Progress on QDP-JIT: Adding support for AMD GPUs

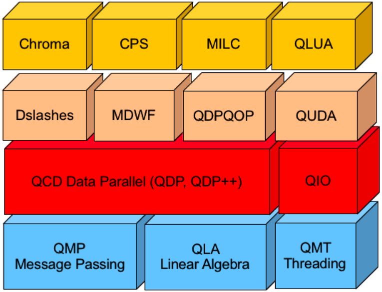
Frank Winter

Jefferson Lab

Lattice Conference 2021

USQCD Lattice QCD Software Stack

- Application: Chroma, implements HMC/RHMC, propagators, sequential sources, contractions, etc.
- Data-parallel interface: QDP++, provides C++ data types and operations
- Message passing: QMP, thin layer for MPI



QDP++ Data Types

	Lattice	Spin	Color	Reality	Type
LatticeColorMatrix	Lattice	Scalar	Matrix	Complex	REAL
LatticeFermion	Lattice	Vector	Vector	Complex	REAL
LatticePropagator	Lattice	Matrix	Matrix	Complex	REAL
Real	Scalar	Scalar	Scalar	Real	REAL

- Implemented as nested C++ template classes
- Configure time parameters:
  Nd (no. of dim), Ns (no. of spin), Nc (no. of color)
- Base precision: REAL (single/double)
- Single lattice geometry

QDP++ Operations

- Operations as C++ expressions

```
LatticeColorMatrix u;
gaussian(u);
LatticePropagator Qanti = Gamma(G5) * Q * Gamma(G5);
LatticeComplex z = trace( adj(Qanti) * Gamma(n) * Q * Gamma(n) );
```

- PETE (Portable Expression Template Engine)
  → standard C++
  → non-intrusive ET traversals (builder, addresses)

QDP++ Operations: Shifts

Overlapping Comms in QDP-JIT

- Shifts perform a circular shift along the specified dimension

```
(LatticePropagator Q,G;)
(multi1d< LatticeColorMatrix > u(Nd);)

G -= u[mu] * shift( Q , FORWARD , mu ) + shift( adj( u[mu] ) * Q , BACKWARD , mu )
```

- blocking in QDP++, 4 separate evaluations:

```
LatticePropagator Qa = shift( Q , FORWARD , mu );
LatticePropagator Qb = adj( u[mu] ) * Q;
LatticePropagator Qc = shift( Qb , BACKWARD , mu );

G -= u[mu] * Qa + Qc;
```

- QDP-JIT:
  shifts evaluate as part of the expression
  $\rightarrow$ single evaluation
  $\rightarrow$ overlapping comms (nearest neighbor)

“Chroma on GPUs”: Requirements

- No code changes to Chroma
  $\rightarrow$ GPU support on the QDP++ API layer
- Efficient GPU kernels
  $\rightarrow$ efficient data layout for GPU
- Minimize (eliminate, ideally) data migration
  $\rightarrow$ automatic memory management with a “cache”

Why Just-In-Time Compilation?

Early days' motivation for JIT:

- CUDA 4/5 (Oak Ridge Titan) had no C++ interface/or pass-by-reference to kernels
  $\rightarrow$ no expression template support
- Still, attempts had been made (CUDA code generator + NVCC callout)
  $\rightarrow$ However, not possible for Titan (missing SDK)
- QDP-JIT/PTX came along
  $\rightarrow$ PTX codegen, dynamic loading with NVIDIA driver
  $\rightarrow$ Math library support was ugly
  $\rightarrow$ (together with QUDA) this enabled the full RHMC to execute on the GPUs

Why Just-In-Time Compilation? (cont.)

- QDP-JIT transitioned to using LLVM to build the kernels
  → Rigorous support for math library
  → PTX versioning done right
- Can use standard compiler like GCC or Clang without relying on vendor-specific ones (NVCC, HIPCC)
  → Fast, mature compilers
- Additional point of inspection
  C++ → LLVM IR → NV PTX/AMD GCN (→ NV SASS)
  → Less likely to hit a compiler bug as transitions are smaller
- No need for target-specific routines (e.g. reductions)
- Dynamic specialization: QDP-JIT optimizes kernels on
  → the virtual machine geometry (more later)
  → (data type size, e.g., matrix size. *possibility)

LLVM Compiler

- Modern compiler infrastructure
- Front-ends for common and new languages (C++, C, Fortran, Julia,...)
- LLVM IR, representation for all stages and languages
- Middle-ends for IR optimization
- Back-ends for all relevant HPC targets (CUDA, GCN, X86, AVX512, ...)

How does it work?

- QDP-JIT can be seen as an LLVM front-end and runtime for the QDP++ API
  (not a stand-alone frontend, however)

Front-end:

- In C++ expression template implementation, replace *evaluate* → *build_evaluate*
- *build_evaluate* uses LLVM IR builder API
  → GPU kernel in LLVM IR

Run-time:

- Dynamically compile the kernel using back-end NVPTX or AMDGPU
- Dynamically load and launch resulting kernel

Adding support for AMD GPUs

- LLVM AMDGPU backend provides ISA code generation for AMD GPUs
- GCN GFX9 (Vega) relevant for Frontier/JLab
- No documentation on how to write kernels at IR level
  → Reverse-engineering from HIP/Clang
- Differences compared to CUDA target:
  → workgroup geometry intrinsics
  → kernel ABI
  → stack allocation
  → math support via OpenCL library
- Call-out to linker (object → shared object)

Recent improvements

Shift operations

- Dynamic specialization of `shifts` on virtual compute node geometry (`-geom X Y Z T`)
- "Fine-grained" single/multi GPU build
  → each lattice dimension selected as single/multi GPU
- Example: Wilson DSlash (even/odd)

```
chi[rb[1]] =
    spinReconstructDir0Minus(u[0] *     shift(spinProjectDir0Minus(psi),  FORWARD, 0))
  + spinReconstructDir0Plus(shift(adj(u[0]) * spinProjectDir0Plus (psi), BACKWARD, 0))
  + spinReconstructDir1Minus(u[1] *     shift(spinProjectDir1Minus(psi),  FORWARD, 1))
  + spinReconstructDir1Plus(shift(adj(u[1]) * spinProjectDir1Plus (psi), BACKWARD, 1))
  + spinReconstructDir2Minus(u[2] *     shift(spinProjectDir2Minus(psi),  FORWARD, 2))
  + spinReconstructDir2Plus(shift(adj(u[2]) * spinProjectDir2Plus (psi), BACKWARD, 2))
  + spinReconstructDir3Minus(u[3] *     shift(spinProjectDir3Minus(psi),  FORWARD, 3))
  + spinReconstructDir3Plus(shift(adj(u[3]) * spinProjectDir3Plus (psi), BACKWARD, 3));
```

- Receive buffers (and code) only for parallelization dimensions

Recent improvements (cont.)

Baryon contractions

- Baryon contractions example:

```
Q = quarkContract13(Q,Q*Q);
q_bar_q = localInnerProduct( Q , Gamma(i) * Q * Gamma(j) );
```

- Previously kernels code was unrolled to register level
  $\rightarrow$ huge program code
  $\rightarrow$ heavy register spilling
  $\rightarrow$ long compilation time
- Now: Add JIT loops on spin level
  Improvement on all propagator expressions!
  Above example (NVIDIA GPU):
  Local mem: 7288 bytes $\rightarrow$ 144 bytes, Registers: 255 $\rightarrow$ 56
  Compilation in order of milli-seconds even on large baryon contraction routines

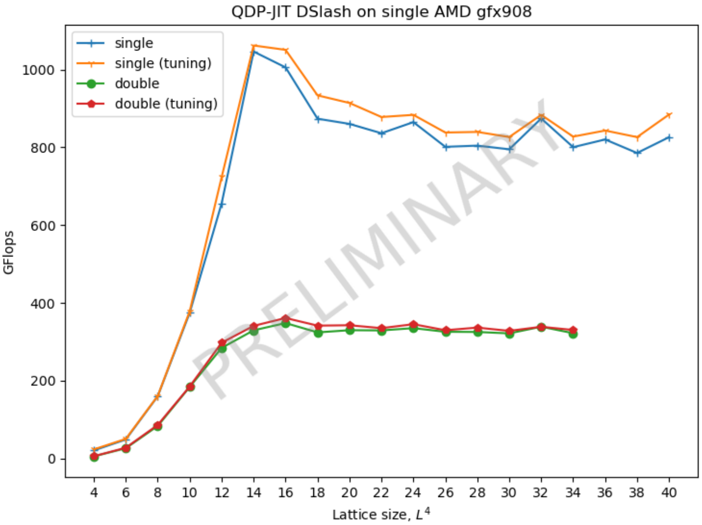
Various recent improvements

- Transitioned to use LLVM 12
- HIP/CUDA context synchronization removed
- Improved handling of Scalars and sumMulti
- Gamma algebra implemented as sparse matrix multiplication
- Cosmetic change for the builder functions:
  Added helpers which mimic high level language constructs such as: 'if', 'for', and 'switch'.

```
JitForLoop loop_k(0,Ns);
{
  JitForLoop loop_j(0,Ns);
  {
    D.elem( loop_k.index() , loop_j.index() ) = rhs.elem();
  }
  loop_j.end();
}
loop_k.end();
```

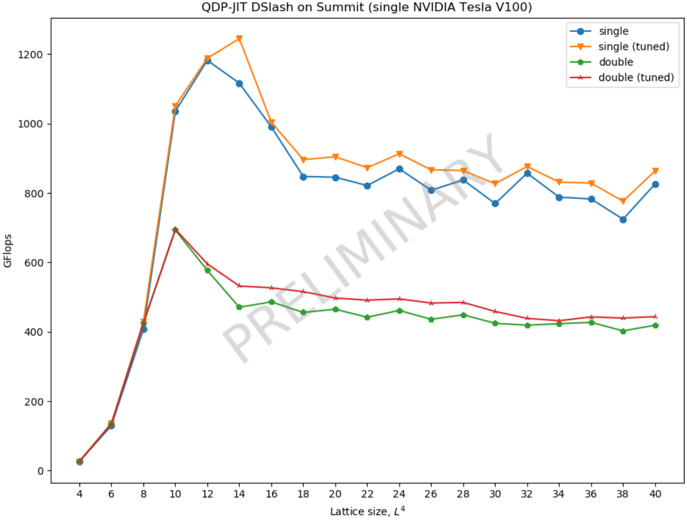
- Reduced time: 40% hadspec, 10-15% three-point/seqsrc, 15% HMC

Performance on MI100 GPU (AMD)



This work utilized the early-access system Tulip hosted by HPE and supported by the HPE and AMD staff of the Frontier Center of Excellence.

Performance on Volta GPU (NVIDIA)



Conclusion & Outlook

- Chroma/QDP-JIT production ready for NVIDIA and AMD GPUs

- Intel GPUs support, via LLVM SPIR-V backend and Intel oneAPI
- AVX support
- Support for multiple lattice geometries

- `https://github.com/JeffersonLab/qdp-jit.git`