TOWARDS QUDA 1.0

Kate Clark, Mathias Wagner and Evan Weinberg

April 27th 2019

PRESENTED BY

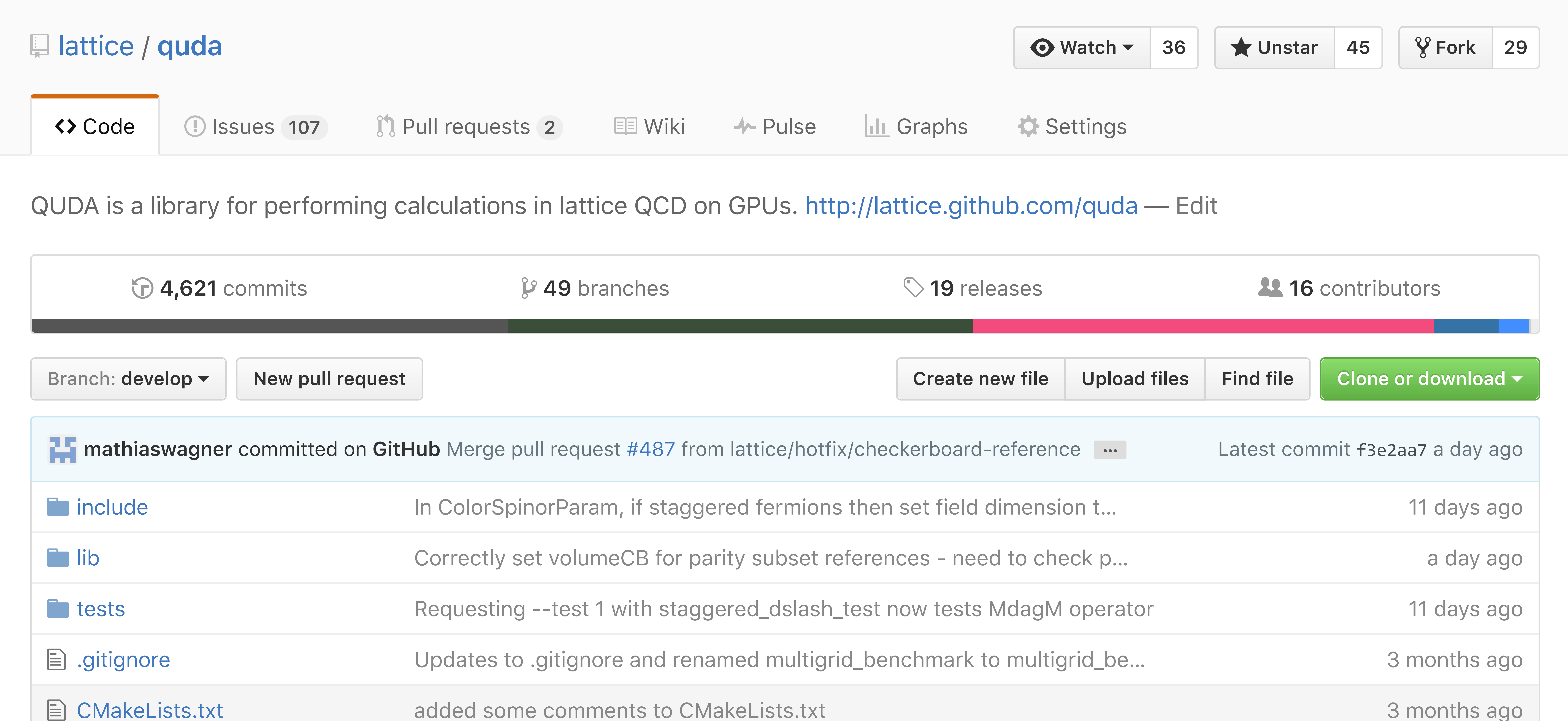
NVIDIA.

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, tmLQCD, etc.
- Provides:
  - Various solvers for all major fermionic discretizations, with multi-GPU support
  - Additional performance-critical routines needed for gauge-field generation
  - Pure gauge evolution, gauge fixing, smearing etc.
- Maximize performance
  - Exploit physical symmetries to minimize memory traffic
  - Mixed-precision methods
  - Autotuning for high performance on all CUDA-capable architectures
  - Domain-decomposed (Schwarz) preconditioners for strong scaling
  - Eigenvector and deflated solvers (Lanczos, EigCG, GMRES-DR)
  - Multi-source solvers
  - Multigrid solvers for optimal convergence
- A research tool for how to reach the exascale

QUDA - LATTICE QCD ON GPUS

http://lattice.github.com/quda



QUDA CONTRIBUTORS

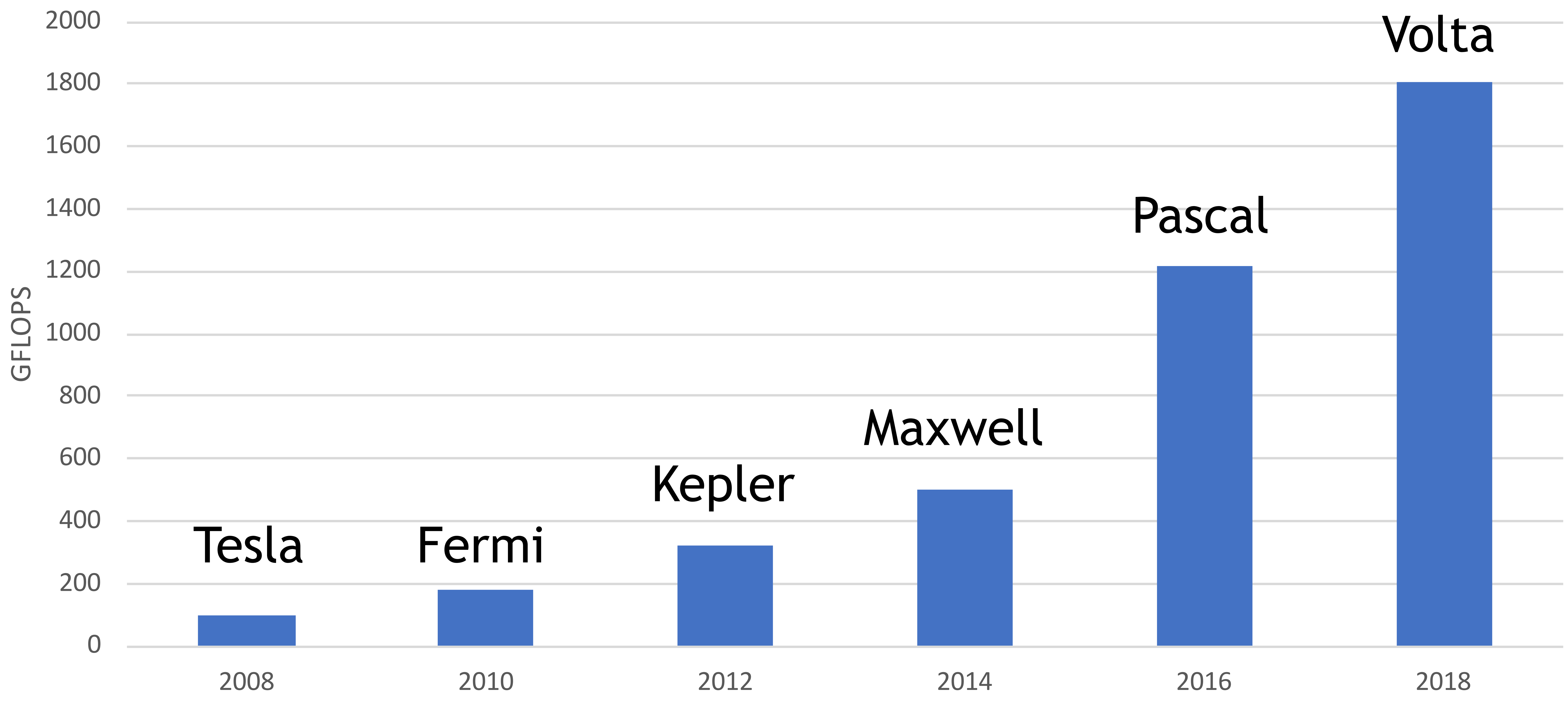
Ron Babich (NVIDIA)
Simone Bacchio (Cyprus)
Michael Baldhauf (Regensburg)
Kip Barros (LANL)
Rich Brower (Boston University)
Nuno Cardoso (NCSA)
Kate Clark (NVIDIA)
Michael Cheng (Boston University)
Carleton DeTar (Utah University)
Justin Foley (Utah -> NIH)
Joel Giedt (Rensselaer Polytechnic Institute)
Arjun Gambhir (William and Mary)
Steve Gottlieb (Indiana University)
Kyriakos Hadjiyiannakou (Cyprus)

Dean Howarth (BU)
Bálint Joó (Jlab)
Hyung-Jin Kim (BNL -> Samsung)
Bartek Kostrzewa (Bonn)
Claudio Rebbi (Boston University)
Hauke Sandmeyer (Bielefeld)
Guochun Shi (NCSA -> Google)
Mario Schröck (INFN)
Alexei Strelchenko (FNAL)
Jiqun Tu (Columbia)
Alejandro Vaquero (INFN)
Mathias Wagner (NVIDIA)
Evan Weinberg (BU -> NVIDIA)
Frank Winter (Jlab)

DSLASH REWRITE

WILSON-DSLASH PERFORMANCE

Same code works over 10 years of GPUs



Single precision

OLD QUDA KERNELS

Wonderful background is the Python generated Wilson dslash kernel

One eighth of it anyway…

All old QUDA code was this hybrid mess of Python and / or C macros

Shouldn't be inflicted on your worst enemy

Performance rely on use of texture cache

THE NEED FOR A COMPLETE REWRITE

Extensibility, composability and maintainability

Ability to add new discretizations easily

Integration with **`jitify`**

Changing representation, N_c etc.

Ability to run on CPU?

DSLASH FRAMEWORK

Wilson	✓
Wilson-clover	✓
Twisted-mass singlet	✓
Twisted-mass doublet	✓
Twisted-clover singlet	✓
Naive staggered	✓
Improved staggered	✓
Shamir 5-d	✓
Shamir 4-d	✓
Möbius	✓
Laplace	✓
Laplace-3d	✓
Covariant derviative	✓
Staggered sextet	WIP
Twisted-clover doublet	WIP
Clover “Hasenbusch”	WIP
Block MDWF preconditioner	WIP

All Dslash kernels now completely rewritten new C++ framework

Reuse the same common cores for multiple stencils

All stencils now support overlapping optimized overlapping comms and compute

Easy to add support for new stencils, change N_c, representation, etc.

Merged into QUDA develop branch a https://github.com/lattice/quda/pull/776

WILSON KERNEL

```
  template <typename Float, int nDim, int nColor, int nParity, bool dagger, KernelType kernel_type,
typename Arg, typename Vector>
  __device__ __host__ inline void applyWilson(Vector &out, Arg &arg, int coord[nDim], int x_cb, int s,
                                        int parity, int idx, int thread_dim, bool &active) {
    typedef typename mapper<Float>::type real;
    typedef ColorSpinor<real,nColor,2> HalfVector;
    typedef Matrix<complex<real>,nColor> Link;
    const int their_spinor_parity = nParity == 2 ? 1-parity : 0;

#pragma unroll
    for (int d = 0; d<nDim; d++) { // loop over dimension
      { // Forward gather - compute fwd offset for vector fetch
        const int fwd_idx = getNeighborIndexCB(coord, d, +1, arg.dc);
        constexpr int proj_dir = dagger ? +1 : -1;
        const bool ghost = (coord[d] + arg.nFace >= arg.dim[d]) &&
          isActive<kernel_type>(active, thread_dim, d, coord, arg);

        if ( doHalo<kernel_type>(d) && ghost ) {
         const int ghost_idx = (kernel_type == EXTERIOR_KERNEL_ALL && d != thread_dim) ?
            ghostFaceIndex<1>(coord, arg.dim, d, arg.nFace) : idx;

          Link U = arg.U(d, x_cb, parity);
          HalfVector in = arg.in.Ghost(d, 1, ghost_idx+s*arg.dc.ghostFaceCB[d], their_spinor_parity);
          if (d == 3) in *= arg.t_proj_scale;
          out += (U * in).reconstruct(d, proj_dir);
        } else if ( doBulk<kernel_type>() && !ghost ) {

          Link U = arg.U(d, x_cb, parity);
          Vector in = arg.in(fwd_idx+s*arg.dc.volume_4d_cb, their_spinor_parity);

          out += (U * in.project(d, proj_dir)).reconstruct(d, proj_dir);
        }
      }

      { // Backward gather - compute back offset for spinor and gauge fetch
        const int back_idx = getNeighborIndexCB(coord, d, -1, arg.dc);
        const int gauge_idx = back_idx;
        constexpr int proj_dir = dagger ? -1 : +1;
        const bool ghost = (coord[d] - arg.nFace < 0) &&
          isActive<kernel_type>(active, thread_dim, d, coord, arg);

        if ( doHalo<kernel_type>(d) && ghost) {
         const int ghost_idx = (kernel_type == EXTERIOR_KERNEL_ALL && d != thread_dim) ?
            ghostFaceIndex<0>(coord, arg.dim, d, arg.nFace) : idx;

          Link U = arg.U.Ghost(d, ghost_idx, 1-parity);
          HalfVector in = arg.in.Ghost(d, 0, ghost_idx+s*arg.dc.ghostFaceCB[d], their_spinor_parity);
          if (d == 3) in *= arg.t_proj_scale;

          out += (conj(U) * in).reconstruct(d, proj_dir);
        } else if ( doBulk<kernel_type>() && !ghost ) {

          Link U = arg.U(d, gauge_idx, 1-parity);
          Vector in = arg.in(back_idx+s*arg.dc.volume_4d_cb, their_spinor_parity);

          out += (conj(U) * in.project(d, proj_dir)).reconstruct(d, proj_dir);
        }
      }
    } //nDim

  }
```

```
  //out(x) = M*in = (-D + m) * in(x-mu)
  template <typename Float, int nDim, int nColor, int nParity, bool dagger, bool xpay, KernelType
kernel_type, typename Arg>
  __device__ __host__ inline void wilson(Arg &arg, int idx, int parity)
  {
    typedef typename mapper<Float>::type real;
    typedef ColorSpinor<real,nColor,4> Vector;

    bool active = kernel_type == EXTERIOR_KERNEL_ALL ? false : true; // is thread active (non-trival for
fused kernel only)
    int thread_dim; // which dimension is thread working on (fused kernel only)
    int coord[nDim];
    int x_cb = getCoords<nDim,QUDA_4D_PC,kernel_type>(coord, arg, idx, parity, thread_dim);

    const int my_spinor_parity = nParity == 2 ? parity : 0;
    Vector out;
    applyWilson<Float,nDim,nColor,nParity,dagger,kernel_type>(out, arg, coord, x_cb, 0, parity, idx,
thread_dim, active);

    if (xpay && kernel_type == INTERIOR_KERNEL) {
      Vector x = arg.x(x_cb, my_spinor_parity);
      out = x + arg.kappa * out;
    } else if (kernel_type != INTERIOR_KERNEL && active) {
      Vector x = arg.out(x_cb, my_spinor_parity);
      out = x + (xpay ? arg.kappa * out : out);
    }

    if (kernel_type != EXTERIOR_KERNEL_ALL || active) arg.out(x_cb, my_spinor_parity) = out;
  }

  // CPU kernel for applying the Wilson operator to a vector
  template <typename Float, int nDim, int nColor, int nParity, bool dagger, bool xpay, KernelType
kernel_type, typename Arg>
  void wilsonCPU(Arg arg)
  {

    for (int parity= 0; parity < nParity; parity++) {
      // for full fields then set parity from loop else use arg setting
      parity = nParity == 2 ? parity : arg.parity;

      for (int x_cb = 0; x_cb < arg.threads; x_cb++) { // 4-d volume
        wilson<Float,nDim,nColor,nParity,dagger,xpay,kernel_type>(arg, x_cb, parity);
      } // 4-d volumeCB
    } // parity

  }

  // GPU Kernel for applying the Wilson operator to a vector
  template <typename Float, int nDim, int nColor, int nParity, bool dagger, bool xpay, KernelType
kernel_type, typename Arg>
  __global__ void wilsonGPU(Arg arg)
  {
    int x_cb = blockIdx.x*blockDim.x + threadIdx.x;
    if (x_cb >= arg.threads) return;

    // for full fields set parity from y thread index else use arg setting
    int parity = nParity == 2 ? blockDim.z*blockIdx.z + threadIdx.z : arg.parity;

    switch(parity) {
    case 0: wilson<Float,nDim,nColor,nParity,dagger,xpay,kernel_type>(arg, x_cb, 0); break;
    case 1: wilson<Float,nDim,nColor,nParity,dagger,xpay,kernel_type>(arg, x_cb, 1); break;
    }

  }
```

WILSON KERNEL

```
template <typename Float, int nDim, int nColor, int nParity, bool dagger, KernelType kernel_type,
typename Arg, typename Vector>
  __device__ __host__ inline void applyWilson(Vector &out, Arg &arg, int coord[nDim], int x_cb, int s,
                                              int parity, int idx, int thread_dim, bool &active) {
    typedef typename mapper<Float>::type real;
    typedef ColorSpinor<real,nColor,2> HalfVector;
    typedef Matrix<complex<real>,nColor> Link;
    const int their_spinor_parity = nParity == 2 ? 1-parity : 0;

#pragma unroll
    for (int d = 0; d<nDim; d++) { // loop over dimension
      { // Forward gather - compute fwd offset for vector fetch
        const int fwd_idx = getNeighborIndexCB(coord, d, +1, arg.dc);
        constexpr int proj_dir = dagger ? +1 : -1;
        const bool ghost = (coord[d] + arg.nFace >= arg.dim[d]) &&
          isActive<kernel_type>(active, thread_dim, d, coord, arg);

        if ( doHalo<kernel_type>(d) && ghost ) {
          const int ghost_idx = (kernel_type == EXTERIOR_KERNEL_ALL && d != thread_dim) ?
             ghostFaceIndex<1>(coord, arg.dim, d, arg.nFace) : idx;

          Link U = arg.U(d, x_cb, parity);
          HalfVector in = arg.in.Ghost(d, 1, ghost_idx+s*arg.dc.ghostFaceCB[d], their_spinor_parity);
          if (d == 3) in *= arg.t_proj_scale;
          out += (U * in).reconstruct(d, proj_dir);
        } else if ( doBulk<kernel_type>() && !ghost ) {

          Link U = arg.U(d, x_cb, parity);
          Vector in = arg.in(fwd_idx+s*arg.dc.volume_4d_cb, their_spinor_parity);

          out += (U * in.project(d, proj_dir)).reconstruct(d, proj_dir);
        }
      }

      { // Backward gather - compute back offset for spinor and gauge fetch
        const int back_idx = getNeighborIndexCB(coord, d, -1, arg.dc);
        const int gauge_idx = back_idx;
        constexpr int proj_dir = dagger ? -1 : +1;
        const bool ghost = (coord[d] - arg.nFace < 0) &&
          isActive<kernel_type>(active, thread_dim, d, coord, arg);

        if ( doHalo<kernel_type>(d) && ghost) {
          const int ghost_idx = (kernel_type == EXTERIOR_KERNEL_ALL && d != thread_dim) ?
             ghostFaceIndex<0>(coord, arg.dim, d, arg.nFace) : idx;

          Link U = arg.U.Ghost(d, ghost_idx, 1-parity);
          HalfVector in = arg.in.Ghost(d, 0, ghost_idx+s*arg.dc.ghostFaceCB[d], their_spinor_parity);
          if (d == 3) in *= arg.t_proj_scale;

          out += (conj(U) * in).reconstruct(d, proj_dir);
        } else if ( doBulk<kernel_type>() && !ghost ) {

          Link U = arg.U(d, gauge_idx, 1-parity);
          Vector in = arg.in(back_idx+s*arg.dc.volume_4d_cb, their_spinor_parity);

          out += (conj(U) * in.project(d, proj_dir)).reconstruct(d, proj_dir);
        }
      }
    } //nDim

  }
```

Halo

Interior

Halo

Interior

```
  //out(x) = M*in = (-D + m) * in(x-mu)
  template <typename Float, int nDim, int nColor, int nParity, bool dagger, bool xpay, KernelType
kernel_type, typename Arg>
  __device__ __host__ inline void wilson(Arg &arg, int idx, int parity)
  {
    typedef typename mapper<Float>::type real;
    typedef ColorSpinor<real,nColor,4> Vector;

    bool active = kernel_type == EXTERIOR_KERNEL_ALL ? false : true; // is thread active (non-trival for
fused kernel only)
    int thread_dim; // which dimension is thread working on (fused kernel only)
    int coord[nDim];
    int x_cb = getCoords<nDim,QUDA_4D_PC,kernel_type>(coord, arg, idx, parity, thread_dim);

    const int my_spinor_parity = nParity == 2 ? parity : 0;
    Vector out;
    applyWilson<Float,nDim,nColor,nParity,dagger,kernel_type>(out, arg, coord, x_cb, 0, parity, idx,
thread_dim, active);

    if (xpay && kernel_type == INTERIOR_KERNEL) {
      Vector x = arg.x(x_cb, my_spinor_parity);
      out = x + arg.kappa * out;
    } else if (kernel_type != INTERIOR_KERNEL && active) {
      Vector x = arg.out(x_cb, my_spinor_parity);
      out = x + (xpay ? arg.kappa * out : out);
    }

    if (kernel_type != EXTERIOR_KERNEL_ALL || active) arg.out(x_cb, my_spinor_parity) = out;
  }

  // CPU kernel for applying the Wilson operator to a vector
  template <typename Float, int nDim, int nColor, int nParity, bool dagger, bool xpay, KernelType
kernel_type, typename Arg>
  void wilsonCPU(Arg arg)
  {

    for (int parity= 0; parity < nParity; parity++) {
      // for full fields then set parity from loop else use arg setting
      parity = nParity == 2 ? parity : arg.parity;

      for (int x_cb = 0; x_cb < arg.threads; x_cb++) { // 4-d volume
        wilson<Float,nDim,nColor,nParity,dagger,xpay,kernel_type>(arg, x_cb, parity);
      } // 4-d volumeCB
    } // parity

  }
```

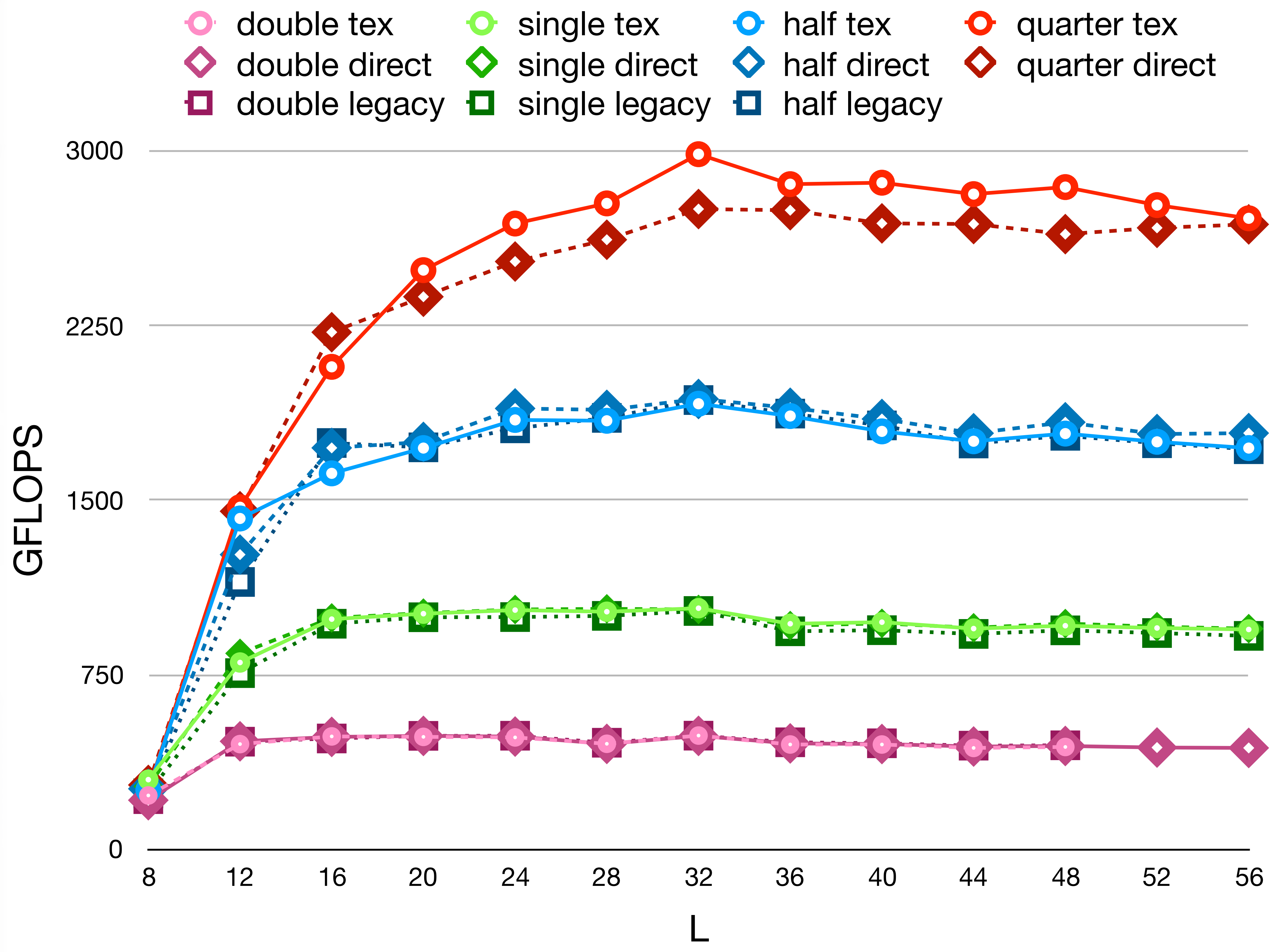
CPU

```
  // GPU Kernel for applying the Wilson operator to a vector
  template <typename Float, int nDim, int nColor, int nParity, bool dagger, bool xpay, KernelType
kernel_type, typename Arg>
  __global__ void wilsonGPU(Arg arg)
  {
    int x_cb = blockIdx.x*blockDim.x + threadIdx.x;
    if (x_cb >= arg.threads) return;

    // for full fields set parity from y thread index else use arg setting
    int parity = nParity == 2 ? blockDim.z*blockIdx.z + threadIdx.z : arg.parity;

    switch(parity) {
    case 0: wilson<Float,nDim,nColor,nParity,dagger,xpay,kernel_type>(arg, x_cb, 0); break;
    case 1: wilson<Float,nDim,nColor,nParity,dagger,xpay,kernel_type>(arg, x_cb, 1); break;
    }

  }
```

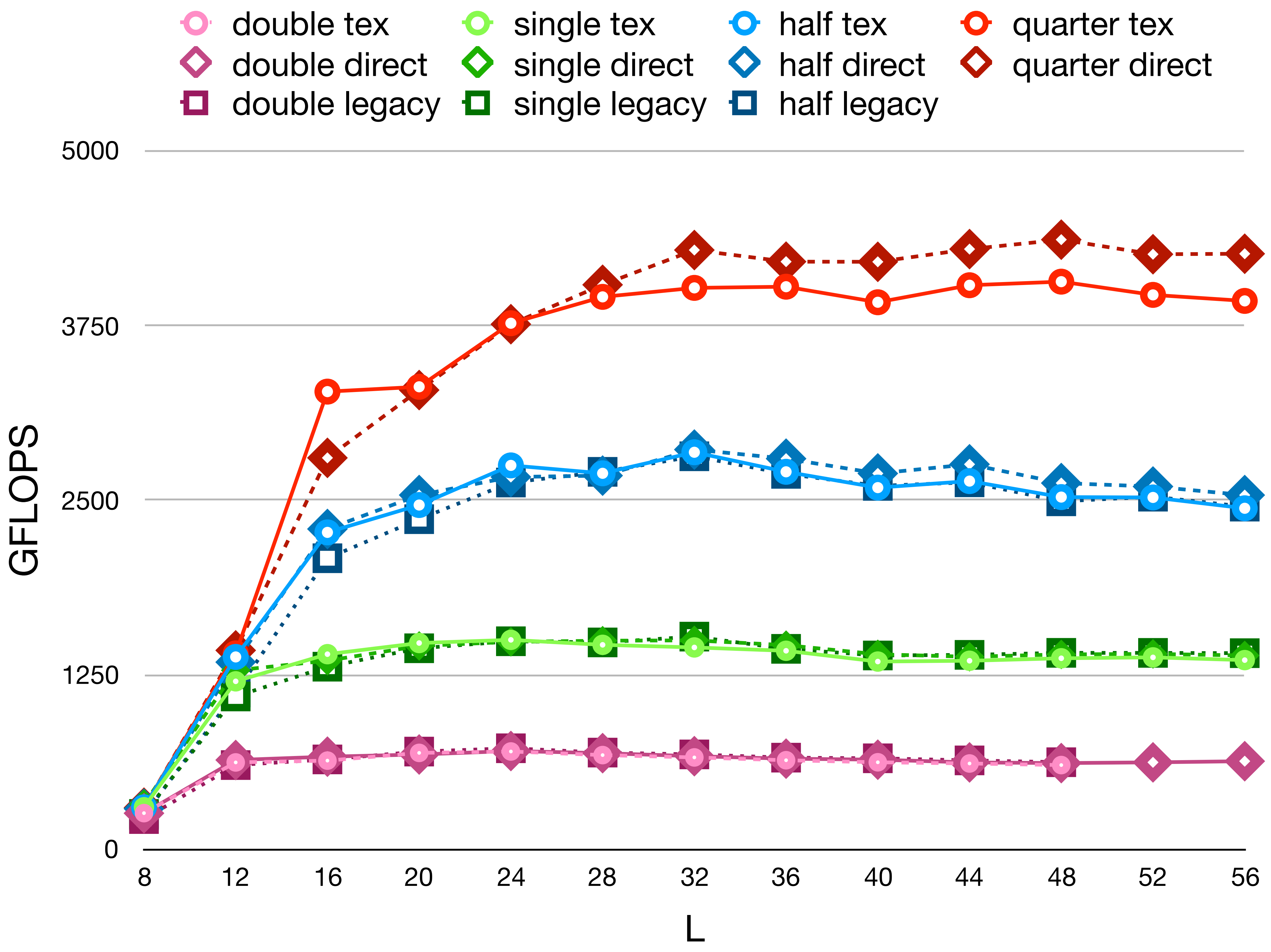
GPU

PERFORMANCE

Legacy ~ 40K LOC total
New ~ 1500 LOC



Pascal



Volta

```
template <typename Float, int nDim, int nColor, int nParity, bool dagger, KernelType kernel_type, typename Arg, typename Vector>
__device__ __host__ inline void applyWilson(Vector &out, Arg &arg, int coord[nDim], int x_cb, int s,
                                            int parity, int idx, int thread_dim, bool &active) {
  typedef typename mapper<Float>::type real;
  typedef ColorSpinor<real,nColor,2> HalfVector;
  typedef Matrix<complex<real>,nColor> Link;
  const int their_spinor_parity = nParity == 2 ? 1-parity : 0;

  // parity for gauge field - include residual parity from 5-d => 4-d checkerboarding
  const int gauge_parity = (nDim == 5 ? (x_cb/arg.dc.volume_4d_cb + parity) % 2 : parity);

#pragma unroll
  for (int d = 0; d<4; d++) { // loop over dimension
    { // Forward gather - compute fwd offset for vector fetch
      const int fwd_idx = getNeighborIndexCB<nDim>(coord, d, +1, arg.dc);
      const int gauge_idx = (nDim == 5 ? x_cb % arg.dc.volume_4d_cb : x_cb);
      constexpr int proj_dir = dagger ? +1 : -1;

      const bool ghost = (coord[d] + arg.nFace >= arg.dim[d]) &&
        isActive<kernel_type>(active, thread_dim, d, coord, arg);

      if ( doHalo<kernel_type>(d) && ghost ) {
        // we need to compute the face index if we are updating a face that isn't ours
        const int ghost_idx = (kernel_type == EXTERIOR_KERNEL_ALL && d != thread_dim) ?
          ghostFaceIndex<1,nDim>(coord, arg.dim, d, arg.nFace) : idx;

        Link U = arg.U(d, gauge_idx, gauge_parity);
        HalfVector in = arg.in.Ghost(d, 1, ghost_idx+s*arg.dc.ghostFaceCB[d], their_spinor_parity);
        if (d == 3) in *= arg.t_proj_scale; // put this in the Ghost accessor and merge with any rescaling?

        out += (U * in).reconstruct(d, proj_dir);
      } else if ( doBulk<kernel_type>() && !ghost ) {

        Link U = arg.U(d, gauge_idx, gauge_parity);
        Vector in = arg.in(fwd_idx+s*arg.dc.volume_4d_cb, their_spinor_parity);

        out += (U * in.project(d, proj_dir)).reconstruct(d, proj_dir);
      }
    }

    { // Backward gather - compute back offset for spinor and gauge fetch
      const int back_idx = getNeighborIndexCB<nDim>(coord, d, -1, arg.dc);
      const int gauge_idx = (nDim == 5 ? back_idx % arg.dc.volume_4d_cb : back_idx);
      constexpr int proj_dir = dagger ? -1 : +1;

      const bool ghost = (coord[d] - arg.nFace < 0) &&
        isActive<kernel_type>(active, thread_dim, d, coord, arg);

      if ( doHalo<kernel_type>(d) && ghost) {
        // we need to compute the face index if we are updating a face that isn't ours
        const int ghost_idx = (kernel_type == EXTERIOR_KERNEL_ALL && d != thread_dim) ?
          ghostFaceIndex<0,nDim>(coord, arg.dim, d, arg.nFace) : idx;

        const int gauge_ghost_idx = (nDim == 5 ? ghost_idx % arg.dc.ghostFaceCB[d] : ghost_idx);
        Link U = arg.U.Ghost(d, gauge_ghost_idx, 1-gauge_parity);
        HalfVector in = arg.in.Ghost(d, 0, ghost_idx+s*arg.dc.ghostFaceCB[d], their_spinor_parity);
        if (d == 3) in *= arg.t_proj_scale;

        out += (conj(U) * in).reconstruct(d, proj_dir);
      } else if ( doBulk<kernel_type>() && !ghost ) {

        Link U = arg.U(d, gauge_idx, 1-gauge_parity);
        Vector in = arg.in(back_idx+s*arg.dc.volume_4d_cb, their_spinor_parity);

        out += (conj(U) * in.project(d, proj_dir)).reconstruct(d, proj_dir);
      }
    }
  } //nDim

}
```

COMPOSABILITY

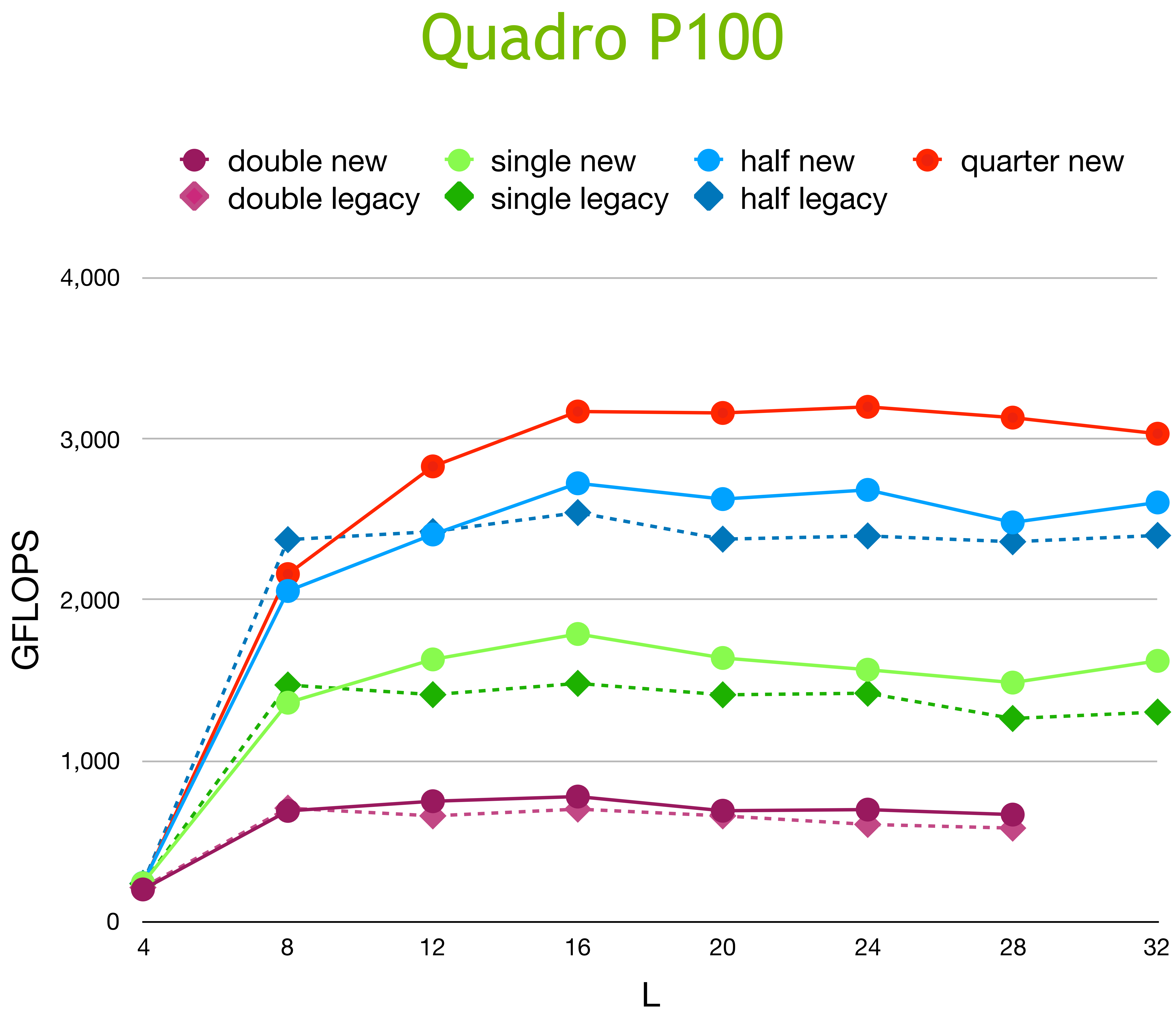
Same Wilson dslash kernel is used by all Wilson-like operators

- Wilson
- Clover
- Twisted mass
- Twisted clover
- Shamir 4-d/5-d
- Möbius

Similar for staggered kernel

- Naive
- Improved
- Sextet fermions

4D-PRECONDITIONED SHAMIR



Sierra CG MDWF Performance
Volume = 48^3x64x12
(Courtesy of CalLat)

Nodes	Performance	Perf/GPU
4[old]	27300	1706
4	35615	2226
3	29100	2425
2	25312	3164

EMBRACING NEW IDEAS

Prior framework meant embracing new algorithms and technologies was very time consuming, requiring lots of hand editing, or simply not tractable

Algorithms	*Technologies*
Multi-rhs solvers	***SHMEM***
New precisions (quarter and quad?)	***Large memory***
Schwarz preconditioners	***CUDA graphs***
Cache and register tiling	
Finer-grain parallelism	

JITIFY INTEGRATION

Framework supports just-in-time compilation

Offline: kernel is an include

JIT: specify kernel name as string

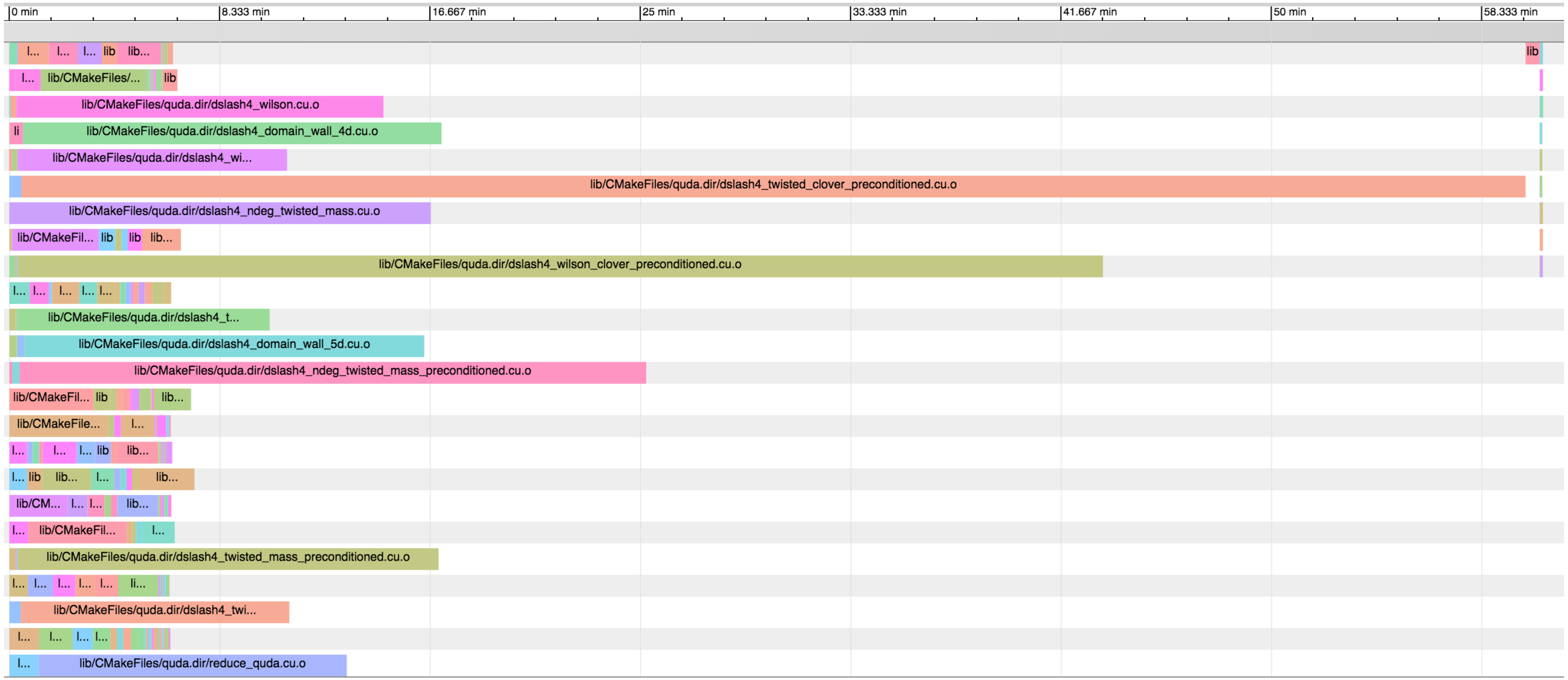
Offline: static instantiation name

JIT: load kernel as string

```
#include <kernels/dslash_wilson.cuh>
namespace quda {

  template <typename Float, int nDim, int nColor, int nParity, bool dagger, bool xpay, KernelType kernel_type, typename Arg>
  struct WilsonLaunch {
    static constexpr const char *kernel = "quda::wilsonGPU"; // kernel name for jit compilation
    template <typename Dslash>
    inline static void launch(Dslash &dslash, TuneParam &tp, Arg &arg, const cudaStream_t &stream) {
      dslash.launch(wilsonGPU<Float,nDim,nColor,nParity,dagger,xpay,kernel_type,Arg>, tp, arg, stream);
    }
  };

  template <typename Float, int nDim, int nColor, typename Arg>
  class Wilson : public Dslash<Float> {

  protected:
    Arg &arg;
    const ColorSpinorField &in;

  public:

    Wilson(Arg &arg, const ColorSpinorField &out, const ColorSpinorField &in)
      : Dslash<Float>(arg, out, in, "kernels/dslash_wilson.cuh"), arg(arg), in(in) { }

    virtual ~Wilson() { }

    void apply(const cudaStream_t &stream) {
      TuneParam tp = tuneLaunch(*this, getTuning(), getVerbosity());
      Dslash<Float>::setParam(arg);
      Dslash<Float>::template instantiate<WilsonLaunch,nDim,nColor>(tp, arg, stream);
    }
};
```

COMPILATION COMPARISON

Offline compilation



NVIDIA

COMPILATION COMPARISON

Just-in-time compilation

NVIDIA

ONGOING QUDA WORK

…or a subset of…

Multigrid

- Continual improvements for clover (with Balint)
- Staggered Multigrid (primarily Evan)

Reworked deflation and eigensolver framework (Dean)

- Lanczos and Arnoldi
- Deflation-accelerated multigrid (prolongator and coarse-grid)

Finish up ripping out old framework - QUDA will half in size from 900K LOC -> 450K LOC

New stencils: Sextet fermions (with Ricky Wong), non-degenerate twisted-clover fermions, etc.

Internal @ NVIDIA: QUDA/LQCD is used for future architecture / compiler / driver developments

- NVSHMEM (see Mathias' talk)

QUDA ROADMAP

Release QUDA 1.0 (this summer)

Post 1.0

- Multi-rhs block solvers for all stencils
- Improved in strong scaling through NVSHMEM (see Mathias' talk)
- Beyond just regular QCD

Longer term

- Investigate how well QUDA runs on C++17 pSTL

Post feature requests here: https://github.com/lattice/quda/issues

SUMMARY

The dslash kernel rewrite is the biggest change to QUDA in 10 years

- Will enable anyone to add support for new Dirac operators
- Enables next generation of algorithm techniques

Continual performance and algorithm improvements

BACK UP

ACCESSORS

99% of QUDA is now written using “accessors”

Originally implemented for QUDA’s copy kernels to support application data layout

Opaque load and store functions that obfuscate data order

Data compression/decompression handled in the accessor

- Gauge reconstruction
- Fixed-point <-> floating point conversion and scaling

Trivial to add support for new data order

MILC GAUGE ACCESSOR

```
template <typename Float, int length> struct MILCOrder : public LegacyOrder<Float,length> {
  typedef typename mapper<Float>::type RegType;
  Float *gauge;
  const int volumeCB;
  const int geometry;
MILCOrder(const GaugeField &u, Float *gauge_=0, Float **ghost_=0) :
  LegacyOrder<Float,length>(u, ghost_), gauge(gauge_ ? gauge_ : (Float*)u.Gauge_p()),
    volumeCB(u.VolumeCB()), geometry(u.Geometry()) { ; }
MILCOrder(const MILCOrder &order) : LegacyOrder<Float,length>(order),
    gauge(order.gauge), volumeCB(order.volumeCB), geometry(order.geometry)
    { ; }
  virtual ~MILCOrder() { ; }
```

load accessor

```
  __device__ __host__ inline void load(RegType v[], int x, int dir, int parity) const {
   for (int i=0; i<length; i++) {
      v[i] = (RegType)gauge[((parity*volumeCB+x)*geometry + dir)*length + i];
    }
 }
```

save accessor

```
  __device__ __host__ inline void save(const RegType v[], int x, int dir, int parity) {
   for (int i=0; i<length; i++) {
      gauge[((parity*volumeCB+x)*geometry + dir)*length + i] = (Float)v[i];
    }
 }
```

lhs wrapper (`U(d,x,parity) = u;`)

```
 __device__ __host__ inline gauge_wrapper<RegType,MILCOrder<Float,length> >
    operator()(int dim, int x_cb, int parity) {
    return gauge_wrapper<RegType,MILCOrder<Float,length> >(*this, dim, x_cb, parity);
  }
```

rhs wrapper (`u = U(d,x,parity);`)

```
 __device__ __host__ inline const gauge_wrapper<RegType,MILCOrder<Float,length> >
    operator()(int dim, int x_cb, int parity) const {
    return gauge_wrapper<RegType,MILCOrder<Float,length> >
      (const_cast<MILCOrder<Float,length>&>(*this), dim, x_cb, parity);
  }

  size_t Bytes() const { return length * sizeof(Float); }
};
```

SPINOR ACCESSOR

```
template <typename Float, int Ns, int Nc>
  struct SpaceColorSpinorOrder {
    typedef typename mapper<Float>::type RegType;
    static const int length = 2 * Ns * Nc;
    Float *field;
    size_t offset;
    Float *ghost[8];
    int volumeCB;
    int faceVolumeCB[4];
    int stride;
    int nParity;
  SpaceColorSpinorOrder(const ColorSpinorField &a, int nFace=1, Float *field_=0, float *dummy=0, Float **ghost_=0)
    : field(field_ ? field_ : (Float*)a.V()), offset(a.Bytes()/(2*sizeof(Float))),
      volumeCB(a.VolumeCB()), stride(a.Stride()), nParity(a.SiteSubset())
    {
      if (volumeCB != stride) errorQuda("Stride must equal volume for this field order");
      for (int i=0; i<4; i++) {
        ghost[2*i] = ghost_ ? ghost_[2*i] : 0;
        ghost[2*i+1] = ghost_ ? ghost_[2*i+1] : 0;
        faceVolumeCB[i] = a.SurfaceCB(i)*nFace;
      }
    }
    virtual ~SpaceColorSpinorOrder() { ; }
```

```
  __device__ __host__ inline void load(RegType v[length], int x, int parity=0) const {
    for (int s=0; s<Ns; s++) {
      for (int c=0; c<Nc; c++) {
        for (int z=0; z<2; z++) {
          v[(s*Nc+c)*2+z] = field[parity*offset + ((x*Nc + c)*Ns + s)*2 + z];
        }
      }
    }
  }
```

load accessor

```
  __device__ __host__ inline void save(const RegType v[length], int x, int parity=0) {
    for (int s=0; s<Ns; s++) {
      for (int c=0; c<Nc; c++) {
        for (int z=0; z<2; z++) {
          field[parity*offset + ((x*Nc + c)*Ns + s)*2 + z] = v[(s*Nc+c)*2+z];
        }
      }
    }
  }
```

save accessor

```
  __device__ __host__ inline colorspinor_wrapper<RegType,SpaceColorSpinorOrder<Float,Ns,Nc> >
    operator()(int x_cb, int parity) {
    return colorspinor_wrapper<RegType,SpaceColorSpinorOrder<Float,Ns,Nc> >(*this, x_cb, parity);
  }

 __device__ __host__ inline const colorspinor_wrapper<RegType,SpaceColorSpinorOrder<Float,Ns,Nc> >
  operator()(int x_cb, int parity) const {
    return colorspinor_wrapper<RegType,SpaceColorSpinorOrder<Float,Ns,Nc> >
    (const_cast<SpaceColorSpinorOrder<Float,Ns,Nc>&>(*this), x_cb, parity);
 }
```

```
  __device__ __host__ inline void loadGhost(RegType v[length], int x, int dim,
                                            int dir, int parity=0) const {
    for (int s=0; s<Ns; s++) {
      for (int c=0; c<Nc; c++) {
        for (int z=0; z<2; z++) {
          v[(s*Nc+c)*2+z] =
            ghost[2*dim+dir][(((parity*faceVolumeCB[dim]+x)*Nc + c)*Ns + s)*2 + z];
        }
      }
    }
  }
```

loadGhost accessor

```
  __device__ __host__ inline void saveGhost(const RegType v[length], int x, int dim,
                                            int dir, int parity=0) {
    for (int s=0; s<Ns; s++) {
      for (int c=0; c<Nc; c++) {
        for (int z=0; z<2; z++) {
          ghost[2*dim+dir][(((parity*faceVolumeCB[dim]+x)*Nc + c)*Ns + s)*2 + z]
             = v[(s*Nc+c)*2+z];
        }
      }
    }
  }
```

saveGhost accessor

```
  size_t Bytes() const { return nParity * volumeCB * Nc * Ns * 2 * sizeof(Float); }
};
```

WILSON-CLOVER KERNEL

```
  template <typename Float, int nDim, int nColor, int nParity, bool dagger, KernelType
kernel_type, typename Arg>
  __device__ __host__ inline void wilsonClover(Arg &arg, int idx, int parity)
  {
    typedef typename mapper<Float>::type real;
    typedef ColorSpinor<real,nColor,4> Vector;
    typedef ColorSpinor<real,nColor,2> HalfVector;

    bool active = kernel_type == EXTERIOR_KERNEL_ALL ? false : true;
    int thread_dim; // which dimension is thread working on (fused kernel only)
    int coord[nDim];
    int x_cb = getCoords<nDim,QUDA_4D_PC,kernel_type>(coord, arg, idx, parity, thread_dim);

    const int my_spinor_parity = nParity == 2 ? parity : 0;
    Vector out;

    // defined in dslash_wilson.cuh
    applyWilson<Float,nDim,nColor,nParity,dagger,kernel_type>(out, arg, coord, x_cb, 0, parity,
idx, thread_dim, active);

    if (kernel_type == INTERIOR_KERNEL) {
      Vector x = arg.x(x_cb, my_spinor_parity);
      x.toRel(); // switch to chiral basis

      Vector tmp;

#pragma unroll
      for (int chirality=0; chirality<2; chirality++) {
        HMatrix<real,nColor*Arg::nSpin/2> A = arg.A(x_cb, parity, chirality);
        HalfVector x_chi = A * x.chiral_project(chirality);
        tmp += x_chi.chiral_reconstruct(chirality);
      }

      tmp.toNonRel(); // switch back to non-chiral basis

      out = tmp + arg.kappa * out;
    } else if (active) {
      Vector x = arg.out(x_cb, my_spinor_parity);
      out = x + arg.kappa * out;
    }

    if (kernel_type != EXTERIOR_KERNEL_ALL || active) arg.out(x_cb, my_spinor_parity) = out;
  }
```

```
  // CPU kernel for applying the Wilson operator to a vector
  template <typename Float, int nDim, int nColor, int nParity, bool dagger,
KernelType kernel_type, typename Arg>
  void wilsonCloverCPU(Arg arg)
  {
    for (int parity= 0; parity < nParity; parity++) {
      // for full fields then set parity from loop else use arg setting
      parity = nParity == 2 ? parity : arg.parity;

      for (int x_cb = 0; x_cb < arg.threads; x_cb++) { // 4-d volume
        wilsonClover<Float,nDim,nColor,nParity,dagger,kernel_type>(arg, x_cb,
parity);
      } // 4-d volumeCB
    } // parity

  }

  // GPU Kernel for applying the Wilson operator to a vector
  template <typename Float, int nDim, int nColor, int nParity, bool dagger,
KernelType kernel_type, typename Arg>
  __global__ void wilsonCloverGPU(Arg arg)
  {
    int x_cb = blockIdx.x*blockDim.x + threadIdx.x;
    if (x_cb >= arg.threads) return;

    // for full fields set parity from y thread index else use arg setting
    int parity = nParity == 2 ? blockDim.z*blockIdx.z + threadIdx.z : arg.parity;

    switch(parity) {
    case 0: wilsonClover<Float,nDim,nColor,nParity,dagger,kernel_type>(arg, x_cb,
0); break;
    case 1: wilsonClover<Float,nDim,nColor,nParity,dagger,kernel_type>(arg, x_cb,
1); break;
    }

  }
```

WILSON-CLOVER PRECONDITIONED

```
  template <typename Float, int nDim, int nColor, int nParity, bool dagger, bool xpay, KernelType
kernel_type, typename Arg>
  __device__ __host__ inline void wilsonClover(Arg &arg, int idx, int parity)
  {
    using namespace linalg; // for Cholesky
    typedef typename mapper<Float>::type real;
    typedef ColorSpinor<real,nColor,4> Vector;
    typedef ColorSpinor<real,nColor,2> HalfVector;

    bool active = kernel_type == EXTERIOR_KERNEL_ALL ? false : true;
    int thread_dim; // which dimension is thread working on (fused kernel only)
    int coord[nDim];
    int x_cb = getCoords<nDim,QUDA_4D_PC,kernel_type>(coord, arg, idx, parity, thread_dim);

    const int my_spinor_parity = nParity == 2 ? parity : 0;

    Vector out;

    // defined in dslash_wilson.cuh
    applyWilson<Float,nDim,nColor,nParity,dagger,kernel_type>(out, arg, coord, x_cb, 0, parity,
idx, thread_dim, active);

    if (kernel_type != INTERIOR_KERNEL && active) {
      // if we're not the interior kernel, then we must sum the partial
      Vector x = arg.out(x_cb, my_spinor_parity);
      out += x;
    }

    if ( isComplete<kernel_type>(arg, coord) && active ) {
      out.toRel(); // switch to chiral basis

      Vector tmp;

#pragma unroll
      for (int chirality=0; chirality<2; chirality++) {

        HMatrix<real,nColor*Arg::nSpin/2> A = arg.A(x_cb, parity, chirality);
        HalfVector out_chi = out.chiral_project(chirality);

        if (arg.dynamic_clover) {
          Cholesky<HMatrix,real,nColor*Arg::nSpin/2> cholesky(A);
          out_chi = static_cast<real>(0.25)*cholesky.backward(cholesky.forward(out_chi));
        } else {
          out_chi = A * out_chi;
        }

        tmp += out_chi.chiral_reconstruct(chirality);
      }

      tmp.toNonRel(); // switch back to non-chiral basis

      if (xpay) {
        Vector x = arg.x(x_cb, my_spinor_parity);
        out = x + arg.kappa * tmp;
      } else {
        out = tmp;
      }
    }

    if (kernel_type != EXTERIOR_KERNEL_ALL || active) arg.out(x_cb, my_spinor_parity) = out;
  }
```

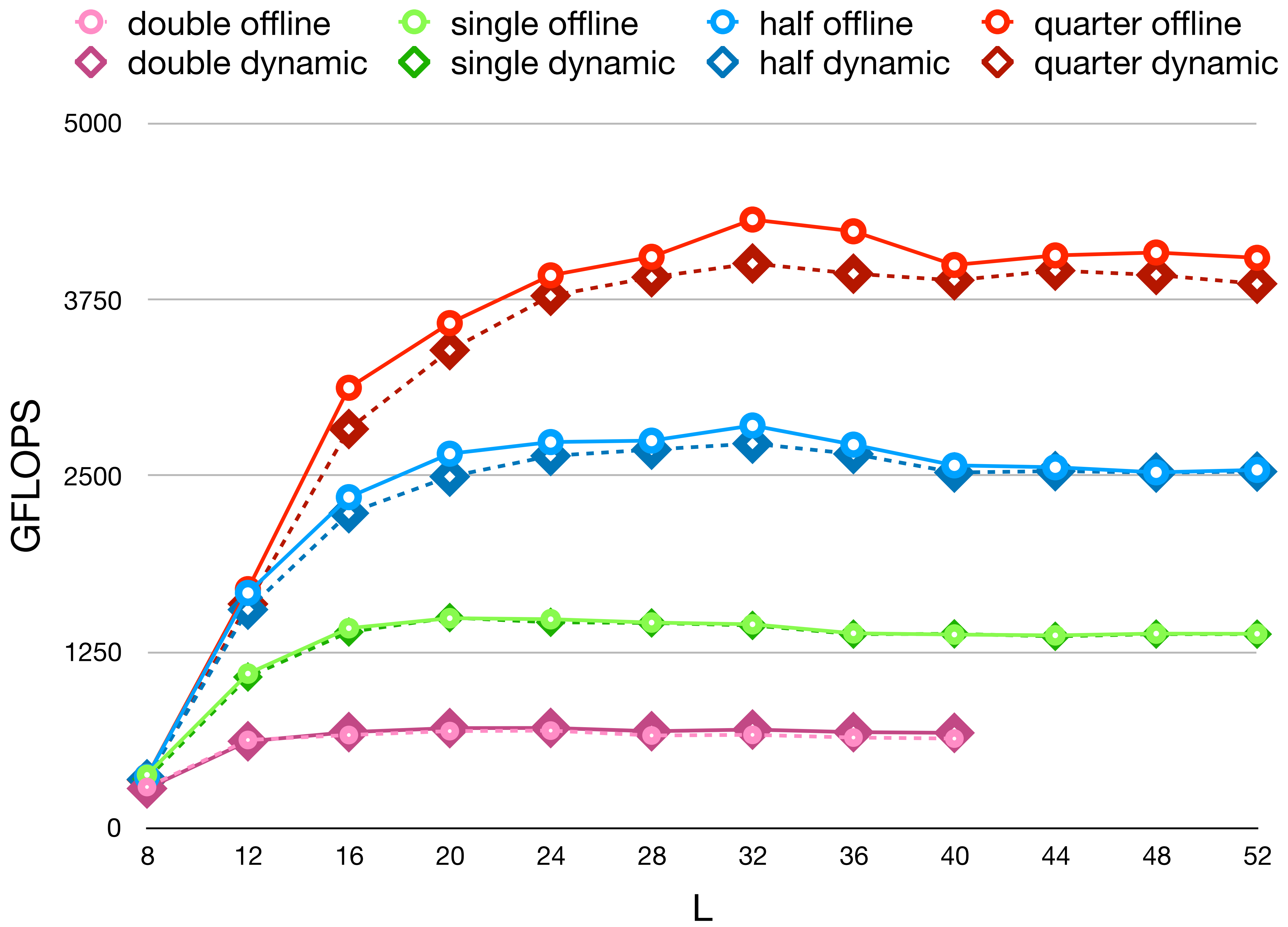
```
  // CPU kernel for applying the Wilson operator to a vector
  template <typename Float, int nDim, int nColor, int nParity, bool dagger, bool xpay, KernelType
kernel_type, typename Arg>
  void wilsonCloverCPU(Arg arg)
  {

    for (int parity= 0; parity < nParity; parity++) {
      // for full fields then set parity from loop else use arg setting
      parity = nParity == 2 ? parity : arg.parity;

      for (int x_cb = 0; x_cb < arg.threads; x_cb++) { // 4-d volume
        wilsonClover<Float,nDim,nColor,nParity,dagger,xpay,kernel_type>(arg, x_cb, parity);
      } // 4-d volumeCB
    } // parity

  }

  // GPU Kernel for applying the Wilson operator to a vector
  template <typename Float, int nDim, int nColor, int nParity, bool dagger, bool xpay, KernelType
kernel_type, typename Arg>
  __global__ void wilsonCloverGPU(Arg arg)
  {
    int x_cb = blockIdx.x*blockDim.x + threadIdx.x;
    if (x_cb >= arg.threads) return;

    // for full fields set parity from y thread index else use arg setting
    int parity = nParity == 2 ? blockDim.z*blockIdx.z + threadIdx.z : arg.parity;

    switch(parity) {
    case 0: wilsonClover<Float,nDim,nColor,nParity,dagger,xpay,kernel_type>(arg, x_cb, 0); break;
    case 1: wilsonClover<Float,nDim,nColor,nParity,dagger,xpay,kernel_type>(arg, x_cb, 1); break;
    }

  }
```

DYNAMIC CLOVER

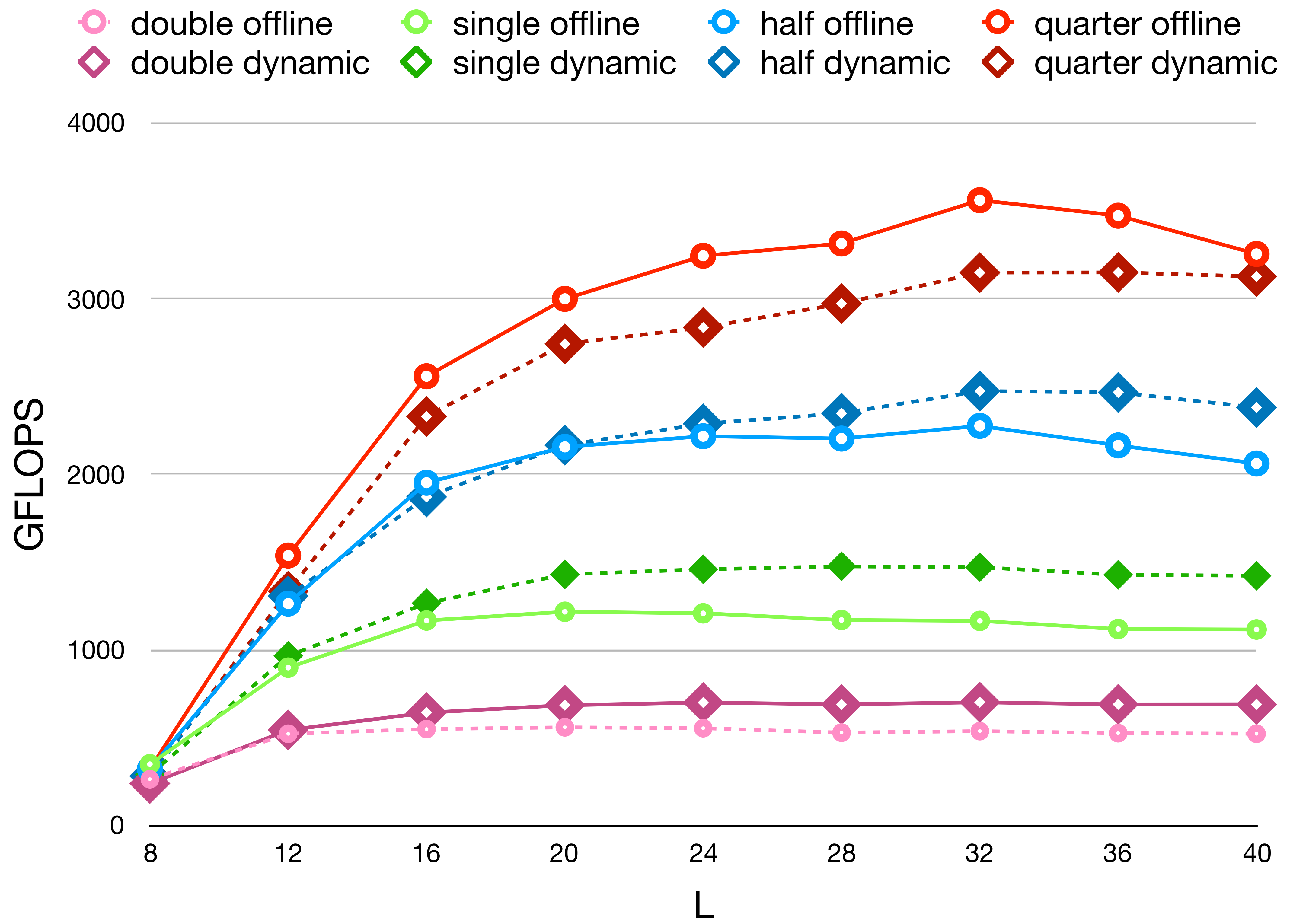
```
#pragma unroll
      for (int chirality=0; chirality<2; chirality++) {

        HMatrix<real,nColor*Arg::nSpin/2> A = arg.A(x_cb, parity, chirality);
        HalfVector out_chi = out.chiral_project(chirality);

        if (arg.dynamic_clover) {
          Cholesky<HMatrix,real,nColor*Arg::nSpin/2> cholesky(A);
          out_chi = cholesky.backward(cholesky.forward(out_chi));
        } else {
          out_chi = A * out_chi;
        }

        tmp += out_chi.chiral_reconstruct(chirality);
      }
```

- Memory reduction strategy
  - No need to store both clover and inverse clover fields
- Performance (and memory reduction) strategy
  - 72 reals per site -> 54 reals per site (TODO)

DYNAMIC CLOVER PERFORMANCE



Clover



Twisted-Clover

```
template <typename Float, typename Arg>
class Stencil : public Tunable {

  protected:
    Arg &arg;
    const Field &meta;

    long long flops() const
    {
      return 48*(long long)meta.VolumeCB();
    }
    long long bytes() const
    {
      return arg.out.Bytes() + 8*arg.in.Bytes();
    }
    bool tuneGridDim() const { return false; }
    unsigned int minThreads() const { return arg.volume; }

  public:
    Stencil(Arg &arg, const Field &meta) : arg(arg), meta(meta)
    {
      strcpy(aux, meta.AuxString());
      strcat(aux, comm_dim_partitioned_string());
    }
    virtual ~Stencil() { }

    void apply(const cudaStream_t &stream) {
        TuneParam tp = tuneLaunch(*this, getTuning(), getVerbosity());
        stencilGPU<Float,nDim,nColor> <<<tp.grid,tp.block,tp.shared_bytes,stream>>>(arg);
      }
    }

    TuneKey tuneKey() const { return TuneKey(meta.VolString(), typeid(*this).name(), aux); }
  };

  template <typename Float>
    void ApplyStencil(Field &out, const Field &in)
  {
    StencilArg<Float> arg(out, in);
    Stencil<Float,StencilArg<Float> > stencil(arg, in);
    stencil.apply(0);
  }
```

Performance metrics

Tuning metadata

Launch work to CPU or GPU depending on field type

Unique "TuneKey" is for every kernel parameter set

1. Create launcher class instance
2. Launch work