PREPARING FOR QUDA 2.0

Kate Clark

OUTLINE

Motivation

Goals

Abstraction Troika

Summary

**ECP benchmarks apps

QUDA

- "QCD on CUDA" – http://lattice.github.com/quda (open source, BSD license)
- Effort started at Boston University in 2008, now in wide use as the GPU backend for BQCD, **Chroma**, CPS**, MILC****, TIFR, etc.
- Provides solvers for all major fermion discretizations, with multi-GPU support
- Maximize performance
  - Mixed-precision methods
  - Multigrid solvers for optimal convergence (see Evan's talk)
  - NVSHMEM for improving strong scaling (see Mathias's talk)
  - Utilize tensor cores for super-linear acceleration (see Jiqun's talk)

- **A research tool for how to reach the exascale (and beyond)**
  - Optimally mapping the problem to hierarchical processors and node topologies

QUDA CONTRIBUTORS

10+ years - lots of contributors

Ron Babich (NVIDIA)
Simone Bacchio (Cyprus)
Kip Barros (LANL)
Rich Brower (Boston University)
Nuno Cardoso (NCSA)
Kate Clark (NVIDIA)
Michael Cheng (Boston University)
Carleton DeTar (Utah University)
Justin Foley (Utah -> NIH)
Joel Giedt (Rensselaer Polytechnic Institute)
Arjun Gambhir (William and Mary)
Steve Gottlieb (Indiana University)
Kyriakos Hadjiyiannakou (Cyprus)
Dean Howarth (LLNL)
Bálint Joó (Jlab)
Hyung-Jin Kim (BNL -> Samsung)
Bartek Kostrzewa (Bonn)
Claudio Rebbi (Boston University)
Eloy Romero (William and Mary)
Hauke Sandmeyer (Bielefeld)
Guochun Shi (NCSA -> Google)
Mario Schröck (INFN)
Alexei Strelchenko (FNAL)
Jiqun Tu (NVIDIA)
Alejandro Vaquero (Utah University)
Mathias Wagner (NVIDIA)
André Walker-Loud (LBL)
Evan Weinberg (NVIDIA)
Frank Winter (Jlab)
Yi-bo Yang (CAS)

QUDA 1.X LIMITATIONS

(not exhaustive…)

No unified style in QUDA

- Unnecessary duplication of code
- Too much boilerplate in adding new functionality

Parts of QUDA are poorly written, documented and / or understood

- E.g., Gauge fixing, pure gauge evolution
- Authors have come and gone

Run-time compilation (Jitify)

- Different code paths for Jitify not sustainable
- Huge potential if we can make it first-class citizen

RGB PORTABILITY

Multiple accelerated non-NVIDIA architectures coming online

- Much interest in having QUDA available everywhere
- CUDA C++, HIP, SYCL, OpenMP, `std::par`

While it's not NVIDIA's job to do this…

- Risk of splintering community if we don't enable it
- We want more folks working on QUDA, not less

Big risks if we don't get the foundation correct

- Churn with PRs that add targets

TEAM

(Bi-)weekly meetings for the past year

Kate Clark: architecture lead, `std::par`

Bálint Joó (ORNL), Dean Howarth (LLNL): HIP

James Osborn (ANL): SYCL

Xiao-Yong Jin (ANL): OpenMP

Alexei Strelchenko (FNAL): DPC++

AMD: Damon McDougall, Corbin Robeck

Intel: Patrick Steinbrecher

OBJECTIVES

Enable QUDA to run on all accelerated architectures

Summit, Jülich Booster, Perlmutter, (CUDA) *and* Frontier (AMD), Aurora (SYCL)

No target specific code should exist outside of targets/arch directories

No compromises to performance on NVIDIA architecture

Enable us to embrace next-generation features while retaining portability

Tensor cores, CUDA graphs, NVSHMEM, etc.

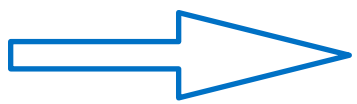
THE ABSTRACTION TROIKA

1. API
2. Generic Kernels
3. Target specialization

We *evolved* to the final abstraction, which came from the unending phone calls

API

All CUDA APIs are abstracted away and not called directly (mostly already in develop)

`cudaMemcpy`  `qudaMemcpy`

Improved debugging

- All API calls now have their errors checked
- All API calls now have automatic source tracking
- Very easy to profile API overheads

To first approximation, all that is needed for HIP

Abstract overall functionality where appropriate as opposed to all API calls

```
    // handles for obtained ghost pointers
    cudaIpcMemHandle_t ipcRemoteGhostDestHandle[2][2][QUDA_MAX_DIM];

    for (int b=0; b<2; b++) {
#ifndef NVSHMEM_COMMS
      for (int dim=0; dim<4; ++dim) {
        if (comm_dim(dim)==1) continue;
        for (int dir=0; dir<2; ++dir) {
          MsgHandle* sendHandle = nullptr;
          MsgHandle* receiveHandle = nullptr;
          int disp = (dir == 1) ? +1 : -1;

          // first set up receive
          if (comm_peer2peer_enabled(1-dir,dim)) {
            receiveHandle = comm_declare_receive_relative(&ipcRemoteGhostDestHandle[b][1-dir][dim],
                                                          dim, -disp,
                                                          sizeof(ipcRemoteGhostDestHandle[b][1-dir][dim]));
          }
          // now send
          cudaIpcMemHandle_t ipcLocalGhostDestHandle;
          if (comm_peer2peer_enabled(dir,dim)) {
            cudaIpcGetMemHandle(&ipcLocalGhostDestHandle, ghost_recv_buffer_d[b]);
            sendHandle = comm_declare_send_relative(&ipcLocalGhostDestHandle,
                                                    dim, disp,
                                                    sizeof(ipcLocalGhostDestHandle));
          }
          if (receiveHandle) comm_start(receiveHandle);
          if (sendHandle) comm_start(sendHandle);

          if (receiveHandle) comm_wait(receiveHandle);
          if (sendHandle) comm_wait(sendHandle);

          if (sendHandle) comm_free(sendHandle);
          if (receiveHandle) comm_free(receiveHandle);
        }
      }

      checkCudaError();
#endif
      // open the remote memory handles and set the send ghost pointers
      for (int dim = 0; dim < 4; ++dim) {
#ifndef NVSHMEM_COMMS
        // TODO: We maybe can force loopback comms to use the IB path here
        if (comm_dim(dim) == 1) continue;
#endif
        // even if comm_dim(2) == 2, we might not have p2p enabled in both directions, so check this
        const int num_dir
          = (comm_dim(dim) == 2 && comm_peer2peer_enabled(0, dim) && comm_peer2peer_enabled(1, dim)) ? 1 : 2;
        for (int dir = 0; dir < num_dir; ++dir) {
#ifndef NVSHMEM_COMMS
          if (!comm_peer2peer_enabled(dir, dim)) continue;
          void **ghostDest = &(ghost_remote_send_buffer_d[b][dim][dir]);
          cudaIpcOpenMemHandle(ghostDest, ipcRemoteGhostDestHandle[b][dir][dim], cudaIpcMemLazyEnablePeerAccess);
#else
          ghost_remote_send_buffer_d[b][dim][dir]
            = nvshmem_ptr(static_cast<char *>(ghost_recv_buffer_d[b]), comm_neighbor_rank(dir, dim));
#endif
        }
        if (num_dir == 1) ghost_remote_send_buffer_d[b][dim][1] = ghost_remote_send_buffer_d[b][dim][0];
      }
    } // buffer index

    checkCudaError();
```

e.g., peer-to-peer initialization

```
    for (int b=0; b<2; b++) {
      // set remote send buffer to ghost receive buffers on neighboring processes
      comm_create_neighbor_memory(ghost_remote_send_buffer_d[b], ghost_recv_buffer_d[b]);
      // get remote events
      comm_create_neighbor_event(ipcRemoteCopyEvent[b], ipcCopyEvent[b]);
    }
```

GENERIC KERNELS

LQCD is trivial

- only a handful of different kernel motifs required

Identify the shape of the parallelism of each kernel

- Reimplemented as *functors* and mapped to one of six generic shapes or *kernels*
  - 1d, 2d, 3d, reduction, multi-reduction, block
- Functors expose parallelism and be called by any runtime that exposes parallelism
  - CUDA / HIP: generic kernels
  - SYCL: wrapped in Lambda launcher
  - `std::par`: functor passed to `std::for_each`, `std::transform_reduce`, etc.

For each generic kernel, launch the kernel using the corresponding Tunable class

- e.g., `TunableKernel1D`, `TunableMultiReduction`

PLAQUETTE EXAMPLE

```
template<int blockSize, typename Arg>
__global__ void computePlaq(Arg arg)
{
  int idx = threadIdx.x + blockIdx.x * blockDim.x;
  int parity = threadIdx.y;

  double2 plaq = make_double2(0.0,0.0);

  while (idx < arg.threads) {
    int x[4];
    getCoords(x, idx, arg.X, parity);
#pragma unroll
    for (int dr=0; dr<4; ++dr) x[dr] += arg.border[dr];

#pragma unroll
    for (int mu = 0; mu < 3; mu++) {
#pragma unroll
      for (int nu = 0; nu < 3; nu++) {
        if (nu >= mu + 1) plaq.x += plaquette(arg, x, parity, mu, nu);
      }

      plaq.y += plaquette(arg, x, parity, mu, 3);
    }

    idx += blockDim.x*gridDim.x;
  }

  // perform final inter-block reduction and write out result
  arg.template reduce2d<blockSize, 2>(plaq);
}
```

Original kernel

```
template <typename Arg> struct Plaquette : plus<vector_type<double, 2>> {
  using reduce_t = vector_type<double, 2>;
  using plus<reduce_t>::operator();
  Arg &arg;
  constexpr Plaquette(Arg &arg) : arg(arg) {}
  static constexpr const char *filename() { return KERNEL_FILE; }

  // return the plaquette at site (x_cb, parity)
  __device__ __host__ inline reduce_t operator()(reduce_t &value, int x_cb, int parity)
  {
    reduce_t plaq;

    int x[4];
    getCoords(x, x_cb, arg.X, parity);
#pragma unroll
    for (int dr=0; dr<4; ++dr) x[dr] += arg.border[dr];
#pragma unroll
    for (int mu = 0; mu < 3; mu++) {
#pragma unroll
      for (int nu = 0; nu < 3; nu++) {
        if (nu >= mu + 1) plaq[0] += plaquette(arg, x, parity, mu, nu);
      }

      plaq[1] += plaquette(arg, x, parity, mu, 3);
    }

    return plus::operator()(plaq, value);
  }
```

Plaquette Functor

PLAQUETTE EXAMPLE

```
template <template <typename> class Transformer, typename Arg, bool grid_stride = true>
__global__ void Reduction_impl(Arg arg)
{
  using reduce_t = typename Transformer<Arg>::reduce_t;
  Transformer<Arg> t(arg);

  auto idx = threadIdx.x + blockIdx.x * blockDim.x;
  auto j = threadIdx.y;

  reduce_t value = arg.init();

  while (idx < arg.threads.x) {
    value = t(value, idx, j);
    if (grid_stride) idx += blockDim.x * gridDim.x; else break;
  }

  // perform final inter-block reduction and write out result
  reduce<Arg::block_size_x, Arg::block_size_y>(arg, t, value);
}
```

Generic CUDA Reduction Kernel

Functor callable from any target

```
template <template <typename> class Functor,
  typename Arg>
auto Reduction_host(const Arg &arg)
{
  using reduce_t = typename Functor<Arg>::reduce_t;
  Functor<Arg> t(arg);

  reduce_t value = arg.init();

  for (int j = 0; j < (int)arg.threads.y; j++) {
    for (int i = 0; i < (int)arg.threads.x; i++) {
      value = t(value, i, j);
    }
  }

  return value;
}
```

Generic CPU Reduction Kernel

```
template <typename Arg> struct Plaquette : plus<vector_type<double, 2>> {
  using reduce_t = vector_type<double, 2>;
  using plus<reduce_t>::operator();
  Arg &arg;
  constexpr Plaquette(Arg &arg) : arg(arg) {}
  static constexpr const char *filename() { return KERNEL_FILE; }

  // return the plaquette at site (x_cb, parity)
  __device__ __host__ inline reduce_t operator()(reduce_t &value, int x_cb, int parity)
  {
    reduce_t plaq;

    int x[4];
    getCoords(x, x_cb, arg.X, parity);
#pragma unroll
    for (int dr=0; dr<4; ++dr) x[dr] += arg.border[dr];
#pragma unroll
    for (int mu = 0; mu < 3; mu++) {
#pragma unroll
      for (int nu = 0; nu < 3; nu++) {
        if (nu >= mu + 1) plaq[0] += plaquette(arg, x, parity, mu, nu);
      }

      plaq[1] += plaquette(arg, x, parity, mu, 3);
    }

    return plus::operator()(plaq, value);
  }
```

Plaquette Functor

PLAQUETTE EXAMPLE

Driver

```
template<typename Float, int nColor, QudaReconstructType recon>
class GaugePlaq : TunableLocalParityReduction {
  const GaugeField &u;
  double2 &plq;

public:
  GaugePlaq(const GaugeField &u, double2 &plq) :
    u(u),
    plq(plq)
  {
#ifdef JITIFY
    create_jitify_program("kernels/gauge_plaq.cuh");
#endif
    strcpy(aux, compile_type_str(u));
    apply(0);
  }

  void apply(const qudaStream_t &stream){
    if (u.Location() == QUDA_CUDA_FIELD_LOCATION) {
      TuneParam tp = tuneLaunch(*this, getTuning(), getVerbosity());
      GaugePlaqArg<Float, nColor, recon> arg(u);
#ifdef JITIFY
      using namespace jitify::reflection;
      jitify_error = program->kernel("quda::computePlaq")
        .instantiate((int)tp.block.x,type_of(arg))
        .configure(tp.grid,tp.block,tp.shared_bytes,stream).launch(arg);
      arg.launch_error = jitify_error == CUDA_SUCCESS ? QUDA_SUCCESS : QUDA_ERROR;
#else
      LAUNCH_KERNEL_LOCAL_PARITY(computePlaq, (*this), tp, stream, arg, decltype(arg));
#endif
      arg.complete(plq);
      if (!activeTuning()) {
        comm_allreduce_array((double*)&plq, 2);
        for (int i = 0; i < 2; i++) ((double*)&plq)[i] /= 9.*2*arg.threads*comm_size();
      }
    } else {
      errorQuda("CPU not supported");
    }
  }

  TuneKey tuneKey() const { return TuneKey(u.VolString(), typeid(*this).name(), aux); }
  long long flops() const
  {
    auto Nc = u.Ncolor();
    return 6ll*u.Volume()*(3 * (8 * Nc * Nc * Nc - 2 * Nc * Nc) + Nc);
  }
  long long bytes() const { return u.Bytes(); }
};
```

```
class TunableReduction2D : public TunableKernel
{
  QudaFieldLocation location;

  template <template <typename> class Functor, typename T, typename Arg>
  void launch(std::vector<T> &result, const TuneParam &tp, const qudaStream_t &stream, Arg &arg)
  {
    if (location == QUDA_CUDA_FIELD_LOCATION) {
      launch_device<Functor>(result, tp, stream, arg);
    } else {
      launch_host<Functor>(result, tp, stream, arg);
    }
  }
};
```

```
template<typename Float, int nColor, QudaReconstructType recon>
class GaugePlaq : public TunableReduction<> {
  const GaugeField &u;
  double2 &plq;

public:
  GaugePlaq(const GaugeField &u, double2 &plq) :
    TunableReduction(u),
    u(u),
    plq(plq)
  {
    apply(device::get_default_stream());
  }

  void apply(const qudaStream_t &stream)
  {
    TuneParam tp = tuneLaunch(*this, getTuning(), getVerbosity());
    GaugePlaqArg<Float, nColor, recon> arg(u);
    launch<Plaquette>(plq, tp, stream, arg);
    for (int i = 0; i < 2; i++) ((double*)&plq)[i] /= 9.*2*arg.threads.x*comm_size();
  }

  long long flops() const
  {
    auto Nc = u.Ncolor();
    return 6ll*u.Volume()*(3 * (8 * Nc * Nc * Nc - 2 * Nc * Nc) + Nc);
  }
  long long bytes() const { return u.Bytes(); }
};
```

TARGET SPECIALIZATION

target::dispatch

```
__host__ __device__ float sinf(float a)
{
#ifdef __CUDA_ARCH__
  return __sinf(a);
#else
  return sinf(a);
#endif
}
```

nvcc / clang

"No target specific code should exist outside of targets/arch directories"

```
// nvcc or clang: compile-time dispatch
template <template <bool, typename ...> class f, typename ...Args>
__host__ __device__ auto dispatch(Args &&... args)
{
#ifdef __CUDA_ARCH__
  return f<true>()(args...);
#else
  return f<false>()(args...);
#endif
}
```

```
template <bool> struct sinf_impl {
  float operator()(float a) { return sinf(a); }
};

template <> struct sinf_impl<true> {
  __device__ float operator()(float a) { return __sinf(a); }
};


__host__ __device__ float sinf(float a) { return target::dispatch<sinf_impl>(a); }
```

Use class template specialization where we need to specialize device code

Dispatcher uses *template template parameters* for generality (compile-time or run-time dispatch)

Macro free and allows for target optimization

intrinsics, inline assembly, target header-only libraries, etc.

Can write generic implementation and template specialize as needed

`include/targets/cuda/aos.h // optimized for CUDA`

`include/targets/generic/aos.h // works on all platforms`

BONUS ROUND

Complete Library Rewrite Presents New Opportunities

QUDA is smaller 

Kernels are on the whole faster: Block orthogonalize 3x faster, prolongator/restrictor 20% faster

Multigrid is more flexible: all block sizes supported

Compilation is faster

- Full Wilson-clover, mixed-precision, multigrid build < 10 mins
- More expansive fast compilation mode for developing

Jitify

- Run-time compilation automatically supported everywhere now

New Physics is now possible

- LSD collaboration: large Nc (Dean Howarth)

PRESENT STATUS

`feature/generic_kernels` functionally complete

- testing, testing, testing
- PR in review: https://github.com/lattice/quda/pull/1056

HIP PR is open from Balint: https://github.com/lattice/quda/pull/1092

- Abstraction is successful
- Basic solvers working 

- Multi-GPU working but performance is still being worked on

SYCL PR is expected from James soon

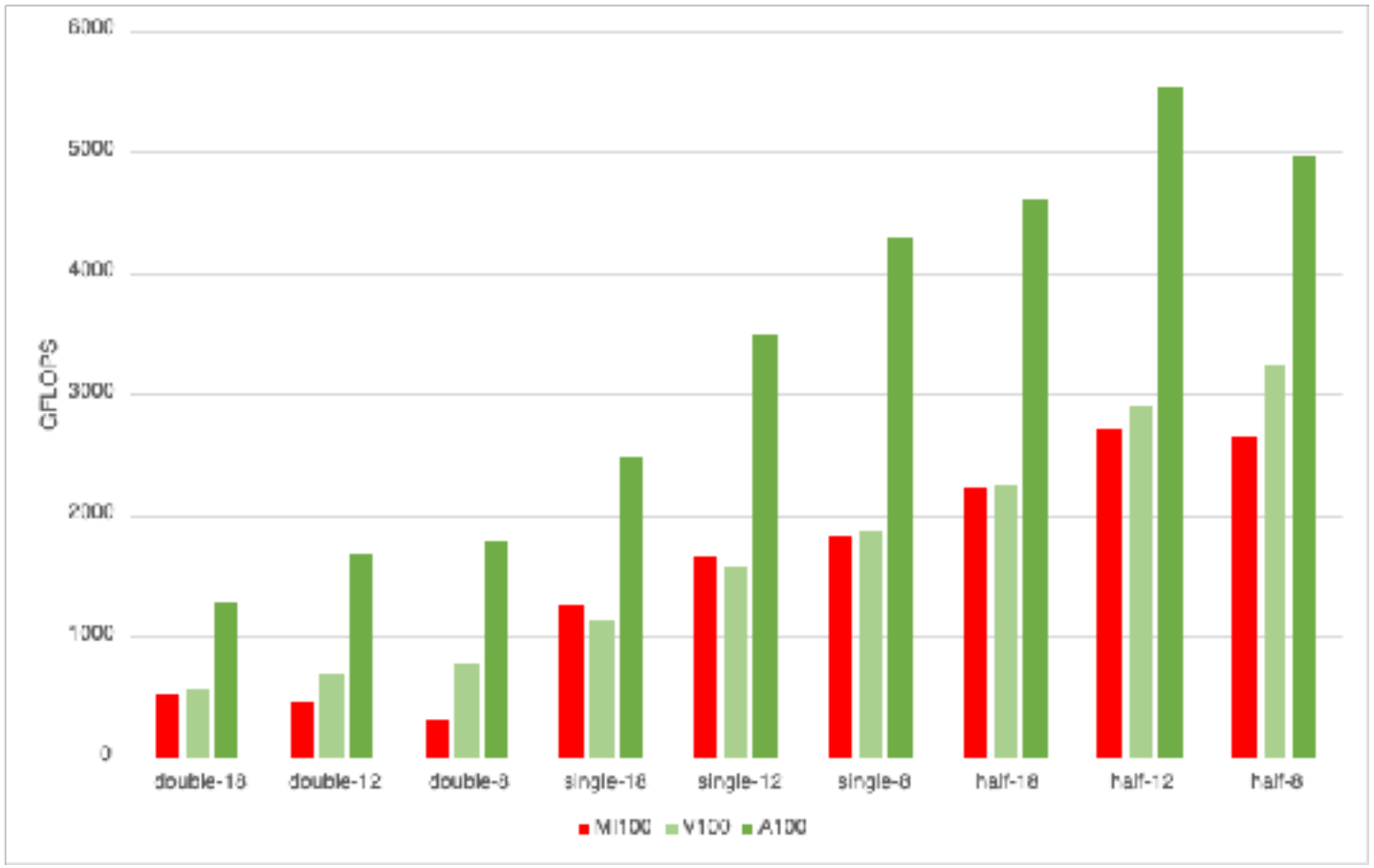
- Basic kernels are working

Longer poles: OpenMP, `std::par`

V100/A100 results from Evan Weinberg
Preliminary MI100 results from Bálint Joó

WILSON DSLASH

32^4



GPU	Machine	Theoretical TB/s
NVIDIA V100	OLCF Summit	0.9
NVIDIA A100-80	NVIDIA Selene	2.0
AMD MI100	OLCF Spock	1.2

CONCLUSION

QUDA 2.0 represents a complete rewrite of QUDA

Portable and will run across the Exascale

Opens new doors

- Optimization
- Machine specialization
- Algorithms
- Physics


NVIDIA