# MAXIMIZING THE BANG PER BIT

Kate Clark @ Lattice 2022

**ECP benchmarks apps

# QUDA

- "QCD on CUDA" – http://lattice.github.com/quda (open source, BSD license)
- Effort started at Boston University in 2008, now in wide use as the GPU backend for BQCD, Chroma**, CPS**, MILC**, TIFR, etc.
- Provides solvers for major fermionic discretizations, pure gauge algorithms, etc.
- Maximize performance
  - Mixed-precision methods
  - Autotuning for high performance on all CUDA-capable architectures
  - Multigrid solvers for optimal convergence
  - NVSHMEM for improving strong scaling
- Portable: HIP (merged), SYCL (in review) and OpenMP (in development)
- **A research tool for how to reach the exascale (and beyond)**
  - Optimally mapping the problem to hierarchical processors and node topologies

# QUDA CONTRIBUTORS

## 10+ years - lots of contributors

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

# ANNOUNCING H100

Unprecedented Performance, Scalability, and Security for Every Data Center

## HIGHEST AI AND HPC PERFORMANCE

4PF FP8 (6X)| 2PF FP16 (3X)| 1PF TF32 (3X)| 60TF FP64 (3X)
3TB/s (1.5X), 80GB HBM3 memory

## TRANSFORMER MODEL OPTIMIZATIONS

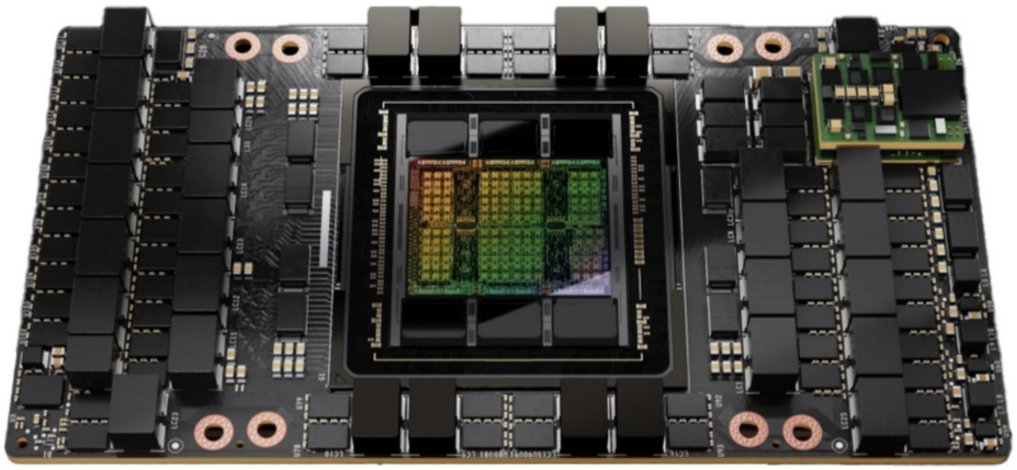
6X faster on largest transformer models

## HIGHEST UTILIZATION EFFICIENCY AND SECURITY

7 Fully isolated & secured instances, guaranteed QoS
2nd Gen MIG | Confidential Computing

## FASTEST, SCALABLE INTERCONNECT

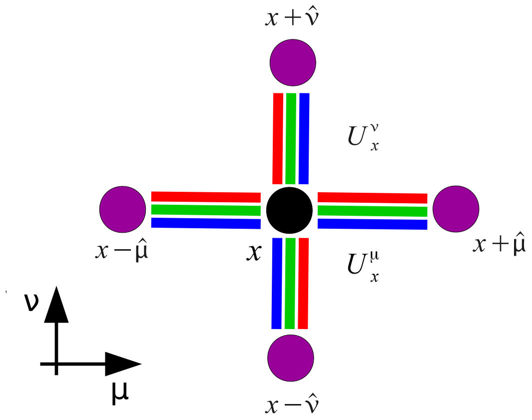
900 GB/s GPU-2-GPU connectivity (1.5X)
up to 256 GPUs with NVLink Switch | 128GB/s PCIe Gen5



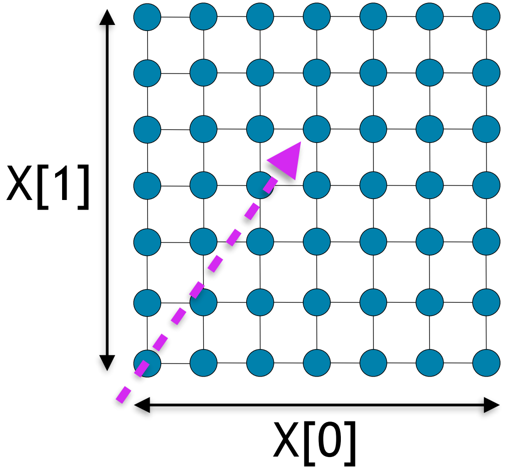
Custom 4N TSMC Process | 80 billion transistors

# MAPPING THE DIRAC OPERATOR TO GPUS

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




- Finite difference operator in LQCD is known as Dslash
- Assign a single space-time point to each thread
  - V = XYZT threads, e.g., V = $24^4$ => $3.3\times10^6$ threads
- Looping over direction each thread must
  - Load the neighboring spinor (24 numbers x8)
  - Load the color matrix connecting the sites (18 numbers x8)
  - Do the computation
  - Save the result (24 numbers)
- Each thread has (Wilson Dslash) 0.92 naive arithmetic intensity
- QUDA reduces memory traffic
  - Exact SU(3) matrix compression (18 => 12 or 8 real numbers)
  - Use 16-bit fixed-point representation with mixed-precision solver



# IEEE FLOATING-POINT NUMBERS

```
struct float32_t {
  unsigned int mantissa : 23;
  unsigned int exponent :  8;
  unsigned int sign     :  1;
};
```

$$(-1)^{b_{31}} \times 2^{(b_{30} b_{29} \ldots b_{23})_2 - 127} \times (1.b_{22} b_{21} \ldots b_0)_2$$

**FP32**

32-bits per real

24-bit mantissa => Precision $\epsilon \sim 5 \times 10^{-8}$

8-bit exponent => Range $\in [1 \times 10^{-38}, 3 \times 10^{38}]$

**FP64**

64-bits per real

53-bit mantissa => Precision $\epsilon \sim 1 \times 10^{-16}$

8-bit exponent => Range $\in [2 \times 10^{-208}, 2 \times 10^{308}]$

# QUDA "HALF" PRECISION

Gauge Field

Element range $\in [-1,1]$

No need to store exponent

Store the matrix elements in 16-bit fixed-point

Fermion fields

No *a priori* bound on the elements range

For each site vector store max element to set range

Perform computation in FP32

16-bit local precision $\epsilon \sim 3 \times 10^{-5}$ with global FP32 range

cf IEEE FP16: $\epsilon \sim 5 \times 10^{-4}$

3x3 Link matrix

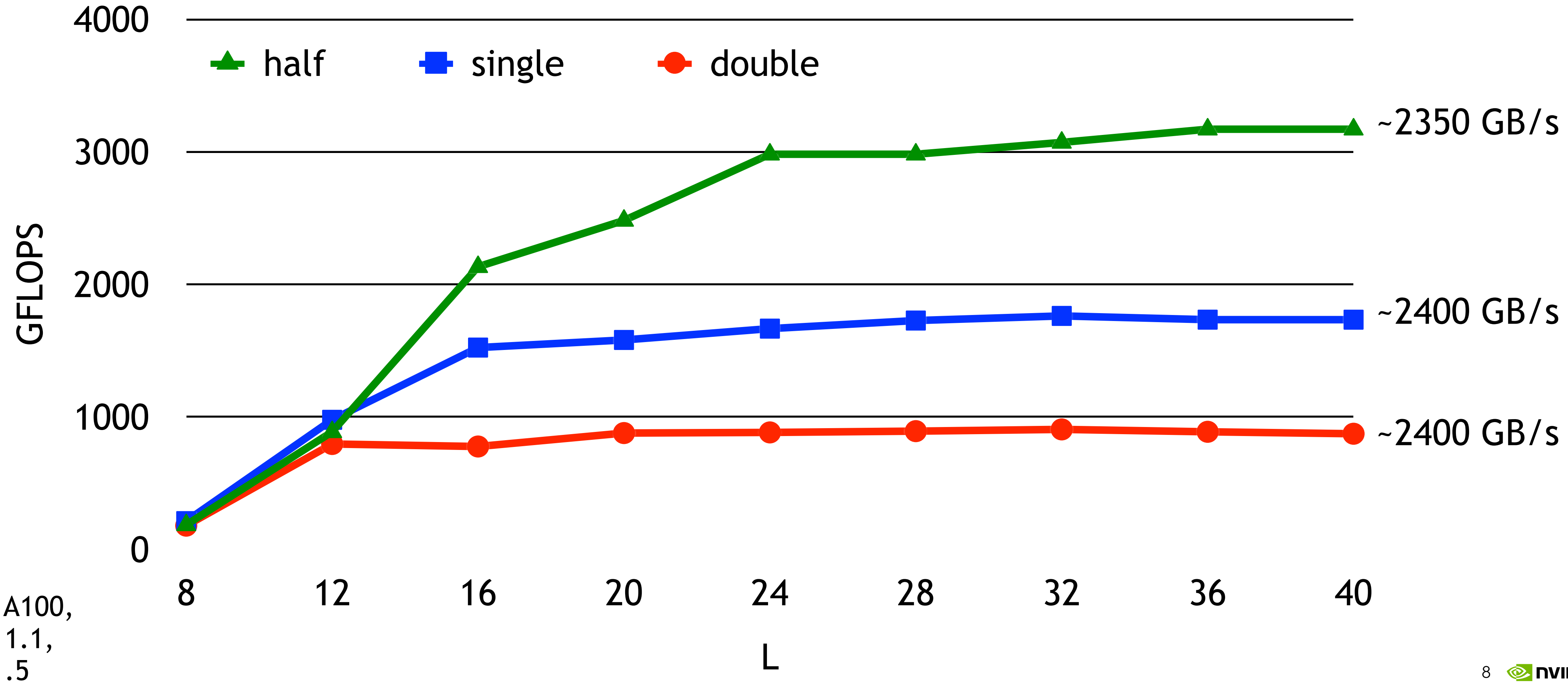
```
struct matrix {
  int16_t v[18];
};
```

Staggered fermion

```
struct vector3 {
  int16_t v[6];
  float max;
};
```

# SINGLE GPU PERFORMANCE

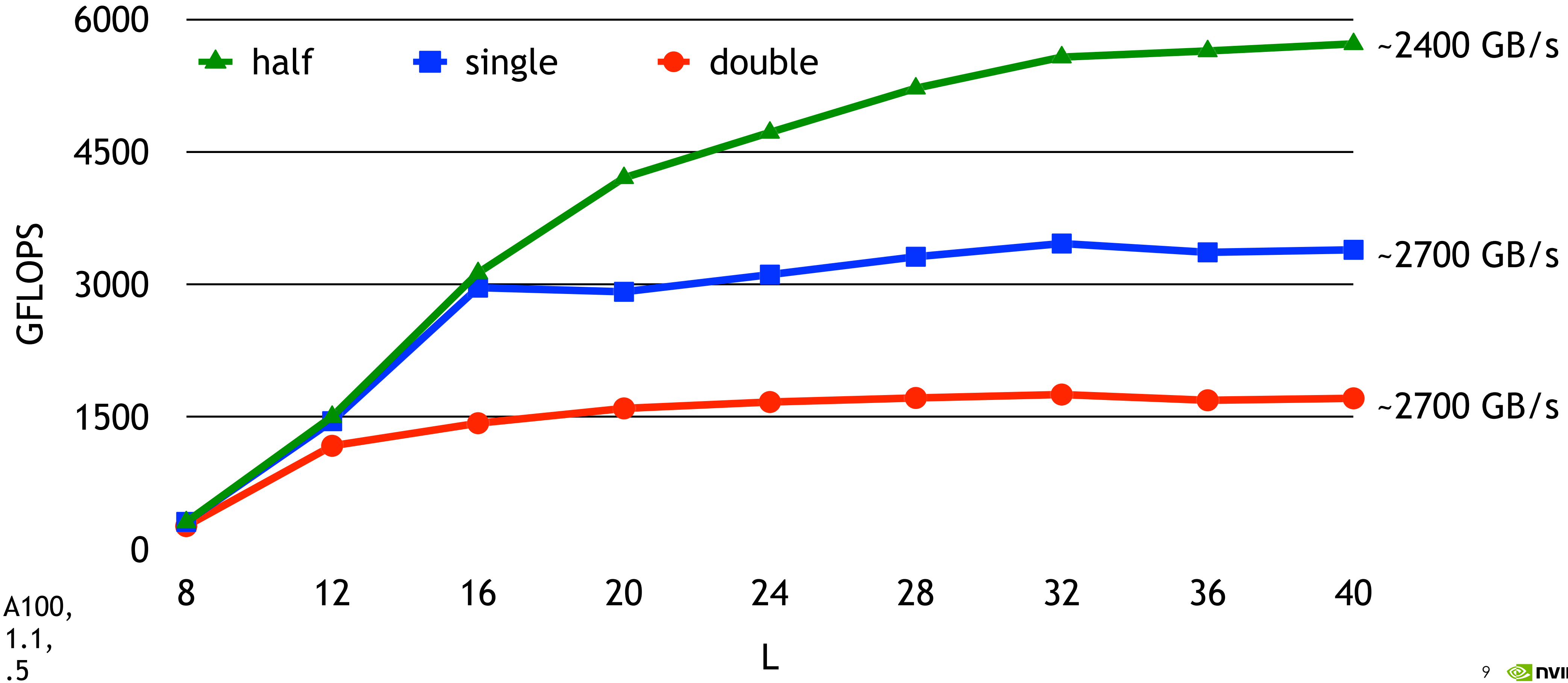
HISQ stencil (Chroma, A100-80)




NVIDIA A100,
CUDA 11.1,
GCC 11.5

# SINGLE GPU PERFORMANCE

## Wilson-clover stencil (Chroma, A100-80)

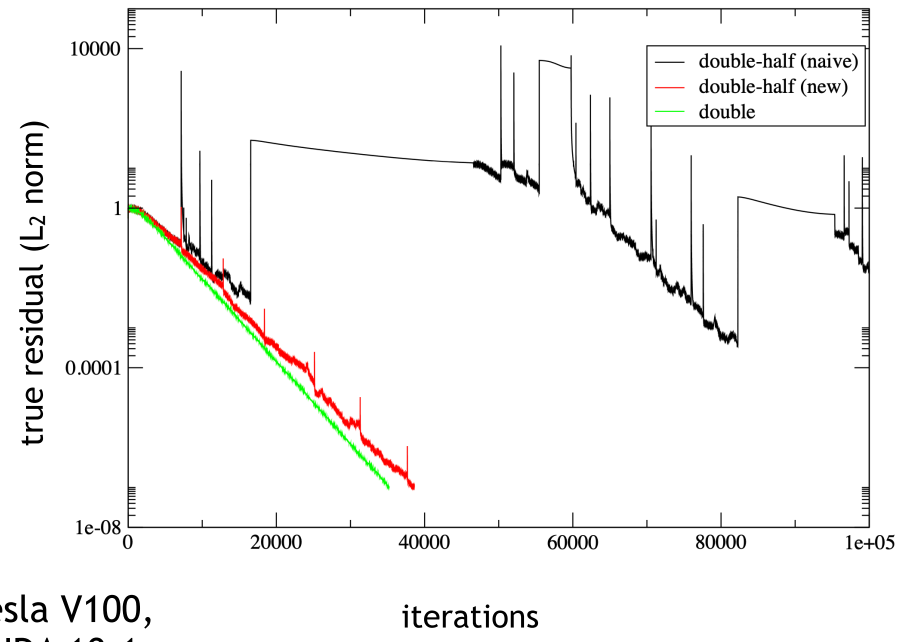



NVIDIA A100,
CUDA 11.1,
GCC 11.5

# MIXED PRECISION

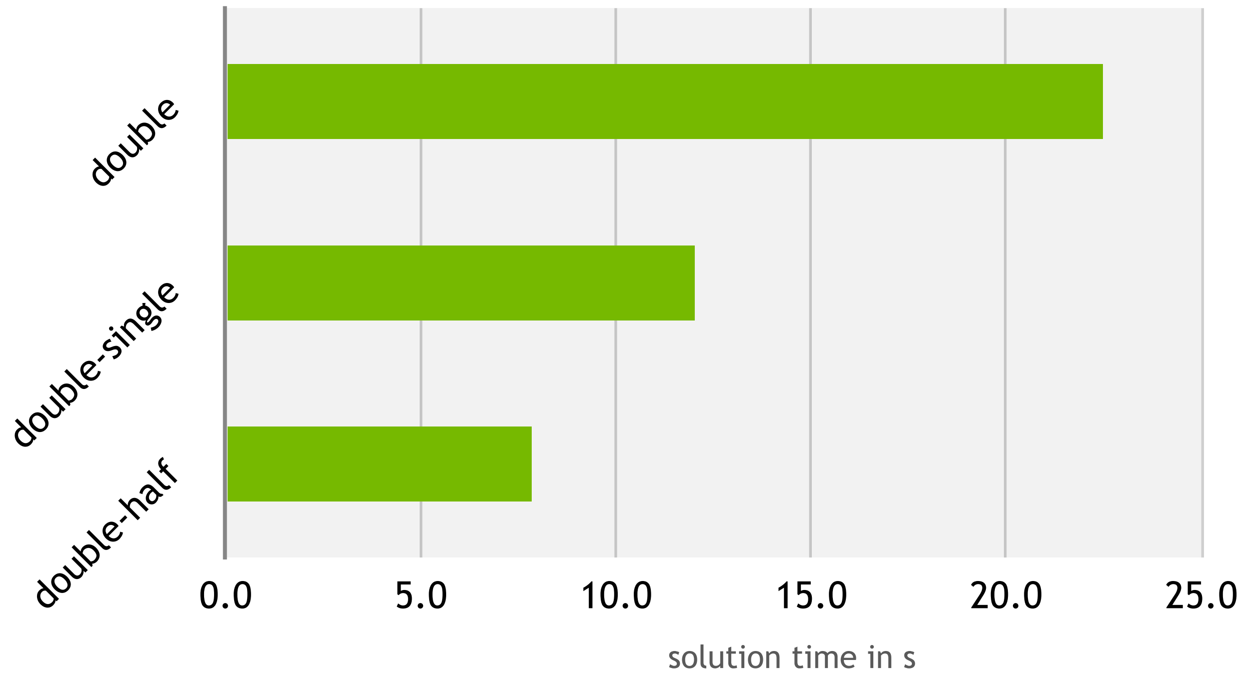
## Using your bits wisely

MILC/QUDA HISQ CG, mass = 0.001 => $\kappa \sim 10^6$




Tesla V100,
CUDA 10.1,
GCC 7.3,
QUDA 1.0

MILC/QUDA HISQ CG solver



# MIXED-PRECISION CG

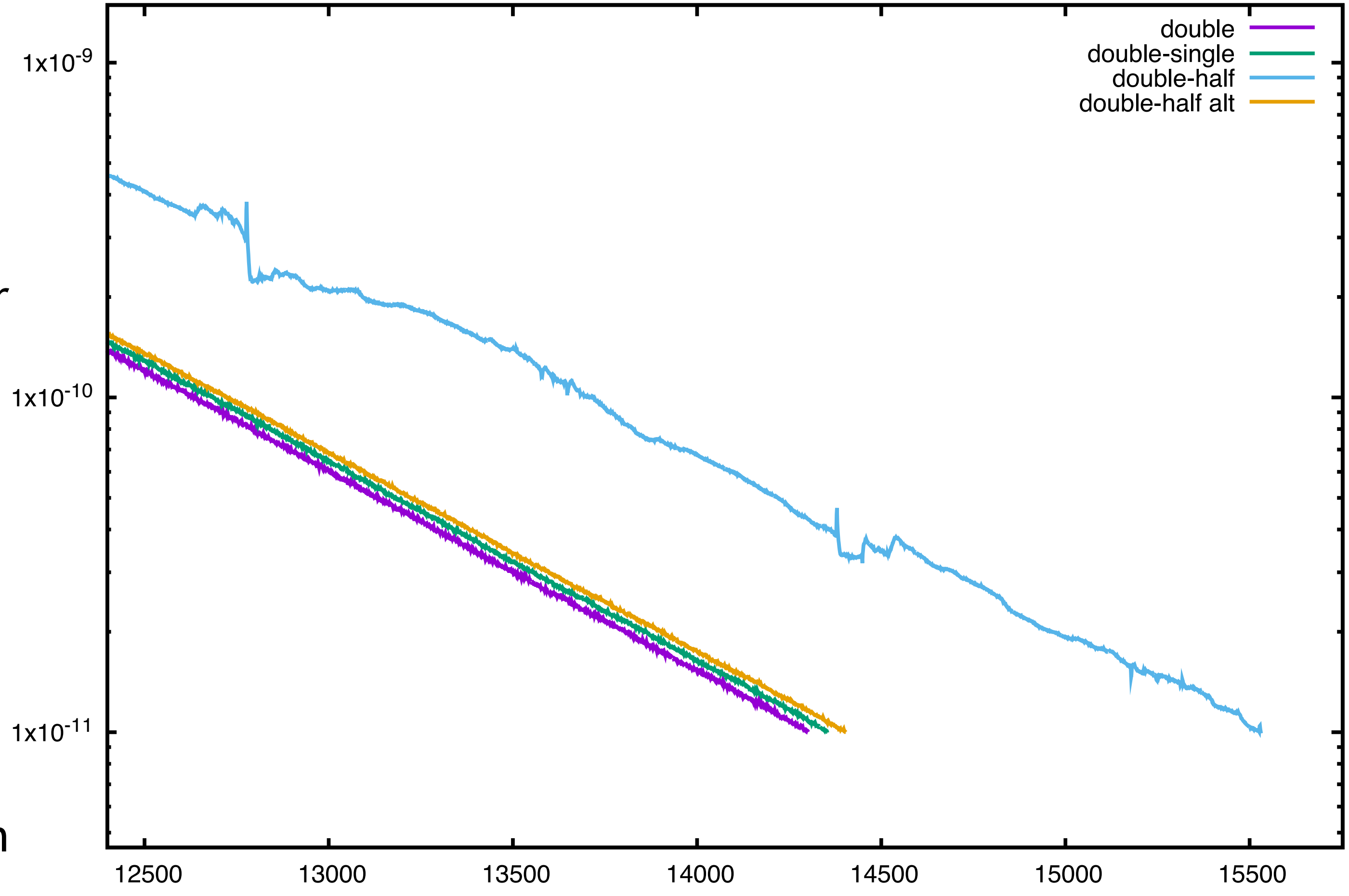
**double-half**

- Reliable update: periodic replacement of the residual with true residual in high precision
- Maintain solution vectors in high precision
    - Including the partial accumulator
- When true residual is injected, re-project the direction vector
- Use Polak-Ribière formula

$$\beta_k := \frac{\mathbf{z}_{k+1}^{\mathsf{T}} \left(\mathbf{r}_{k+1} - \mathbf{r}_k\right)}{\mathbf{z}_k^{\mathsf{T}} \mathbf{r}_k}$$

**double-half alt**

- Residual replacement strategy of van der Worst and Ye



# NEED FOR MORE PRECISION

Mixed-precision solvers have their limits

- Can break down once we can longer represent the linear system

$$\frac{\lambda_{\min}}{\lambda_{\max}} = \kappa^{-1} < \epsilon$$

- Explicit orthogonolization can become unstable
- Co-linearity break down (multi-shift solver)

Performance is dictated by memory bandwidth

- => Can we increase precision without increasing the memory traffic?

# MORE PRECISION AT CONSTANT BITS

$\epsilon \sim 3 \times 10^{-5}$

```
struct vector3_half {
  int16_t v[6];
  float max;
};
```

$\epsilon \gtrsim 3 \times 10^{-6}$

```
struct spinor_20 {
  int20_t v[6];
  uint8_t exponent;
};
```

128 bits

```
struct spinor3_fp32 {
  float v[6];
};
```

$\epsilon \sim 1 \times 10^{-7}$

```
struct spinor_30 {
  int30_t v[6];
  uint8_t exponent;
};
```

$\epsilon \gtrsim 2 \times 10^{-9}$

192 bits

```
template <> struct spinor_packed<30> {
  static constexpr unsigned int bitwidth = 30;
  static constexpr float scale = get_scale<bitwidth>();

  unsigned int a_re : bitwidth;
  unsigned int exponent0: 2;

  unsigned int a_im : bitwidth;
  unsigned int exponent1: 2;

  unsigned int b_re : bitwidth;
  unsigned int exponent2: 2;

  unsigned int b_im : bitwidth;
  unsigned int exponent3: 2;

  unsigned int c_re : bitwidth;
  unsigned int dummy0: 2;

  unsigned int c_im : bitwidth;
  unsigned int dummy1: 2;

  spinor_packed() = default;
  template <typename spinor> __host__ __device__ spinor_packed(const spinor &in) { pack(in); }


  template <typename spinor> __host__ __device__ inline void unpack(spinor &v)
  {
    // reconstruct 30-bit numbers
    unsigned int vu[6];
    vu[0] = a_re;
    vu[1] = a_im;
    vu[2] = b_re;
    vu[3] = b_im;
    vu[4] = c_re;
    vu[5] = c_im;

    // convert to signed
    int vs[6];
    for (int i = 0; i < 6; i++) memcpy(vs + i, vu + i, sizeof(int));

    // signed extend to 32 bits and rescale
    float_structure fs;
    fs.f = 0;
    fs.s.exponent = exponent0 + (exponent1 << 2) + (exponent2 << 4) + (exponent3 << 6);

    using real = decltype(v[0].real());
    for (int i = 0; i < 3; i++) {
      v[i].real(static_cast<real>(signextend<bitwidth>(vs[2 * i + 0])) * fs.f);
      v[i].imag(static_cast<real>(signextend<bitwidth>(vs[2 * i + 1])) * fs.f);
    }
  }
```

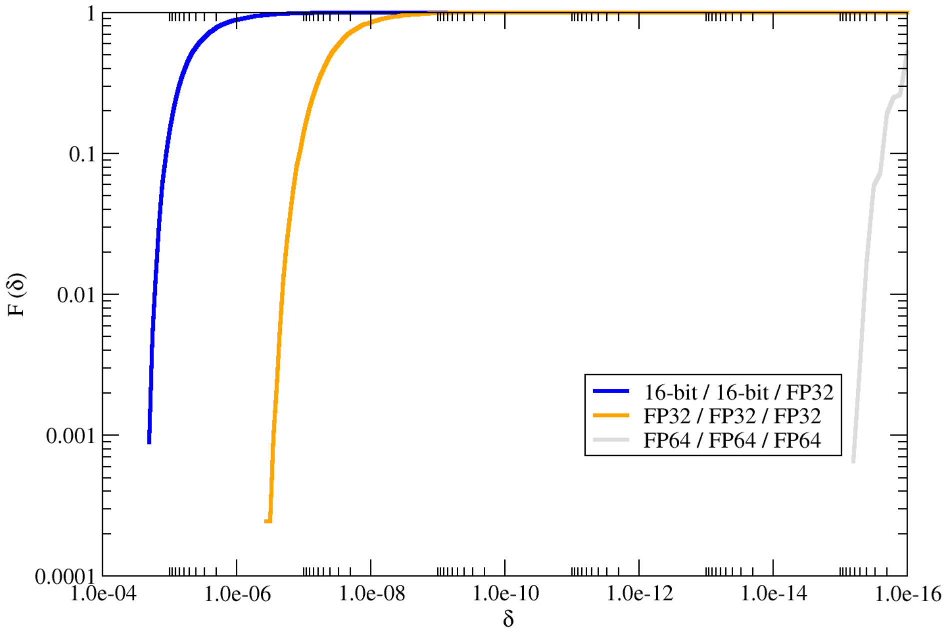
```
template <typename spinor>  __host__ __device__ inline void pack(const spinor &in)
 {
   // find the max
   float max[2] = {fabsf(in[0].real()), fabsf(in[0].imag())};
   for (int i = 1; i < 3; i++) {
     max[0] = fmaxf(max[0], fabsf(in[i].real()));
     max[1] = fmaxf(max[1], fabsf(in[i].imag()));
   }
   max[0] = fmaxf(max[0], max[1]);

   // ensures correct max covers all values if input vector is higher precision
   if (sizeof(in[0].real()) > sizeof(float))
     max[0] += max[0] * std::numeric_limits<float>::epsilon();

   // compute rounded up exponent for rescaling
   float_structure fs;
   fs.f = max[0] / scale;
   fs.s.exponent++;
   fs.s.mantissa = 0;

   // pack the exponent
   exponent0 = fs.s.exponent >> 0;
   exponent1 = fs.s.exponent >> 2;
   exponent2 = fs.s.exponent >> 4;
   exponent3 = fs.s.exponent >> 6;

   // rescale and convert to integer
   int vs[6];
   for (int i = 0; i < 3; i++) {
     vs[2 * i + 0] = lrint(in[i].real() / fs.f);
     vs[2 * i + 1] = lrint(in[i].imag() / fs.f);
   }

   unsigned int vu[6];
   for (int i = 0; i < 6; i++) memcpy(vu + i, vs + i, sizeof(int));

   // split into required bitfields
   a_re = vu[0];
   a_im = vu[1];
   b_re = vu[2];
   b_im = vu[3];
   c_re = vu[4];
   c_im = vu[5];
 }
```

# Hidden in the QUDA accessors
# Write once and used library wide

# HOW WELL DOES IT WORK?

## Precision: gauge / fermion / compute

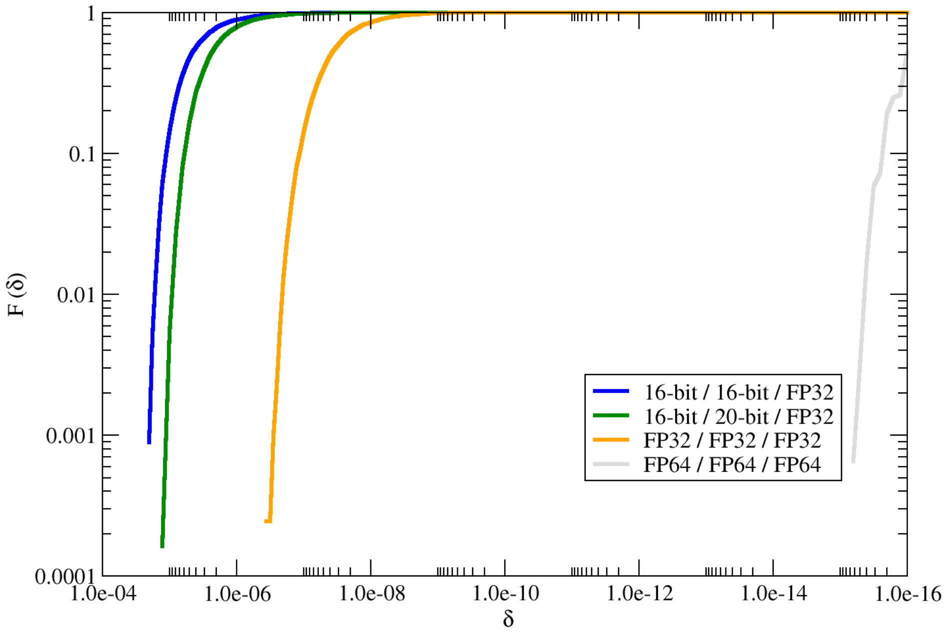
HISQ Dslash element-by-element absolute deviation CDF vs FP64 reference



# HOW WELL DOES IT WORK?

## Precision: gauge / fermion / compute

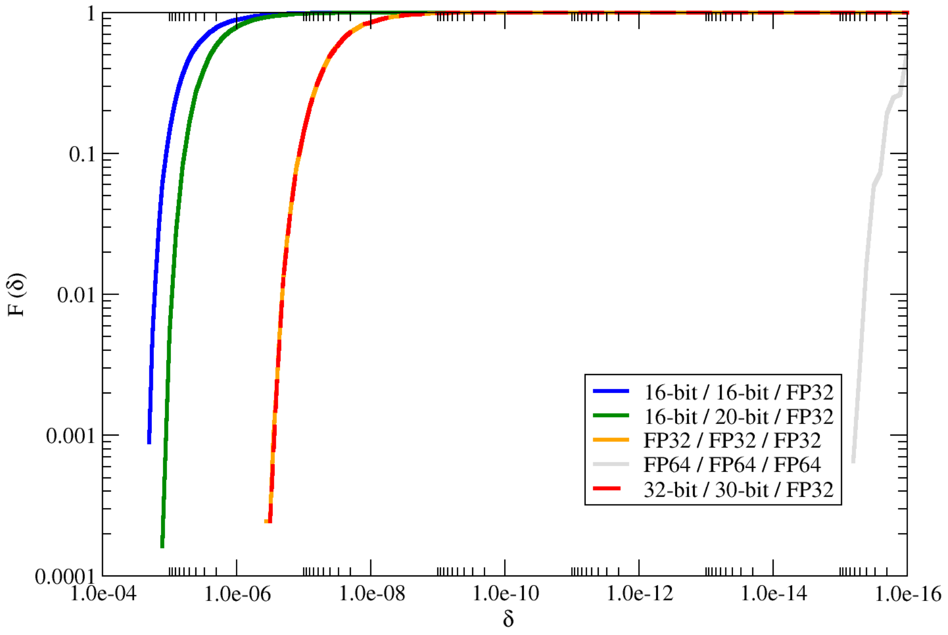
HISQ Dslash element-by-element absolute deviation CDF vs FP64 reference



# HOW WELL DOES IT WORK?

## Precision: gauge / fermion / compute

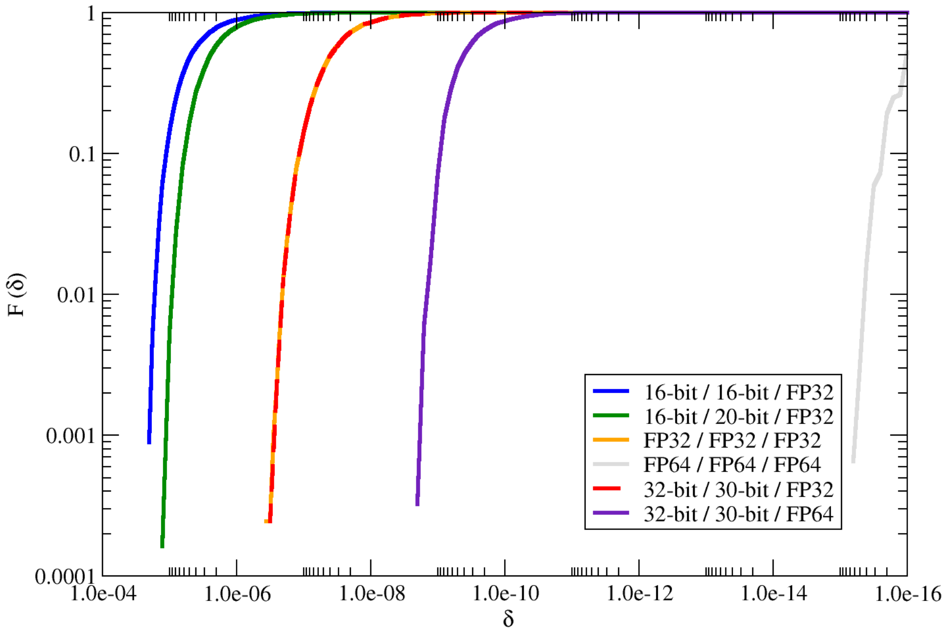
HISQ Dslash element-by-element absolute deviation CDF vs FP64 reference



# HOW WELL DOES IT WORK?

## Precision: gauge / fermion / compute

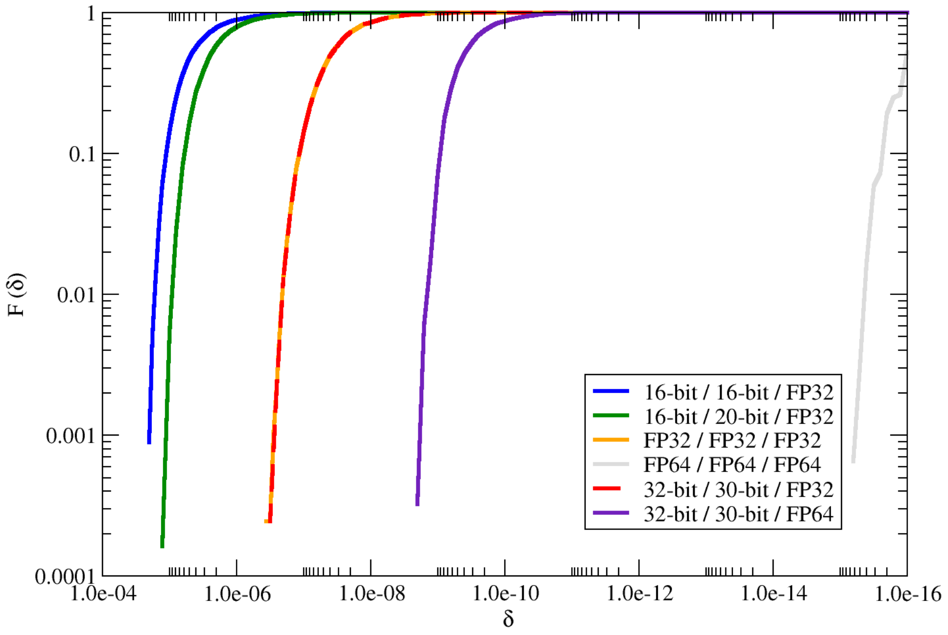
HISQ Dslash element-by-element absolute deviation CDF vs FP64 reference



# HOW WELL DOES IT WORK?

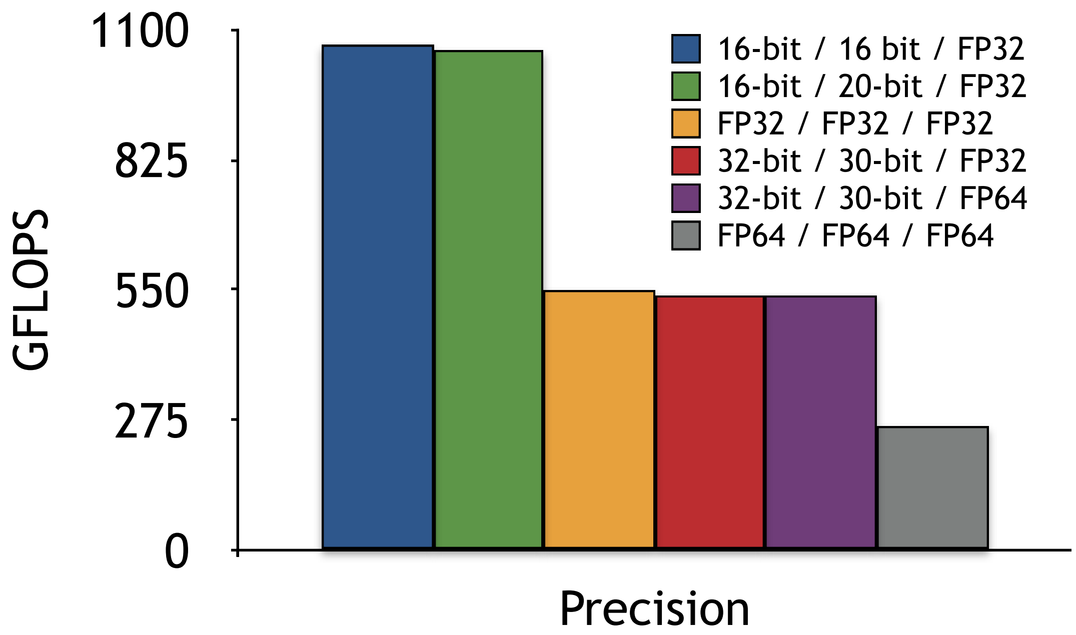
## Precision: gauge / fermion / compute

HISQ Dslash element-by-element absolute deviation CDF vs FP64 reference




HISQ Dslash Performance

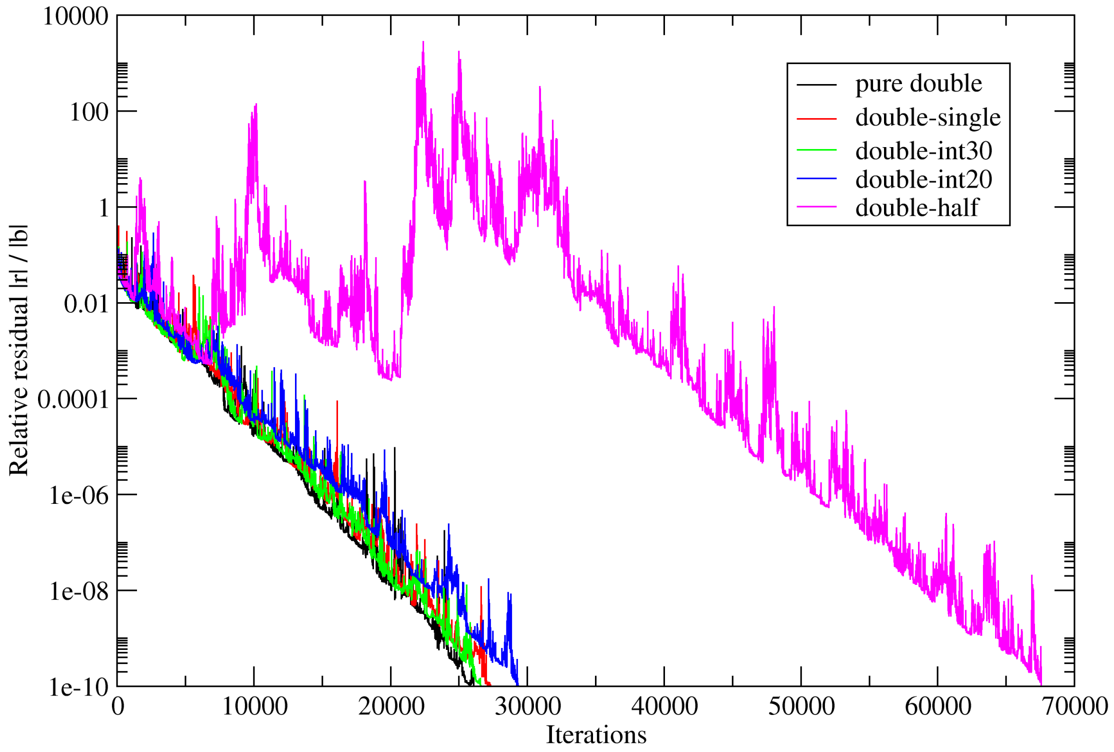
V = $32^4$, Quadro GV100




Gain two orders of magnitude in precision with no performance cost

# BICGSTAB(4)

## HISQ, V = $36^3$x72, β = 6.3, m = 0.001




| | Iterations | Time (s) |
|---|---|---|
| pure double | 26064 | 307 |
| double-single | 27308 | 159 |
| double-int30 | 26580 | 150 |
| double-int20 | 29336 | 106 |
| double-half | 67552 | 247 |

# MULTI-SHIFT CG SOLVER

Used for RHMC and multi-mass solver propagators

Mixed-precision multi-shift CG

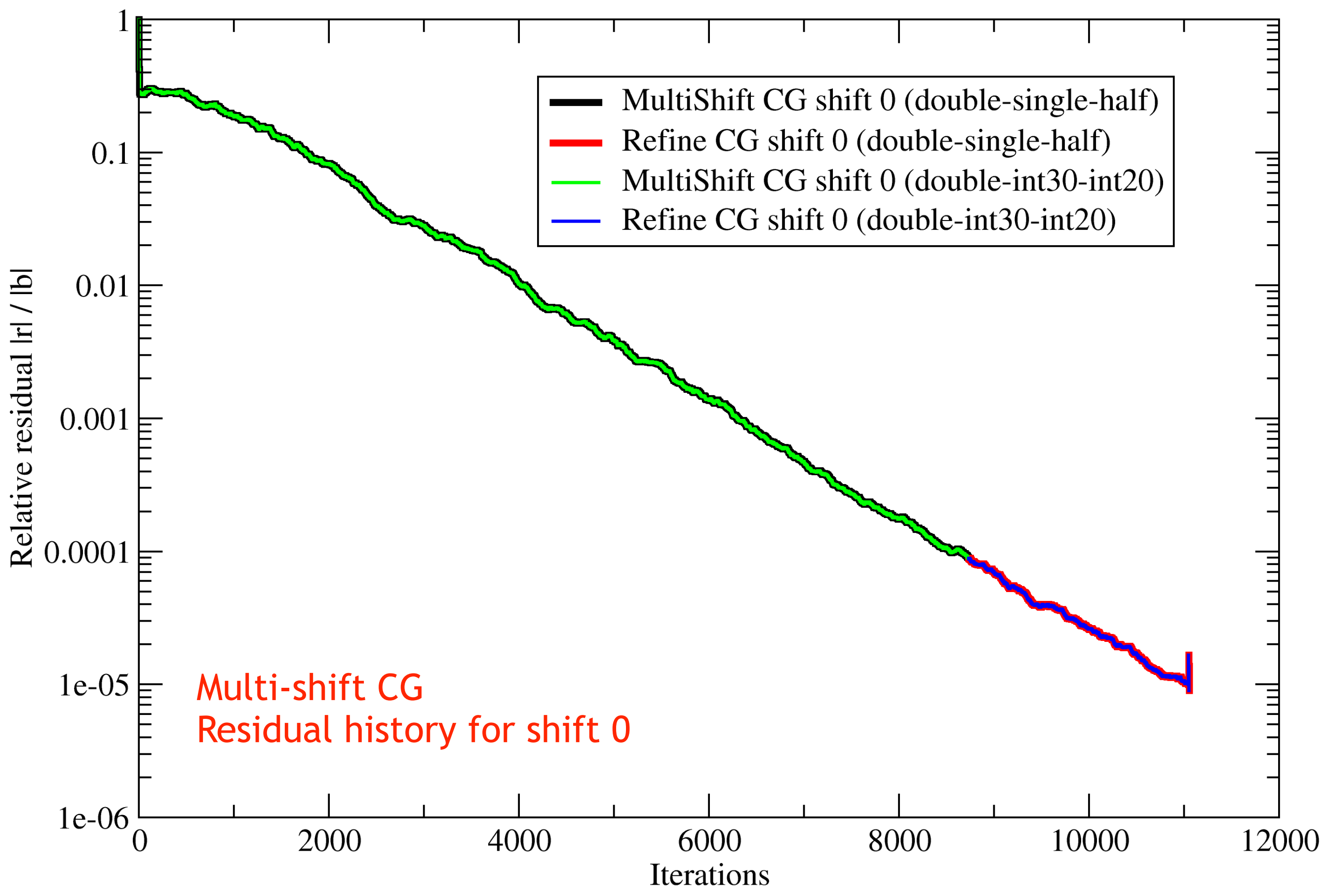
- Essentially mixed-precision CG on shift 0
- Shifted iterated residuals drift away true residual
- Refine each shifted system to correct for lack of residual collinearity
- Many additional iterations can be required

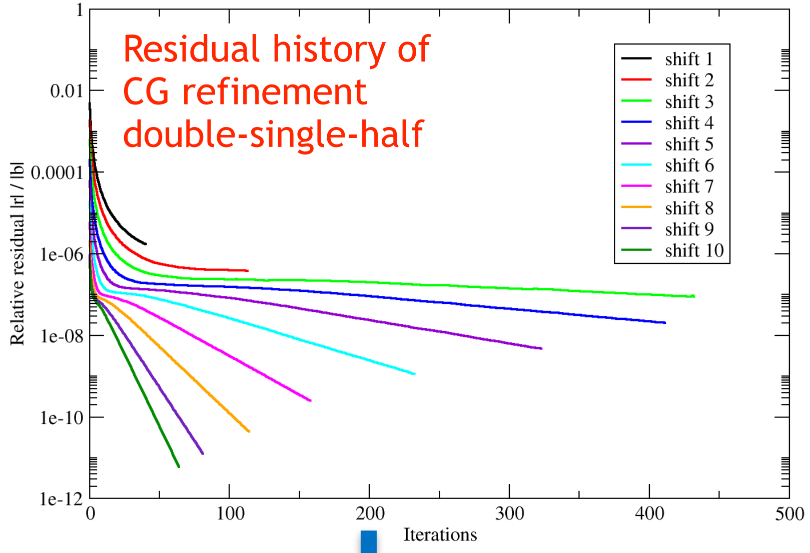
Prior optimal QUDA strategy

- double-single multi-shift-CG
- double-half per shift refinement

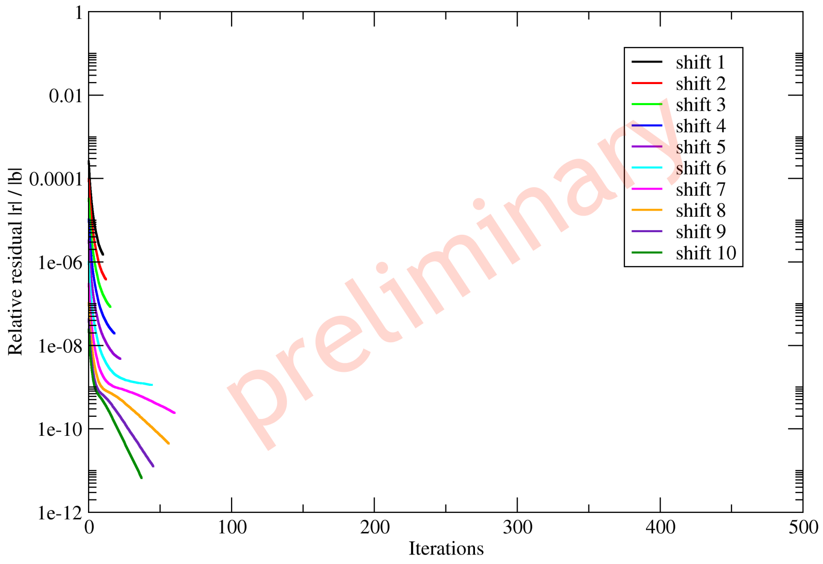
# MULTI-SHIFT SOLVER

HISQ RHMC, $V = 36^3x72$, $\beta = 6.3$, $m = 0.001$, 11 shifts




Significant reduction in refinement iterations



# SUMMARY

LQCD has different precision requirements than IEEE floating-point

No need to couple the computation precision to storage format

Custom precision formats can do significantly better while having negligible overhead

Dramatic improvement in solver stability is possible

Outlook

- Bit-packed storage formats ideal for offline storage, e.g., eigenvectors
- For going beyond double precision, we can do better than float128

NVIDIA