QDP-JIT: Portable JIT Compilation for Lattice QCD

Multi-Architecture GPU Support and Multi-Grid Development

Frank Winter

Jefferson Lab

Lattice 2026
University of Maryland, College Park

Chroma, QDP++, and QDP-JIT: Overview

Chroma

- `github.com/JeffersonLab/chroma`
- Developed under SciDAC
- High-level Lattice QCD C++ application framework
- Implements physics algorithms and workflows
  - Gauge generation (e.g. HMC)
  - Linear solvers
  - Measurements and analysis
- Physics code is written in terms of lattice-wide objects and expressions
- Intended to be portable across architectures

QDP++

- `github.com/JeffersonLab/qdpxx`
- Data-parallel programming layer used by Chroma
- Provides lattice data types:
  - Gauge fields
  - Fermion fields
  - Scalars, vectors, matrices
- Provides lattice operations and expression templates
- Lets Chroma describe *what* to compute, independent of low-level implementation

Chroma, QDP++, and QDP-JIT: Overview

QDP-JIT

- `github.com/JeffersonLab/qdp-jit`
- Re-implementation of the QDP++ interface
- Preserves the same high-level programming model
- Generates and launches **optimized GPU kernels** at run time
- Targets modern accelerator architectures
- Keeps data resident on the device and minimizes host/device data traffic

Key Idea

- Chroma algorithms can remain essentially unchanged
- QDP++ / QDP-JIT forms the abstraction layer underneath
- Backend execution can evolve without rewriting the physics code

Software Stack

Chroma $\Longrightarrow$ QDP++ / QDP-JIT $\Longrightarrow$ CPU / GPU Execution

Why JIT for QDP-JIT? No binary bloat

Static expression-template builds

- Every expression variant must be compiled ahead of time
- Matrix type, precision, backend, and expression combinations multiply quickly
- Host compilers can spend hours instantiating templates
- Linkers may struggle with very large generated object code

Result

Large binaries full of kernels that a given HMC run may never execute.

QDP-JIT approach

- Keep the host build lightweight
- Build the expression AST at runtime
- Emit LLVM IR only for expressions that are actually evaluated
- Generate exactly the kernels needed for the current run

Main advantage

- Zero binary bloat
- No monolithic template-instantiation explosion.

Why JIT for QDP-JIT? Toolchain Decoupling

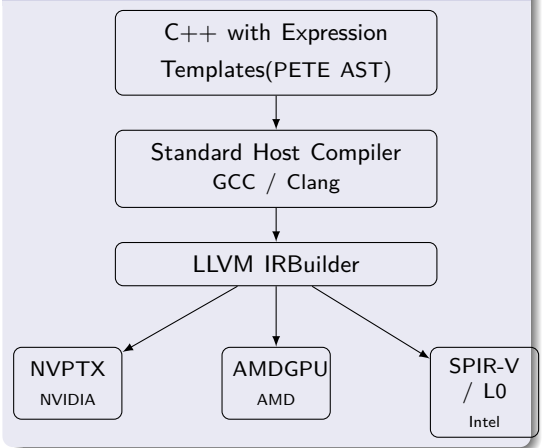
Problem with vendor C++ offload toolchains

- Heavy C++ expression templates stress SYCL / OpenMP / CUDA front-ends
- Compiler bugs and regressions are common during early-access bring-up

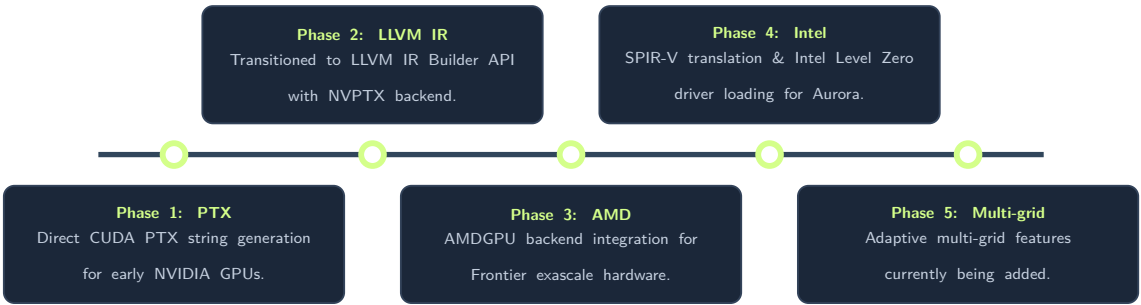
QDP-JIT strategy

- Compile host code with a standard compiler (e.g. GCC, Clang)
- Lower the parallel expression AST directly with LLVM IRBuilder
- Skip vendor C++ front-ends entirely

QDP-JIT compilation path



Evolution of QDP-JIT

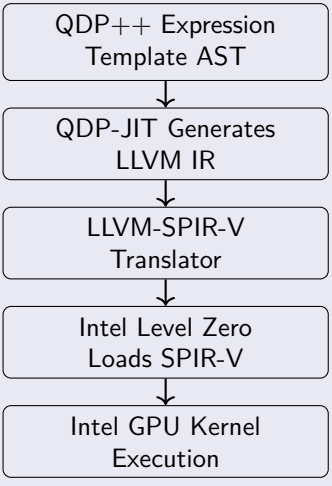


SPIR-V as the Intel GPU Kernel Boundary

What is SPIR-V?

- Standard Portable Intermediate Representation
- Open binary IR defined by the Khronos Group
- Used for GPU compute and graphics kernels
- Consumed by runtimes such as OpenCL, Vulkan, and Intel Level Zero
- Not native GPU machine code: the driver still lowers it to device instructions

QDP-JIT Intel GPU path



$N_f = 2 + 1$ dynamical Wilson Clover HMC on ANL Aurora

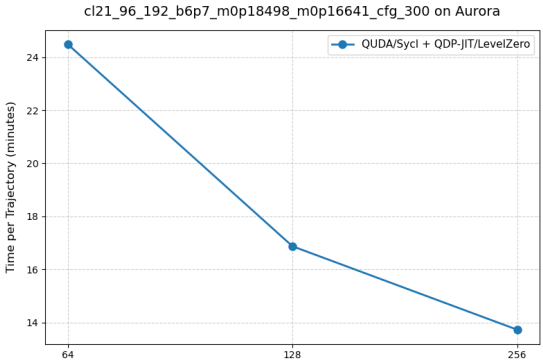
Aurora Nodes	QDP-JIT +QUDA	QDPXX +QUDA	QUDA $M^\dagger M$	QUDA $M + \rho_i$	total QUDA	QUDA / Chroma-JIT	QUDA / Chroma-CPU
64	1469.1 s	14979.7 s	1138.62 s	284.02 s	1422.641 s	96.84%	9.50%
128	1012.3 s	7755.9 s	794.01 s	214.34 s	1008.352 s	99.61%	13.00%
256	823.5 s	4085.1 s	648.31 s	161.66 s	809.964 s	98.36%	19.83%

Setup

$\tau = 1.414$, 14 MD steps, force-gradient integrator. Light 2-flavor $M^\dagger M$ solves use QUDA multigrid. One-flavor RHMC uses QUDA multi-shift CG and chrono extrapolator.

Parameters

$m_\pi = 170\,\text{MeV}$, $\beta = 6.7$, $L_x/a = 96$, $L_t/a = 192$,



Scaling diagnosis: the missing time is large bulk Chroma/QDP work

Nodes	Extra QDPXX time	Ratio
64	13510.6 s	–
128	6743.6 s	$2.00\times$
256	3261.5 s	$2.07\times$

$$T_{\mathrm{QDPXX}} \approx T_{\mathrm{QUDA}} + T_{\mathrm{Chroma,CPU}},$$
$$T_{\mathrm{JIT}} \approx T_{\mathrm{QUDA}} + T_{\mathrm{Chroma,GPU}}.$$

Interpretation

The QDPXX–QDP-JIT difference nearly halves when the node count doubles. That is the signature of full-volume lattice operations, not a fixed startup cost.

Consequence

After QDP-JIT removes CPU-side lattice algebra, the trajectory exposes the stronger-scaling limit: QUDA coarse-grid work, communication, global reductions, and latency.

Expensive non-solver components

1. Wilson–Clover fermion-force assembly
2. RHMC force accumulation over rational poles
3. Clover-term derivative
4. Extra force work from the force-gradient integrator
5. M, $M^\dagger$, or Schur-complement applications outside the inverter
6. Gauge force and gauge updates
7. Pseudofermion generation
8. Hamiltonian evaluation
9. Residual checks

Key point

At near-physical light mass, solves are expensive, but the trajectory also contains many repeated full-lattice algebra operations surrounding those solves.

New QDP-JIT Features for Adaptive Multi-Grid

Variable Current Lattice Size

- Introduce notion of a *current lattice size*, can be changed dynamically.
- Newly created lattice objects use the current lattice size.
- Fine-grid and coarse-grid coexist naturally.

Mixed-Size Expressions

- Expressions may combine lattice objects of different sizes.
- Smaller lattices propagate their site values to the corresponding larger-lattice sites.
- This provides the basis for coarse-to-fine operations.

Blocked Reductions

- Reductions can be restricted to lattice blocks, e.g. local inner products on each aggregate.
- Needed for coarse basis construction and adaptive AMG setup.

Example

```
multi1d<int> blocking = {4,4,4,4};
MG::push_blocking(blocking);

LatticeComplexD a =
  blockSum(
    localInnerProduct(vecs_1[j], vecs_1[i]),
    blocking
  );
```

Build Chroma + QDP-JIT + QUDA

https://github.com/fwinter/package

1. Get the package & software

```
git clone https://github.com/fwinter/package
cd package
./download-sources.sh
```

2. Select backend

```
cd cuda/         # NVIDIA GPUs
cd rocm/         # AMD GPUs
cd level-zero/   # Intel GPUs
```

3. Configure environment

```
vim env.sh
```

4. Build everything

```
./build-all.sh
```