LATTICE QCD ON GPU SYSTEMS

Mathias Wagner, Lattice Practices 2024

**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
  - NVSHMEM for improving strong scaling
  - Utilize tensor cores for super-linear acceleration

- **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

Buck Babich (NVIDIA)
Simone Bacchio (Cyprus)
Michael Balfhauf (Regensburg)
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)
Marco Garofalo (Bonn)
Steve Gottlieb (Indiana University)
Kyriakos Hadjiyiannakou (Cyprus)
Ben Hoerz (Intel)
Dean Howarth (LBL)
Xiangyu Jiang (ITP, Chinese Academy of Sciences)
Xiao-Yong Jin (ANL)
Bálint Joó (Jlab)
Hyung-Jin Kim (BNL -> Samsung)
Bartek Kostrzewa (Bonn)
Damon McDougall (AMD)
Colin Morningstar (CMU)
James Osborn (ANL)
Ferenc Pittler (Cyprus)
Claudio Rebbi (Boston University)
Eloy Romero (William and Mary)
Hauke Sandmeyer (Bielefeld)
Aniket Sen (Bonn)
Guochun Shi (NCSA -> Google)
Mario Schröck (INFN)
Alexei Strelchenko (FNAL)
Jiqun Tu (NVIDIA)
Alejandro Vaquero (Utah University)
Michael Wagman (FNAL)
Mathias Wagner (NVIDIA)
André Walker-Loud (LBL)
Evan Weinberg (NVIDIA)
Frank Winter (Jlab)
Yi-bo Yang (CAS)

TEN YEARS OF QUDA

in use as GPU backend for BQCD, Chroma, CPS, MILC, TIFR, etc.

Solvers for all major fermionic discretizations

Routines needed for gauge-field generation

Maximize performance

- Exploit symmetries to minimize memory traffic
- Mixed-precision methods (16 bit / 8 bit)
- Domain-decomposed (Schwarz) preconditioners for strong scaling
- Eigenvector and deflated solvers (Lanczos, EigCG, GMRES-DR)
- Multi-source solvers
- Multigrid solvers for optimal convergence

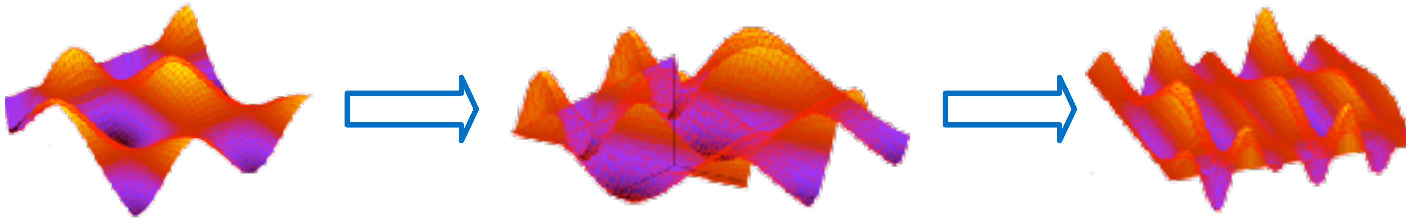
STEPS IN AN LQCD CALCULATION

$$D_{ij}^{\alpha\beta}(x,y;U)\psi_j^{\beta}(y) = \eta_i^{\alpha}(x)$$

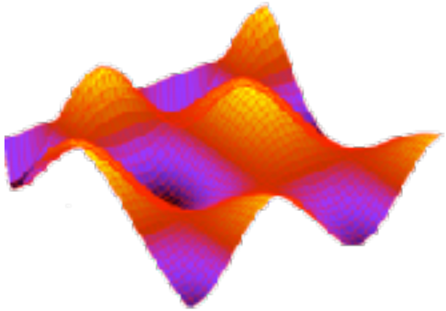
or Ax = b

1. Generate an ensemble of gluon field configurations “gauge generation”
    - Produced in sequence, with hundreds needed per ensemble
    - Strong scaling required with 100-1000 TFLOPS sustained for several months
    - 50-90% of the runtime is in the linear solver
    - O(1) solve per linear system
    - Target 16^4 per GPU

Simulation Cost ~ $a^{-6} V^{5/4}$

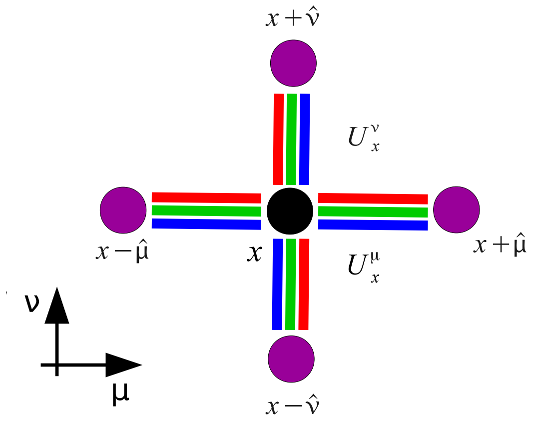


2. “Analyze” the configurations
    - Can be farmed out, assuming ~10 TFLOPS per job
    - Task parallelism means that clusters reign supreme here
    - 80-99% of the runtime is in the linear solver
    - Many solves per system, e.g., $O(10^6)$
    - Target 24^4-32^4 per GPU



MAPPING THE DIRAC OPERATOR TO CUDA

$$D_{x,x'} =$$



Finite difference operator in LQCD is known as Dslash

Assign a single space-time point to each thread

- V = XYZT threads, e.g., V = 24^4 => 3.3×10^6 threads

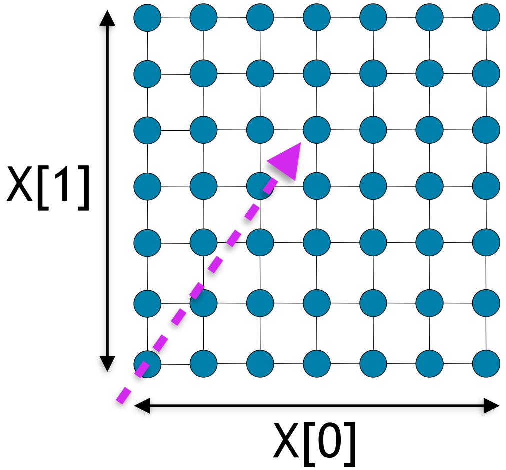
Looping over direction each thread must

- Load the neighboring spinor (24 numbers x8)
- Load the color matrix connecting the sites (18 numbers x8)
- Do the computation
- Save the result (24 numbers)

Each thread has (Wilson Dslash) 0.92 naive arithmetic intensity

QUDA reduces memory traffic

- Exact SU(3) matrix compression (18 => 12 or 8 real numbers)
- Use 16-bit fixed-point representation with mixed-precision solver



LINEAR SOLVERS

QUDA supports a wide range of linear solvers

- CG, BiCGstab, GCR, Multi-shift solvers, etc.

Condition number inversely proportional to mass

- Light (realistic) masses are highly singular
- Naive Krylov solvers suffer from critical slowing down at decreasing mass

Entire solver algorithm must run on GPUs

- Time-critical kernel is the stencil application
- Also require BLAS level-1 type operations

```
while (|r_k| > ε) {
    β_k = (r_k, r_k)/(r_{k-1}, r_{k-1})
    p_{k+1} = r_k - β_k p_k
    q_{k+1} = A p_{k+1}
    α = (r_k, r_k)/(p_{k+1}, q_{k+1})
    r_{k+1} = r_k - α q_{k+1}
    x_{k+1} = x_k + α p_{k+1}
    k = k+1
}
```

conjugate gradient

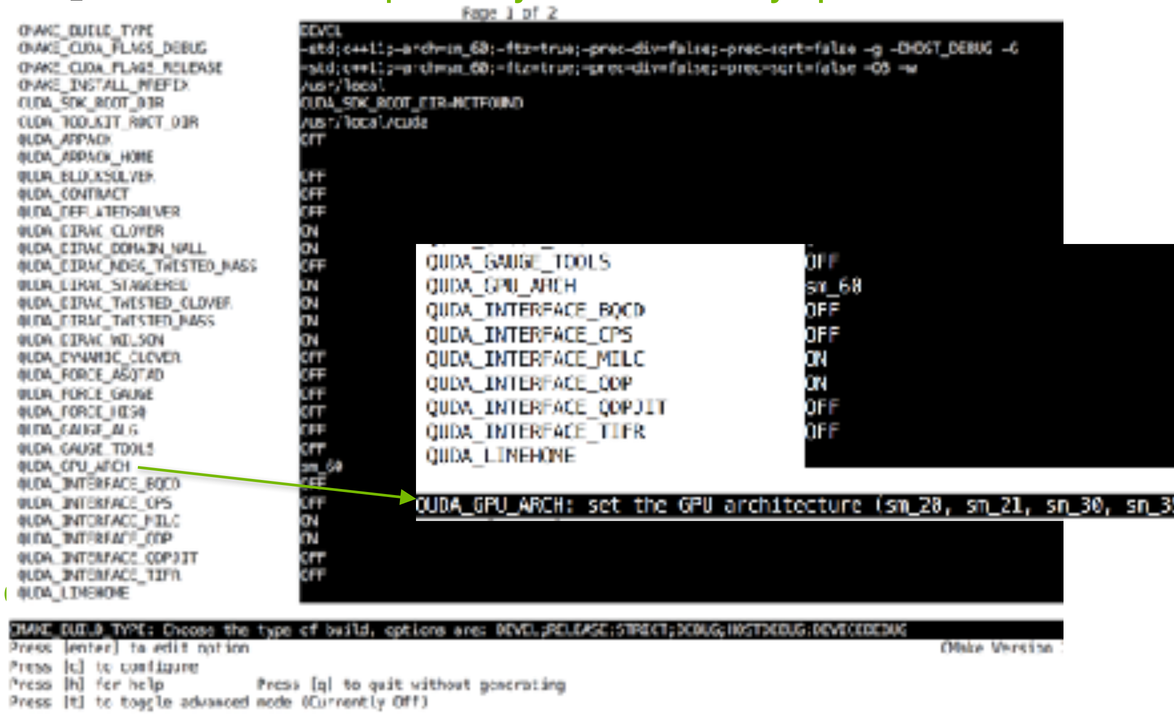
THE MANY WAYS TO QUDA

BUILDING QUDA

using CMAKE

```
git clone https://github.com/lattice/quda.git   // clones repository into directory quda
mkdir build; cd build   // use a build directory
cmake ../quda      // setup build system
ccmake     // interface to set the options
           // shows a description of options
           // may also be passed on command line
           // cmake GUI for exploring options

make -j16  // you can also use other build systems,
```



NVIDIA.

DOCUMENTATION

https://github.com/lattice/quda/wiki

Compilation guide

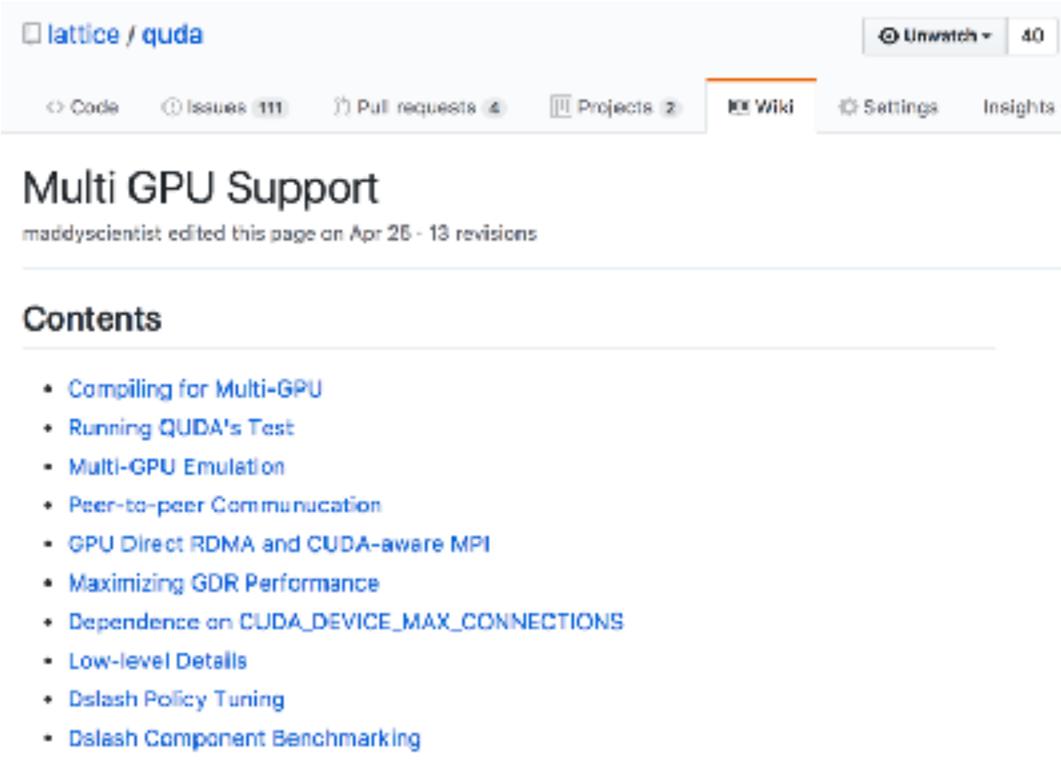
How to contribute to QUDA

C and Fortran Interface Guide

Maximizing multi-GPU performance

Parameter tuning for deflation and multigrid

More coming soon...



CHROMA

https://jeffersonlab.github.io/chroma

QUDA has close integration with the USQCD-developed Chroma package

- Developed primarily at Jlab

QUDA provides GPU acceleration for HMC and analysis

- Wilson, Wilson-clover solvers
  - BiCGStab, CG, CGNR, CGNE, multi-shift CG, GCR-MG, GCR-DD
- Mobius solvers
  - CG, CGNR, CGNE

Chroma + QUDA comes in two flavours

CHROMA

Traditional approach (QDP++)

GPUs (using QUDA) used only as a linear solver accelerator

Everything non-solver remains on the CPU

- Chroma QDP++ expressions are run with usual OpenMP CPU parallelization

Ideal for analysis that is solver bound

https://github.com/JeffersonLab/qdp-jit/commits/llvm-nvptx

CHROMA

Offload Approach (QDP-JIT)

arXiv/1408.5925 Winter *et al*

Just-in-time compilation of QDP++ expressions into GPU kernels

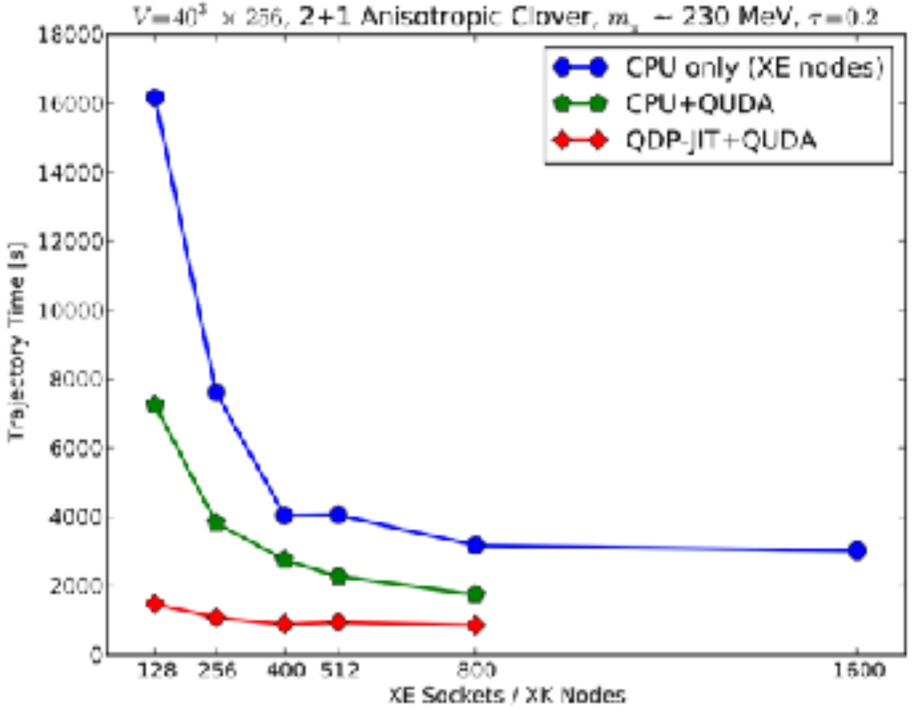
- All of Chroma runs on the GPU
- CPU used for I/O and task marshaling only

QUDA still used to accelerate the linear solver

- No data movement between CPU and GPU, all data remains on the GPU

GPU memory *can* be limiting factor

Ideal for gauge generation where we want to strong scale (less memory footprint per node)



CHROMA

Running with QUDA

All relevant QUDA solver parameters exposed to Chroma's XML input

E.g., Multi-shift CG solver for clover fermions

```xml
<InvertParam>
  <invType>MULTI_CG_QUDA_CLOVER_INVERTER</invType>
  <CloverParams>
    <Mass>-0.2450</Mass>
    <clovCoeff>1.24930970916466</clovCoeff>
    <AnisoParam>
      <anisoP>false</anisoP>
      <t_dir>3</t_dir>
      <xi_0>1</xi_0>
      <nu>1</nu>
    </AnisoParam>
  </CloverParams>
  <RsdTarget>1e-10 1e-10 1e-10 1e-10 1e-10 1e-10 1e-10 1e-10 1e-10 1e-10 1e-10 1e-10 1e-10 1e-10 1e-10</RsdTarget>
  <Delta>1.0e-14</Delta>
  <MaxIter>50000</MaxIter>
  <RsdToleranceFactor>100</RsdToleranceFactor>
  <AntiPeriodicT>true</AntiPeriodicT>
  <SolverType>CG</SolverType>
  <Verbose>false</Verbose>
  <CheckShifts>true</CheckShifts>
  <AsymmetricLinop>false</AsymmetricLinop>
  <CudaReconstruct>RECONS_12</CudaReconstruct>
  <CudaSloppyPrecision>DOUBLE</CudaSloppyPrecision>
  <CudaSloppyReconstruct>RECONS_12</CudaSloppyReconstruct>
  <AxialGaugeFix>false</AxialGaugeFix>
  <AutotuneDslash>true</AutotuneDslash>
</InvertParam>
```

MILC

https://github.com/milc-qcd/milc_qcd

MILC is a C-based application commonly used for HISQ fermion simulations

MILC includes built in support for QUDA

- HISQ CG and multi-shift CG solvers
- HISQ and gauge forces
- HISQ link construction
- Clover CG and multi-shift CG solvers

QUDA can be used for both gauge generation and analysis acceleration

MILC

Building and Installing

Edit Makefile as needed or set environment variables

Need to point to QUDA, CUDA and enable which routines to offload

Example: compile MILC with full QUDA HISQ support (ks_imp_rhmc/compile_su3_rhmd_hisq_quda.sh)

```
#!/bin/bash

CC=mpicc \
CXX=mpicxx \
CUDA_HOME=/usr/local/cuda-9.2 \
QUDA_HOME=/scratch/kate/quda-develop \
WANTQUDA=true \
WANT_FN_CG_GPU=true \
WANT_FL_GPU=true \
WANT_GF_GPU=true \
WANT_FF_GPU=true \
PRECISION=2 \
MPP=true \
OMP=true \
make -j 1 su3_rhmd_hisq
```

More instructions found here: https://github.com/lattice/quda/wiki/MILC-with-QUDA

TRY YOURSELF

Running NERSC benchmarks

Once QUDA support has been enabled, nothing needs to be done

- Parameters are set in the QUDA interface to match those of MILC running on CPU
- Some QUDA-specific parameters can be set using environment variables
- https://github.com/lattice/quda/wiki/QUDA-Environment-Variables

Detailed instructions

- https://github.com/lattice/quda/wiki/Running-the-NERSC-MILC-Benchmarks

EXERCISE: Run NERSC small (1/2 GPUs) or NERSC Medium (2+ GPUs)

TRY YOURSELF

Running NERSC benchmarks

Detailed instructions

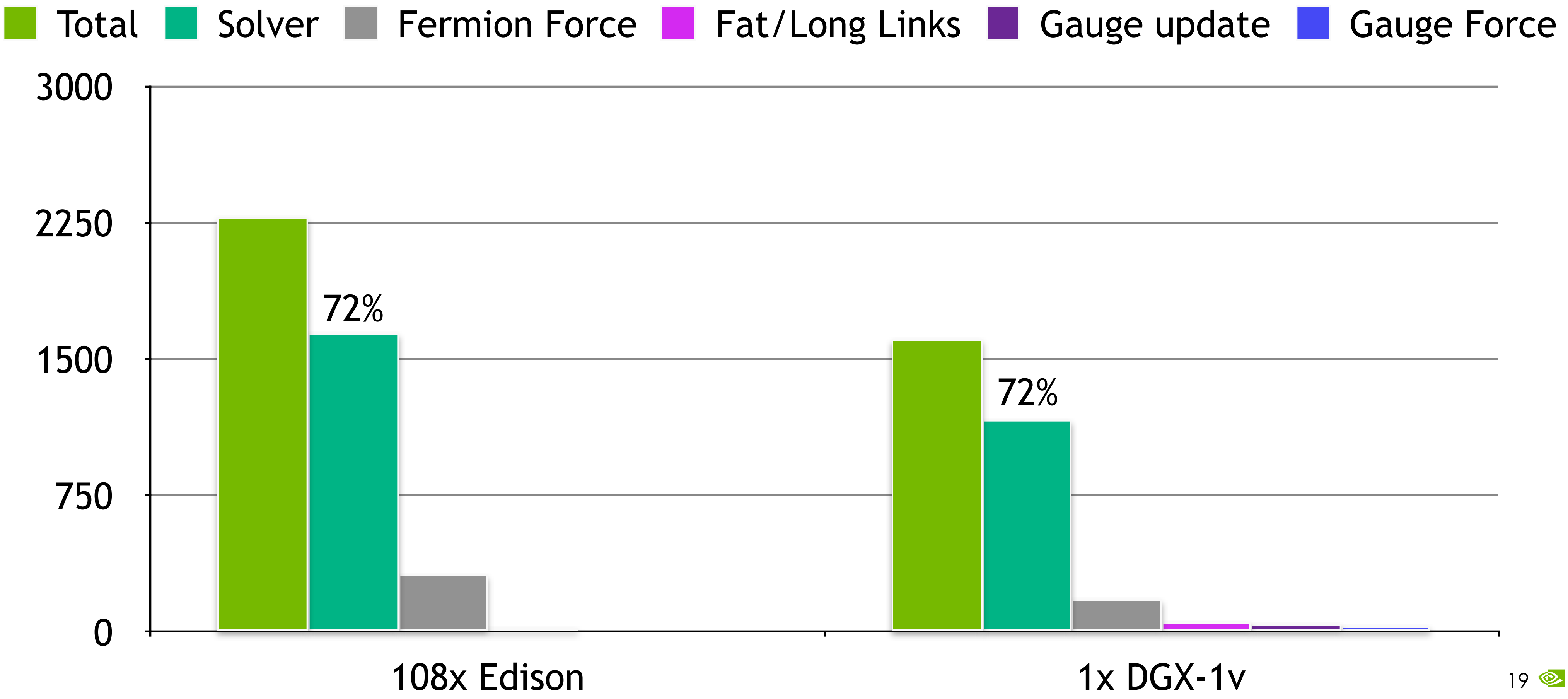
https://github.com/lattice/quda/wiki/Running-the-NERSC-MILC-Benchmarks

EXERCISE: Run NERSC small (1/2 GPUs) or NERSC Medium (2+ GPUs)

```
export PATH_TO_CUDA=/nvme/h/buildsets/eb_cyclone_rl/software/CUDA/11.8.0/
export PATH_TO_QUDA=/nvme/h/lap24mw1/qudabuild118/usqcd/
export PATH_TO_QIO=/nvme/h/lap24mw1/qudabuild118/usqcd/
export PATH_TO_QMP=/nvme/h/lap24mw1/qudabuild118/usqcd/
export PATH_TO_NVHPCSDK=""
```

MILC RHMD ON GPUS

NERSC Medium benchmark (RHMD on $V=36^3x72$)



NATIVE QUDA

QUDA includes a variety of internal test applications

```
heatbath_test               # pure gauge evolution
invert_test                 # linear solver test
multigrid_invert_test       # multigrid solver
multigrid_evolve_test       # pure gauge evolution with MG evolution
etc.
```

These have been hacked evolved directly into actual physics applications

E.g, `https://github.com/ETMC-QUDA/quda-QKXTM-Multigrid`

NATIVE QUDA

Working directly with QUDA is ideal for algorithm development

Perfect the method before interfacing with external application

Advantage: Full exposure to the algorithm parameters that control QUDA

Disadvantage: Full exposure to the algorithm parameters that control QUDA

```
typedef struct QudaMultigridParam_s {

  QudaInvertParam *invert_param;

  /** Number of multigrid levels */
  int n_level;

  /** Geometric block sizes to use on each level */
  int geo_block_size[QUDA_MAX_MG_LEVEL][QUDA_MAX_DIM];

  /** Spin block sizes to use on each level */
  int spin_block_size[QUDA_MAX_MG_LEVEL];

  /** Number of null-space vectors to use on each level */
  int n_vec[QUDA_MAX_MG_LEVEL];

  /** Precision to store the null-space vectors in (post block orthogonalization) */
  QudaPrecision precision_null[QUDA_MAX_MG_LEVEL];

  /** Verbosity on each level of the multigrid */
  QudaVerbosity verbosity[QUDA_MAX_MG_LEVEL];

  /** Inverter to use in the setup phase */
  QudaInverterType setup_inv_type[QUDA_MAX_MG_LEVEL];

  /** Number of setup iterations */
  int num_setup_iter[QUDA_MAX_MG_LEVEL];

  /** Tolerance to use in the setup phase */
  double setup_tol[QUDA_MAX_MG_LEVEL];

  /** Maximum number of iterations for each setup solver */
  int setup_maxiter[QUDA_MAX_MG_LEVEL];

  /** Maximum number of iterations for refreshing the null-space vectors */
  int setup_maxiter_refresh[QUDA_MAX_MG_LEVEL];

  /** Null-space type to use in the setup phase */
  QudaSetupType setup_type;

  /** Pre orthonormalize vectors in the setup phase */
  QudaBoolean pre_orthonormalize;

  /** Post orthonormalize vectors in the setup phase */
  QudaBoolean post_orthonormalize;

  /** The solver that wraps around the coarse grid correction and smoother */
  QudaInverterType coarse_solver[QUDA_MAX_MG_LEVEL];

  /** Tolerance for the solver that wraps around the coarse grid correction and smoother */
  double coarse_solver_tol[QUDA_MAX_MG_LEVEL];

  /** Tolerance for the solver that wraps around the coarse grid correction and smoother */
  double coarse_solver_maxiter[QUDA_MAX_MG_LEVEL];

  /** Smoother to use on each level */
  QudaInverterType smoother[QUDA_MAX_MG_LEVEL];

  /** Tolerance to use for the smoother / solver on each level */
  double smoother_tol[QUDA_MAX_MG_LEVEL];

  /** Number of pre-smoother applications on each level */
  int nu_pre[QUDA_MAX_MG_LEVEL];

  /** Number of post-smoother applications on each level */
  int nu_post[QUDA_MAX_MG_LEVEL];

  /** Over/under relaxation factor for the smoother at each level */
  double omega[QUDA_MAX_MG_LEVEL];

  /** Precision to use for halo communication in the smoother */
  QudaPrecision smoother_halo_precision[QUDA_MAX_MG_LEVEL];

  /** Whether to use additive or multiplicative Schwarz preconditioning in the smoother */
  QudaSchwarzType smoother_schwarz_type[QUDA_MAX_MG_LEVEL];

  /** Number of Schwarz cycles to apply */
  int smoother_schwarz_cycle[QUDA_MAX_MG_LEVEL];

  /** The type of residual to send to the next coarse grid, and thus the
      type of solution to receive back from this coarse grid */
  QudaSolutionType coarse_grid_solution_type[QUDA_MAX_MG_LEVEL];

  /** The type of smoother solve to do on each grid (e/o preconditioning or not)*/
  QudaSolveType smoother_solve_type[QUDA_MAX_MG_LEVEL];

  /** The type of multigrid cycle to perform at each level */
  QudaMultigridCycleType cycle_type[QUDA_MAX_MG_LEVEL];

  /** Whether to use global reductions or not for the smoother / solver at each level */
  QudaBoolean global_reduction[QUDA_MAX_MG_LEVEL];

  /** Location where each level should be done */
  QudaFieldLocation location[QUDA_MAX_MG_LEVEL];

  /** Location where the coarse-operator construction will be computedn */
  QudaFieldLocation setup_location[QUDA_MAX_MG_LEVEL];

  /** Minimize device memory allocations during the adaptive setup,
      placing temporary fields in mapped memory instad of device
      memory */
  QudaBoolean setup_minimize_memory;

  /** Whether to compute the null vectors or reload them */
  QudaComputeNullVector compute_null_vector;

  /** Whether to generate on all levels or just on level 0 */
  QudaBoolean generate_all_levels;

  /** Whether to run the verification checks once set up is complete */
  QudaBoolean run_verify;

  /** Filename prefix where to load the null-space vectors */
  char vec_infile[256];

  /** Filename prefix for where to save the null-space vectors */
  char vec_outfile[256];

  /** The Gflops rate of the multigrid solver setup */
  double gflops;

  /**< The time taken by the multigrid solver setup */
  double secs;

  /** Multiplicative factor for the mu parameter */
  double mu_factor[QUDA_MAX_MG_LEVEL];

} QudaMultigridParam;
```

OTHER WAYS TO QUDA

QUDA hooks also built into

- BQCD (Wilson and Wilson-clover CG solver)
- tmLQCD (Twisted-mass and twisted-clover CG and multigrid solvers)
- CPS (Wilson, Domain-wall and Möbius solvers)
- FUEL (staggered CG solver)

Always looking to extend the range and applicability of QUDA

QUDA 2.0

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

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

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

```
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

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

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

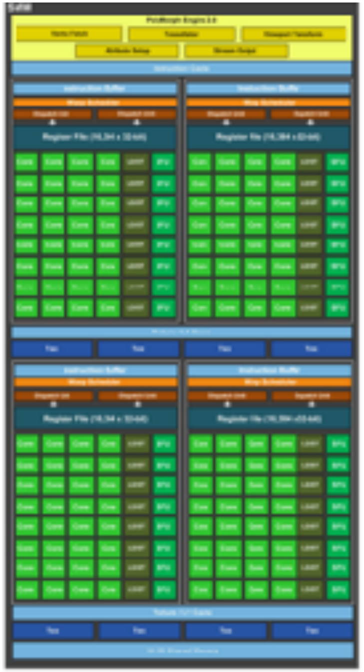
```
include/targets/cuda/aos.h // optimized for CUDA
include/targets/generic/aos.h // works on all platforms
```

AUTOTUNING

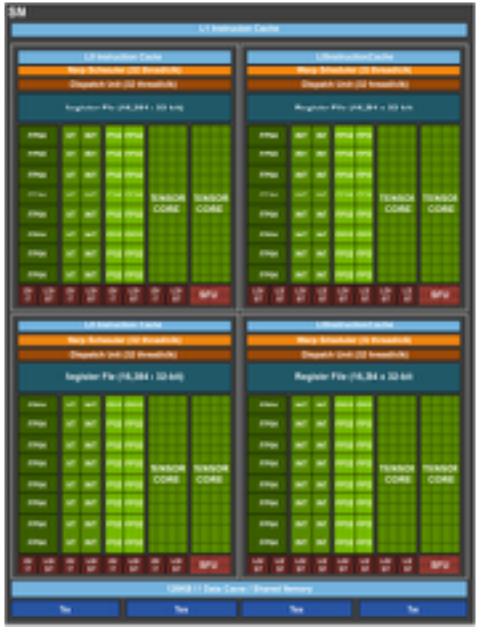
WHY AUTOTUNING MATTERS

Autotuning provides performance portability

Kepler SM (2012)
192 cores, 48 KiB L1



Maxwell SM (2014)
128 cores, 48 KiB L1



Volta SM (2017)
64 cores, 128 KiB L1

QUDA'S AUTOTUNER

QUDA includes an autotuner for ensuring optimal kernel performance

- virtual C++ class "`Tunable`" that is derived for each kernel you want to autotune

By default `Tunable` classes will autotune 1-d CTA size, shared memory size, grid size

- Derived specializations do 2-d and 3-d CTA tuning

Tuned parameters are stored in a `std::map` and dumped to disk for later reuse

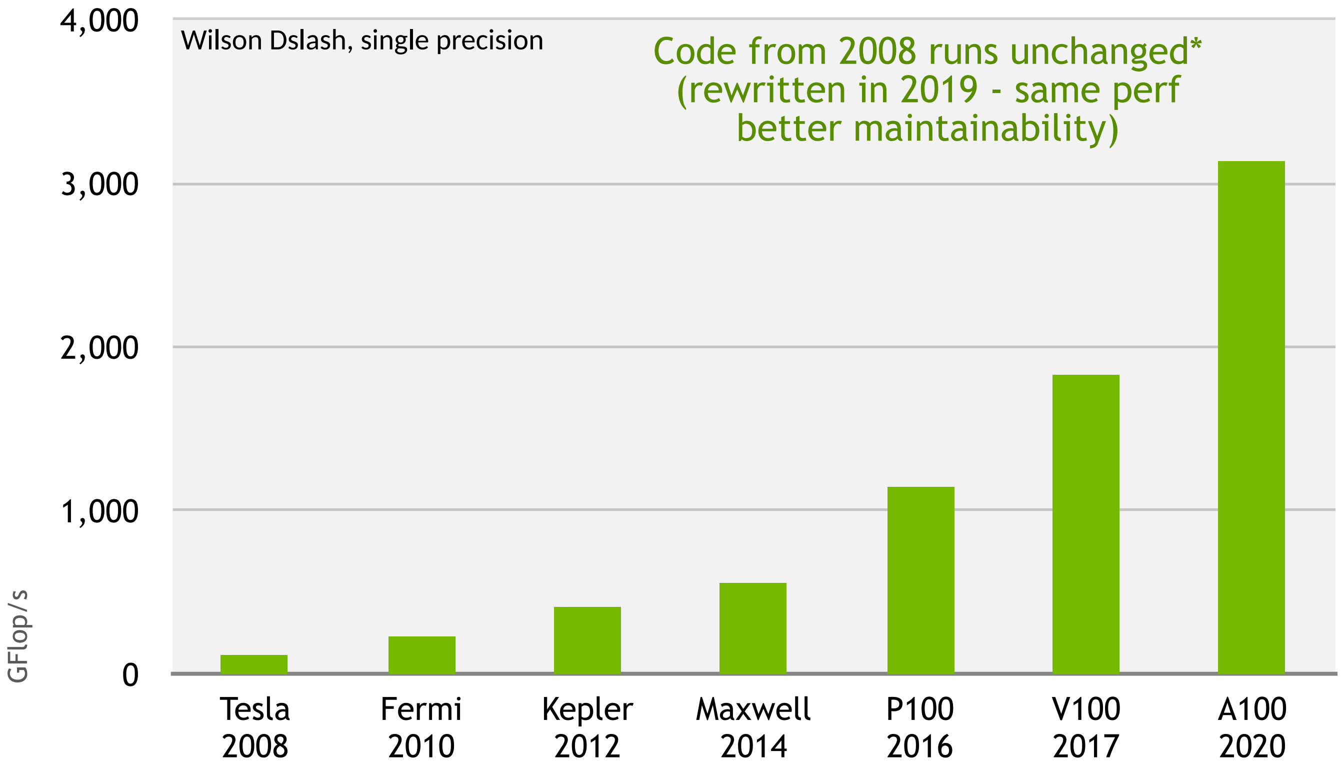
Built in performance metrics and profiling

User just needs to

- State resource requirements: shared memory per thread and/or per CTA, total number of threads
- Specify a tuneKey which gives each kernel a unique entry and break any degeneracy

SINGLE GPU PERFORMANCE

Wilson Dslash Kernel



SAVE/RESTORE

Autotuner serializes tunecache to disk and can be restored for subsequent runs

```
tunecache     0.9.0     v0.9.0a1-with_v.0.8_milc_interface-136-g2eaa400-sm_70 cpu_arch=x86_64,gpu_arch=sm_70,cuda_version=9010  # Last updated Sat Mar 24 13:26:19 2018

          volume   name aux  block.x   block.y   block.z   grid.x    grid.y    grid.z    shared_bytes  aux.x     aux.y     aux.z     aux.w     time comment
     12x24x24x24   N4quda14PackFaceWilsonI6short4fEE    vol=165888,stride=165888,precision=2,Ns=4,Nc=3,comm=0011,kernelPackT,nFace=1,location=     96   1    1    288  1    1    0    1    1    1    1    5.80267e-06   # 57.18 Gflop/s, 381.18 GB/s, 
     12x24x24x24   N4quda14PackFaceWilsonI6short4fEE    vol=165888,stride=165888,precision=2,Ns=4,Nc=3,comm=0011,nFace=1,location=  64   1    1    216  1    1    1537 1    1    1    1    5.12e-06 # 32.40 Gflop/s, 216.00 GB/s, tuned Sat Mar 24 1
     12x24x24x24   N4quda15CopyColorSpinorIdsLi4ELi3ENS_18CopyColorSpinorArgINS_11colorspinor21SpaceSpinorColorOrderIdLi4ELi3EEENS2_11FloatNOrderIsLi4ELi3ELi4ELb0EEEEEEE    out_stride=165888,in_stride=165888,RelBasis  672  1    1    247  1
 2018
     12x24x24x24   N4quda15CopyColorSpinorIsdLi4ELi3ENS_18CopyColorSpinorArgINS_11colorspinor11FloatNOrderIsLi4ELi3ELi4ELb0EEENS2_21SpaceSpinorColorOrderIdLi4ELi3EEEEEEE    out_stride=165888,in_stride=165888,NonRelBasis    128  1    1    129
 2018
     12x24x24x24   N4quda16WilsonDslashCudaI6short4S1_EE    policy,comm=0011,reconstruct=12,topo=1122     32   1    1    1    1    1    0    1    1    0    0    0.00012639    # 1732.51 Gflop/s, 803.25 GB/s, tuned Sat Mar 24 13:26:16 2018
     12x24x24x24   N4quda16WilsonDslashCudaI6short4S1_EE    type=exterior_all,comm=0011,reconstruct=12    64   1    1    432  1    1    1586 1    1    1    1    1.2288e-05    # 351.00 Gflop/s, 234.00 GB/s, tuned Sat Mar 24 13:26:16 2018
     12x24x24x24   N4quda16WilsonDslashCudaI6short4S1_EE    type=exterior_all,comm=0011,reconstruct=12,zero_copy  768  1    1    36   1    1    24577     1    1    1    1    7.68e-05 # 56.16 Gflop/s, 37.44 GB/s, tuned Sat Mar 24 13:26:16 2018
     12x24x24x24   N4quda16WilsonDslashCudaI6short4S1_EE    type=exterior_t,comm=0011,reconstruct=12 32   1    1    432  1    1    2587 1    1    1    1    8.192e-06     # 263.25 Gflop/s, 175.50 GB/s, tuned Sat Mar 24 13:26:16 2018
     12x24x24x24   N4quda16WilsonDslashCudaI6short4S1_EE    type=exterior_t,comm=0011,reconstruct=12,zero_copy    32   1    1    432  1    1    24577     1    1    1    1    4.3008e-05    # 50.14 Gflop/s, 33.43 GB/s, tuned Sat Mar 24 13:26:16 20
     12x24x24x24   N4quda16WilsonDslashCudaI6short4S1_EE    type=exterior_z,comm=0011,reconstruct=12 128  1    1    108  1    1    4097 1    1    1    1    7.168e-06     # 300.86 Gflop/s, 200.57 GB/s, tuned Sat Mar 24 13:26:16 2018
     12x24x24x24   N4quda16WilsonDslashCudaI6short4S1_EE    type=exterior_z,comm=0011,reconstruct=12,zero_copy    384  1    1    36   1    1    24577     1    1    1    1    4.1984e-05    # 51.37 Gflop/s, 34.24 GB/s, tuned Sat Mar 24 13:26:16 20
     12x24x24x24   N4quda16WilsonDslashCudaI6short4S1_EE    type=interior,comm=0011,ghost=0011,reconstruct=12 736  1    1    226  1    1    24577     1    1    1    1    8.2944e-05    # 2588.00 Gflop/s, 1189.33 GB/s, tuned Sat Mar 24 13:26:16 201
     12x24x24x24   N4quda4blas5Norm2Id6float26float4EE vol=165888,stride=165888,precision=2,Ns=4,Nc=3    576  1    1    133  1    1    0    1    1    1    1    2.21867e-05   # 358.89 Gflop/s, 388.80 GB/s, tuned Sat Mar 24 13:26:15 2018
     24x24x24x24   N4quda12ExtractGhostIsLi18ELi4ENS_5gauge11FloatNOrderIsLi18ELi4ELi12EL20QudaStaggeredPhase_s0ELb1EEEEE  stride=172800 96   1    1    144  1    1    0    1    1    1    1    2.4576e-05    # 0.00 Gflop/s, 108.00 GB/s, tuned Sat 
     24x24x24x24   N4quda9CopyGaugeIsdLi18ENS_5gauge11FloatNOrderIsLi18ELi4ELi12EL20QudaStaggeredPhase_s0ELb1EEENS1_8QDPOrderIdLi18EEELb0EEE out_stride=172800,in_stride=165888,geometry=4      64   1    1    2592 4    1    0    1    1    1    1
     24x24x24x24   N4quda9CopyGaugeIssLi18ENS_5gauge11FloatNOrderIsLi18ELi4ELi12EL20QudaStaggeredPhase_s0ELb1EEES4_Lb1EEE  out_stride=172800,in_stride=172800,geometry=4     96   1    1    72   4    1    0    1    1    1    1    1.4336e-05    # 0
   bytes=1327104   cudaMemcpyDeviceToDevice    exchangeGhost,cuda_gauge_field.cu,243     32   1    1    1    1    1    0    1    1    1    1    2.56e-05 # 0.00 Gflop/s, 103.68 GB/s, tuned Sat Mar 24 13:26:15 2018
  bytes=31850496   cudaMemcpyDeviceToHost saveSpinorField,cuda_color_spinor_field.cu,630     32   1    1    1    1    1    0    1    1    0    0    0.010999 # 0.00 Gflop/s, 2.90 GB/s, tuned Sat Mar 24 13:26:16 2018
  bytes=31850496   cudaMemcpyHostToDevice loadSpinorField,cuda_color_spinor_field.cu,586     32   1    1    1    1    1    0    1    1    1    1    0.00445088    # 0.00 Gflop/s, 7.16 GB/s, tuned Sat Mar 24 13:26:15 2018
  bytes=47775744   cudaMemcpyHostToDevice copy,cuda_gauge_field.cu,640    32   1    1    1    1    1    0    1    1    1    1    0.00701014    # 0.00 Gflop/s, 6.82 GB/s, tuned Sat Mar 24 13:26:15 2018
```

Autotuner also dumps a profile to disk when application ends

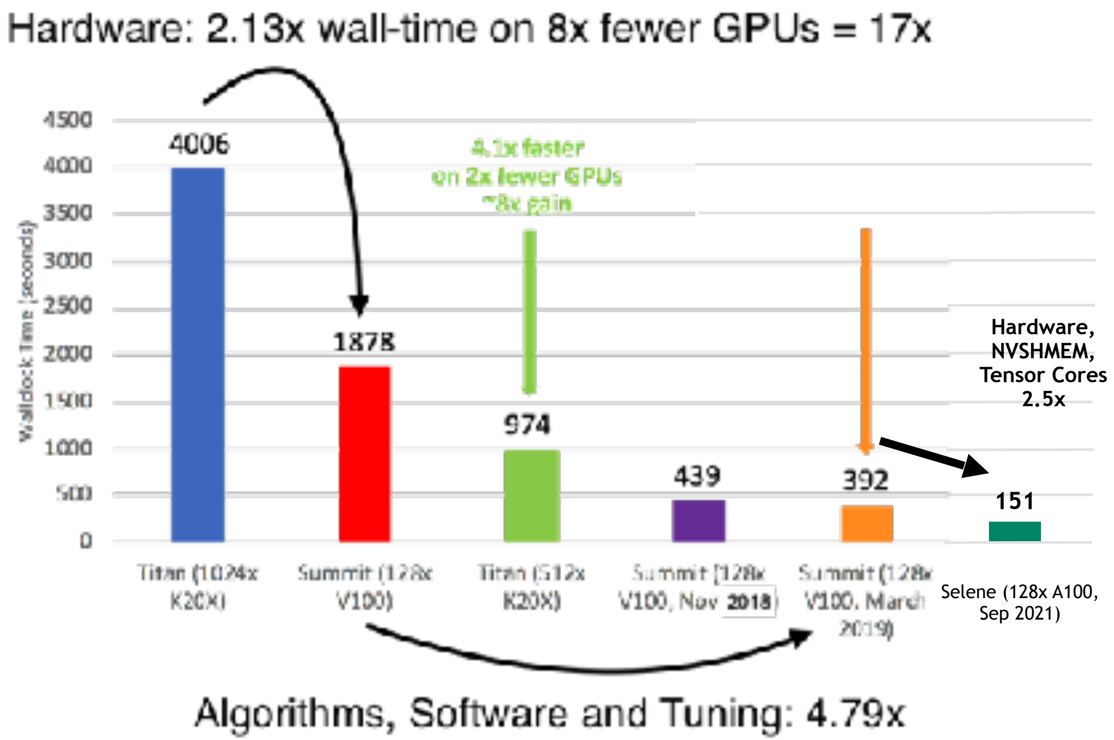
```
profile  0.9.0    v0.8.0-1644-g7684193-sm_60    cpu_arch=x86_64,gpu_arch=sm_60,cuda_version=8000# Last updated Fri Apr 28 13:07:27 2017

  total time   percentage        calls  time / call          volume   name   aux   comment
    0.227616      43.4696         1000  0.000227616     24x24x24x24   N4quda7LaplaceIfLi4ELi3ENS_10LaplaceArgIfLi3EL21QudaReconstructType_s12ELb1EEEEE  vol=331776,stride=172800,precision=4,comm=0000,xpay# 848.33 Gflop/s, 909.55 GB/s
     0.10025      19.1456          918  0.000109205     24x24x24x24   N4quda4blas9axpyZpbx_I7double2S2_EE  vol=331776,stride=172800,precision=4,vol=331776,stride=172800,precision=8# 72.91 Gflop/s, 510.40 GB/s
   0.0555307      10.6051         1000  5.55307e-05     24x24x24x24   N4quda4blas11axpyCGNorm2I7double26float2S3_EE  vol=331776,stride=172800,precision=4# 224.05 Gflop/s, 448.10 GB/s
    0.039441      7.53236           84  0.000469536     24x24x24x24   N4quda7LaplaceIdLi4ELi3ENS_10LaplaceArgIdLi3EL21QudaReconstructType_s12ELb1EEEEE  vol=331776,stride=172800,precision=8,comm=0000,xpay# 411.24 Gflop/s, 881.84 GB/s
    0.037984       7.2541         1000   3.7984e-05     24x24x24x24   N4quda4blas3DotId6float2S2_EE  vol=331776,stride=172800,precision=4# 109.18 Gflop/s, 436.73 GB/s
```

FULL STACK CONTINOUS IMPROVEMENT

CHROMA WILSON-CLOVER HMC

- Dominated by QUDA Multigrid
- Few solves per gauge configuration, can be bound by “heavy” (well-conditioned) solves
  - Evolve and refresh coarse space as the gauge field evolves
- Mixed precision an important piece of the puzzle
  - Double – outer solve precision
  - Single – GCR preconditioner
  - Half – Coarse operator precision
  - Int32 – deterministic parallel coarsening
- Wilson-clover MG implemented in **QUDA**, hooked into **Chroma** ; support in **Grid**
- **Latest and greatest:** Wilson-clover NVSHMEM plus tensor-core-accelerated setup



Chroma w/ QDP-JIT and QUDA, ECP FOM data, $V=64^3\times128$ sites, m_π ~172 MeV, (QDP-JIT by F. Winter, Jefferson Lab)

Original figure credit Balint Joo

NVIDIA

CONTRIBUTING

CONTRIBUTING

BSD license

permissive license, do whatever you want to with the code

better if you contribute back (pull request @ http://lattice.github.com/quda)

makes your contribution visible

let's other build on your work

get feedback on your changes

Get in touch: https://quda.slack.com

ask what is already there, what you can use, …

MODERN SOFTWARE ENGINEERING

gitflow and continuous integration

Use branches for new features, hot fixes, current development head and released (master) version

Require pull request for new contributions

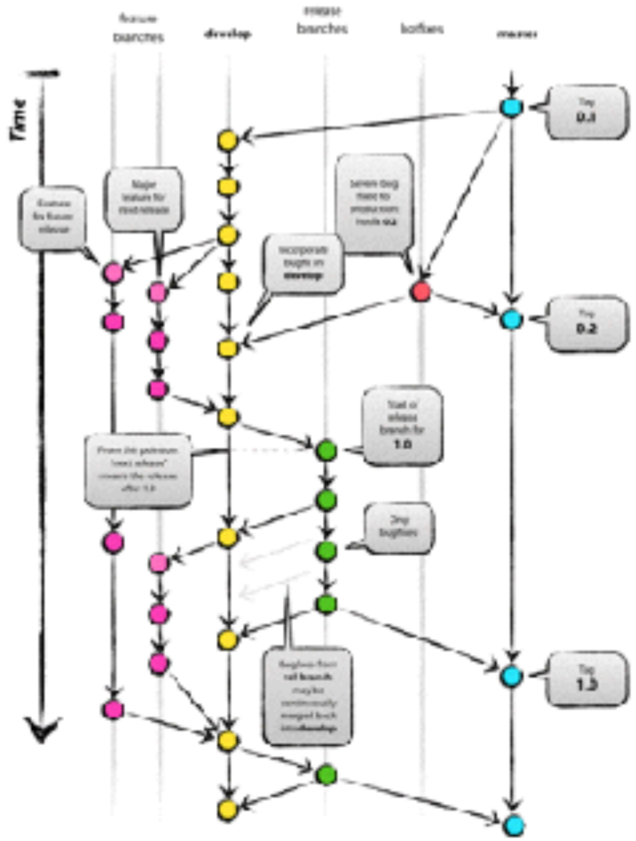
→ peer review of code required before merging

→ automated build testing (Jenkins)

→ automated tests (Jenkins)

Give feedback to new contributors

Prevents fragmentation



Source: http://nvie.com/posts/a-successful-git-branching-model/

MIXED PRECISION

LINEAR SOLVERS

QUDA supports a wide range of linear solvers

- CG, BiCGstab, GCR, Multi-shift solvers, etc.

Condition number inversely proportional to mass

- Light (realistic) masses are highly singular
- Naive Krylov solvers suffer from critical slowing down at decreasing mass

Entire solver algorithm must run on GPUs

- Time-critical kernel is the stencil application
- Also require BLAS level-1 type operations

```
while (|r_k| > ε) {
    β_k = (r_k, r_k)/(r_{k-1}, r_{k-1})
    p_{k+1} = r_k - β_k p_k
    q_{k+1} = A p_{k+1}
    α = (r_k, r_k)/(p_{k+1}, q_{k+1})
    r_{k+1} = r_k - α q_{k+1}
    x_{k+1} = x_k + α p_{k+1}
    k = k+1
}
```

conjugate gradient

MIXED-PRECISION CG

apply Dslash in sloppy precision (single, half)

reliable residual replacement in high precision

ensures double-precision accuracy of final result

half precision **storage**:

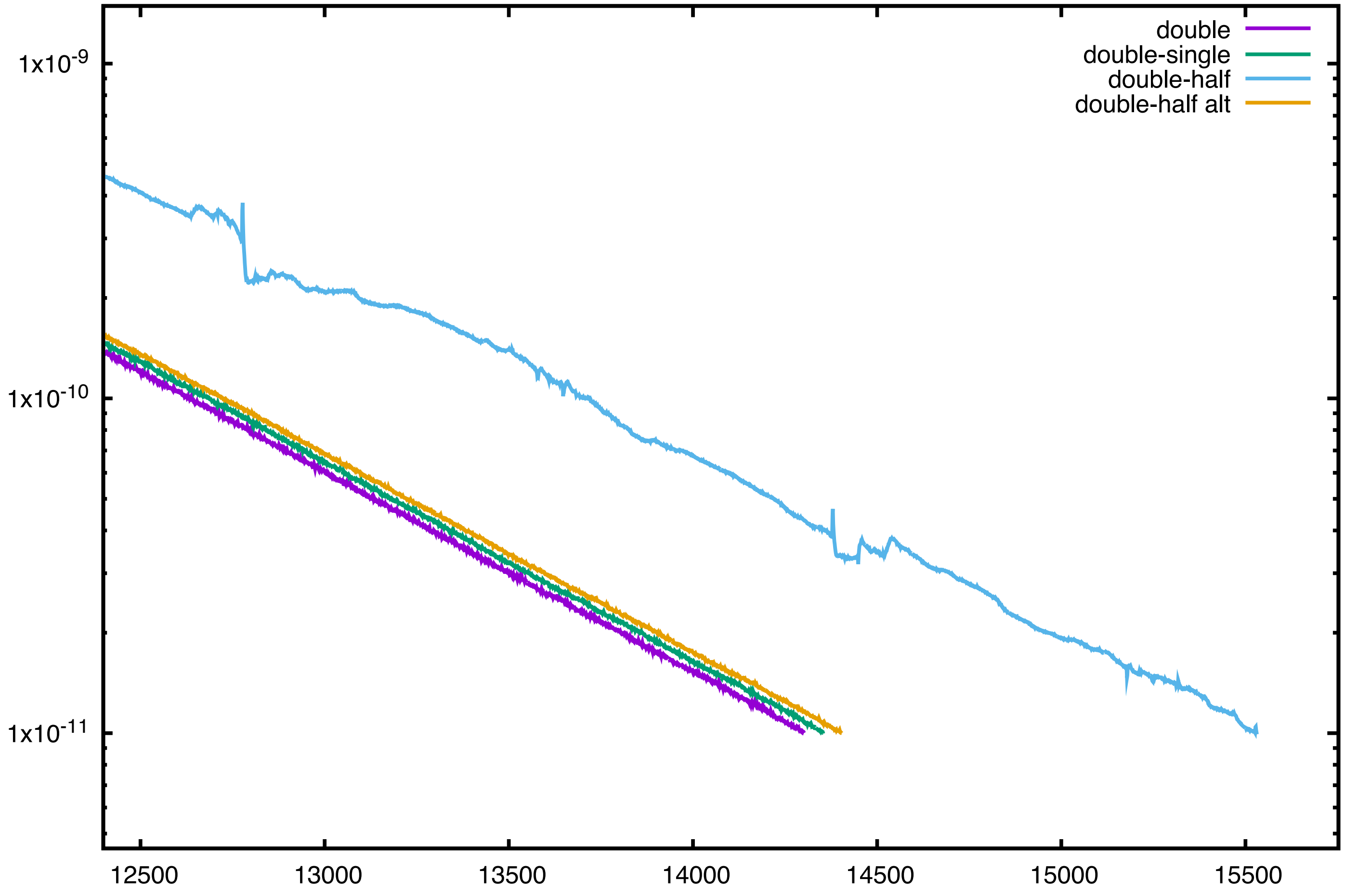
`Links: [-1,1] 16 bit fixed`

`Spinor:`

`16bit fixed (24 numbers)`

`float (exponent, 1)`

use fp32 for actual arithmetics



MIXED-PRECISION CG

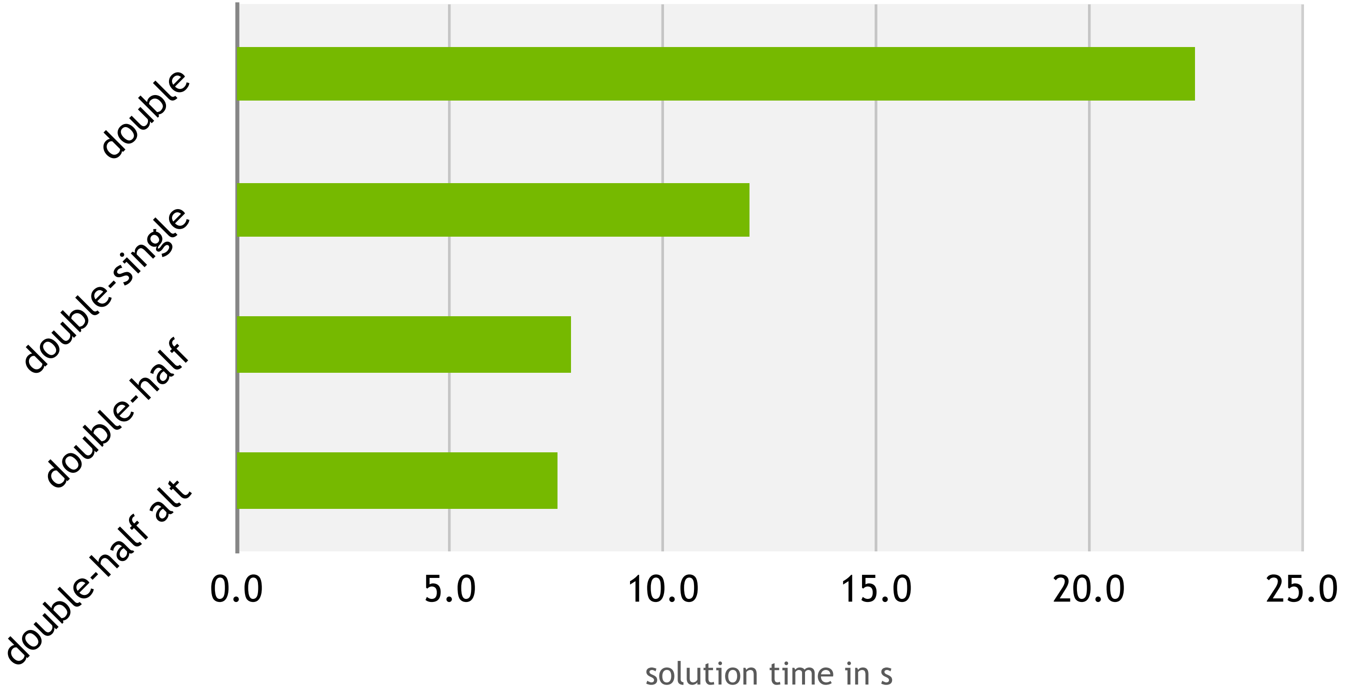
mixed precision:

apply Dslash in sloppy precision (single, half)

reliable residual replacement in high precision

ensures double-precision accuracy of final result

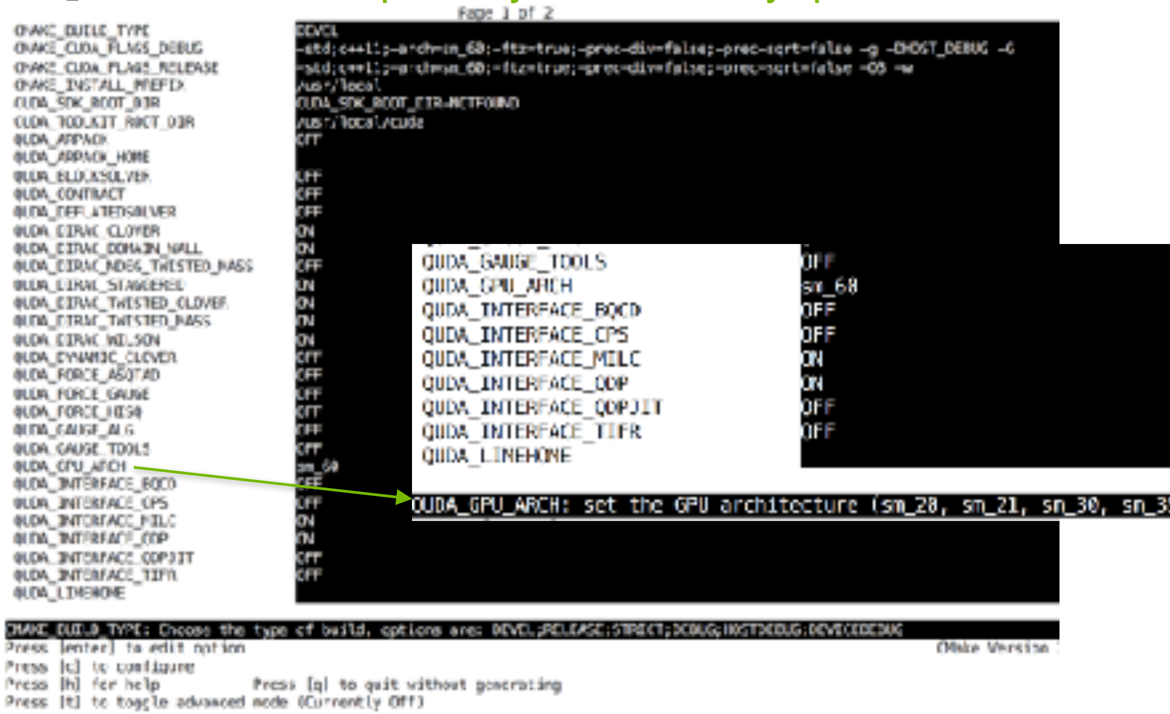
virtually identical iteration count



BUILDING QUDA

using CMAKE

```
git clone https://github.com/lattice/quda.git  // clones repository into directory quda
mkdir build; cd build  // use a build directory
cmake ../quda     // setup build system
ccmake    // interface to set the options
          // shows a description of options
          // may also be passed on command line
          // cmake GUI for exploring options
```

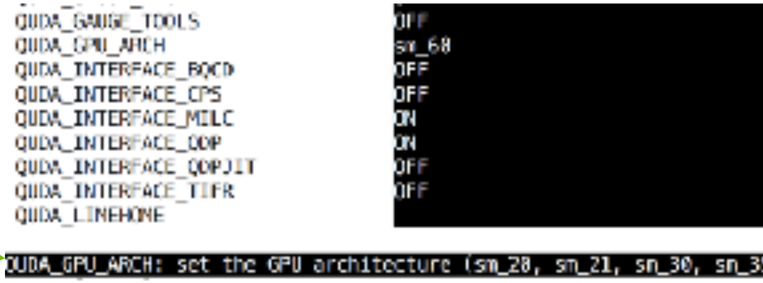


BUILDING QUDA

using CMAKE

// Enable/disable the following options in ccmake

CMAKE_BUILD_TYPE	RELEASE
QUDA_DIRAC_CLOVER	OFF
QUDA_DIRAC_CLOVER_HASENBUSCH	OFF
QUDA_DIRAC_DOMAIN_WALL	OFF
QUDA_DIRAC_NDEG_TWISTED_MASS	OFF
QUDA_DIRAC_STAGGERED	ON
QUDA_DIRAC_TWISTED_CLOVER	OFF
QUDA_DIRAC_TWISTED_MASS	OFF
QUDA_DIRAC_WILSON	ON
QUDA_DOWNLOAD_USQCD	ON
QUDA_GPU_ARCH	sm_80
QUDA_QIO	ON
QUDA_QMP	ON



`make -j16` // you can also use other build systems, e.g. ninja (recommended) by using `-GNinja`
`make install` // installs to <build_dir>/usqcd by default

TRY YOURSELF

in build/tests

`export QUDA_RESOURCE_PATH=.`	where tunecache is stored
`mpirun -np 2`	use 2 MPI rank
`./staggered_invert_test`	executable
`--load-gauge milc18181836.lat`	gauge file (in ~lap24mw1/gpu/quda)
`--dim 18 18 18 18  --gridsize 1 1 1 2`	local dimension and size of MPI grid
`--test 3 -`	test to run (CG)
`--mass 0.02`	mass
`--niter 2000 --tol 1e-10`	number of iteration and tolerance
`--prec double`	precision to use
`--prec-sloppy half`	sloppy prec for mixed precision solver (use double / half / single)

rerun to see tuned performance (first exec. does autotuning)

NVIDIA