DEVELOPING QCD ALGORITHMS FOR NVIDIA GPUS USING THE QUDA FRAMEWORK

Kate Clark and Mathias Wagner, June 23rd 2017


SciDAC
Scientific Discovery through Advanced Computing


QUDA

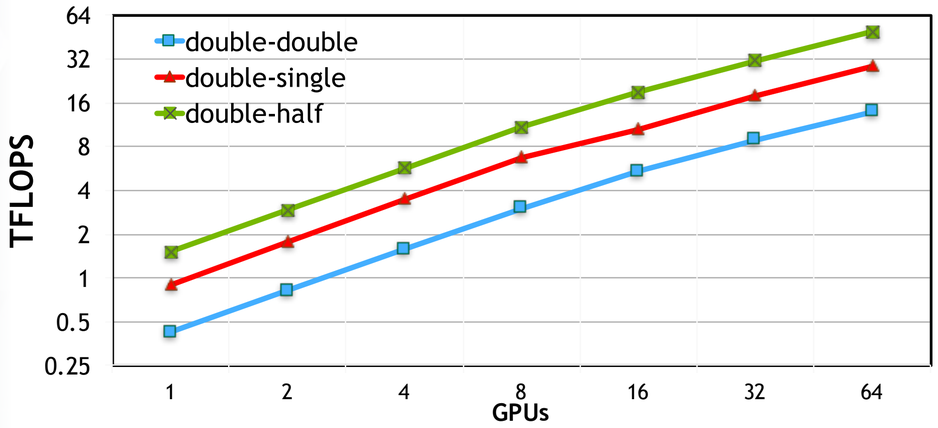
- "QCD on CUDA" – http://lattice.github.com/quda (open source, BSD license)
- Effort started at Boston University in 2008, now in wide use as the GPU backend for BQCD, Chroma, CPS, MILC, TIFR, tmLQCD, etc.
- Provides:
  - Various solvers for all major fermionic discretizations, with multi-GPU support
  - Additional performance-critical routines needed for gauge-field generation
- 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 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 (Temple)
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)
Alejandro Vaquero (INFN)
Mathias Wagner (NVIDIA)
Evan Weinberg (BU)
Frank Winter (Jlab)
Insert your name here!

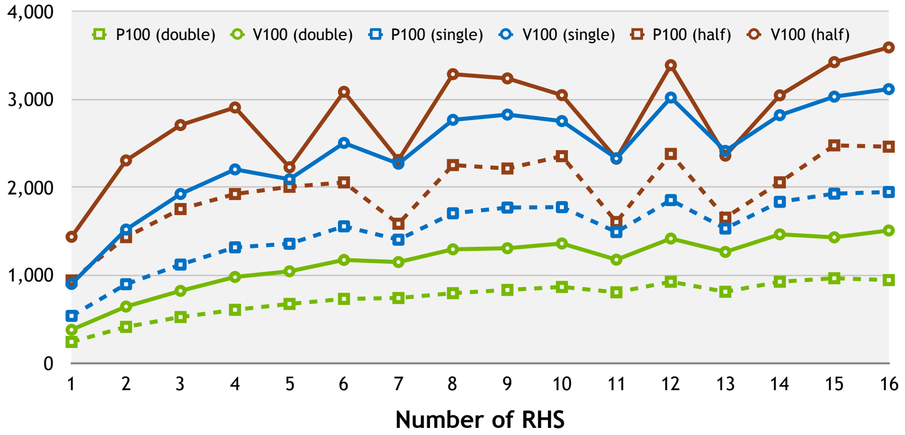
RECENT FOCUS

CG weak scaling, V=24^4x16, Mixed-precision Shamir, Saturn V



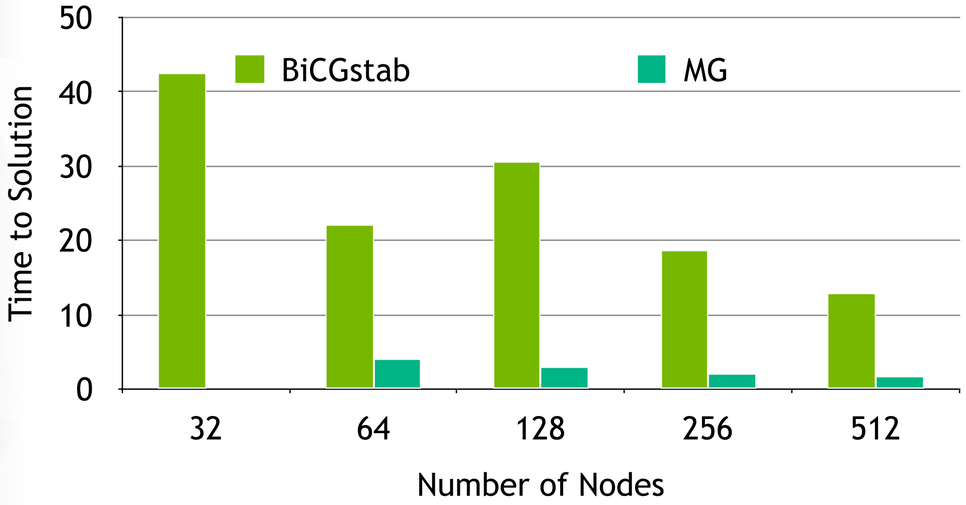
Improving multi-GPU scaling

HISQ Dslash multi-RHS, 24^4



Multi-RHS and block solvers

Strong scaling, Wilson-clover, V = 64^3x128, m_π = 197 MeV



Adaptive Multigrid

Just (if not *more*) important

- Code hygiene
- Documentation
- Development workflow

NVIDIA

A DAY IN THE LIFE OF A QUDA ALGORITHM

EASE OF DEVELOPMENT

QUDA has developed into an LQCD framework that

- ensures first-class GPU performance
- extensible to new theories, algorithms, etc.
- is simple to write new code for

EASE OF DEVELOPMENT

QUDA has developed into an LQCD framework that

- ensures first-class GPU performance
- extensible to new theories, algorithms, etc.
- is simple to write new code for - Really?

EASE OF DEVELOPMENT

QUDA has developed into an LQCD framework that

- ensures first-class GPU performance
- extensible to new theories, algorithms, etc.
- is simple to write new code for - Really?

Let's walk through how to write a kernel from scratch: example Gauge Laplace Operator

- Work through how to instantiate computation based on run-time meta data
- Using the autotuner
- Understand the performance
- Then we'll modify it into a Wilson Dslash

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

GAUGE LAPLACE STENCIL

We want to implement

$$M_{x,y}\phi_y = (\delta_{x,y} - \kappa D_{x,y})\phi_y = \delta_{x,y} - \kappa \sum_{\mu} \left(U_{\mu}(x)\delta_{x,y+\mu} + U^{\dagger}_{\mu}(x-\mu)\delta_{x,y-\mu} \right) \phi_y$$

Assumed starting point is after we have already created our data fields

- ColorSpinorField out - result field
- GaugeField U - gauge field
- ColorSpinorField in - input field

Use actual QUDA code with not slideware

```
/**
   @brief Applies the off-diagonal part of the Laplace operator

   @param[out] out The out result field
   @param[in] U The gauge field
   @param[in] kappa Kappa value
   @param[in] in The input field
   @param[in] parity The site parity
   @param[in] x_cb The checker-boarded site index
 */
extern __shared__ float s[];
template <typename Float, int nDim, int nColor, typename Vector, typename Arg>
__device__ __host__ inline void applyLaplace(Vector &out, Arg &arg, int x_cb, int parity) {
  typedef Matrix<complex<Float>,nColor> Link;
  const int their_spinor_parity = (arg.nParity == 2) ? 1-parity : 0;

  int coord[nDim];
  getCoords(coord, x_cb, arg.dim, parity);

#pragma unroll
  for (int d = 0; d<nDim; d++) // loop over dimension
  {
    //Forward gather - compute fwd offset for vector fetch
    const int fwd_idx = linkIndexP1(coord, arg.dim, d);

    if ( arg.commDim[d] && (coord[d] + arg.nFace >= arg.dim[d]) ) {
      const int ghost_idx = ghostFaceIndex<1>(coord, arg.dim, d, arg.nFace);

      const Link U = arg.U(d, x_cb, parity);
      const Vector in = arg.in.Ghost(d, 1, ghost_idx, their_spinor_parity);

      out += U * in;
    } else {

      const Link U = arg.U(d, x_cb, parity);
      const Vector in = arg.in(fwd_idx, their_spinor_parity);

      out += U * in;
    }

    //Backward gather - compute back offset for spinor and gauge fetch
    const int back_idx = linkIndexM1(coord, arg.dim, d);
    const int gauge_idx = back_idx;

    if ( arg.commDim[d] && (coord[d] - arg.nFace < 0) ) {
      const int ghost_idx = ghostFaceIndex<0>(coord, arg.dim, d, arg.nFace);

      const Link U = arg.U.Ghost(d, ghost_idx, 1-parity);
      const Vector in = arg.in.Ghost(d, 0, ghost_idx, their_spinor_parity);

      out += conj(U) * in;
    } else {

      const Link U = arg.U(d, gauge_idx, 1-parity);
      const Vector in = arg.in(back_idx, their_spinor_parity);

      out += conj(U) * in;
    }
  } //nDim

}
```

LAPLACE KERNEL

```
//out(x) = M*in = (-D + m) * in(x-mu)
template <typename Float, int nDim, int nColor, typename Arg>
__device__ __host__ inline void laplace(Arg &arg, int x_cb, int parity)
{
  typedef ColorSpinor<Float,nColor,1> Vector;
  Vector out;

  applyLaplace<Float,nDim,nColor>(out, arg, x_cb, parity);

  if (arg.isXpay()) {
    Vector x = arg.x(x_cb, parity);
    out = x + arg.kappa * out;
  }
  arg.out(x_cb, parity) = out;
}

// CPU kernel for applying the Laplace operator to a vector
template <typename Float, int nDim, int nColor, typename Arg>
void laplaceCPU(Arg arg)
{

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

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

}

// GPU Kernel for applying the Laplace operator to a vector
template <typename Float, int nDim, int nColor, typename Arg>
__global__ void laplaceGPU(Arg arg)
{
  int x_cb = blockIdx.x*blockDim.x + threadIdx.x;

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

  if (x_cb >= arg.volumeCB) return;
  if (parity >= arg.nParity) return;

  laplace<Float,nDim,nColor>(arg, x_cb, parity);
}
```

LAPLACE KERNEL

```
/**
   @brief Applies the off-diagonal part of the Laplace operator

   @param[out] out The out result field
   @param[in] U The gauge field
   @param[in] kappa Kappa value
   @param[in] in The input field
   @param[in] parity The site parity
   @param[in] x_cb The checker-boarded site index
 */
extern __shared__ float s[];
template <typename Float, int nDim, int nColor, typename Vector, typename Arg>
__device__ __host__ inline void applyLaplace(Vector &out, Arg &arg, int x_cb, int parity) {
  typedef Matrix<complex<Float>,nColor> Link;
  const int their_spinor_parity = (arg.nParity == 2) ? 1-parity : 0;

  int coord[nDim];
  getCoords(coord, x_cb, arg.dim, parity);

#pragma unroll
  for (int d = 0; d<nDim; d++) // loop over dimension
  {
    //Forward gather - compute fwd offset for vector fetch
    const int fwd_idx = linkIndexP1(coord, arg.dim, d);

    if ( arg.commDim[d] && (coord[d] + arg.nFace >= arg.dim[d]) ) {
      const int ghost_idx = ghostFaceIndex<1>(coord, arg.dim, d, arg.nFace);

      const Link U = arg.U(d, x_cb, parity);
      const Vector in = arg.in.Ghost(d, 1, ghost_idx, their_spinor_parity);

      out += U * in;
    } else {

      const Link U = arg.U(d, x_cb, parity);
      const Vector in = arg.in(fwd_idx, their_spinor_parity);

      out += U * in;
    }

    //Backward gather - compute back offset for spinor and gauge fetch
    const int back_idx = linkIndexM1(coord, arg.dim, d);
    const int gauge_idx = back_idx;

    if ( arg.commDim[d] && (coord[d] - arg.nFace < 0) ) {
      const int ghost_idx = ghostFaceIndex<0>(coord, arg.dim, d, arg.nFace);

      const Link U = arg.U.Ghost(d, ghost_idx, 1-parity);
      const Vector in = arg.in.Ghost(d, 0, ghost_idx, their_spinor_parity);

      out += conj(U) * in;
    } else {

      const Link U = arg.U(d, gauge_idx, 1-parity);
      const Vector in = arg.in(back_idx, their_spinor_parity);

      out += conj(U) * in;
    }
  } //nDim

}
```

CPU Kernel

indices from loops

```
//out(x) = M*in = (-D + m) * in(x-mu)
template <typename Float, int nDim, int nColor, typename Arg>
__device__ __host__ inline void laplace(Arg &arg, int x_cb, int parity)
{
  typedef ColorSpinor<Float,nColor,1> Vector;
  Vector out;

  applyLaplace<Float,nDim,nColor>(out, arg, x_cb, parity);

  if (arg.isXpay()) {
    Vector x = arg.x(x_cb, parity);
    out = x + arg.kappa * out;
  }
  arg.out(x_cb, parity) = out;
}
```

```
// CPU kernel for applying the Laplace operator to a vector
template <typename Float, int nDim, int nColor, typename Arg>
void laplaceCPU(Arg arg)
{

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

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

}
```

```
// GPU Kernel for applying the Laplace operator to a vector
template <typename Float, int nDim, int nColor, typename Arg>
__global__ void laplaceGPU(Arg arg)
{
  int x_cb = blockIdx.x*blockDim.x + threadIdx.x;

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

  if (x_cb >= arg.volumeCB) return;
  if (parity >= arg.nParity) return;

  laplace<Float,nDim,nColor>(arg, x_cb, parity);
}
```

```
/**
   @brief Applies the off-diagonal part of the Laplace operator

   @param[out] out The out result field
   @param[in] U The gauge field
   @param[in] kappa Kappa value
   @param[in] in The input field
   @param[in] parity The site parity
   @param[in] x_cb The checker-boarded site index
 */
extern __shared__ float s[];
template <typename Float, int nDim, int nColor, typename Vector, typename Arg>
__device__ __host__ inline void applyLaplace(Vector &out, Arg &arg, int x_cb, int parity) {
  typedef Matrix<complex<Float>,nColor> Link;
  const int their_spinor_parity = (arg.nParity == 2) ? 1-parity : 0;

  int coord[nDim];
  getCoords(coord, x_cb, arg.dim, parity);

#pragma unroll
  for (int d = 0; d<nDim; d++) // loop over dimension
  {
    //Forward gather - compute fwd offset for vector fetch
    const int fwd_idx = linkIndexP1(coord, arg.dim, d);

    if ( arg.commDim[d] && (coord[d] + arg.nFace >= arg.dim[d]) ) {
      const int ghost_idx = ghostFaceIndex<1>(coord, arg.dim, d, arg.nFace);

      const Link U = arg.U(d, x_cb, parity);
      const Vector in = arg.in.Ghost(d, 1, ghost_idx, their_spinor_parity);

      out += U * in;
    } else {

      const Link U = arg.U(d, x_cb, parity);
      const Vector in = arg.in(fwd_idx, their_spinor_parity);

      out += U * in;
    }

    //Backward gather - compute back offset for spinor and gauge fetch
    const int back_idx = linkIndexM1(coord, arg.dim, d);
    const int gauge_idx = back_idx;

    if ( arg.commDim[d] && (coord[d] - arg.nFace < 0) ) {
      const int ghost_idx = ghostFaceIndex<0>(coord, arg.dim, d, arg.nFace);

      const Link U = arg.U.Ghost(d, ghost_idx, 1-parity);
      const Vector in = arg.in.Ghost(d, 0, ghost_idx, their_spinor_parity);

      out += conj(U) * in;
    } else {

      const Link U = arg.U(d, gauge_idx, 1-parity);
      const Vector in = arg.in(back_idx, their_spinor_parity);

      out += conj(U) * in;
    }
  } //nDim

}
```

LAPLACE KERNEL

```
//out(x) = M*in = (-D + m) * in(x-mu)
template <typename Float, int nDim, int nColor, typename Arg>
__device__ __host__ inline void laplace(Arg &arg, int x_cb, int parity)
{
  typedef ColorSpinor<Float,nColor,1> Vector;
  Vector out;

  applyLaplace<Float,nDim,nColor>(out, arg, x_cb, parity);

  if (arg.isXpay()) {
    Vector x = arg.x(x_cb, parity);
    out = x + arg.kappa * out;
  }
  arg.out(x_cb, parity) = out;
}

// CPU kernel for applying the Laplace operator to a vector
template <typename Float, int nDim, int nColor, typename Arg>
void laplaceCPU(Arg arg)
{

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

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

}

// GPU Kernel for applying the Laplace operator to a vector
template <typename Float, int nDim, int nColor, typename Arg>
__global__ void laplaceGPU(Arg arg)
{
  int x_cb = blockIdx.x*blockDim.x + threadIdx.x;

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

  if (x_cb >= arg.volumeCB) return;
  if (parity >= arg.nParity) return;

  laplace<Float,nDim,nColor>(arg, x_cb, parity);
}
```

LAPLACE KERNEL

```
/**
   @brief Applies the off-diagonal part of the Laplace operator

   @param[out] out The out result field
   @param[in] U The gauge field
   @param[in] kappa Kappa value
   @param[in] in The input field
   @param[in] parity The site parity
   @param[in] x_cb The checker-boarded site index
 */
extern __shared__ float s[];
template <typename Float, int nDim, int nColor, typename Vector, typename Arg>
__device__ __host__ inline void applyLaplace(Vector &out, Arg &arg, int x_cb, int parity) {
  typedef Matrix<complex<Float>,nColor> Link;
  const int their_spinor_parity = (arg.nParity == 2) ? 1-parity : 0;

  int coord[nDim];
  getCoords(coord, x_cb, arg.dim, parity);

#pragma unroll
  for (int d = 0; d<nDim; d++) // loop over dimension
  {
    //Forward gather - compute fwd offset for vector fetch
    const int fwd_idx = linkIndexP1(coord, arg.dim, d);

    if ( arg.commDim[d] && (coord[d] + arg.nFace >= arg.dim[d]) ) {
      const int ghost_idx = ghostFaceIndex<1>(coord, arg.dim, d, arg.nFace);

      const Link U = arg.U(d, x_cb, parity);
      const Vector in = arg.in.Ghost(d, 1, ghost_idx, their_spinor_parity);

      out += U * in;
    } else {

      const Link U = arg.U(d, x_cb, parity);
      const Vector in = arg.in(fwd_idx, their_spinor_parity);

      out += U * in;
    }

    //Backward gather - compute back offset for spinor and gauge fetch
    const int back_idx = linkIndexM1(coord, arg.dim, d);
    const int gauge_idx = back_idx;

    if ( arg.commDim[d] && (coord[d] - arg.nFace < 0) ) {
      const int ghost_idx = ghostFaceIndex<0>(coord, arg.dim, d, arg.nFace);

      const Link U = arg.U.Ghost(d, ghost_idx, 1-parity);
      const Vector in = arg.in.Ghost(d, 0, ghost_idx, their_spinor_parity);

      out += conj(U) * in;
    } else {

      const Link U = arg.U(d, gauge_idx, 1-parity);
      const Vector in = arg.in(back_idx, their_spinor_parity);

      out += conj(U) * in;
    }
  } //nDim

}
```

```
//out(x) = M*in = (-D + m) * in(x-mu)
template <typename Float, int nDim, int nColor, typename Arg>
__device__ __host__ inline void laplace(Arg &arg, int x_cb, int parity)
{
  typedef ColorSpinor<Float,nColor,1> Vector;
  Vector out;

  applyLaplace<Float,nDim,nColor>(out, arg, x_cb, parity);

  if (arg.isXpay()) {
    Vector x = arg.x(x_cb, parity);
    out = x + arg.kappa * out;
  }
  arg.out(x_cb, parity) = out;
}

// CPU kernel for applying the Laplace operator to a vector
template <typename Float, int nDim, int nColor, typename Arg>
void laplaceCPU(Arg arg)
{

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

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

}
```

GPU Kernel

x_cb from x dim
parity from y dim

```
// GPU Kernel for applying the Laplace operator to a vector
template <typename Float, int nDim, int nColor, typename Arg>
__global__ void laplaceGPU(Arg arg)
{
  int x_cb = blockIdx.x*blockDim.x + threadIdx.x;

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

  if (x_cb >= arg.volumeCB) return;
  if (parity >= arg.nParity) return;

  laplace<Float,nDim,nColor>(arg, x_cb, parity);
}
```

```
/**
   @brief Applies the off-diagonal part of the Laplace operator

   @param[out] out The out result field
   @param[in] U The gauge field
   @param[in] kappa Kappa value
   @param[in] in The input field
   @param[in] parity The site parity
   @param[in] x_cb The checker-boarded site index
 */
extern __shared__ float s[];
template <typename Float, int nDim, int nColor, typename Vector, typename Arg>
__device__ __host__ inline void applyLaplace(Vector &out, Arg &arg, int x_cb, int parity) {
  typedef Matrix<complex<Float>,nColor> Link;
  const int their_spinor_parity = (arg.nParity == 2) ? 1-parity : 0;

  int coord[nDim];
  getCoords(coord, x_cb, arg.dim, parity);

#pragma unroll
  for (int d = 0; d<nDim; d++) // loop over dimension
  {
    //Forward gather - compute fwd offset for vector fetch
    const int fwd_idx = linkIndexP1(coord, arg.dim, d);

    if ( arg.commDim[d] && (coord[d] + arg.nFace >= arg.dim[d]) ) {
      const int ghost_idx = ghostFaceIndex<1>(coord, arg.dim, d, arg.nFace);

      const Link U = arg.U(d, x_cb, parity);
      const Vector in = arg.in.Ghost(d, 1, ghost_idx, their_spinor_parity);

      out += U * in;
    } else {

      const Link U = arg.U(d, x_cb, parity);
      const Vector in = arg.in(fwd_idx, their_spinor_parity);

      out += U * in;
    }

    //Backward gather - compute back offset for spinor and gauge fetch
    const int back_idx = linkIndexM1(coord, arg.dim, d);
    const int gauge_idx = back_idx;

    if ( arg.commDim[d] && (coord[d] - arg.nFace < 0) ) {
      const int ghost_idx = ghostFaceIndex<0>(coord, arg.dim, d, arg.nFace);

      const Link U = arg.U.Ghost(d, ghost_idx, 1-parity);
      const Vector in = arg.in.Ghost(d, 0, ghost_idx, their_spinor_parity);

      out += conj(U) * in;
    } else {

      const Link U = arg.U(d, gauge_idx, 1-parity);
      const Vector in = arg.in(back_idx, their_spinor_parity);

      out += conj(U) * in;
    }
  } //nDim

}
```

LAPLACE KERNEL

```
//out(x) = M*in = (-D + m) * in(x-mu)
template <typename Float, int nDim, int nColor, typename Arg>
__device__ __host__ inline void laplace(Arg &arg, int x_cb, int parity)
{
  typedef ColorSpinor<Float,nColor,1> Vector;
  Vector out;

  applyLaplace<Float,nDim,nColor>(out, arg, x_cb, parity);

  if (arg.isXpay()) {
    Vector x = arg.x(x_cb, parity);
    out = x + arg.kappa * out;
  }
  arg.out(x_cb, parity) = out;
}

// CPU kernel for applying the Laplace operator to a vector
template <typename Float, int nDim, int nColor, typename Arg>
void laplaceCPU(Arg arg)
{

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

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

}

// GPU Kernel for applying the Laplace operator to a vector
template <typename Float, int nDim, int nColor, typename Arg>
__global__ void laplaceGPU(Arg arg)
{
  int x_cb = blockIdx.x*blockDim.x + threadIdx.x;

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

  if (x_cb >= arg.volumeCB) return;
  if (parity >= arg.nParity) return;

  laplace<Float,nDim,nColor>(arg, x_cb, parity);
}
```

```
/**
   @brief Applies the off-diagonal part of the Laplace operator

   @param[out] out The out result field
   @param[in] U The gauge field
   @param[in] kappa Kappa value
   @param[in] in The input field
   @param[in] parity The site parity
   @param[in] x_cb The checker-boarded site index
 */
extern __shared__ float s[];
template <typename Float, int nDim, int nColor, typename Vector, typename Arg>
__device__ __host__ inline void applyLaplace(Vector &out, Arg &arg, int x_cb, int parity) {
  typedef Matrix<complex<Float>,nColor> Link;
  const int their_spinor_parity = (arg.nParity == 2) ? 1-parity : 0;

  int coord[nDim];
  getCoords(coord, x_cb, arg.dim, parity);

#pragma unroll
  for (int d = 0; d<nDim; d++) // loop over dimension
  {
    //Forward gather - compute fwd offset for vector fetch
    const int fwd_idx = linkIndexP1(coord, arg.dim, d);

    if ( arg.commDim[d] && (coord[d] + arg.nFace >= arg.dim[d]) ) {
      const int ghost_idx = ghostFaceIndex<1>(coord, arg.dim, d, arg.nFace);

      const Link U = arg.U(d, x_cb, parity);
      const Vector in = arg.in.Ghost(d, 1, ghost_idx, their_spinor_parity);

      out += U * in;
    } else {

      const Link U = arg.U(d, x_cb, parity);
      const Vector in = arg.in(fwd_idx, their_spinor_parity);

      out += U * in;
    }

    //Backward gather - compute back offset for spinor and gauge fetch
    const int back_idx = linkIndexM1(coord, arg.dim, d);
    const int gauge_idx = back_idx;

    if ( arg.commDim[d] && (coord[d] - arg.nFace < 0) ) {
      const int ghost_idx = ghostFaceIndex<0>(coord, arg.dim, d, arg.nFace);

      const Link U = arg.U.Ghost(d, ghost_idx, 1-parity);
      const Vector in = arg.in.Ghost(d, 0, ghost_idx, their_spinor_parity);

      out += conj(U) * in;
    } else {

      const Link U = arg.U(d, gauge_idx, 1-parity);
      const Vector in = arg.in(back_idx, their_spinor_parity);

      out += conj(U) * in;
    }
  } //nDim

}
```

Compute grid indices

Forward gather

Backward gather

LAPLACE KERNEL

```
//out(x) = M*in = (-D + m) * in(x-mu)
template <typename Float, int nDim, int nColor, typename Arg>
__device__ __host__ inline void laplace(Arg &arg, int x_cb, int parity)
{
  typedef ColorSpinor<Float,nColor,1> Vector;
  Vector out;

  applyLaplace<Float,nDim,nColor>(out, arg, x_cb, parity);

  if (arg.isXpay()) {
    Vector x = arg.x(x_cb, parity);
    out = x + arg.kappa * out;
  }
  arg.out(x_cb, parity) = out;
}

// CPU kernel for applying the Laplace operator to a vector
template <typename Float, int nDim, int nColor, typename Arg>
void laplaceCPU(Arg arg)
{

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

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

}

// GPU Kernel for applying the Laplace operator to a vector
template <typename Float, int nDim, int nColor, typename Arg>
__global__ void laplaceGPU(Arg arg)
{
  int x_cb = blockIdx.x*blockDim.x + threadIdx.x;

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

  if (x_cb >= arg.volumeCB) return;
  if (parity >= arg.nParity) return;

  laplace<Float,nDim,nColor>(arg, x_cb, parity);
}
```

```
/**
   @brief Applies the off-diagonal part of the Laplace operator

   @param[out] out The out result field
   @param[in] U The gauge field
   @param[in] kappa Kappa value
   @param[in] in The input field
   @param[in] parity The site parity
   @param[in] x_cb The checker-boarded site index
 */
extern __shared__ float s[];
template <typename Float, int nDim, int nColor, typename Vector, typename Arg>
__device__ __host__ inline void applyLaplace(Vector &out, Arg &arg, int x_cb, int parity) {
  typedef Matrix<complex<Float>,nColor> Link;
  const int their_spinor_parity = (arg.nParity == 2) ? 1-parity : 0;

  int coord[nDim];
  getCoords(coord, x_cb, arg.dim, parity);

#pragma unroll
  for (int d = 0; d<nDim; d++) // loop over dimension
  {
    //Forward gather - compute fwd offset for vector fetch
    const int fwd_idx = linkIndexP1(coord, arg.dim, d);

    if ( arg.commDim[d] && (coord[d] + arg.nFace >= arg.dim[d]) ) {
      const int ghost_idx = ghostFaceIndex<1>(coord, arg.dim, d, arg.nFace);

      const Link U = arg.U(d, x_cb, parity);
      const Vector in = arg.in.Ghost(d, 1, ghost_idx, their_spinor_parity);

      out += U * in;
    } else {

      const Link U = arg.U(d, x_cb, parity);
      const Vector in = arg.in(fwd_idx, their_spinor_parity);

      out += U * in;
    }

    //Backward gather - compute back offset for spinor and gauge fetch
    const int back_idx = linkIndexM1(coord, arg.dim, d);
    const int gauge_idx = back_idx;

    if ( arg.commDim[d] && (coord[d] - arg.nFace < 0) ) {
      const int ghost_idx = ghostFaceIndex<0>(coord, arg.dim, d, arg.nFace);

      const Link U = arg.U.Ghost(d, ghost_idx, 1-parity);
      const Vector in = arg.in.Ghost(d, 0, ghost_idx, their_spinor_parity);

      out += conj(U) * in;
    } else {

      const Link U = arg.U(d, gauge_idx, 1-parity);
      const Vector in = arg.in(back_idx, their_spinor_parity);

      out += conj(U) * in;
    }
  } //nDim

}
```

LAPLACE KERNEL

```
//out(x) = M*in = (-D + m) * in(x-mu)
template <typename Float, int nDim, int nColor, typename Arg>
__device__ __host__ inline void laplace(Arg &arg, int x_cb, int parity)
{
  typedef ColorSpinor<Float,nColor,1> Vector;
  Vector out;

  applyLaplace<Float,nDim,nColor>(out, arg, x_cb, parity);

  if (arg.isXpay()) {
    Vector x = arg.x(x_cb, parity);
    out = x + arg.kappa * out;
  }
  arg.out(x_cb, parity) = out;
}

// CPU kernel for applying the Laplace operator to a vector
template <typename Float, int nDim, int nColor, typename Arg>
void laplaceCPU(Arg arg)
{

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

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

}

// GPU Kernel for applying the Laplace operator to a vector
template <typename Float, int nDim, int nColor, typename Arg>
__global__ void laplaceGPU(Arg arg)
{
  int x_cb = blockIdx.x*blockDim.x + threadIdx.x;

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

  if (x_cb >= arg.volumeCB) return;
  if (parity >= arg.nParity) return;

  laplace<Float,nDim,nColor>(arg, x_cb, parity);
}
```

LAPLACE KERNEL

```cpp
/**
   @brief Applies the off-diagonal part of the Laplace operator

   @param[out] out The out result field
   @param[in] U The gauge field
   @param[in] kappa Kappa value
   @param[in] in The input field
   @param[in] parity The site parity
   @param[in] x_cb The checker-boarded site index
 */
extern __shared__ float s[];
template <typename Float, int nDim, int nColor, typename Vector, typename Arg>
__device__ __host__ inline void applyLaplace(Vector &out, Arg &arg, int x_cb, int parity) {
  typedef Matrix<complex<Float>,nColor> Link;
  const int their_spinor_parity = (arg.nParity == 2) ? 1-parity : 0;

  int coord[nDim];
  getCoords(coord, x_cb, arg.dim, parity);

#pragma unroll
  for (int d = 0; d<nDim; d++) // loop over dimension
  {
    //Forward gather - compute fwd offset for vector fetch
    const int fwd_idx = linkIndexP1(coord, arg.dim, d);

    if ( arg.commDim[d] && (coord[d] + arg.nFace >= arg.dim[d]) ) {
      const int ghost_idx = ghostFaceIndex<1>(coord, arg.dim, d, arg.nFace);

      const Link U = arg.U(d, x_cb, parity);
      const Vector in = arg.in.Ghost(d, 1, ghost_idx, their_spinor_parity);

      out += U * in;
    } else {
      const Link U = arg.U(d, x_cb, parity);
      const Vector in = arg.in(fwd_idx, their_spinor_parity);

      out += U * in;
    }

    //Backward gather - compute back offset for spinor and gauge fetch
    const int back_idx = linkIndexM1(coord, arg.dim, d);
    const int gauge_idx = back_idx;

    if ( arg.commDim[d] && (coord[d] - arg.nFace < 0) ) {
      const int ghost_idx = ghostFaceIndex<0>(coord, arg.dim, d, arg.nFace);

      const Link U = arg.U.Ghost(d, ghost_idx, 1-parity);
      const Vector in = arg.in.Ghost(d, 0, ghost_idx, their_spinor_parity);

      out += conj(U) * in;
    } else {

      const Link U = arg.U(d, gauge_idx, 1-parity);
      const Vector in = arg.in(back_idx, their_spinor_parity);

      out += conj(U) * in;
    }
  } //nDim

}
```

Read accessors

Ghost accessors

Matrix-vector

Write accessor

```cpp
//out(x) = M*in = (-D + m) * in(x-mu)
template <typename Float, int nDim, int nColor, typename Arg>
__device__ __host__ inline void laplace(Arg &arg, int x_cb, int parity)
{
  typedef ColorSpinor<Float,nColor,1> Vector;
  Vector out;

  applyLaplace<Float,nDim,nColor>(out, arg, x_cb, parity);

  if (arg.isXpay()) {
    Vector x = arg.x(x_cb, parity);
    out = x + arg.kappa * out;
  }
  arg.out(x_cb, parity) = out;
}

// CPU kernel for applying the Laplace operator to a vector
template <typename Float, int nDim, int nColor, typename Arg>
void laplaceCPU(Arg arg)
{

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

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

}

// GPU Kernel for applying the Laplace operator to a vector
template <typename Float, int nDim, int nColor, typename Arg>
__global__ void laplaceGPU(Arg arg)
{
  int x_cb = blockIdx.x*blockDim.x + threadIdx.x;

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

  if (x_cb >= arg.volumeCB) return;
  if (parity >= arg.nParity) return;

  laplace<Float,nDim,nColor>(arg, x_cb, parity);
}
```

```
/**
   @brief Applies the off-diagonal part of the Laplace operator

   @param[out] out The out result field
   @param[in] U The gauge field
   @param[in] kappa Kappa value
   @param[in] in The input field
   @param[in] parity The site parity
   @param[in] x_cb The checker-boarded site index
 */
extern __shared__ float s[];
template <typename Float, int nDim, int nColor, typename Vector, typename Arg>
__device__ __host__ inline void applyLaplace(Vector &out, Arg &arg, int x_cb, int parity) {
  typedef Matrix<complex<Float>,nColor> Link;
  const int their_spinor_parity = (arg.nParity == 2) ? 1-parity : 0;

  int coord[nDim];
  getCoords(coord, x_cb, arg.dim, parity);

#pragma unroll
  for (int d = 0; d<nDim; d++) // loop over dimension
  {
    //Forward gather - compute fwd offset for vector fetch
    const int fwd_idx = linkIndexP1(coord, arg.dim, d);

    if ( arg.commDim[d] && (coord[d] + arg.nFace >= arg.dim[d]) ) {
      const int ghost_idx = ghostFaceIndex<1>(coord, arg.dim, d, arg.nFace);

      const Link U = arg.U(d, x_cb, parity);
      const Vector in = arg.in.Ghost(d, 1, ghost_idx, their_spinor_parity);

      out += U * in;
    } else {

      const Link U = arg.U(d, x_cb, parity);
      const Vector in = arg.in(fwd_idx, their_spinor_parity);

      out += U * in;
    }

    //Backward gather - compute back offset for spinor and gauge fetch
    const int back_idx = linkIndexM1(coord, arg.dim, d);
    const int gauge_idx = back_idx;

    if ( arg.commDim[d] && (coord[d] - arg.nFace < 0) ) {
      const int ghost_idx = ghostFaceIndex<0>(coord, arg.dim, d, arg.nFace);

      const Link U = arg.U.Ghost(d, ghost_idx, 1-parity);
      const Vector in = arg.in.Ghost(d, 0, ghost_idx, their_spinor_parity);

      out += conj(U) * in;
    } else {

      const Link U = arg.U(d, gauge_idx, 1-parity);
      const Vector in = arg.in(back_idx, their_spinor_parity);

      out += conj(U) * in;
    }
  } //nDim

}
```

LAPLACE KERNEL

```
//out(x) = M*in = (-D + m) * in(x-mu)
template <typename Float, int nDim, int nColor, typename Arg>
__device__ __host__ inline void laplace(Arg &arg, int x_cb, int parity)
{
  typedef ColorSpinor<Float,nColor,1> Vector;
  Vector out;

  applyLaplace<Float,nDim,nColor>(out, arg, x_cb, parity);

  if (arg.isXpay()) {
    Vector x = arg.x(x_cb, parity);
    out = x + arg.kappa * out;
  }
  arg.out(x_cb, parity) = out;
}

// CPU kernel for applying the Laplace operator to a vector
template <typename Float, int nDim, int nColor, typename Arg>
void laplaceCPU(Arg arg)
{

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

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

}

// GPU Kernel for applying the Laplace operator to a vector
template <typename Float, int nDim, int nColor, typename Arg>
__global__ void laplaceGPU(Arg arg)
{
  int x_cb = blockIdx.x*blockDim.x + threadIdx.x;

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

  if (x_cb >= arg.volumeCB) return;
  if (parity >= arg.nParity) return;

  laplace<Float,nDim,nColor>(arg, x_cb, parity);
}
```

ENTRY POINT

```
//Apply the Laplace operator
//out(x) = M*in = - kappa*\sum_mu U_{-\mu}(x)in(x+mu) + U^\dagger_mu(x-mu)in(x-mu)
//Uses the kappa normalization for the Wilson operator.
void ApplyLaplace(ColorSpinorField &out, const ColorSpinorField &in, const GaugeField &U,
                  double kappa, const ColorSpinorField *x, int parity)
{
  if (in.V() == out.V()) errorQuda("Aliasing pointers");
  Precision(in, out, U));  // check all precisions match
  Location(out, in, U);    // check all locations match

  in.exchangeGhost(1-parity);

  if (dslash::aux_worker) dslash::aux_worker->apply(0);

  if (U.Precision() == QUDA_DOUBLE_PRECISION) {
    ApplyLaplace<double>(out, in, U, kappa, x, parity);
  } else if (U.Precision() == QUDA_SINGLE_PRECISION) {
    ApplyLaplace<float>(out, in, U, kappa, x, parity);
  } else {
    errorQuda("Unsupported precision %d\n", U.Precision());
  }
}
```

Initial runtime checks

Halo exchange

Auxiliary worker

Template on precision

TEMPLATE INSTANTIATION

```
// template on the number of colors
template <typename Float>
  void ApplyLaplace(ColorSpinorField &out, const ColorSpinorField &in, const GaugeField &U,
                    double kappa, const ColorSpinorField *x, int parity)
{
  if (in.Ncolor() == 3) {
    ApplyLaplace<Float,3>(out, in, U, kappa, x, parity);
  } else {
    errorQuda("Unsupported number of colors %d\n", U.Ncolor());
  }
}
```

Template on Nc

```
// template on the gauge reconstruction
template <typename Float, int nColor>
  void ApplyLaplace(ColorSpinorField &out, const ColorSpinorField &in, const GaugeField &U,
                    double kappa, const ColorSpinorField *x, int parity)
{
  if (U.Reconstruct()== QUDA_RECONSTRUCT_NO) {
    ApplyLaplace<Float,nColor,QUDA_RECONSTRUCT_NO>(out, in, U, kappa, x, parity);
  } else if (U.Reconstruct()== QUDA_RECONSTRUCT_12) {
    ApplyLaplace<Float,nColor,QUDA_RECONSTRUCT_12>(out, in, U, kappa, x, parity);
  } else if (U.Reconstruct()== QUDA_RECONSTRUCT_8) {
    ApplyLaplace<Float,nColor,QUDA_RECONSTRUCT_8>(out, in, U, kappa, x, parity);
  } else {
    errorQuda("Unsupported reconstruct type %d\n", U.Reconstruct());
  }
}
```

Template on gauge reconstruct

```
template <typename Float, int nColor, QudaReconstructType recon>
  void ApplyLaplace(ColorSpinorField &out, const ColorSpinorField &in, const GaugeField &U,
                    double kappa, const ColorSpinorField *x, int parity)
{
  if (x) {
    ApplyLaplace<Float,nColor,recon,true>(out, in, U, kappa, x, parity);
  } else {
    ApplyLaplace<Float,nColor,recon,false>(out, in, U, kappa, x, parity);
  }
}
```

Template on (1 - kappa D) vs D

CREATE ARGUMENT STRUCT

- All kernels driven by a single argument struct
- All kernel meta data is baked into these using the instantiated templates
- Convert dynamic run-time fields into static accessors

```
/**
   @brief Parameter structure for driving the Laplace operator
 */
template <typename Float, int nColor, QudaReconstructType reconstruct, bool xpay>
struct LaplaceArg {
  typedef typename colorspinor_order_mapper<Float,QUDA_FLOAT2_FIELD_ORDER,1,nColor>::type F;
  typedef typename gauge_mapper<Float,reconstruct>::type G;

  F out;                // output vector field
  const F in;           // input vector field
  const F x;            // input vector when doing xpay
  const G U;            // the gauge field
  const Float kappa;    // kappa parameter = 1/(2d+m)
  const int parity;     // only use this for single parity fields
  const int nParity;    // number of parities we're working on
  const int nFace;      // hard code to 1 for now
  const int dim[4];     // full lattice dimensions
  const int commDim[4]; // whether a given dimension is partitioned or not
  const int volumeCB;   // checker-boarded volume

  __host__ __device__ static constexpr bool isXpay() { return xpay; }

  LaplaceArg(ColorSpinorField &out, const ColorSpinorField &in, const GaugeField &U,
       Float kappa, const ColorSpinorField *x, int parity)
    : out(out), in(in), U(U), kappa(kappa), x(xpay ? *x : in), parity(parity), nParity(in.SiteSubset()),
nFace(1),
      dim{ (3-nParity) * in.X(0), in.X(1), in.X(2), in.X(3) },
    commDim{comm_dim_partitioned(0), comm_dim_partitioned(1), comm_dim_partitioned(2), comm_dim_partitioned(3)},
    volumeCB(in.VolumeCB())
  {
    if (in.FieldOrder() != QUDA_FLOAT2_FIELD_ORDER || !U.isNative())
    errorQuda("Unsupported field order colorspinor=%d gauge=%d combination\n", in.FieldOrder(), U.FieldOrder());
  }
};

template <typename Float, int nColor, QudaReconstructType recon, bool xpay>
  void ApplyLaplace(ColorSpinorField &out, const ColorSpinorField &in, const GaugeField &U,
                    double kappa, const ColorSpinorField *x, int parity)
{
  constexpr int nDim = 4;
  LaplaceArg<Float,nColor,recon,xpay> arg(out, in, U, kappa, x, parity);
  Laplace<Float,nDim,nColor,LaplaceArg<Float,nColor,recon,xpay> > laplace(arg, in);
  laplace.apply(0);
}
```

CREATE ARGUMENT STRUCT

- All kernels driven by a single argument struct
- All kernel meta data is baked into these using the instantiated templates
- Convert dynamic run-time fields into static accessors

```
/**
   @brief Parameter structure for driving the Laplace operator
 */
template <typename Float, int nColor, QudaReconstructType reconstruct, bool xpay>
struct LaplaceArg {
  typedef typename colorspinor_order_mapper<Float,QUDA_FLOAT2_FIELD_ORDER,1,nColor>::type F;
  typedef typename gauge_mapper<Float,reconstruct>::type G;

  F out;                // output vector field
  const F in;           // input vector field
  const F x;            // input vector when doing xpay
  const G U;            // the gauge field
  const Float kappa;    // kappa parameter = 1/(2d+m)
  const int parity;     // only use this for single parity fields
  const int nParity;    // number of parities we're working on
  const int nFace;      // hard code to 1 for now
  const int dim[4];     // full lattice dimensions
  const int commDim[4]; // whether a given dimension is partitioned or not
  const int volumeCB;   // checker-boarded volume

  __host__ __device__ static constexpr bool isXpay() { return xpay; }

  LaplaceArg(ColorSpinorField &out, const ColorSpinorField &in, const GaugeField &U,
       Float kappa, const ColorSpinorField *x, int parity)
    : out(out), in(in), U(U), kappa(kappa), x(xpay ? *x : in), parity(parity), nParity(in.SiteSubset()),
nFace(1),
      dim{ (3-nParity) * in.X(0), in.X(1), in.X(2), in.X(3) },
    commDim{comm_dim_partitioned(0), comm_dim_partitioned(1), comm_dim_partitioned(2), comm_dim_partitioned(3)},
    volumeCB(in.VolumeCB())
  {
    if (in.FieldOrder() != QUDA_FLOAT2_FIELD_ORDER || !U.isNative())
    errorQuda("Unsupported field order colorspinor=%d gauge=%d combination\n", in.FieldOrder(), U.FieldOrder());
  }
};

template <typename Float, int nColor, QudaReconstructType recon, bool xpay>
  void ApplyLaplace(ColorSpinorField &out, const ColorSpinorField &in, const GaugeField &U,
                    double kappa, const ColorSpinorField *x, int parity)
{
  constexpr int nDim = 4;
  LaplaceArg<Float,nColor,recon,xpay> arg(out, in, U, kappa, x, parity);
  Laplace<Float,nDim,nColor,LaplaceArg<Float,nColor,recon,xpay> > laplace(arg, in);
  laplace.apply(0);
}
```

Create argument struct

CREATE ARGUMENT STRUCT

- All kernels driven by a single argument struct
- All kernel meta data is baked into these using the instantiated templates
- Convert dynamic run-time fields into static accessors

```
/**
   @brief Parameter structure for driving the Laplace operator
 */
template <typename Float, int nColor, QudaReconstructType reconstruct, bool xpay>
struct LaplaceArg {
  typedef typename colorspinor_order_mapper<Float,QUDA_FLOAT2_FIELD_ORDER,1,nColor>::type F;
  typedef typename gauge_mapper<Float,reconstruct>::type G;

  F out;                // output vector field
  const F in;           // input vector field
  const F x;            // input vector when doing xpay
  const G U;            // the gauge field
  const Float kappa;    // kappa parameter = 1/(2d+m)
  const int parity;     // only use this for single parity fields
  const int nParity;    // number of parities we're working on
  const int nFace;      // hard code to 1 for now
  const int dim[4];     // full lattice dimensions
  const int commDim[4]; // whether a given dimension is partitioned or not
  const int volumeCB;   // checker-boarded volume

  __host__ __device__ static constexpr bool isXpay() { return xpay; }

  LaplaceArg(ColorSpinorField &out, const ColorSpinorField &in, const GaugeField &U,
       Float kappa, const ColorSpinorField *x, int parity)
    : out(out), in(in), U(U), kappa(kappa), x(xpay ? *x : in), parity(parity), nParity(in.SiteSubset()),
nFace(1),
      dim{ (3-nParity) * in.X(0), in.X(1), in.X(2), in.X(3) },
    commDim{comm_dim_partitioned(0), comm_dim_partitioned(1), comm_dim_partitioned(2), comm_dim_partitioned(3)},
    volumeCB(in.VolumeCB())
  {
    if (in.FieldOrder() != QUDA_FLOAT2_FIELD_ORDER || !U.isNative())
    errorQuda("Unsupported field order colorspinor=%d gauge=%d combination\n", in.FieldOrder(), U.FieldOrder());
  }
};

template <typename Float, int nColor, QudaReconstructType recon, bool xpay>
  void ApplyLaplace(ColorSpinorField &out, const ColorSpinorField &in, const GaugeField &U,
                    double kappa, const ColorSpinorField *x, int parity)
{
  constexpr int nDim = 4;
  LaplaceArg<Float,nColor,recon,xpay> arg(out, in, U, kappa, x, parity);
  Laplace<Float,nDim,nColor,LaplaceArg<Float,nColor,recon,xpay> > laplace(arg, in);
  laplace.apply(0);
}
```

CREATE ARGUMENT STRUCT

- All kernels driven by a single argument struct
- All kernel meta data is baked into these using the instantiated templates
- Convert dynamic run-time fields into static accessors

Static accessors

Bind field into static accessor

```
/**
   @brief Parameter structure for driving the Laplace operator
 */
template <typename Float, int nColor, QudaReconstructType reconstruct, bool xpay>
struct LaplaceArg {
  typedef typename colorspinor_order_mapper<Float,QUDA_FLOAT2_FIELD_ORDER,1,nColor>::type F;
  typedef typename gauge_mapper<Float,reconstruct>::type G;

  F out;                // output vector field
  const F in;           // input vector field
  const F x;            // input vector when doing xpay
  const G U;            // the gauge field
  const Float kappa;    // kappa parameter = 1/(2d+m)
  const int parity;     // only use this for single parity fields
  const int nParity;    // number of parities we're working on
  const int nFace;      // hard code to 1 for now
  const int dim[4];     // full lattice dimensions
  const int commDim[4]; // whether a given dimension is partitioned or not
  const int volumeCB;   // checker-boarded volume

  __host__ __device__ static constexpr bool isXpay() { return xpay; }

  LaplaceArg(ColorSpinorField &out, const ColorSpinorField &in, const GaugeField &U,
             Float kappa, const ColorSpinorField *x, int parity)
    : out(out), in(in), U(U), kappa(kappa), x(xpay ? *x : in), parity(parity), nParity(in.SiteSubset()),
nFace(1),
      dim{ (3-nParity) * in.X(0), in.X(1), in.X(2), in.X(3) },
    commDim{comm_dim_partitioned(0), comm_dim_partitioned(1), comm_dim_partitioned(2), comm_dim_partitioned(3)},
    volumeCB(in.VolumeCB())
  {
    if (in.FieldOrder() != QUDA_FLOAT2_FIELD_ORDER || !U.isNative())
    errorQuda("Unsupported field order colorspinor=%d gauge=%d combination\n", in.FieldOrder(), U.FieldOrder());
  }
};

template <typename Float, int nColor, QudaReconstructType recon, bool xpay>
  void ApplyLaplace(ColorSpinorField &out, const ColorSpinorField &in, const GaugeField &U,
                    double kappa, const ColorSpinorField *x, int parity)
{
  constexpr int nDim = 4;
  LaplaceArg<Float,nColor,recon,xpay> arg(out, in, U, kappa, x, parity);
  Laplace<Float,nDim,nColor,LaplaceArg<Float,nColor,recon,xpay> > laplace(arg, in);
  laplace.apply(0);
}
```

CREATE ARGUMENT STRUCT

- All kernels driven by a single argument struct
- All kernel meta data is baked into these using the instantiated templates
- Convert dynamic run-time fields into static accessors

```
/**
   @brief Parameter structure for driving the Laplace operator
 */
template <typename Float, int nColor, QudaReconstructType reconstruct, bool xpay>
struct LaplaceArg {
  typedef typename colorspinor_order_mapper<Float,QUDA_FLOAT2_FIELD_ORDER,1,nColor>::type F;
  typedef typename gauge_mapper<Float,reconstruct>::type G;

  F out;                // output vector field
  const F in;           // input vector field
  const F x;            // input vector when doing xpay
  const G U;            // the gauge field
  const Float kappa;    // kappa parameter = 1/(2d+m)
  const int parity;     // only use this for single parity fields
  const int nParity;    // number of parities we're working on
  const int nFace;      // hard code to 1 for now
  const int dim[4];     // full lattice dimensions
  const int commDim[4]; // whether a given dimension is partitioned or not
  const int volumeCB;   // checker-boarded volume

  __host__ __device__ static constexpr bool isXpay() { return xpay; }

  LaplaceArg(ColorSpinorField &out, const ColorSpinorField &in, const GaugeField &U,
       Float kappa, const ColorSpinorField *x, int parity)
    : out(out), in(in), U(U), kappa(kappa), x(xpay ? *x : in), parity(parity), nParity(in.SiteSubset()),
nFace(1),
      dim{ (3-nParity) * in.X(0), in.X(1), in.X(2), in.X(3) },
    commDim{comm_dim_partitioned(0), comm_dim_partitioned(1), comm_dim_partitioned(2), comm_dim_partitioned(3)},
    volumeCB(in.VolumeCB())
  {
    if (in.FieldOrder() != QUDA_FLOAT2_FIELD_ORDER || !U.isNative())
    errorQuda("Unsupported field order colorspinor=%d gauge=%d combination\n", in.FieldOrder(), U.FieldOrder());
  }
};

template <typename Float, int nColor, QudaReconstructType recon, bool xpay>
  void ApplyLaplace(ColorSpinorField &out, const ColorSpinorField &in, const GaugeField &U,
                    double kappa, const ColorSpinorField *x, int parity)
{
  constexpr int nDim = 4;
  LaplaceArg<Float,nColor,recon,xpay> arg(out, in, U, kappa, x, parity);
  Laplace<Float,nDim,nColor,LaplaceArg<Float,nColor,recon,xpay> > laplace(arg, in);
  laplace.apply(0);
}
```

```
template <typename Float, int nDim, int nColor, typename Arg>
class Laplace : public TunableVectorY {

  protected:
    Arg &arg;
    const ColorSpinorField &meta;

    long long flops() const
    {
      return (2*nDim*(8*nColor*nColor)-2*nColor + (arg.isXpay() ? 2*2*nColor : 0) )
             *arg.nParity*(long long)meta.VolumeCB();
    }
    long long bytes() const
    {
      return arg.out.Bytes() + 2*nDim*arg.in.Bytes() +
             arg.nParity*2*nDim*arg.U.Bytes()*meta.VolumeCB() + (arg.isXpay() ? arg.x.Bytes() : 0);
    }
    bool tuneGridDim() const { return false; }
    unsigned int minThreads() const { return arg.volumeCB; }

  public:
    Laplace(Arg &arg, const ColorSpinorField &meta) : TunableVectorY(arg.nParity), arg(arg), meta(meta)
    {
      strcpy(aux, meta.AuxString());
      strcat(aux, comm_dim_partitioned_string());
      if (arg.isXpay()) strcat(aux,",xpay");
    }
    virtual ~Laplace() { }

    void apply(const cudaStream_t &stream) {
      if (meta.Location() == QUDA_CPU_FIELD_LOCATION) {
        laplaceCPU<Float,nDim,nColor>(arg);
      } else {
        TuneParam tp = tuneLaunch(*this, getTuning(), getVerbosity());
        laplaceGPU<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, int nColor, QudaReconstructType recon, bool xpay>
    void ApplyLaplace(ColorSpinorField &out, const ColorSpinorField &in, const GaugeField &U,
                      double kappa, const ColorSpinorField *x, int parity)
  {
    constexpr int nDim = 4;
    LaplaceArg<Float,nColor,recon,xpay> arg(out, in, U, kappa, x, parity);
    Laplace<Float,nDim,nColor,LaplaceArg<Float,nColor,recon,xpay> > laplace(arg, in);
    laplace.apply(0);
  }
```

AUTOTUNING

- Kernels use launcher class derived from “Tunable”
- Provides autotuning for work distribution, size of thread blocks, number of thread blocks, etc.
- Ensures optimal performance
- Tuned parameters are stored in a std::map and dumped to disk for later reuse

```
template <typename Float, int nDim, int nColor, typename Arg>
class Laplace : public TunableVectorY {

  protected:
    Arg &arg;
    const ColorSpinorField &meta;

    long long flops() const
    {
      return (2*nDim*(8*nColor*nColor)-2*nColor + (arg.isXpay() ? 2*2*nColor : 0) )
             *arg.nParity*(long long)meta.VolumeCB();
    }
    long long bytes() const
    {
      return arg.out.Bytes() + 2*nDim*arg.in.Bytes() +
             arg.nParity*2*nDim*arg.U.Bytes()*meta.VolumeCB() + (arg.isXpay() ? arg.x.Bytes() : 0);
    }
    bool tuneGridDim() const { return false; }
    unsigned int minThreads() const { return arg.volumeCB; }

  public:
    Laplace(Arg &arg, const ColorSpinorField &meta) : TunableVectorY(arg.nParity), arg(arg), meta(meta)
    {
      strcpy(aux, meta.AuxString());
      strcat(aux, comm_dim_partitioned_string());
      if (arg.isXpay()) strcat(aux,",xpay");
    }
    virtual ~Laplace() { }

    void apply(const cudaStream_t &stream) {
      if (meta.Location() == QUDA_CPU_FIELD_LOCATION) {
        laplaceCPU<Float,nDim,nColor>(arg);
      } else {
        TuneParam tp = tuneLaunch(*this, getTuning(), getVerbosity());
        laplaceGPU<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, int nColor, QudaReconstructType recon, bool xpay>
    void ApplyLaplace(ColorSpinorField &out, const ColorSpinorField &in, const GaugeField &U,
                      double kappa, const ColorSpinorField *x, int parity)
  {
    constexpr int nDim = 4;
    LaplaceArg<Float,nColor,recon,xpay> arg(out, in, U, kappa, x, parity);
    Laplace<Float,nDim,nColor,LaplaceArg<Float,nColor,recon,xpay> > laplace(arg, in);
    laplace.apply(0);
  }
```

AUTOTUNING

- Kernels use launcher class derived from “Tunable”
- Provides autotuning for work distribution, size of thread blocks, number of thread blocks, etc.
- Ensures optimal performance
- Tuned parameters are stored in a std::map and dumped to disk for later reuse

1. Create launcher class instance
2. Launch work

```
template <typename Float, int nDim, int nColor, typename Arg>
class Laplace : public TunableVectorY {

  protected:
    Arg &arg;
    const ColorSpinorField &meta;

    long long flops() const
    {
      return (2*nDim*(8*nColor*nColor)-2*nColor + (arg.isXpay() ? 2*2*nColor : 0) )
             *arg.nParity*(long long)meta.VolumeCB();
    }
    long long bytes() const
    {
      return arg.out.Bytes() + 2*nDim*arg.in.Bytes() +
             arg.nParity*2*nDim*arg.U.Bytes()*meta.VolumeCB() + (arg.isXpay() ? arg.x.Bytes() : 0);
    }
    bool tuneGridDim() const { return false; }
    unsigned int minThreads() const { return arg.volumeCB; }

  public:
    Laplace(Arg &arg, const ColorSpinorField &meta) : TunableVectorY(arg.nParity), arg(arg), meta(meta)
    {
      strcpy(aux, meta.AuxString());
      strcat(aux, comm_dim_partitioned_string());
      if (arg.isXpay()) strcat(aux,",xpay");
    }
    virtual ~Laplace() { }

    void apply(const cudaStream_t &stream) {
      if (meta.Location() == QUDA_CPU_FIELD_LOCATION) {
        laplaceCPU<Float,nDim,nColor>(arg);
      } else {
        TuneParam tp = tuneLaunch(*this, getTuning(), getVerbosity());
        laplaceGPU<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, int nColor, QudaReconstructType recon, bool xpay>
    void ApplyLaplace(ColorSpinorField &out, const ColorSpinorField &in, const GaugeField &U,
                      double kappa, const ColorSpinorField *x, int parity)
  {
    constexpr int nDim = 4;
    LaplaceArg<Float,nColor,recon,xpay> arg(out, in, U, kappa, x, parity);
    Laplace<Float,nDim,nColor,LaplaceArg<Float,nColor,recon,xpay> > laplace(arg, in);
    laplace.apply(0);
  }
```

AUTOTUNING

- Kernels use launcher class derived from “Tunable”
- Provides autotuning for work distribution, size of thread blocks, number of thread blocks, etc.
- Ensures optimal performance
- Tuned parameters are stored in a std::map and dumped to disk for later reuse

```
template <typename Float, int nDim, int nColor, typename Arg>
class Laplace : public TunableVectorY {

  protected:
    Arg &arg;
    const ColorSpinorField &meta;

    long long flops() const
    {
      return (2*nDim*(8*nColor*nColor)-2*nColor + (arg.isXpay() ? 2*2*nColor : 0) )
             *arg.nParity*(long long)meta.VolumeCB();
    }
    long long bytes() const
    {
      return arg.out.Bytes() + 2*nDim*arg.in.Bytes() +
             arg.nParity*2*nDim*arg.U.Bytes()*meta.VolumeCB() + (arg.isXpay() ? arg.x.Bytes() : 0);
    }
    bool tuneGridDim() const { return false; }
    unsigned int minThreads() const { return arg.volumeCB; }

  public:
    Laplace(Arg &arg, const ColorSpinorField &meta) : TunableVectorY(arg.nParity), arg(arg), meta(meta)
    {
      strcpy(aux, meta.AuxString());
      strcat(aux, comm_dim_partitioned_string());
      if (arg.isXpay()) strcat(aux,",xpay");
    }
    virtual ~Laplace() { }

    void apply(const cudaStream_t &stream) {
      if (meta.Location() == QUDA_CPU_FIELD_LOCATION) {
        laplaceCPU<Float,nDim,nColor>(arg);
      } else {
        TuneParam tp = tuneLaunch(*this, getTuning(), getVerbosity());
        laplaceGPU<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, int nColor, QudaReconstructType recon, bool xpay>
    void ApplyLaplace(ColorSpinorField &out, const ColorSpinorField &in, const GaugeField &U,
                      double kappa, const ColorSpinorField *x, int parity)
  {
    constexpr int nDim = 4;
    LaplaceArg<Float,nColor,recon,xpay> arg(out, in, U, kappa, x, parity);
    Laplace<Float,nDim,nColor,LaplaceArg<Float,nColor,recon,xpay> > laplace(arg, in);
    laplace.apply(0);
  }
```

AUTOTUNING

- Kernels use launcher class derived from “Tunable”
- Provides autotuning for work distribution, size of thread blocks, number of thread blocks, etc.
- Ensures optimal performance
- Tuned parameters are stored in a std::map and dumped to disk for later reuse

Unique “TuneKey” is for every kernel parameter set

```
template <typename Float, int nDim, int nColor, typename Arg>
class Laplace : public TunableVectorY {

  protected:
    Arg &arg;
    const ColorSpinorField &meta;

    long long flops() const
    {
      return (2*nDim*(8*nColor*nColor)-2*nColor + (arg.isXpay() ? 2*2*nColor : 0) )
             *arg.nParity*(long long)meta.VolumeCB();
    }
    long long bytes() const
    {
      return arg.out.Bytes() + 2*nDim*arg.in.Bytes() +
             arg.nParity*2*nDim*arg.U.Bytes()*meta.VolumeCB() + (arg.isXpay() ? arg.x.Bytes() : 0);
    }
    bool tuneGridDim() const { return false; }
    unsigned int minThreads() const { return arg.volumeCB; }

  public:
    Laplace(Arg &arg, const ColorSpinorField &meta) : TunableVectorY(arg.nParity), arg(arg), meta(meta)
    {
      strcpy(aux, meta.AuxString());
      strcat(aux, comm_dim_partitioned_string());
      if (arg.isXpay()) strcat(aux,",xpay");
    }
    virtual ~Laplace() { }

    void apply(const cudaStream_t &stream) {
      if (meta.Location() == QUDA_CPU_FIELD_LOCATION) {
        laplaceCPU<Float,nDim,nColor>(arg);
      } else {
        TuneParam tp = tuneLaunch(*this, getTuning(), getVerbosity());
        laplaceGPU<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, int nColor, QudaReconstructType recon, bool xpay>
    void ApplyLaplace(ColorSpinorField &out, const ColorSpinorField &in, const GaugeField &U,
                      double kappa, const ColorSpinorField *x, int parity)
  {
    constexpr int nDim = 4;
    LaplaceArg<Float,nColor,recon,xpay> arg(out, in, U, kappa, x, parity);
    Laplace<Float,nDim,nColor,LaplaceArg<Float,nColor,recon,xpay> > laplace(arg, in);
    laplace.apply(0);
  }
```

AUTOTUNING

- Kernels use launcher class derived from “Tunable”
- Provides autotuning for work distribution, size of thread blocks, number of thread blocks, etc.
- Ensures optimal performance
- Tuned parameters are stored in a std::map and dumped to disk for later reuse

```
template <typename Float, int nDim, int nColor, typename Arg>
class Laplace : public TunableVectorY {

  protected:
    Arg &arg;
    const ColorSpinorField &meta;

    long long flops() const
    {
      return (2*nDim*(8*nColor*nColor)-2*nColor + (arg.isXpay() ? 2*2*nColor : 0) )
             *arg.nParity*(long long)meta.VolumeCB();
    }
    long long bytes() const
    {
      return arg.out.Bytes() + 2*nDim*arg.in.Bytes() +
             arg.nParity*2*nDim*arg.U.Bytes()*meta.VolumeCB() + (arg.isXpay() ? arg.x.Bytes() : 0);
    }
    bool tuneGridDim() const { return false; }
    unsigned int minThreads() const { return arg.volumeCB; }
  public:
    Laplace(Arg &arg, const ColorSpinorField &meta) : TunableVectorY(arg.nParity), arg(arg), meta(meta)
    {
      strcpy(aux, meta.AuxString());
      strcat(aux, comm_dim_partitioned_string());
      if (arg.isXpay()) strcat(aux,",xpay");
    }
    virtual ~Laplace() { }

    void apply(const cudaStream_t &stream) {
      if (meta.Location() == QUDA_CPU_FIELD_LOCATION) {
        laplaceCPU<Float,nDim,nColor>(arg);
      } else {
        TuneParam tp = tuneLaunch(*this, getTuning(), getVerbosity());
        laplaceGPU<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, int nColor, QudaReconstructType recon, bool xpay>
    void ApplyLaplace(ColorSpinorField &out, const ColorSpinorField &in, const GaugeField &U,
                      double kappa, const ColorSpinorField *x, int parity)
  {
    constexpr int nDim = 4;
    LaplaceArg<Float,nColor,recon,xpay> arg(out, in, U, kappa, x, parity);
    Laplace<Float,nDim,nColor,LaplaceArg<Float,nColor,recon,xpay> > laplace(arg, in);
    laplace.apply(0);
  }
```

Tuning metadata

AUTOTUNING

- Kernels use launcher class derived from “Tunable”
- Provides autotuning for work distribution, size of thread blocks, number of thread blocks, etc.
- Ensures optimal performance
- Tuned parameters are stored in a std::map and dumped to disk for later reuse

```
template <typename Float, int nDim, int nColor, typename Arg>
class Laplace : public TunableVectorY {

  protected:
    Arg &arg;
    const ColorSpinorField &meta;

    long long flops() const
    {
      return (2*nDim*(8*nColor*nColor)-2*nColor + (arg.isXpay() ? 2*2*nColor : 0) )
             *arg.nParity*(long long)meta.VolumeCB();
    }
    long long bytes() const
    {
      return arg.out.Bytes() + 2*nDim*arg.in.Bytes() +
             arg.nParity*2*nDim*arg.U.Bytes()*meta.VolumeCB() + (arg.isXpay() ? arg.x.Bytes() : 0);
    }
    bool tuneGridDim() const { return false; }
    unsigned int minThreads() const { return arg.volumeCB; }

  public:
    Laplace(Arg &arg, const ColorSpinorField &meta) : TunableVectorY(arg.nParity), arg(arg), meta(meta)
    {
      strcpy(aux, meta.AuxString());
      strcat(aux, comm_dim_partitioned_string());
      if (arg.isXpay()) strcat(aux,",xpay");
    }
    virtual ~Laplace() { }

    void apply(const cudaStream_t &stream) {
      if (meta.Location() == QUDA_CPU_FIELD_LOCATION) {
        laplaceCPU<Float,nDim,nColor>(arg);
      } else {
        TuneParam tp = tuneLaunch(*this, getTuning(), getVerbosity());
        laplaceGPU<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, int nColor, QudaReconstructType recon, bool xpay>
    void ApplyLaplace(ColorSpinorField &out, const ColorSpinorField &in, const GaugeField &U,
                      double kappa, const ColorSpinorField *x, int parity)
  {
    constexpr int nDim = 4;
    LaplaceArg<Float,nColor,recon,xpay> arg(out, in, U, kappa, x, parity);
    Laplace<Float,nDim,nColor,LaplaceArg<Float,nColor,recon,xpay> > laplace(arg, in);
    laplace.apply(0);
  }
```

AUTOTUNING

- Kernels use launcher class derived from “Tunable”
- Provides autotuning for work distribution, size of thread blocks, number of thread blocks, etc.
- Ensures optimal performance
- Tuned parameters are stored in a std::map and dumped to disk for later reuse

```
template <typename Float, int nDim, int nColor, typename Arg>
class Laplace : public TunableVectorY {

  protected:
    Arg &arg;
    const ColorSpinorField &meta;

    long long flops() const
    {
      return (2*nDim*(8*nColor*nColor)-2*nColor + (arg.isXpay() ? 2*2*nColor : 0) )
             *arg.nParity*(long long)meta.VolumeCB();
    }
    long long bytes() const
    {
      return arg.out.Bytes() + 2*nDim*arg.in.Bytes() +
             arg.nParity*2*nDim*arg.U.Bytes()*meta.VolumeCB() + (arg.isXpay() ? arg.x.Bytes() : 0);
    }
    bool tuneGridDim() const { return false; }
    unsigned int minThreads() const { return arg.volumeCB; }

  public:
    Laplace(Arg &arg, const ColorSpinorField &meta) : TunableVectorY(arg.nParity), arg(arg), meta(meta)
    {
      strcpy(aux, meta.AuxString());
      strcat(aux, comm_dim_partitioned_string());
      if (arg.isXpay()) strcat(aux,",xpay");
    }
    virtual ~Laplace() { }

    void apply(const cudaStream_t &stream) {
      if (meta.Location() == QUDA_CPU_FIELD_LOCATION) {
        laplaceCPU<Float,nDim,nColor>(arg);
      } else {
        TuneParam tp = tuneLaunch(*this, getTuning(), getVerbosity());
        laplaceGPU<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, int nColor, QudaReconstructType recon, bool xpay>
    void ApplyLaplace(ColorSpinorField &out, const ColorSpinorField &in, const GaugeField &U,
                      double kappa, const ColorSpinorField *x, int parity)
  {
    constexpr int nDim = 4;
    LaplaceArg<Float,nColor,recon,xpay> arg(out, in, U, kappa, x, parity);
    Laplace<Float,nDim,nColor,LaplaceArg<Float,nColor,recon,xpay> > laplace(arg, in);
    laplace.apply(0);
  }
```

Performance metrics

AUTOTUNING

- Kernels use launcher class derived from “Tunable”
- Provides autotuning for work distribution, size of thread blocks, number of thread blocks, etc.
- Ensures optimal performance
- Tuned parameters are stored in a std::map and dumped to disk for later reuse

```
template <typename Float, int nDim, int nColor, typename Arg>
class Laplace : public TunableVectorY {

  protected:
    Arg &arg;
    const ColorSpinorField &meta;

    long long flops() const
    {
      return (2*nDim*(8*nColor*nColor)-2*nColor + (arg.isXpay() ? 2*2*nColor : 0) )
             *arg.nParity*(long long)meta.VolumeCB();
    }
    long long bytes() const
    {
      return arg.out.Bytes() + 2*nDim*arg.in.Bytes() +
             arg.nParity*2*nDim*arg.U.Bytes()*meta.VolumeCB() + (arg.isXpay() ? arg.x.Bytes() : 0);
    }
    bool tuneGridDim() const { return false; }
    unsigned int minThreads() const { return arg.volumeCB; }

  public:
    Laplace(Arg &arg, const ColorSpinorField &meta) : TunableVectorY(arg.nParity), arg(arg), meta(meta)
    {
      strcpy(aux, meta.AuxString());
      strcat(aux, comm_dim_partitioned_string());
      if (arg.isXpay()) strcat(aux,",xpay");
    }
    virtual ~Laplace() { }

    void apply(const cudaStream_t &stream) {
      if (meta.Location() == QUDA_CPU_FIELD_LOCATION) {
        laplaceCPU<Float,nDim,nColor>(arg);
      } else {
        TuneParam tp = tuneLaunch(*this, getTuning(), getVerbosity());
        laplaceGPU<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, int nColor, QudaReconstructType recon, bool xpay>
    void ApplyLaplace(ColorSpinorField &out, const ColorSpinorField &in, const GaugeField &U,
                      double kappa, const ColorSpinorField *x, int parity)
  {
    constexpr int nDim = 4;
    LaplaceArg<Float,nColor,recon,xpay> arg(out, in, U, kappa, x, parity);
    Laplace<Float,nDim,nColor,LaplaceArg<Float,nColor,recon,xpay> > laplace(arg, in);
    laplace.apply(0);
  }
```

AUTOTUNING

- Kernels use launcher class derived from “Tunable”
- Provides autotuning for work distribution, size of thread blocks, number of thread blocks, etc.
- Ensures optimal performance
- Tuned parameters are stored in a std::map and dumped to disk for later reuse

```
template <typename Float, int nDim, int nColor, typename Arg>
class Laplace : public TunableVectorY {

  protected:
    Arg &arg;
    const ColorSpinorField &meta;

    long long flops() const
    {
      return (2*nDim*(8*nColor*nColor)-2*nColor + (arg.isXpay() ? 2*2*nColor : 0) )
             *arg.nParity*(long long)meta.VolumeCB();
    }
    long long bytes() const
    {
      return arg.out.Bytes() + 2*nDim*arg.in.Bytes() +
             arg.nParity*2*nDim*arg.U.Bytes()*meta.VolumeCB() + (arg.isXpay() ? arg.x.Bytes() : 0);
    }
    bool tuneGridDim() const { return false; }
    unsigned int minThreads() const { return arg.volumeCB; }

  public:
    Laplace(Arg &arg, const ColorSpinorField &meta) : TunableVectorY(arg.nParity), arg(arg), meta(meta)
    {
      strcpy(aux, meta.AuxString());
      strcat(aux, comm_dim_partitioned_string());
      if (arg.isXpay()) strcat(aux,",xpay");
    }
    virtual ~Laplace() { }

    void apply(const cudaStream_t &stream) {
      if (meta.Location() == QUDA_CPU_FIELD_LOCATION) {
        laplaceCPU<Float,nDim,nColor>(arg);
      } else {
        TuneParam tp = tuneLaunch(*this, getTuning(), getVerbosity());
        laplaceGPU<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, int nColor, QudaReconstructType recon, bool xpay>
    void ApplyLaplace(ColorSpinorField &out, const ColorSpinorField &in, const GaugeField &U,
                      double kappa, const ColorSpinorField *x, int parity)
  {
    constexpr int nDim = 4;
    LaplaceArg<Float,nColor,recon,xpay> arg(out, in, U, kappa, x, parity);
    Laplace<Float,nDim,nColor,LaplaceArg<Float,nColor,recon,xpay> > laplace(arg, in);
    laplace.apply(0);
  }
```

AUTOTUNING

- Kernels use launcher class derived from “Tunable”
- Provides autotuning for work distribution, size of thread blocks, number of thread blocks, etc.
- Ensures optimal performance
- Tuned parameters are stored in a std::map and dumped to disk for later reuse

Launch work to CPU or GPU depending on field type

```
template <typename Float, int nDim, int nColor, typename Arg>
class Laplace : public TunableVectorY {

  protected:
    Arg &arg;
    const ColorSpinorField &meta;

    long long flops() const
    {
      return (2*nDim*(8*nColor*nColor)-2*nColor + (arg.isXpay() ? 2*2*nColor : 0) )
             *arg.nParity*(long long)meta.VolumeCB();
    }
    long long bytes() const
    {
      return arg.out.Bytes() + 2*nDim*arg.in.Bytes() +
             arg.nParity*2*nDim*arg.U.Bytes()*meta.VolumeCB() + (arg.isXpay() ? arg.x.Bytes() : 0);
    }
    bool tuneGridDim() const { return false; }
    unsigned int minThreads() const { return arg.volumeCB; }

  public:
    Laplace(Arg &arg, const ColorSpinorField &meta) : TunableVectorY(arg.nParity), arg(arg), meta(meta)
    {
      strcpy(aux, meta.AuxString());
      strcat(aux, comm_dim_partitioned_string());
      if (arg.isXpay()) strcat(aux,",xpay");
    }
    virtual ~Laplace() { }

    void apply(const cudaStream_t &stream) {
      if (meta.Location() == QUDA_CPU_FIELD_LOCATION) {
        laplaceCPU<Float,nDim,nColor>(arg);
      } else {
        TuneParam tp = tuneLaunch(*this, getTuning(), getVerbosity());
        laplaceGPU<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, int nColor, QudaReconstructType recon, bool xpay>
    void ApplyLaplace(ColorSpinorField &out, const ColorSpinorField &in, const GaugeField &U,
                      double kappa, const ColorSpinorField *x, int parity)
  {
    constexpr int nDim = 4;
    LaplaceArg<Float,nColor,recon,xpay> arg(out, in, U, kappa, x, parity);
    Laplace<Float,nDim,nColor,LaplaceArg<Float,nColor,recon,xpay> > laplace(arg, in);
    laplace.apply(0);
  }
```

AUTOTUNING

- Kernels use launcher class derived from “Tunable”
- Provides autotuning for work distribution, size of thread blocks, number of thread blocks, etc.
- Ensures optimal performance
- Tuned parameters are stored in a std::map and dumped to disk for later reuse

PERFORMANCE ANALYSIS

Autotuner dumps a profile to disk when endQuda called

Preconditioned CG

```
profile0.9.0  v0.8.0-1644-g7684193-sm_60  cpu_arch=x86_64,gpu_arch=sm_60,cuda_version=8000# Last updated Fri Apr 28 13:09:30 2017

  total time    percentage         calls    time / call          volumenameauxcomment
    0.165536        32.202          1000    0.000165536     12x24x24x24N4quda7LaplaceIfLi4ELi3ENS_10LaplaceArgIfLi3EL21QudaReconstructType_s12ELb1EEEEEvol=165888,stride=172800,precision=4,comm=0000,xpay# 583.24 Gflop/s, 625.33 GB/s
    0.158816        30.8948         1000    0.000158816     12x24x24x24N4quda7LaplaceIfLi4ELi3ENS_10LaplaceArgIfLi3EL21QudaReconstructType_s12ELb0EEEEEvol=165888,stride=172800,precision=4,comm=0000# 595.38 Gflop/s, 626.72 GB/s
   0.0532917        10.3669          940    5.66933e-05     12x24x24x24N4quda4blas9axpyZpbx_I7double2S2_EEvol=165888,stride=172800,precision=4,vol=165888,stride=172800,precision=8# 70.23 Gflop/s, 491.58 GB/s
   0.0331733        6.45326         1000    3.31733e-05     12x24x24x24N4quda4blas11axpyCGNorm2I7double26float2S3_EEvol=165888,stride=172800,precision=4# 187.52 Gflop/s, 375.05 GB/s
   0.0243413        4.73516         1000    2.43413e-05     12x24x24x24N4quda4blas3DotId6float2S2_EEvol=165888,stride=172800,precision=4# 85.19 Gflop/s, 340.75 GB/s
   0.0202383          3.937           64    0.000316224     12x24x24x24N4quda7LaplaceIdLi4ELi3ENS_10LaplaceArgIdLi3EL21QudaReconstructType_s12ELb1EEEEEvol=165888,stride=172800,precision=8,comm=0000,xpay# 305.31 Gflop/s, 654.69 GB/s
```

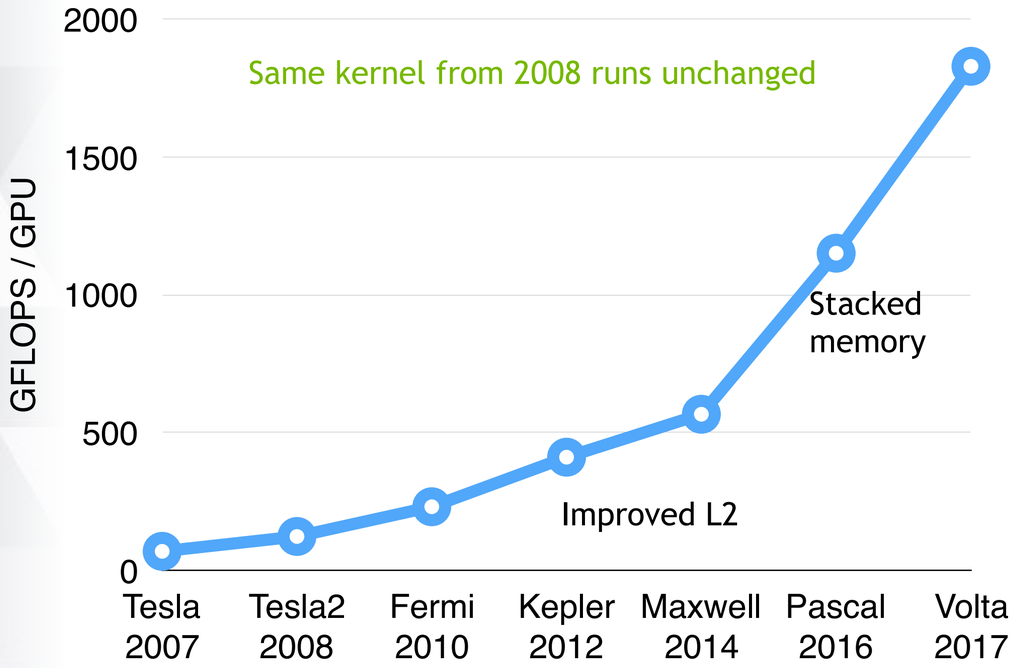
Direct CG

```
profile0.9.0  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          volumenameauxcomment
    0.227616        43.4696         1000    0.000227616     24x24x24x24N4quda7LaplaceIfLi4ELi3ENS_10LaplaceArgIfLi3EL21QudaReconstructType_s12ELb1EEEEEvol=331776,stride=172800,precision=4,comm=0000,xpay# 848.33 Gflop/s, 909.55 GB/s
     0.10025        19.1456          918    0.000109205     24x24x24x24N4quda4blas9axpyZpbx_I7double2S2_EEvol=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     24x24x24x24N4quda4blas11axpyCGNorm2I7double26float2S3_EEvol=331776,stride=172800,precision=4# 224.05 Gflop/s, 448.10 GB/s
    0.039441        7.53236           84    0.000469536     24x24x24x24N4quda7LaplaceIdLi4ELi3ENS_10LaplaceArgIdLi3EL21QudaReconstructType_s12ELb1EEEEEvol=331776,stride=172800,precision=8,comm=0000,xpay# 411.24 Gflop/s, 881.84 GB/s
    0.037984          7.2541         1000     3.7984e-05     24x24x24x24N4quda4blas3DotId6float2S2_EEvol=331776,stride=172800,precision=4# 109.18 Gflop/s, 436.73 GB/s
```

Simple high-level code can achieve high performance

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

```
/**
   @brief Applies the off-diagonal part of the Laplace operator

   @param[out] out The out result field
   @param[in] U The gauge field
   @param[in] kappa Kappa value
   @param[in] in The input field
   @param[in] parity The site parity
   @param[in] x_cb The checker-boarded site index
 */
extern __shared__ float s[];
template <typename Float, int nDim, int nColor, typename Vector, typename Arg>
__device__ __host__ inline void applyLaplace(Vector &out, Arg &arg, int x_cb, int parity) {
  typedef Matrix<complex<Float>,nColor> Link;
  const int their_spinor_parity = (arg.nParity == 2) ? 1-parity : 0;

  int coord[nDim];
  getCoords(coord, x_cb, arg.dim, parity);

#pragma unroll
  for (int d = 0; d<nDim; d++) // loop over dimension
  {
    //Forward gather - compute fwd offset for vector fetch
    const int fwd_idx = linkIndexP1(coord, arg.dim, d);

    if ( arg.commDim[d] && (coord[d] + arg.nFace >= arg.dim[d]) ) {
      const int ghost_idx = ghostFaceIndex<1>(coord, arg.dim, d, arg.nFace);

      const Link U = arg.U(d, x_cb, parity);
      const Vector in = arg.in.Ghost(d, 1, ghost_idx, their_spinor_parity);

      out += U * in;
    } else {

      const Link U = arg.U(d, x_cb, parity);
      const Vector in = arg.in(fwd_idx, their_spinor_parity);

      out += U * in;
    }

    //Backward gather - compute back offset for spinor and gauge fetch
    const int back_idx = linkIndexM1(coord, arg.dim, d);
    const int gauge_idx = back_idx;

    if ( arg.commDim[d] && (coord[d] - arg.nFace < 0) ) {
      const int ghost_idx = ghostFaceIndex<0>(coord, arg.dim, d, arg.nFace);

      const Link U = arg.U.Ghost(d, ghost_idx, 1-parity);
      const Vector in = arg.in.Ghost(d, 0, ghost_idx, their_spinor_parity);

      out += conj(U) * in;
    } else {

      const Link U = arg.U(d, gauge_idx, 1-parity);
      const Vector in = arg.in(back_idx, their_spinor_parity);

      out += conj(U) * in;
    }
  } //nDim

}
```

LAPLACE KERNEL

```
//out(x) = M*in = (-D + m) * in(x-mu)
template <typename Float, int nDim, int nColor, typename Arg>
__device__ __host__ inline void laplace(Arg &arg, int x_cb, int parity)
{
  typedef ColorSpinor<Float,nColor,1> Vector;
  Vector out;

  applyLaplace<Float,nDim,nColor>(out, arg, x_cb, parity);

  if (arg.isXpay()) {
    Vector x = arg.x(x_cb, parity);
    out = x + arg.kappa * out;
  }
  arg.out(x_cb, parity) = out;
}

// CPU kernel for applying the Laplace operator to a vector
template <typename Float, int nDim, int nColor, typename Arg>
void laplaceCPU(Arg arg)
{

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

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

}

// GPU Kernel for applying the Laplace operator to a vector
template <typename Float, int nDim, int nColor, typename Arg>
__global__ void laplaceGPU(Arg arg)
{
  int x_cb = blockIdx.x*blockDim.x + threadIdx.x;

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

  if (x_cb >= arg.volumeCB) return;
  if (parity >= arg.nParity) return;

  laplace<Float,nDim,nColor>(arg, x_cb, parity);
}
```

WILSON KERNEL

```
/**
   @brief Applies the off-diagonal part of the Wilson operator

   @param[out] out The out result field
   @param[in] U The gauge field
   @param[in] kappa Kappa value
   @param[in] in The input field
   @param[in] parity The site parity
   @param[in] x_cb The checker-boarded site index
 */
extern __shared__ float s[];
template <typename Float, int nDim, int nColor, typename Vector, typename Arg>
__device__ __host__ inline void applyWilson(Vector &out, Arg &arg, int x_cb, int parity) {
  typedef ColorSpinor<Float,nColor,2> HalfVector;
  typedef Matrix<complex<Float>,nColor> Link;
  const int their_spinor_parity = (arg.nParity == 2) ? 1-parity : 0;

  int coord[nDim];
  getCoords(coord, x_cb, arg.dim, parity);

#pragma unroll
  for (int d = 0; d<nDim; d++) // loop over dimension
  {
    //Forward gather - compute fwd offset for vector fetch
    const int fwd_idx = linkIndexP1(coord, arg.dim, d);

    if ( arg.commDim[d] && (coord[d] + arg.nFace >= arg.dim[d]) ) {
      const int ghost_idx = ghostFaceIndex<1>(coord, arg.dim, d, arg.nFace);

      const Link U = arg.U(d, x_cb, parity);
      const HalfVector in = arg.in.Ghost(d, 1, ghost_idx, their_spinor_parity);

      out += (U * in).reconstruct(d, -1);
    } else {

      const Link U = arg.U(d, x_cb, parity);
      const Vector in = arg.in(fwd_idx, their_spinor_parity);

      out += (U * in.project(d, -1)).reconstruct(d, -1);
    }

    //Backward gather - compute back offset for spinor and gauge fetch
    const int back_idx = linkIndexM1(coord, arg.dim, d);
    const int gauge_idx = back_idx;

    if ( arg.commDim[d] && (coord[d] - arg.nFace < 0) ) {
      const int ghost_idx = ghostFaceIndex<0>(coord, arg.dim, d, arg.nFace);

      const Link U = arg.U.Ghost(d, ghost_idx, 1-parity);
      const HalfVector in = arg.in.Ghost(d, 0, ghost_idx, their_spinor_parity);

      out += (conj(U) * in).reconstruct(d, +1);
    } else {

      const Link U = arg.U(d, gauge_idx, 1-parity);
      const Vector in = arg.in(back_idx, their_spinor_parity);

      out += (conj(U) * in.project(d, +1).reconstruct(d, +1);
    }
  } //nDim

}
```

```
//out(x) = M*in = (-D + m) * in(x-mu)
template <typename Float, int nDim, int nColor, typename Arg>
__device__ __host__ inline void wilson(Arg &arg, int x_cb, int parity)
{
  typedef ColorSpinor<Float,nColor,4> Vector;
  Vector out;

  applyWilson<Float,nDim,nColor>(out, arg, x_cb, parity);

  if (arg.isXpay()) {
    Vector x = arg.x(x_cb, parity);
    out = x + arg.kappa * out;
  }
  arg.out(x_cb, parity) = out;
}

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

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

    for (int x_cb = 0; x_cb < arg.volumeCB; x_cb++) { // 4-d volume
      wilson<Float,nDim,nColor>(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, typename Arg>
__global__ void wilsonGPU(Arg arg)
{
  int x_cb = blockIdx.x*blockDim.x + threadIdx.x;

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

  if (x_cb >= arg.volumeCB) return;
  if (parity >= arg.nParity) return;

  wilson<Float,nDim,nColor>(arg, x_cb, parity);
}
```

WILSON KERNEL

```
/**
   @brief Applies the off-diagonal part of the Wilson operator

   @param[out] out The out result field
   @param[in] U The gauge field
   @param[in] kappa Kappa value
   @param[in] in The input field
   @param[in] parity The site parity
   @param[in] x_cb The checker-boarded site index
 */
extern __shared__ float s[];
template <typename Float, int nDim, int nColor, typename Vector, typename Arg>
__device__ __host__ inline void applyWilson(Vector &out, Arg &arg, int x_cb, int parity) {
  typedef ColorSpinor<Float,nColor,2> HalfVector;
  typedef Matrix<complex<Float>,nColor> Link;
  const int their_spinor_parity = (arg.nParity == 2) ? 1-parity : 0;

  int coord[nDim];
  getCoords(coord, x_cb, arg.dim, parity);

#pragma unroll
  for (int d = 0; d<nDim; d++) // loop over dimension
  {
    //Forward gather - compute fwd offset for vector fetch
    const int fwd_idx = linkIndexP1(coord, arg.dim, d);

    if ( arg.commDim[d] && (coord[d] + arg.nFace >= arg.dim[d]) ) {
      const int ghost_idx = ghostFaceIndex<1>(coord, arg.dim, d, arg.nFace);

      const Link U = arg.U(d, x_cb, parity);
      const HalfVector in = arg.in.Ghost(d, 1, ghost_idx, their_spinor_parity);

      out += (U * in).reconstruct(d, -1);
    } else {

      const Link U = arg.U(d, x_cb, parity);
      const Vector in = arg.in(fwd_idx, their_spinor_parity);

      out += (U * in.project(d, -1)).reconstruct(d, -1);
    }

    //Backward gather - compute back offset for spinor and gauge fetch
    const int back_idx = linkIndexM1(coord, arg.dim, d);
    const int gauge_idx = back_idx;

    if ( arg.commDim[d] && (coord[d] - arg.nFace < 0) ) {
      const int ghost_idx = ghostFaceIndex<0>(coord, arg.dim, d, arg.nFace);

      const Link U = arg.U.Ghost(d, ghost_idx, 1-parity);
      const HalfVector in = arg.in.Ghost(d, 0, ghost_idx, their_spinor_parity);

      out += (conj(U) * in).reconstruct(d, +1);
    } else {

      const Link U = arg.U(d, gauge_idx, 1-parity);
      const Vector in = arg.in(back_idx, their_spinor_parity);

      out += (conj(U) * in.project(d, +1).reconstruct(d, +1);
    }
  } //nDim

}
```

We have spin!

Spin reconstruct

Spin project and reconstruct

Spin reconstruct

Spin project and reconstruct

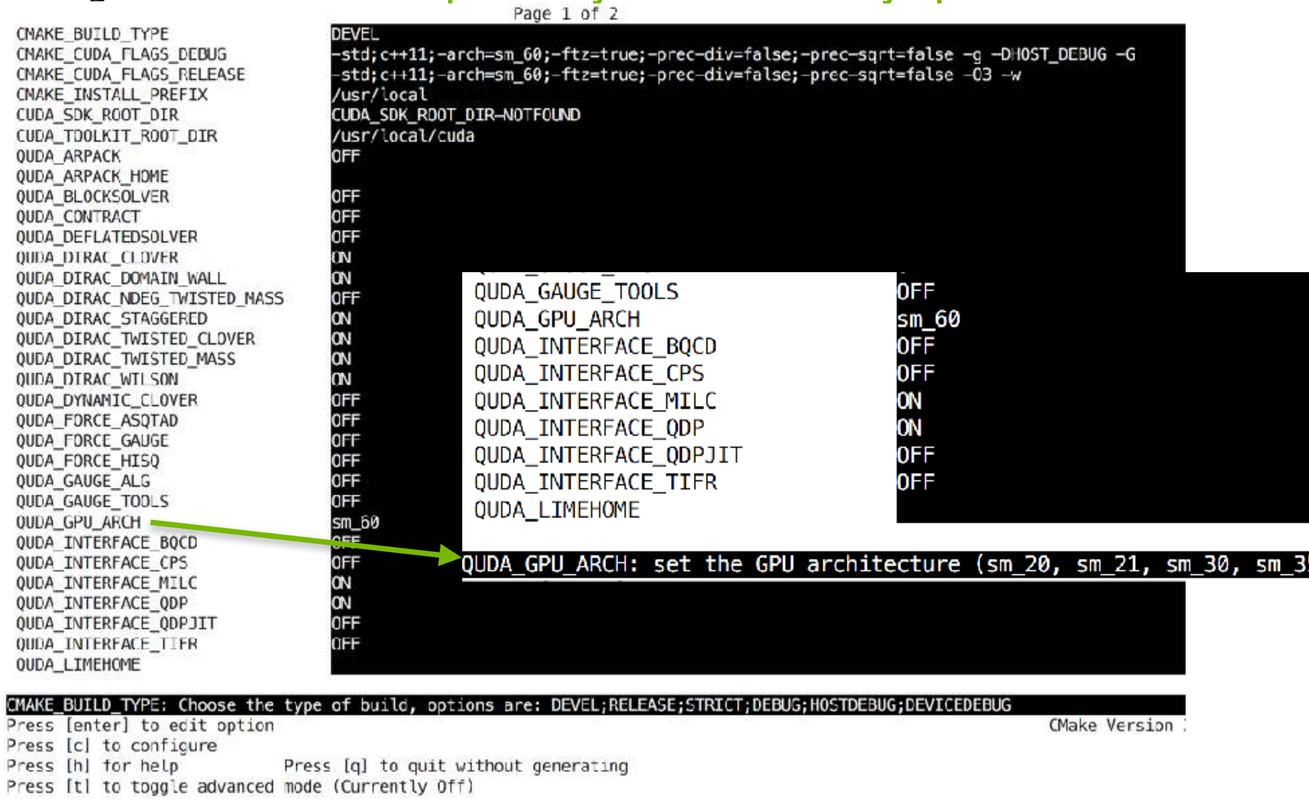
```
//out(x) = M*in = (-D + m) * in(x-mu)
template <typename Float, int nDim, int nColor, typename Arg>
__device__ __host__ inline void wilson(Arg &arg, int x_cb, int parity)
{
  typedef ColorSpinor<Float,nColor,4> Vector;
  Vector out;

  applyWilson<Float,nDim,nColor>(out, arg, x_cb, parity);

  if (arg.isXpay()) {
    Vector x = arg.x(x_cb, parity);
    out = x + arg.kappa * out;
  }
  arg.out(x_cb, parity) = out;
}

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

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

    for (int x_cb = 0; x_cb < arg.volumeCB; x_cb++) { // 4-d volume
      wilson<Float,nDim,nColor>(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, typename Arg>
__global__ void wilsonGPU(Arg arg)
{
  int x_cb = blockIdx.x*blockDim.x + threadIdx.x;

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

  if (x_cb >= arg.volumeCB) return;
  if (parity >= arg.nParity) return;

  wilson<Float,nDim,nColor>(arg, x_cb, parity);
}
```

QUDA DEVELOPMENT WORKFLOW

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 GIU for exploring options
```




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

QUDA AS AN APPLICATION FRAMEWORK

add *additional* lower level C++ interface

High level C/Fortran interface

targets application accelerations (e.g. MILC)
QUDA manages GPU allocation and transfer

```
// SETUP, pass host data (raw pointers) to QUDA
// QUDA internally keeps loaded gauge field
loadGaugeQuda(const_cast<void*>(gaugefield), &gaugeParam);

// CALCULATION
// QUDA uses loaded gauge field and host pointers for src, solution
invertQuda((char*)solution, ((char*)source), &invertParam);
doSomething(solution, source, ….)
invertQuda((char*)solution, ((char*)other_source), &invertParam);
doSomethingElse(solution, source, ….)
```

makes it hard to control residency of data

missing bits and pieces (doSomething) are either on the host or need to be hacked into QUDA

Future low level C++-interface (WIP)

write C++ application with QUDA data structures
control data lifetime on GPU

```
// SETUP, ALLOCATE MEMORY
cudaGaugeQudaField *gauge = new cudaGaugeField(gParam);
colorSpinorField *b = new cudaColorSpinorField(*h_b, cudaParam);
colorSpinorField *x = new cudaColorSpinorField(*h_b, cudaParam);

// CALCULATION, ALLOCATE MEMORY
invertQuda(x, b, gauge, invertParam);
doSomething(x, b, ….)
invertQuda(y, b, gauge, invertParam);
doSomethingElse( x, y, b )

delete b;   delete x;
```

missing bits *can* but *don't need* to be part of QUDA

allows writing e.g. of a fully GPU resident RHMC application with QUDA acceleration

Data on host in blue / Data on GPU in green

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

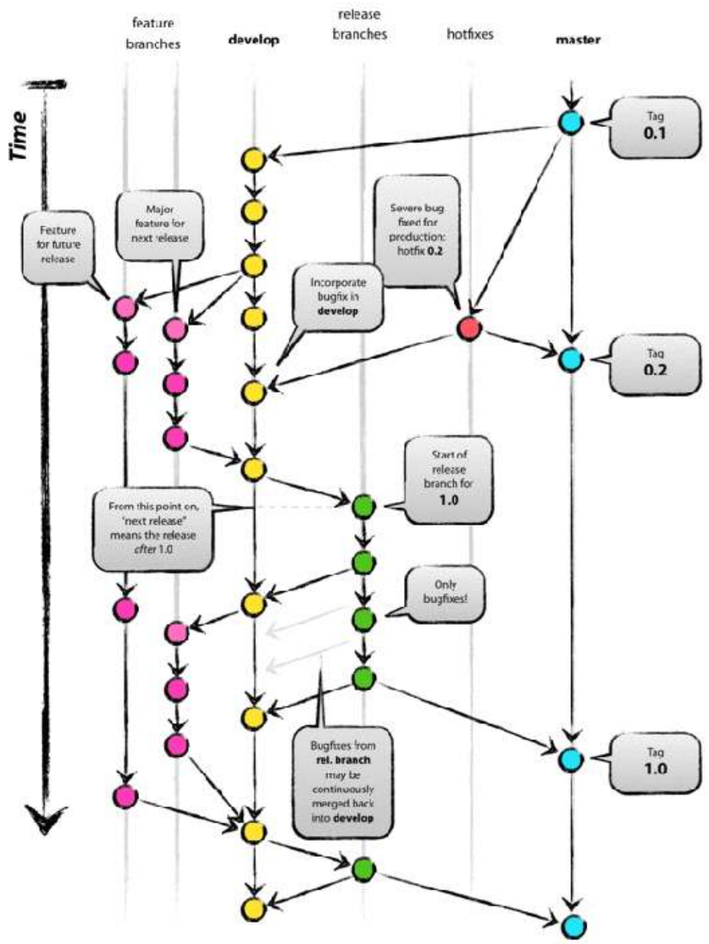
→ peer review of code required before merging

→ automated build testing (Jenkins)

→ automated tests (Jenkins)

Give feedback to new contributors

Prevents fragmentation



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

CONTINUOUS INTEGRATIONS

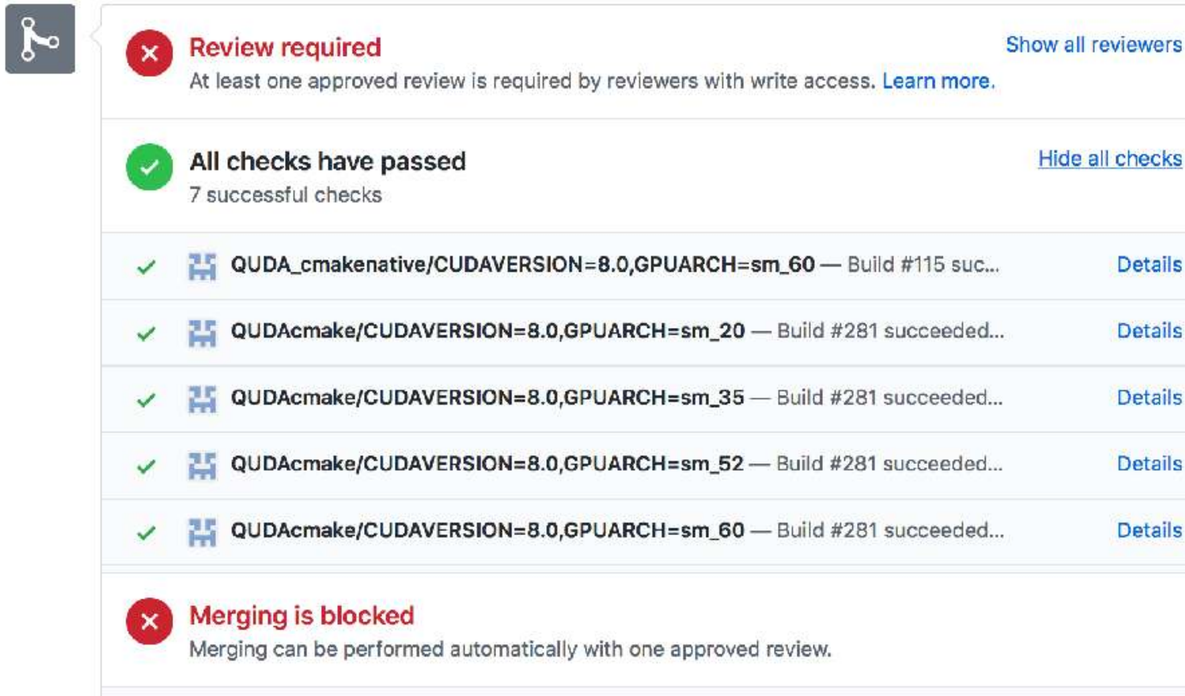
based on Jenkins

integrated into github pull requests

manual review
automatically build QUDA with various options
run and verify QUDA internal tests

Future plans:

increase coverage of parameter space
increase coverage of internal tests
add application tests e.g. Chroma, MILC with QUDA



TESTING

using jenkins, ctest and googletest

internal tests leverage googletest framework

tests can be driven through ctest

integrated in Jenkins

```
[----------] 1 test from caxpyBzpx_double/BlasTest
[ RUN      ] caxpyBzpx_double/BlasTest.verify/0
caxpyBzpx                       error = 7.368517e-14
[       OK ] caxpyBzpx_double/BlasTest.verify/0 (195 ms)
[----------] 1 test from caxpyBzpx_double/BlasTest (195 ms total)

[----------] 1 test from multicDotProductNorm_double/BlasTest
[ RUN      ] multicDotProductNorm_double/BlasTest.verify/0
cDotProductNorm (block)         error = 1.069540e-20
[       OK ] multicDotProductNorm_double/BlasTest.verify/0 (165 ms)
[----------] 1 test from multicDotProductNorm_double/BlasTest (165 ms total)

[----------] 1 test from multicDotProduct_double/BlasTest
[ RUN      ] multicDotProduct_double/BlasTest.verify/0
cDotProduct (block)             error = 1.170183e-16
[       OK ] multicDotProduct_double/BlasTest.verify/0 (169 ms)
```

Test Result : QUDA/BlasTest

0 failures

168 tests

Took 4 min 14 sec.

add description

All Tests

Test name	Duration	Status
benchmark/HeavyQuarkResidualNorm_half ((0, 31))	0.74 sec	Passed
benchmark/HeavyQuarkResidualNorm_single ((1, 31))	0.72 sec	Passed
benchmark/ax_half ((0, 7))	0.73 sec	Passed
benchmark/ax_single ((1, 7))	0.73 sec	Passed
benchmark/axpby_half ((0, 2))	0.74 sec	Passed
benchmark/axpby_single ((1, 2))	0.73 sec	Passed
benchmark/axpyBzpcx_block_half ((0, 37))	0.74 sec	Passed
benchmark/axpyBzpcx_block_single ((1, 37))	0.73 sec	Passed
benchmark/axpyBzpcx_half ((0, 11))	0.74 sec	Passed
benchmark/axpyBzpcx_single ((1, 11))	0.73 sec	Passed
benchmark/axpyNorm_half ((0, 20))	0.73 sec	Passed
benchmark/axpyNorm_single ((1, 20))	0.72 sec	Passed
benchmark/axpyReDot_half ((0, 35))	0.74 sec	Passed

DOCUMENTATION

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

Compilation guide

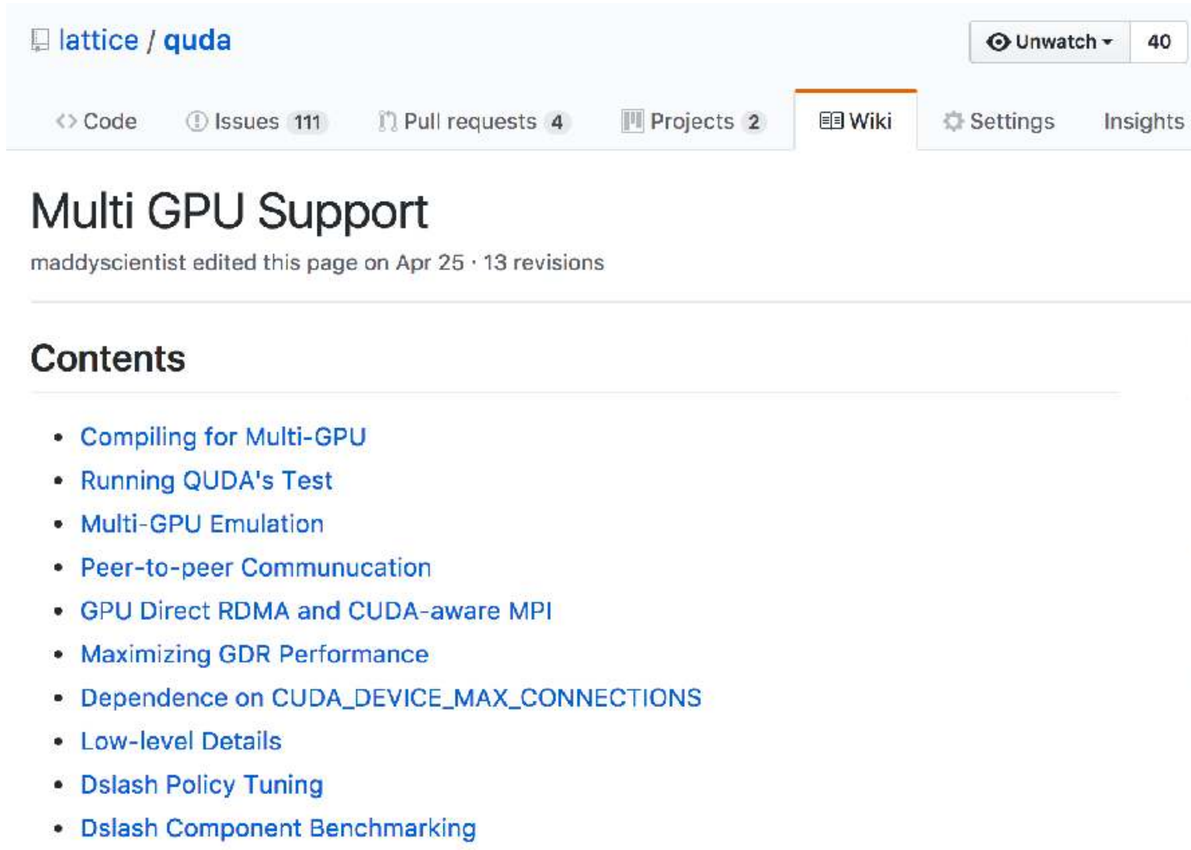
How to contribute to QUDA

C and Fortran Interface Guide

Maximizing multi-GPU performance

Parameter tuning for deflation and multigrid

More coming soon…



SUMMARY

QUDA continues to be under active development

Much recent focus on improving development workflow and code hygiene

Come and join the fun!