UNC/SUPER Status

Rob Fowler

Oct 19, 2014

(With Diptorup Deb and Allan Porterfield)

Overview

- Goal: Use compiler technology and auto-tuning to help close the "Ninja programmer performance gap" between simple, high-level code and hand-optimized LQCD routines, especially for new computer architectures.
- Problem statement: QDP++ is an embedded distributed-memory domain-specific language for LQCD computation implemented using C++ template meta-programming. QDP-JIT extends QDP++ to generate locally optimized Nvidia PTX code.
- Issues
  - Code for each assignment expression (statement) is generated independently without global analysis.
  - Memory management through run time software cache, no compiler analysis.
    - Excessive data movement, though most of this is to/from fast memory.
    - On Nvidia GPUs, memory bandwidth and data transfer are the rate limiters.
  - Per expression data movement (MPI) without global planning/coalescing/scheduling.
  - Pressure in the memory hierarchy: registers, caches, effects on pipelining.

Direction:

- **Tasks: With JLAB, add compiler analysis and code transformations to QDP-JIT.**
  - **Loop (expression) fusion to reduce memory traffic and generate tighter inner loops.**
  - **Static analysis to improve memory management and reduce CPU↔GPU traffic.**
  - **Optimize cluster-wide messaging operations across large units of code.**
  - **Generate auto-tuning hooks to address hard problems, *e.g.*, performance portability.**
  - **(LLVM chosen to leverage Intel and Nvidia infrastructures for accelerator devices.)**

Recent Activities

- Graduate Student (Diptorup Deb) hired and being brought up to speed.

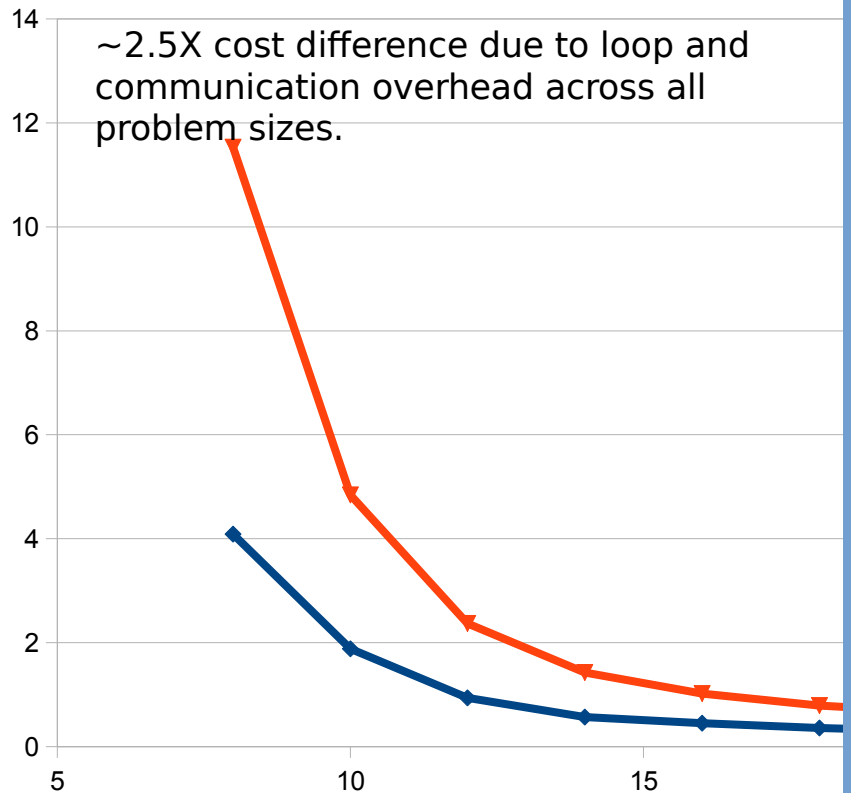
- Selected LLVM as the compilation framework.
  - Adopted by Nvidia as its compilation base.
  - Modern, clean, open source design.
  - Growing support in academia and industry.
    - We can leverage other projects..

- Selected a few simple examples to drive experimentation.
  - QDP-JIT examples to expose CPU to device communication issues as well as on-device memory and code generation issues.
  - Traditional QDP++ and Chroma examples generating CPU code to experiment with performance issues there and as a simpler target for compiler experiments.

Toy Problem : Performance effects of loop splitting/fusion.

One expression vs. One operator per expression on a single node using one Tesla K20c

Cost per site for 100th repetition.



N	LatticeReal a,b,c
	V =N^4
		a = -(log(b)/c);	a = log(b);
			a = (a/c);
			A = -a;

2	1.600E+001	15751	45996
4	2.560E+002	15801	45525
6	1.296E+003	16911	46709
8	4.096E+003	16737	47239
10	1.000E+004	18834	48435
12	2.074E+004	19497	49130
14	3.842E+004	21895	54790
16	6.554E+004	29458	66973
18	1.050E+005	37683	82597
20	1.600E+005	47163	106913
22	2.343E+005	62389	137678
24	3.318E+005	77733	199135
26	4.570E+005	98372	245277
28	6.147E+005	128187	279876

Most of the performance difference can be attributed to excess CPU <--> GPU data movement.

Discussion

- Almost all of the differences can be explained in terms of Device --> Host data movement.
  - Specifically, host memory is updated after every kernel, even if the data object is not needed on the host.
- Problem: run time code instantiation/generation using expression templates is strictly on an (assignment expression) by expression basis.
  - Static compiler analysis of whole program is not available.
  - Instantiated code looks neither forward nor backwards.
  - Runtime support libraries for data caching can record history, but can't see forward.
  -

A More Realistic example.

Timing variants on the Dslash test from the QDP++ examples directory.

- Evaluate the tradeoffs between monolithic and fragmented versions.
- Tests over lattice size range from 2^4 to 28^4 for Nd= 4.
- Compilers used :
- QDP++ : GCC 4.8.1, ICC14.0.1,and Clang 3.4
- QDP-JIT : GCC 4.8.1

Dslash operator base case

```
if(isign> 0)
{
chi[rb[cb]] = 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))
+ spinRecnstructDir3Plus(shift(adj(u[3]) * spinProjectDir3Plus(psi), BACKWARD, 3))
;
}
```

Dslash operator two sub-expressions

```
if (isign> 0)
{
chi[rb[cb]] = 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));

chi[rb[cb]] += 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));
}
```

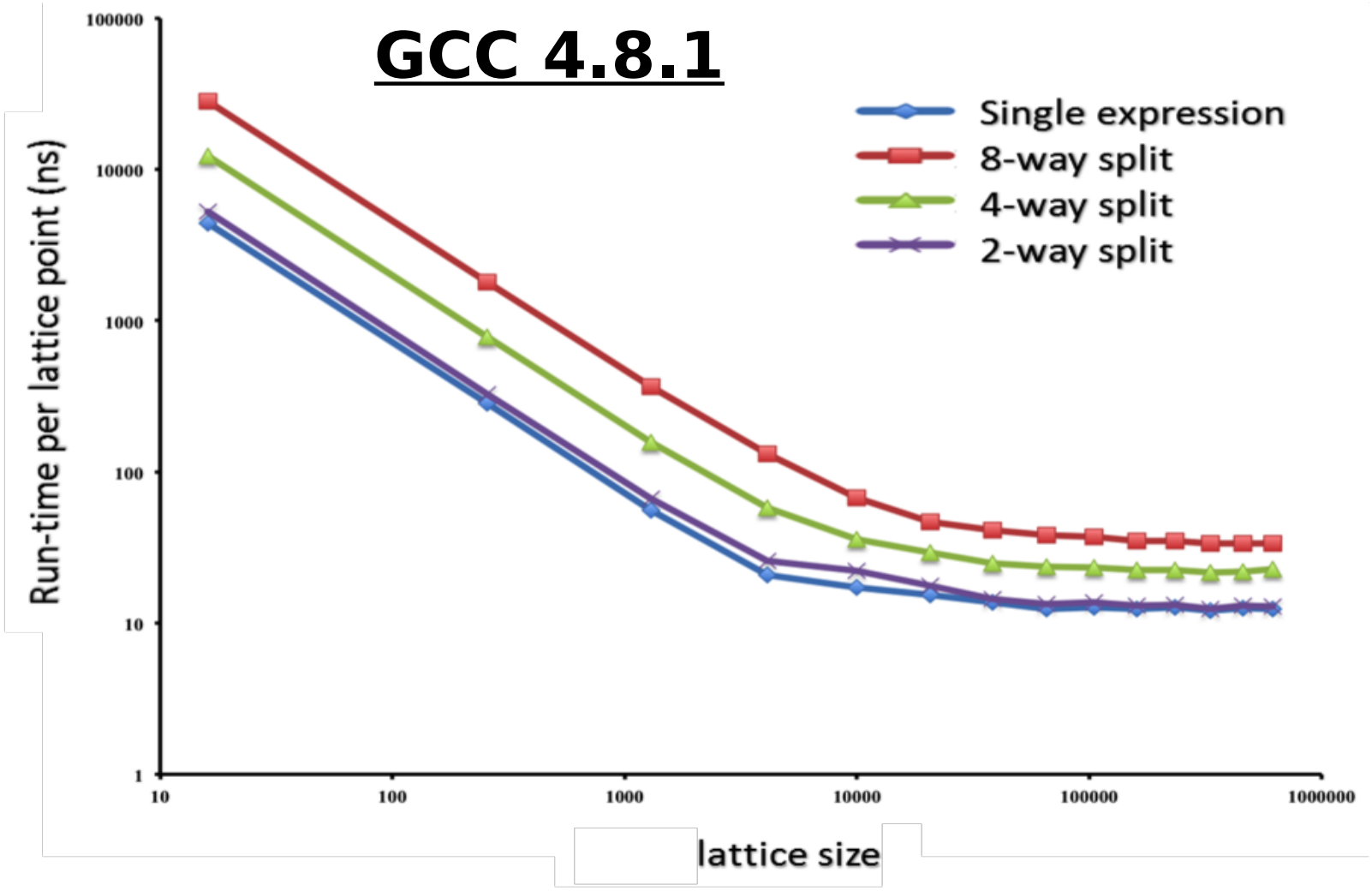
Dslash operator four sub-expressions

```
chi[rb[cb]] =
spinReconstructDir0Minus(u[0]*shift(spinProjectDir0Minus(psi), FORWARD, 0))
+ spinReconstructDir0Plus(shift(adj(u[0]) * spinProjectDir0Plus(psi), BACKWARD,
0));

chi[rb[cb]] +=
spinReconstructDir1Minus(u[1] * shift(spinProjectDir1Minus(psi), FORWARD, 1))
+ spinReconstructDir1Plus(shift(adj(u[1]) * spinProjectDir1Plus(psi), BACKWARD,
1));

chi[rb[cb]] +=
spinReconstructDir2Minus(u[2] * shift(spinProjectDir2Minus(psi), FORWARD, 2))
+ spinReconstructDir2Plus(shift(adj(u[2]) * spinProjectDir2Plus(psi), BACKWARD,
2));

chi[rb[cb]] +=
spinReconstructDir3Minus(u[3] * shift(spinProjectDir3Minus(psi), FORWARD, 3))
+ spinReconstructDir3Plus(shift(adj(u[3]) * spinProjectDir3Plus(psi), BACKWARD,
3));
```

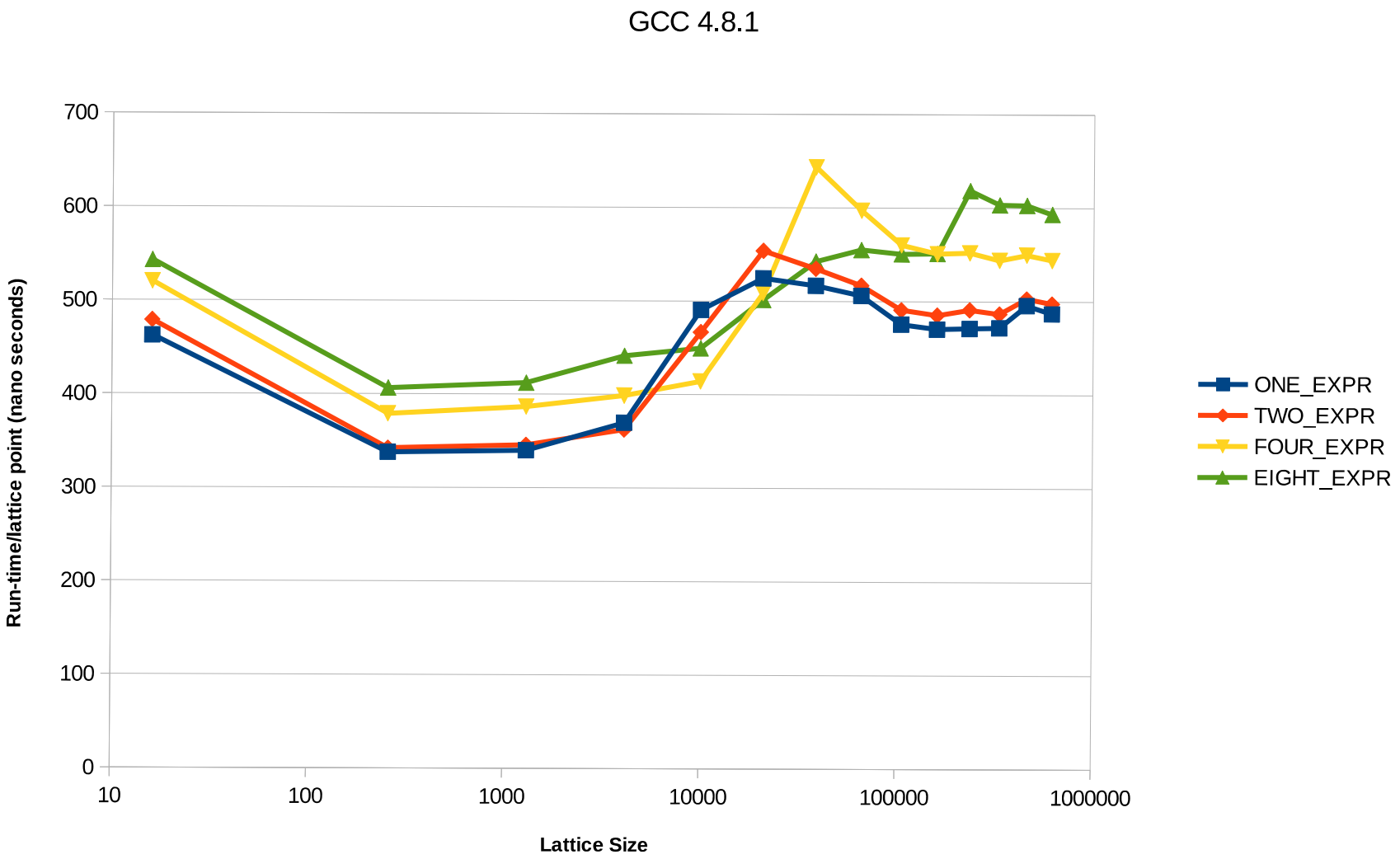
Dslash operator eight sub-expressions

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

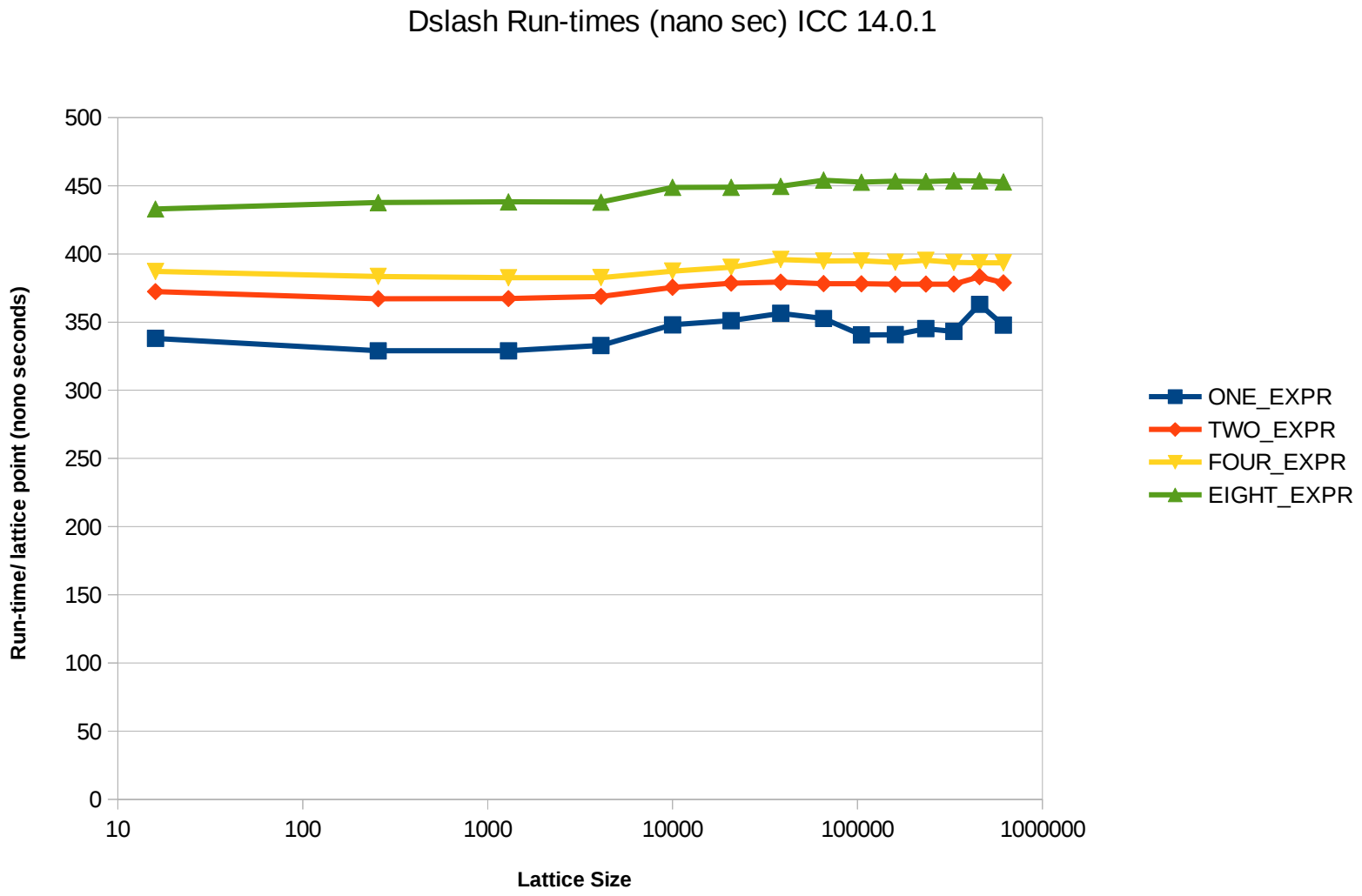
Evaluating effect of loop splitting/fusion (QDP-JIT)



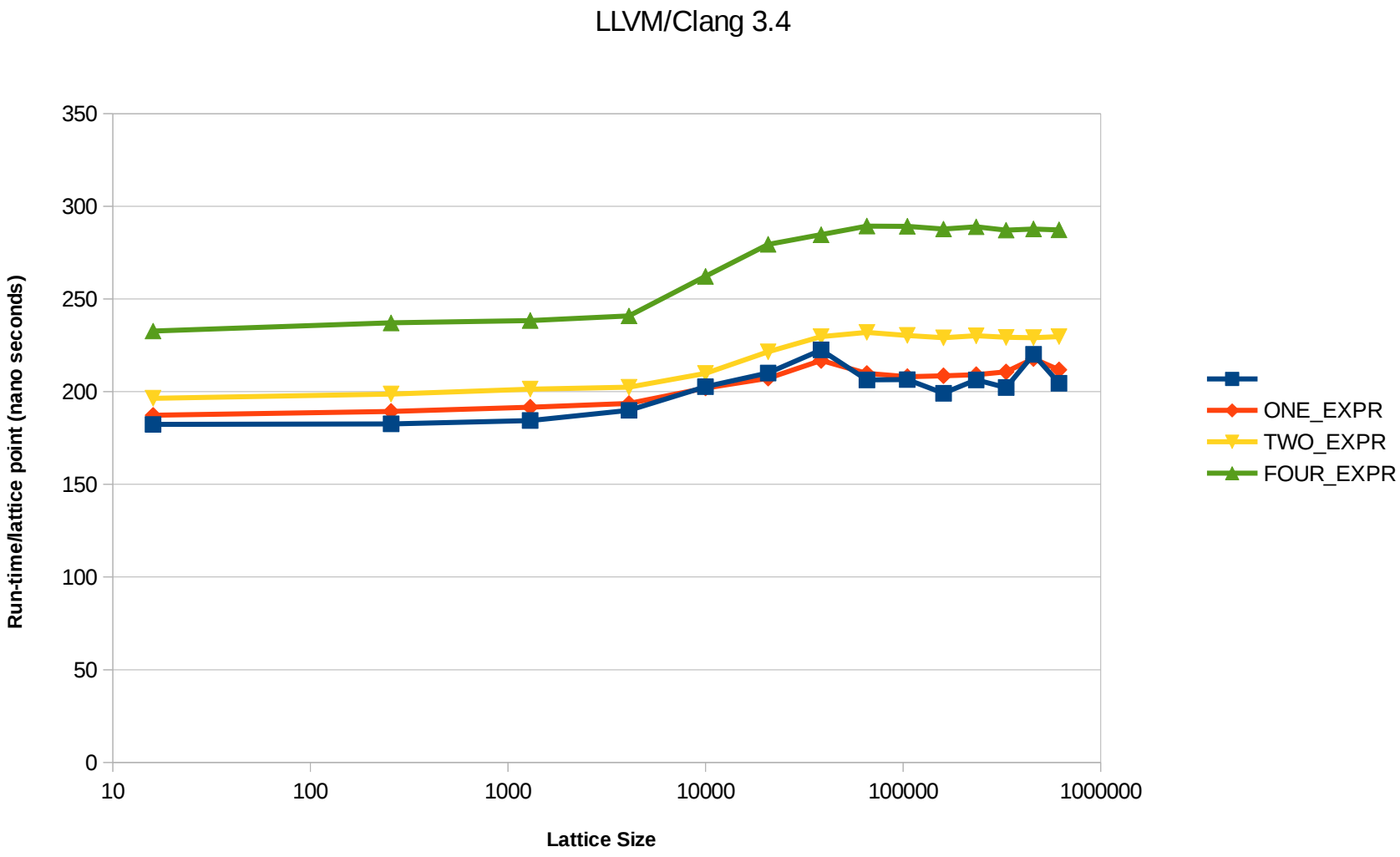
Evaluating effect of loop splitting (QDP++)



Evaluating effect of loop splitting (QDP++)



Evaluating effect of loop splitting (QDP++)



Observations regarding Dslash experiments.

- The QDP-JIT results are dominated by CPU <--> GPU communication.
- The GCC results raise interesting questions:
  - The steep rise in all curves is the expected rise in cache misses.
  - The improvements at very large sizes not definitively explained.
    - Hypotheses: Hardware prefetching becomes effective as loop sizes increase. Loop optimizations might become effective.
    - Measurements and analyses are in progress.
- ICC is better than GCC but both are dominated by Clang.
  - Where's the cache size cliff?
  - Code generated out of simple ETs looks very clean and should be amenable to loop transformations. Examination of Dslash object code is on the agenda.
  - Preliminary measurements of ICC-generated code indicates a high prefetch buffer hit rate.
- Experiments will be repeated with better instrumentation.

Going forward

- Inter-expression/statement analysis and transformation is the key to improving performance for high level programming.
  - Better code in the inner loops through loop fusion.
  - Improving data management and transport in QDP-JIT
    - Eliminate GPU-->CPU copy back for temporaries never used on CPU. (Manual experiments in progress.)
    - Consolidate and schedule device data transfers.
    - Asynchrony?
  - Apply consolidation and scheduling to inter-node communication.

- Next step: Send Diptorup to JLab for an extended visit.
  - Extensive cross visits in SOW and funds are available.

- Other problems and directions (driven by Jlab collaborators).
  - Whole program performance characterization.
  - Customized data allocators for QDP++.