable
    add_executable(${TESTNAME} ${ARGN})

   # link libmg (the library I am testing) and my testutils
    target_link_libraries(${TESTNAME} libmg testutils )

    # Add the test to CTest
    add_test(NAME ${TESTNAME} COMMAND ${TESTNAME})
endmacro()
```

transitively links Googletest etc.

creates test

Apply Macro

```
package_add_test(test_kokkos test_kokkos.cpp)
package_add_test(test_kokkos_perf test_kokkos_perf.cpp)
```

MiniApp Example: Kokkos Dslash

- Test: ensure that the Dslash written in Kokkos is the same as an unoptimized trusted one in the framework.

```
void dslash(LatticeFermion& chi, const LatticeFermion& psi, enum PlusMinus isign, int cb) const
{
  switch (isign) {
     case PLUS:
      chi[rb[cb]] = spinReconstructDir0Minus(u[0] * shift(spinProjectDir0Minus(psi), FORWARD, 0))
              + spinReconstructDir0Plus(shift(adj(u[0]) * spinProjectDir0Plus(psi), BACKWARD, 0))
              + spinReconstructDir1Minus(u[1] * shift(spinProjectDir1Minus(psi), FORWARD, 1))
              + spinReconstructDir1Plus(shift(adj(u[1]) * spinProjectDir1Plus(psi), BACKWARD, 1))
              + spinReconstructDir2Minus(u[2] * shift(spinProjectDir2Minus(psi), FORWARD, 2))
              + spinReconstructDir2Plus(shift(adj(u[2]) * spinProjectDir2Plus(psi), BACKWARD, 2))
              + spinReconstructDir3Minus(u[3] * shift(spinProjectDir3Minus(psi), FORWARD, 3))
              + spinReconstructDir3Plus(shift(adj(u[3]) * spinProjectDir3Plus(psi), BACKWARD, 3));
      break;
    case MINUS:
      chi[rb[cb]] = spinReconstructDir0Plus(u[0] * shift(spinProjectDir0Plus(psi), FORWARD, 0))
              + spinReconstructDir0Minus(shift(adj(u[0]) * spinProjectDir0Minus(psi), BACKWARD, 0))
              + spinReconstructDir1Plus(u[1] * shift(spinProjectDir1Plus(psi), FORWARD, 1))
              + spinReconstructDir1Minus(shift(adj(u[1]) * spinProjectDir1Minus(psi), BACKWARD, 1))
              + spinReconstructDir2Plus(u[2] * shift(spinProjectDir2Plus(psi), FORWARD, 2))
              + spinReconstructDir2Minus(shift(adj(u[2]) * spinProjectDir2Minus(psi), BACKWARD, 2))
            + spinReconstructDir3Plus(u[3] * shift(spinProjectDir3Plus(psi), FORWARD, 3))
              + spinReconstructDir3Minus(shift(adj(u[3]) * spinProjectDir3Minus(psi), BACKWARD, 3))
      break;
    }
}
```

QDP++ reference: pretty simple

Test Code: Apply both QDP++ and MiniApp opertors. Check difference

```
TEST(TestKokkos, TestDslash)
{
  IndexArray latdims={{32,32,32,32}};
  initQDPXXLattice(latdims);

  multi1d<LatticeColorMatrix> gauge_in(n_dim);        // Synthetic QDP++ input gauge field
  for(int mu=0; mu < n_dim; ++mu) {
    gaussian(gauge_in[mu]); reunit(gauge_in[mu]);     // gaussian noise, then reunitarize
  }

  LatticeFermion psi_in=zero;   gaussian(psi_in);     // synthetic QDP++ input spinor, gaussian noise

  LatticeInfo info(latdims,4,3,NodeInfo());                         // Kokkos framework objects: info
  KokkosCBFineSpinor<MGComplex<REAL>,4> kokkos_spinor_even(info,EVEN);  // Spinor on half lattice
  KokkosCBFineSpinor<MGComplex<REAL>,4> kokkos_spinor_odd(info,ODD);    // Spinor on other half of lattice
  KokkosFineGaugeField<MGComplex<REAL>>  kokkos_gauge(info);            // Gauge field.

  // Import Gauge Field to kokkos based container
  QDPGaugeFieldToKokkosGaugeField(gauge_in, kokkos_gauge);

  // Create Kokkos Dslash object
  KokkosDslash<MGComplex<REAL>,MGComplex<REAL>,MGComplex<REAL>> D(info);

  LatticeFermion psi_out = zero;      LatticeFermion kokkos_out=zero; // fields for test results
  for(int cb=0; cb < 2; ++cb) {
     KokkosCBFineSpinor<MGComplex<REAL>,4>& out_spinor = (cb == EVEN) ? kokkos_spinor_even : kokkos_spinor_odd;
     KokkosCBFineSpinor<MGComplex<REAL>,4>& in_spinor = (cb == EVEN) ? kokkos_spinor_odd: kokkos_spinor_even;

     for(int isign=-1; isign < 2; isign+=2) {.                     //  The active part of the tests:
        psi_out = zero; kokkos_out = zero;                         //  Zero output
        dslash(psi_out,gauge_in,psi_in,isign,cb);                  //  Apply QDP++ Dslash

        QDPLatticeFermionToKokkosCBSpinor(psi_in, in_spinor);      // Import input to Kokkos-spinor
        D(in_spinor,kokkos_gauge,out_spinor,isign);                //  Apply New Kokkos based Dslash
        KokkosCBSpinorToQDPLatticeFermion(out_spinor, kokkos_out); // Export back into QDP++ frameworkd

        psi_out[rb[cb]] -= kokkos_out;                             // Compute norm of the diff between
        double norm_diff = toDouble(sqrt(norm2(psi_out,rb[cb])));  // QDP++ result and Kokkos one
        ASSERT_LT( norm_diff, 1.0e-5);                             // Assert that it is small enough
}}}
```

MiniApp Example: Kokkos Dslash

- Test: ensure that the Dslash written in Kokkos is the same as an unoptimized trusted one in the framework.

```
void dslash(LatticeFermion& chi, const LatticeFermion& psi, enum PlusMinus isign, int cb) const
{
  switch (isign) {
     case PLUS:
      chi[rb[cb]] = spinReconstructDir0Minus(u[0] * shift(spinProjectDir0Minus(psi), FORWARD, 0))
              + spinReconstructDir0Plus(shift(adj(u[0]) * spinProjectDir0Plus(psi), BACKWARD, 0))
              + spinReconstructDir1Minus(u[1] * shift(spinProjectDir1Minus(psi), FORWARD, 1))
              + spinReconstructDir1Plus(shift(adj(u[1]) * spinProjectDir1Plus(psi), BACKWARD, 1))
              + spinReconstructDir2Minus(u[2] * shift(spinProjectDir2Minus(psi), FORWARD, 2))
              + spinReconstructDir2Plus(shift(adj(u[2]) * spinProjectDir2Plus(psi), BACKWARD, 2))
              + spinReconstructDir3Minus(u[3] * shift(spinProjectDir3Minus(psi), FORWARD, 3))
              + spinReconstructDir3Plus(shift(adj(u[3]) * spinProjectDir3Plus(psi), BACKWARD, 3));
      break;
    case MINUS:
      chi[rb[cb]] = spinReconstructDir0Plus(u[0] * shift(spinProjectDir0Plus(psi), FORWARD, 0))
              + spinReconstructDir0Minus(shift(adj(u[0]) * spinProjectDir0Minus(psi), BACKWARD, 0))
              + spinReconstructDir1Plus(u[1] * shift(spinProjectDir1Plus(psi), FORWARD, 1))
              + spinReconstructDir1Minus(shift(adj(u[1]) * spinProjectDir1Minus(psi), BACKWARD, 1))
              + spinReconstructDir2Plus(u[2] * shift(spinProjectDir2Plus(psi), FORWARD, 2))
              + spinReconstructDir2Minus(shift(adj(u[2]) * spinProjectDir2Minus(psi), BACKWARD, 2))
            + spinReconstructDir3Plus(u[3] * shift(spinProjectDir3Plus(psi), FORWARD, 3))
              + spinReconstructDir3Minus(shift(adj(u[3]) * spinProjectDir3Minus(psi), BACKWARD, 3))
      break;
   }
}
```

QDP++ reference: pretty simple

Test Code: Apply both QDP++ and MiniApp opertors. Check difference

```
TEST(TestKokkos, TestDslash)
{
  IndexArray latdims={{32,32,32,32}};
  initQDPXXLattice(latdims);

  multi1d<LatticeColorMatrix> gauge_in(n_dim);
  for(int mu=0; mu < n_dim; ++mu) {
    gaussian(gauge_in[mu]); reunit(gauge_in[mu]);
  }
  LatticeFermion psi_in=zero;   gaussian(psi_in);
```

Generate Synthetic Data in known framework (QDP++)

```
  LatticeInfo info(latdims,4,3,NodeInfo());
  KokkosCBFineSpinor<MGComplex<REAL>,4> kokkos_spinor_even(info,EVEN);
  KokkosCBFineSpinor<MGComplex<REAL>,4> kokkos_spinor_odd(info,ODD);
  KokkosFineGaugeField<MGComplex<REAL>>  kokkos_gauge(info);

  // Import Gauge Field to kokkos based container
  QDPGaugeFieldToKokkosGaugeField(gauge_in, kokkos_gauge);
```

Generate Datatypes used in the test and import the data

```
  // Create Kokkos Dslash object
  KokkosDslash<MGComplex<REAL>,MGComplex<REAL>,MGComplex<REAL>> D(info);

  LatticeFermion psi_out = zero;      LatticeFermion kokkos_out=zero; // fields for test results
  for(int cb=0; cb < 2; ++cb) {
     KokkosCBFineSpinor<MGComplex<REAL>,4>& out_spinor = (cb == EVEN) ? kokkos_spinor_even : kokkos_spinor_odd;
     KokkosCBFineSpinor<MGComplex<REAL>,4>& in_spinor = (cb == EVEN) ? kokkos_spinor_odd: kokkos_spinor_even;

     for(int isign=-1; isign < 2; isign+=2) {.                         //  The active part of the tests:
          psi_out = zero; kokkos_out = zero;
          dslash(psi_out,gauge_in,psi_in,isign,cb);
```

Perform reference computation

```
          QDPLatticeFermionToKokkosCBSpinor(psi_in, in_spinor);
          D(in_spinor,kokkos_gauge,out_spinor,isign);
          KokkosCBSpinorToQDPLatticeFermion(out_spinor, kokkos_out);
```

Test the optimized code and export result

```
          psi_out[rb[cb]] -= kokkos_out;
          double norm_diff = toDouble(sqrt(norm2(psi_out,rb[cb])));
          ASSERT_LT( norm_diff, 1.0e-5);
}}}
```

Check correctness

Test output…

```
Running tests...
Test project /home/users/coe0071/HIP-Kokkos/KokkosDslashWorkspace/build/build_kokkos_dslash/test
    Start 1: test_kokkos
1/1 Test #1: test_kokkos ......................   Passed    0.71 sec

100% tests passed, 0 tests failed out of 1

Total Test time (real) =   0.72 sec
```

make test
or
ctest

./test_kokkos

```
INFO: Initializing Kokkos
INFO: Initializing QDP++
INFO: QDP++ Initialized
[==========] Running 1 test from 1 test case.
[----------] Global test environment set-up.
[----------] 1 test from TestKokkos
[ RUN      ] TestKokkos.TestDslash
Lattice initialized:
  problem size = 32 32 32 32
  layout size = 32 32 32 32
  logical machine size = 1 1 1 1
  subgrid size = 32 32 32 32
  total number of nodes = 1
  total volume = 1048576
  subgrid volume = 1048576
Initializing QDPDefaultAllocator.
Finished init of RNG
Finished lattice layout
[       OK ] TestKokkos.TestDslash (26206 ms)
[----------] 1 test from TestKokkos (26206 ms total)

[----------] Global test environment tear-down
[==========] 1 test from 1 test case ran. (26206 ms total)
[  PASSED  ] 1 test.
INFO: Finalizing QDP++
INFO: Finalizing Kokkos
```

QUDA-Chroma Integration

- Chroma wraps QUDA solvers etc.
- Chroma objects instantiated via “object factories”
  - (xml parameters) -> objects
- Test library integration:
  - have test XML parameters
  - SetUp() creates chroma objects
  - Factory dowcasts to base type
  - I added getSolver() function which upcasts back to original type so I can get at its public internals.

```
namespace SymmPrecTesting
{
   std::string inv_param_quda_bicgstba_xml =              \
     "<?xml version='1.0'?>                                \
     <Param>                                               \
       <InvertParam>                                       \
           <invType>QUDA_CLOVER_INVERTER</invType>         \
            …  /* stuff hidden for space reasons */ \
          “;
};
```

Store Parameters as a string in .h file

```
template<typename TestType>
class QudaFixtureT : public TestType {
public:
  void SetUp() {

    /* … detail removed to save space */

    // Turn parameter into an input stream
    std::istringstream inv_param_xml_stream(inv_param_quda_bicgstab_xml);

    // Convert XML to something internal
    GroupXML_t inv_param = readXMLGroup(xml_in, "//InvertParam", “invType”);

    // Factory create the QUDA solver
    linop_solver = S_symm->invLinOp(state,inv_param);
  }
```

Create objects from parameters

```
  // Return Upcasted version of linop solvers to access public innards
  LinOpSysSolverQUDAClover& getSolver() {
    return dynamic_cast<LinOpSysSolverQUDAClover&>(*linop_solver);
  }
  …
```

Upcast to original type to be able to inspect innards

QUDA-Chroma Integration

- Now the test:
  - In this case compare linear operators match.
  - use the quda_inv_param struct to control QUDA behaviour (this struct was set up when QUDA solver was created)
  - but can change behaviour by changing the quda_inv_param struct members (e.g op. vs. hermitian conj.)
  - Can call QUDA directly to apply it's linear operator
  - Compare with the Chroma one

```
class QudaFixture : public QudaFixtureT<::testing::Test> {};

TEST_F(QudaFixture, TestCloverMat)
{
  auto the_quda_solver = getSolver();
  auto quda_inv_param = the_quda_solver.getQudaInvertParam();

  for(int dagger = 0; dagger < 2; ++dagger) {
     enum PlusMinus isign = ( dagger == 0 ) ? PLUS : MINUS;

     /* Set Op vs. Hermitian Conj. op */
     quda_inv_param.dagger = (dagger == 0 ) ? QUDA_DAG_NO : QUDA_DAG_YES;

     /* Prepare source and result vector */
     T src=zero; T res=zero; T res_quda = zero;
     gaussian(src,rb[1]);

     (*M_symm)(res,src,isign);



     auto src_ptr = (void *)
                  &(src.elem(rb[1].start()).elem(0).elem(0).real());
    auto res_quda_ptr = (void *)
                  &(res_quda.elem(rb[1].start()).elem(0).elem(0).real());

     /* Call QUDA. This will Import the vectors */
     MatQuda(res_quda_ptr, src_ptr, &quda_inv_param);


     T diff = zero;  diff[rb[1]] = res_quda - res;
     Double norm_diff_per_site = sqrt(norm2(diff,rb[1]))/sites;
     ASSERT_LT( toDouble(norm_diff_per_site), 1.0e-14);
}}
```

QUDA-Chroma Integration

- Now the test:
  - In this case compare linear operators match.
  - use the quda_inv_param struct to control QUDA behaviour (this struct was set up when QUDA solver was created)
  - but can change behaviour by changing the quda_inv_param struct members (e.g op. vs. hermitian conj.)
  - Can call QUDA directly to apply it's linear operator
  - Compare with the Chroma one

```
class QudaFixture : public QudaFixtureT<::testing::Test> {};

TEST_F(QudaFixture, TestCloverMat)
{
  auto the_quda_solver = getSolver();
  auto quda_inv_param = the_quda_solver.getQudaInvertParam();

  for(int dagger = 0; dagger < 2; ++dagger) {
    enum PlusMinus isign = ( dagger == 0 ) ? PLUS : MINUS;

    /* Set Op vs. Hermitian Conj. op */
    quda_inv_param.dagger = (dagger == 0 ) ? QUDA_DAG_NO : QUDA_DAG_YES;

    /* Prepare source and result vector */
    T src=zero; T res=zero; T res_quda = zero;
    gaussian(src,rb[1]);

    (*M_symm)(res,src,isign);

    auto src_ptr = (void *)
                &(src.elem(rb[1].start()).elem(0).elem(0).real());
   auto res_quda_ptr = (void *)
                &(res_quda.elem(rb[1].start()).elem(0).elem(0).real());

    /* Call QUDA. This will Import the vectors */
    MatQuda(res_quda_ptr, src_ptr, &quda_inv_param);

    T diff = zero;  diff[rb[1]] = res_quda - res;
    Double norm_diff_per_site = sqrt(norm2(diff,rb[1]))/sites;
    ASSERT_LT( toDouble(norm_diff_per_site), 1.0e-14);
}}
```

Get Upcasted object quda_inv_params

Init Data and apply Chroma's own operator

Apply QUDA's operator - will import & export QDP++ types due to settings in quda_inv_param

Check result

QUDA Chroma Integration

- This test is very handy
  - if QUDA Changes I can check the integration is still sound.
  - if QUDA-Chroma users report errors, I can liaise with Kate Clark, the Lead Developer of QUDA at NVIDIA to try and see whether my tests break too — handy for finding bugs e.g. in the Input XMLs as well as our code

What else can I be testing?

- Performance
  - assert runtime of test is appropriate (not overlong)
- Symmetries & other invariants - helpful if there is no reference to test against
  - E.g. In LQCD: gauge invariance / gauge covariance
  - Conservation laws
  - …

Summary

- Testing is good practice: can range from small unit tests all the way up to full CI
- Testing in my opinion, is a good way to learn about programming models
  - especially in combination with mini-apps
  - e.g. SYCL buffer management, single tasks, parallel_for constructs, SIMD issues
- It is useful to have a reliable reference code (in my case QDP++)
- For C++ programmers there are many good testing frameworks
  - I focused here in GoogleTest, but there are others: Catch, Boost.Test, CppUnit …
- For clients of rapidly co-developing libraries, it is helpful to have integration tests
- It is easy to add tests to CMake build systems
- I hope the examples here will help you write useful tests.

References

- The IDEAS project: https://ideas-productivity.org/
- GoogleTest API: https://github.com/google/googletest
  - check out the README, and follow the links to the GoogleTest Primer
- Fortran Users May care to check out pFUnit: https://www.exascaleproject.org/event/pFUnit/
- An introduction to Modern CMake: https://cliutils.gitlab.io/modern-cmake/
  - free electronic book with examples on GoogleTest integration
- Two mini apps from me:
  - KokkosDslash: https://github.com/bjoo/KokkosDslash.git
  - SYCLDslash: https://github.com/bjoo/SyCLDslash.git
- The QUDA Library from NVIDIA: http://lattice.github.io/quda/
- Kokkos: https://github.com/kokkos/kokkos, https://www.exascaleproject.org/event/introduction-to-kokkos/
- SYCL: https://www.khronos.org/registry/SYCL/specs/sycl-1.2.1.pdf
- Intel OneAPI: https://software.intel.com/sites/default/files/oneAPIProgrammingGuide_8.pdf
- StackOverflow: https://stackoverflow.com/
- Other IDEAS talks: https://www.exascaleproject.org/event/ci2sl/

Acknowledgments

- B. Joo acknowledges funding from the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research under the Exascale Computing Project (2.2.1.01 ADSE03 Lattice QCD)
- B. Joo acknowledges funding from the U.S. Department of Energy, Office of Science, Offices of Nuclear Physics, High Energy Physics and Advanced Scientific Computing Research under the SciDAC-4 program.
- B. Joo acknowledges travel funding from NERSC for a summer Affiliate Appointment for work on Kokkos