OPENMP FOR QCD

What Works for Us and What We Still Need

Meifeng Lin

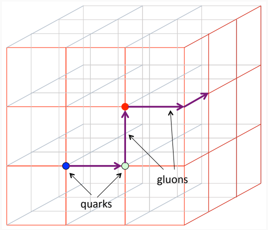
Brookhaven National Laboratory

OpenMPCon, Stony Brook University, September 18-20, 2017

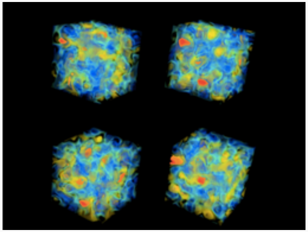
OUTLINE

LATTICE QCD

LATTICE QUANTUM CHROMODYNAMICS (QCD)



$\Longrightarrow$



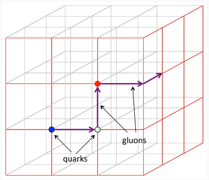
- Lattice QCD is a numerical framework to simulate quarks and gluons, the fundamental particles involved in strong interactions, from the theory of QCD.
- It is formulated on a discrete four-dimensional space-time grid or lattice.
- Quarks live on the lattice sites, and can propagate through the gluon ”lattice links”.
- Monte Carlo simulations are performed to generate the quantum fields of the gluons or ”the gauge field ensemble”.
- Complex calculations are made on these gauge ensembles to obtain physics results of relevance to experiments or other theoretical predictions.

LATTICE QCD COMPUTE KERNEL

- The core kernel of lattice QCD is matrix vector multiplications - the so-called *Dslash* operator.

$$D^{ij}_{\alpha\beta}(x,y)\psi^j_\beta(y) = \sum_{\mu=1}^{4}\Big[(1-\gamma_\mu)_{\alpha\beta}\,{U_\mu}^{ij}(x)\delta_{x+\hat\mu,y} + (1+\gamma_\mu)_{\alpha\beta}\,{U^\dagger_\mu}^{ij}(x+\hat\mu)\delta_{x-\hat\mu,y}\Big]\,\psi^j_\beta(y)$$

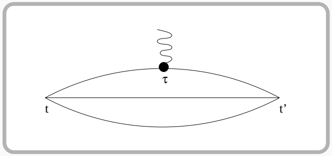
  - x, y - regular 4-dimensional grid points.
  - γ_μ - 4×4 matrices (fixed).
  - $U_\mu(x)$ - complex SU(3) matrices.
  - $\psi(y)$ - complex 12-component vectors.
  - nearest-neighbor, 9-point stencil operator.

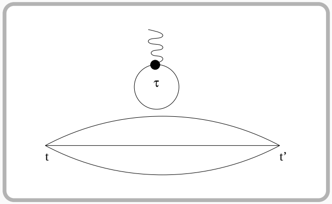


- The Dslash operations make up 70-90% of the computation time.

OTHER COMPONENTS OF LATTICE QCD

- Numerical algorithms
  - Monte Carlo sampling: Metropolis, Heatbath, ...
  - Molecular Dynamics (combined with Monte Carlo $\rightarrow$ Hybrid Monte Carlo)
  - Linear equation solvers: $Ax = b$
  - Eigenvalue solvers: $Ax = \lambda x$
- Physics applications
  - Actions: discretization schemes for the quarks and gluons
  - Measurements: evaluation of Feynman-diagram like graphs.





GOING INTO EXASCALE

Physics Objectives

- Increase the precision of certain critical calculations to understand fundamental symmetries in high-energy physics by an order of magnitude.
- Extend the calculations of the light nuclei and multi-nucleon systems in nuclear physics with quark masses that are closer to their values in nature.

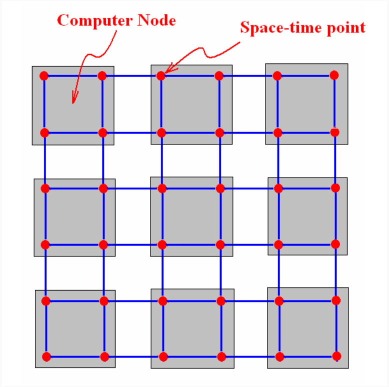
Software Requirements

- **Efficiency**: Should be able to efficiently exploit the expected multiple levels of parallelism on the exascale architectures. Need to conquer the communication bottleneck.
- **Flexibility**: Should be flexible for the users to implement different algorithms and physics calculations, and can provide easy access to multi-layered abstractions for the users.
- **Performance Portability**: Should be portable to minimize code changes for different architectures while maintaining competitive performance.

THE EVOLUTION OF LQCD HARDWARE AND PROGRAMMING MODELS

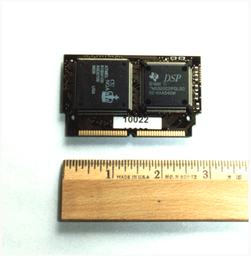
LATTICE QCD AND COMPUTER TECHNOLOGY

- Lattice QCD calculations require a lot of computing power.
- Uniform space-time structure $\rightarrow$ suitable for parallel computing.
- Assign a sub-lattice to each computing processor (with conventional data layout).



THE QCDSP SUPERCOMPUTER

- **QCDSP** (1998 – 2004):
  Quantum Chromodynamics on Digital Signal Processors


A QCDSP node

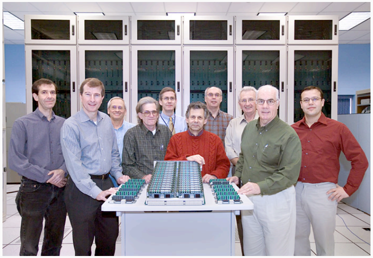
QCDSP at Columbia, 1 TFlops (10^{12}) Peak

- Designed by a group at Columbia and Brookhaven National Lab.
- Digital Signal Processor-based node
- 4-D mesh for communications.
- $\sim$ 12,288 nodes installed at BNL and Columbia.
- Won 1998 Gordon Bell Prize for price performance: $10/Mflops(sustained)

An MIMD machine, but typically programmed in SPMD. No threading.
Hardware-specific message passing. No MPI.

THE QCDOC SUPERCOMPUTER

- **QCDOC** (2004 – 2010): QCD On a Chip



QCDOC, 20 TFlops Peak

- Designed by lattice theorists at Columbia, RBRC, Edinburgh, in collaboration with IBM.
- PowerPC-based node, operating at 400 MHz.
- 6-D mesh for communications.
- $12,288 \times 2$ nodes installed at BNL, and $12,288$ nodes at Edinburgh
- \$1/Mflops(sustained)

Still no threading.
Internode communication through QCD Message Passing (QMP) interface (may or may not depend on MPI).

QCDCQ AND BLUEGENE/Q

QCDCQ (QCD with Chiral Quarks)

- Designed in-house by Columbia, Edinburgh and RBRC in collaboration with IBM.
- Follow-on to QCDOC and BlueGene machines.

	QCDOC	QCDCQ
1 node	1 core	16 cores
peak performance/node	0.8 GFlops	200 GFlops
peak performance/rack (1024 nodes)	0.8 TFlops	200 TFlops

- Much more cost efficient: \$0.02/Mflops(sustained)
- Several racks installed at BNL and University of Edinburgh.
- Prototype machine for BlueGene/Q

Threading became very important. MPI+X model used.

PC CLUSTERS

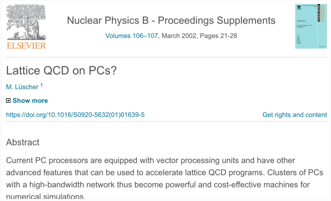
- Clusters with PC processors are now commonplace.
- Some clusters dedicated to lattice QCD simulations in the US:

10q cluster at Jefferson Lab

JPsi cluster at Fermilab

KNL cluster at JLab

- Things have changed a lot since 15 years ago...


Nuclear Physics B - Proceedings Supplements

Volumes 106–107, March 2002, Pages 21-28

Lattice QCD on PCs?

M. Lüscher [1]

Show more

https://doi.org/10.1016/S0920-5632(01)01639-5

Get rights and content

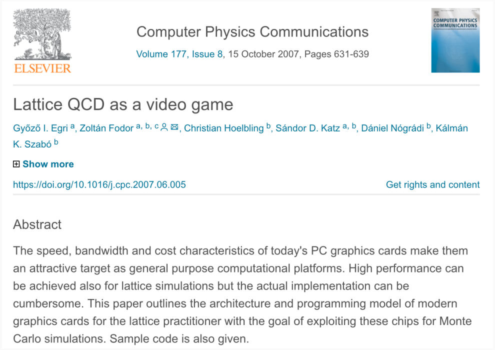
Abstract

Current PC processors are equipped with vector processing units and have other advanced features that can be used to accelerate lattice QCD programs. Clusters of PCs with a high-bandwidth network thus become powerful and cost-effective machines for numerical simulations.

Some OpenMP usage. But often running multiple MPI processes per node. Intel's new many integrated core (MIC) architecture (Knights Corner and Knights Landing) makes it very important to have OpenMP threading in the code.

LATTICE QCD ON GPGPUS

- LQCD theorists have been active users of Titan at OLCF and other GPU computing resources.



Computer Physics Communications
Volume 177, Issue 8, 15 October 2007, Pages 631-639

Lattice QCD as a video game

Győző I. Egri [a], Zoltán Fodor [a, b, c], Christian Hoelbling [b], Sándor D. Katz [a, b], Dániel Nógrádi [b], Kálmán K. Szabó [b]

⊞ Show more

https://doi.org/10.1016/j.cpc.2007.06.005 Get rights and content

Abstract

The speed, bandwidth and cost characteristics of today's PC graphics cards make them an attractive target as general purpose computational platforms. High performance can be achieved also for lattice simulations but the actual implementation can be cumbersome. This paper outlines the architecture and programming model of modern graphics cards for the lattice practitioner with the goal of exploiting these chips for Monte Carlo simulations. Sample code is also given.

OpenCL, OpenACC, CUDA all have been explored by different groups.
Most popular now: CUDA.

CURRENT US LATTICE QCD SOFTWARE STACK

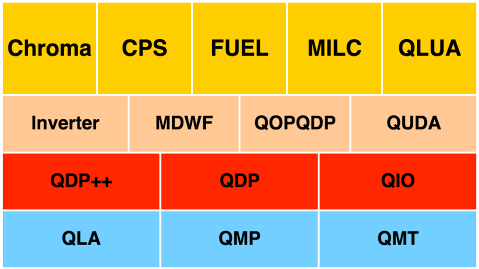


Figure 1: The SciDAC Layers and the software module architecture.

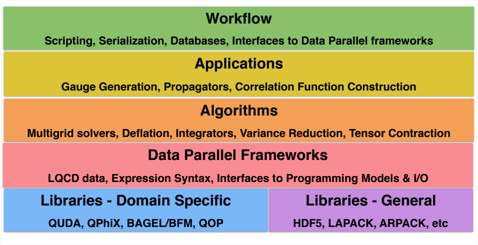
Current USQCD[1] software stack consists of four levels:

- **Level 1 - QLA, QMT, QMP**: BLAS, communication and threading support.
- **Level 2 - QDP++, QDP, QIO**: Lattice data parallel layer and I/O.
- **Level 3 - Inverter, MDWF, QOPQDP, QUDA**[2]: Lattice QCD APIs.
- **Level 4 - Chroma, CPS, FUEL, MILC, QLUA**: Application suites.

[1] The USQCD Collaboration is a consortium of US scientists working on lattice QCD.
[2] QUDA is a lattice QCD API written specifically for GPUs with CUDA C.

PROPOSED EXASCALE QCD SOFTWARE LAYER

- Because the LQCD software has evolved overtime with the target hardware, the programming models and styles are a mixture of many things.
- Over the years, MPI+X has become the most popular choice.
- A new software stack is being developed under the ECP application development project.
- Performance portability is one of the key considerations for the new design.


Workflow
Scripting, Serialization, Databases, Interfaces to Data Parallel frameworks
Applications
Gauge Generation, Propagators, Correlation Function Construction
Algorithms
Multigrid solvers, Deflation, Integrators, Variance Reduction, Tensor Contraction
Data Parallel Frameworks
LQCD data, Expression Syntax, Interfaces to Programming Models & I/O
Libraries - Domain Specific
QUDA, QPhiX, BAGEL/BFM, QOP
Libraries - General
HDF5, LAPACK, ARPACK, etc

SOME SPECIFIC EXAMPLES

Optimizing the Dslash operator in Columbia Physics System (CPS)

People Involved:

- Stony Brook University

  Eric Papenhausen

- Reservoir Labs Inc.

  M. Harper Langston
  Benoit Meister
  Muthu Baskaran

- BNL

  Chulwoo Jung
  Taku Izubuchi
  ML

THE DOMAIN WALL DSLASH OPERATOR

- The Domain Wall (DW) fermion matrix can be written as

$$M^{DW}_{x,s;x',s'} = (4 - m_5)\delta_{x,x'}\delta_{s,s'} - \frac{1}{2} D^{W}_{x,x'}\delta_{s,s'} + D^{5}_{s,s'}\delta_{x,x'}, \tag{1}$$

where m_5 is the domain wall height, $D^W_{x,x'}$ is the Wilson Dslash operator, and $D^5_{ss'}$ is the fermion mass term that couples the two boundaries in the 5th dimension,

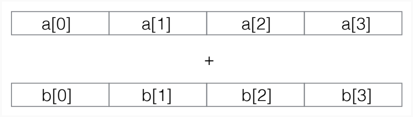
$$\begin{aligned} D^5_{ss'} \quad = \quad & -\frac{1}{2}\left[(1-\gamma_5)\delta_{s+1,s'} + (1+\gamma_5)\delta_{s-1,s'} - 2\delta_{s,s'}\right] \\ & + \frac{m_f}{2}\left[(1-\gamma_5)\delta_{s,L_s-1}\delta_{0,s'} + (1+\gamma_5)\delta_{s,0}\delta_{L_s-1,s'}\right]. \end{aligned} \tag{2}$$

- Most FLOPs are in the 4D derivative term ($(4 - m_5)\delta_{x,x'}\delta_{s,s'} - \frac{1}{2}D^W_{x,x'}\delta_{s,s'}$) in Eq.(1): 1320 flops per site.

SINGLE INSTRUCTION MULTIPLE DATA (SIMD)

- Modern CPUs, both by Intel and AMD, support vector instructions.
  - **SSE**: 128-bit vector register, capable of 2 DP/4 SP flops per cycle.
  - **AVX**: 256-bit vector register, capable of 4 DP/8 SP flops per cycle.
  - **AVX2**: AVX with fused multiply-add (FMA).
  - AVX512: Intel KNL, Skylake, ...
- Data layout is the key: Data in one SIMD operation need to fit into the same vector register. With AVX, the following instructions should be able to execute in one clock cycle.

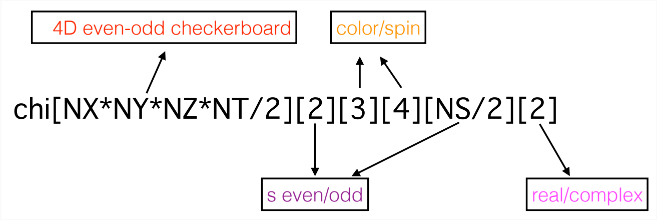
```
double a[4], b[4], c[4];
for (int n=0; n<4; n++) c[n] = a[n] + b[n];
```



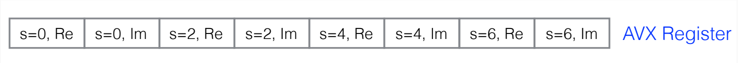
- There also cannot be any data dependencies among the SIMD data.
- In DWF 4D Dslash, the s coordinates are completely independent. $\hookrightarrow$ Good place to vectorize.

DWF DATA LAYOUT

- We chose the following data layout to enable us to vectorize in the fifth (s) dimension.



- In one AVX register, with single precision, the data mapping goes



- SIMD intrinsics were used to implement the vectorized DWF Dslash.
- Tried `omp simd`, but performance was very poor.

OTHER OPTIMIZATIONS

- **FMA**: AVX2 provides intrinsics to perform fused multiply-add. However, we found that simply turning on -mfma compiler option for gcc gave us the same performance boost as using intrinsics.
- **Improved data locality**:
  - We studied tiling to increase memory reuse, but didn't gain any performance.
  - We also explored using a space-filling curve, implemented as the Z-curve, to improve data locality, but the performance boost was minimal.
- **Prefetching:** Before the computation of each stencil operation, prefetch data needed for the next stencil. Led to 10% performance improvement.

- On Intel(R) Xeon(R) CPU E5-2690 v3 @ 2.60GHz processor (Haswell), with $8^4 \times 8$ lattice, we achieved 34% peak single-core performance in single precision.

Optimization	AVX2	Tiling	Z-Curve	Prefetching
time [ms]	0.86	0.92	1.0	0.76
Gflops	25.1	23.5	21.6	28.5

MULTITHREADING WITH OPENMP

- Within the node, we use OpenMP for multithreading.
- Three strategies have been explored:
  - Simple Pragma: Thread the outer loop, usually the t loop.
    $\hookrightarrow$ Parallelism is limited by the t dimension size, won't scale well in many-core systems.

```
#pragma omp parallel for private(tmp_tst)// collapse(4)
for(t=1; t<lt+1; t++)
  for(z=1; z<lz+1; z++)
    for(y=1; y<ly+1; y++)
      for(x=2; x<lx+2; x++)
      ...
```

  - Compressed Loop: Compress the nested loops into one single loop.
  - Explicit Work Distribution: Similar to Compressed Loop, but explicitly assign work to each thread.

```
#pragma omp parallel
  {
   int nthreads = omp_get_num_threads();
   int tid = omp_get_thread_num();
   int work = NT*NZ*NY*(NX/2)/nthreads;
   int start = tid * work;
   int end = (tid+1) * work;
   for(lat_idx = start; lat_idx < end; lat_idx++)
   ......
  }
```

OPENMP PERFORMANCE

Performance was measured on LIRED, with dual-socket Haswell per node @ 2.6 GHz (24 cores).

- $8^4 \times 8$

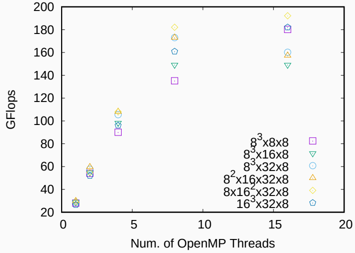
Num. Threads	Simple Pragma	Compressed Loop	Explicit Dist.
1	28.4 GF/s	28.0 GF/s	28.0 GF/s
2	51.5 GF/s	54.1 GF/s	54.1 GF/s
4	90.1 GF/s	90.1 GF/s	90.1 GF/s
8	135.2 GF/s	135.2 GF/s	144.2 GF/s
16	127.2 GF/s	180.2 GF/s	154.4 GF/s

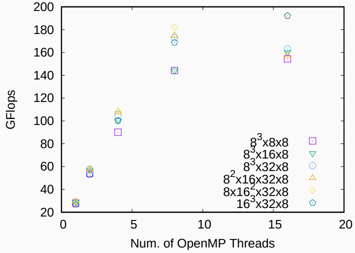
- $16^3 \times 32 \times 8$:

Num. Threads	Simple Pragma	Compressed Loop	Explicit Dist.
1	26.9 GF/s	26.5 GF/s	26.8 GF/s
2	54.5 GF/s	52.0 GF/s	52.8 GF/s
4	100.3 GF/s	96.1 GF/s	100.3 GF/s
8	168.8 GF/s	160.9 GF/s	168.8 GF/s
16	197.7 GF/s	182.1 GF/s	192.2 GF/s

OPENMP SUMMARY

- Three threading approaches result in similar performances, except when the problem size is small, Simple Pragma doesn't scale as well.
- Surprisingly, the performance does not deteriorate with a much larger lattice size $\hookrightarrow$ possible indication of poor cache reuse.
- Volume comparison:
  Left - Compressed Loop. Right - Explicit Work Distribution.



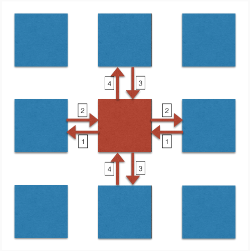


We also found that that binding OpenMP threads to the processors was key to improve OpenMP performance. With gcc, this is done through

```
export OMP_PROC_BIND=true
```

OVERLAPPING COMMS AND COMPUTE

- To improve strong scaling, we also overlapped comms and compute.
- The communication pattern is illustrated in the following. There is blocking for each transfer sequence.

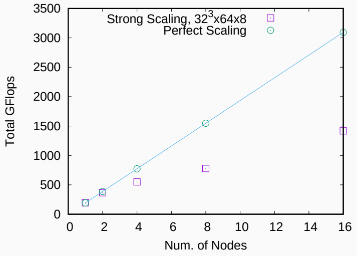


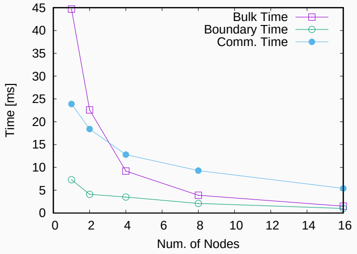
- The best performance is obtained with 2 MPI processes per node (1 MPI process per socket, improved data locality).
- With each MPI process, a number of threads equal to the number of compute cores are used.
- We dedicate one thread (the `master` thread) to do the communications, and the rest of the threads for computation[3].
- Do bulk computation first while waiting for the communication to complete, then do the boundary computation.

[3] May also be done with OpenMP tasking

MULTINODE PERFORMANCE

- Strong scaling study of a $32^3 \times 64 \times 8$ calculation was performed on LIRED, with dual-socket Intel Haswell CPUs and Mellanox 56 Gigabit FDR interconnect.
- The performance scales well up to 4 nodes, and scales sublinearly from 8 to 16 nodes.
- After 4 nodes, the total time is dominated by the communication time.
- Bulk computation itself scales well with the number of nodes.
- Rediscovered the old truth: Communication is the bottleneck for strong scaling!





Exascale Performance Portability for LQCD

People Involved:

Peter Boyle	University of Edinburgh
Kate Clark	NVIDIA
Carleton DeTar	University of Utah
ML	BNL
Verinder Rana	Brookhaven National Laboratory
Alejandro Vaquero	University of Utah

WHY DO WE CARE ABOUT PERFORMANCE PORTABILITY?

- A single version of portable code is easier to maintain.
- Less time spent on integrating the low-level APIs with the application layer, and more time on physics and algorithm development.
- Question: how much performance are we willing to lose in exchange for portability?
- The answer may be ”0”. But looking towards the future, with potentially more diverse architectures, are we able to continue our current approach?

WAYS TO PERFORMANCE PORTABILITY?

Various tools are under development for performance portability.

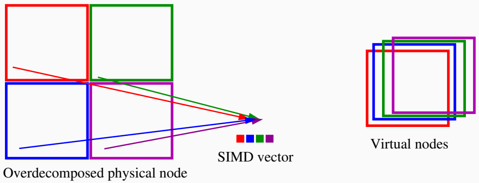
- High-level compiler directives
  - OpenMP
  - OpenACC
- High-level programming abstractions:
  - RAJA (LLNL)
  - Kokkos (Sandia)
  - SYCL (Kronos)
  - C++ AMP (Microsoft)
  - ...
- Code generators/Source-to-source compilers
  LQCD community have investigated:
  - JIT: QDP-JIT (JLab/Frank Winter)
  - Nim: QEX (ANL/James Osborn)
  - R-Stream compiler (Reservoir Labs)

Question: should we design our new software with portability in mind first and then optimize for performance later, or the other way around? Can we design our software with performance portability in mind from the beginning?

GRID

- Grid[4] is a next-generation C++ lattice QCD library being developed by Peter Boyle, Guido Cossu, Antonin Portelli and Azusa Yamaguchi at the University of Edinburgh.
  https://github.com/paboyle/Grid
- Originally developed and optimized for CPUs. Being used as a testbed for QCD ECP performance portability.
- It uses new features in C++11 for abstractions and programming flexibility.
- Data layout designed to match CPU SIMD lanes.

- Vector data layout: Decompose four-dimensional grids into sub-domains that map perfectly onto the target SIMD length.



[4] Peter Boyle et al. “Grid: A next generation data parallel C++ QCD library”. In: (2015). arXiv: 1512.03487 [hep-lat].

VECTORIZATION FOR CPUS

- Vectorization is achieved in different ways on different targets, either using intrinsics, or explicit short scalar loops for compiler vectorization, and possibly using OpenMP SIMD pragmas depending on target.
- But the implementation details are abstracted inside templated data types.

```
//Vectorization
#ifdef GEN
#include "Grid_generic.h"
#endif
#ifdef SSE4
#include "Grid_sse4.h"
#endif
#if defined(AVX1) || defined (AVXFMA) || defined(AVX2) || defined(AVXFMA4)
#include "Grid_avx.h"
#endif
#if defined AVX512
#include "Grid_avx512.h"
#endif

// Abstract Data Types
typedef Grid_simd< float, SIMD_Ftype > vRealF;
typedef Grid_simd< double, SIMD_Dtype > vRealD;
typedef Grid_simd< std::complex< float > , SIMD_Ftype > vComplexF;
typedef Grid_simd< std::complex< double >, SIMD_Dtype > vComplexD;
typedef Grid_simd< Integer, SIMD_Itype > vInteger;
```

PARALLELISM IN GRID

- Grid uses OpenMP for on-node threading and MPI for inter-node communications.
- Lattice-wide operations are done in a big *for* loop over the **outer** lattice sites.

```
PARALLEL_FOR_LOOP
        for(int ss=0;ss<lhs._grid->oSites();ss++){
            ret._odata[ss] = trace(lhs._odata[ss]);
        }
```

- PARALLEL_FOR_LOOP is a macro currently defined as an OpenMP parallel construct. It potentially can be replaced with OpenACC for GPU.

```
#ifdef GRID_OMP
#include <omp.h>
#define PARALLEL_FOR_LOOP _Pragma("omp parallel for ")
#define PARALLEL_NESTED_LOOP2 _Pragma("omp parallel for collapse(2)")
#else
#define PARALLEL_FOR_LOOP
#define PARALLEL_NESTED_LOOP2
#endif
```

GRID EXPRESSION TEMPLATE

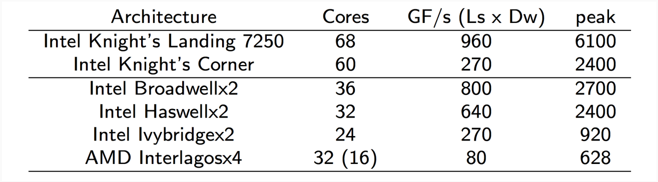
- Extensive use of templates to allow for high-level abstractions.

```
GridCartesian     Grid(latt_size,simd_layout,mpi_layout);

LatticeColourMatrix A(&Grid);
LatticeColourMatrix B(&Grid);
LatticeColourMatrix C(&Grid);

C = A * B
```

- Expression template makes this possible.
- Many architectures supported with impressive performance.



Architecture	Cores	GF/s (Ls x Dw)	peak
Intel Knight's Landing 7250	68	960	6100
Intel Knight's Corner	60	270	2400
Intel Broadwellx2	36	800	2700
Intel Haswellx2	32	640	2400
Intel Ivybridgex2	24	270	920
AMD Interlagosx4	32 (16)	80	628

P. Boyle

GRID C++ EXPRESSION TEMPLATE

Some Background

- GPU is not among the supported architectures at the moment.
- Initial GPU porting effort started last year using OpenACC.
  - Ran into many issues due to Grid's complex data structures. $\hookrightarrow$ deep copy
  - PGI compiler did not sufficiently support C++11 code.
  - STL not supported on GPUs.
  - Porting whole Grid turned out to be rather difficult.
- Proof-of-concept studies using stripped-down version of Grid expression template (ET).

Grid ET

- ~ 200 lines of self-contained code, provided by P. Boyle.
- Arithmetic operations contained in the recursive `eval` function
  $\hookrightarrow$ `for` loop is target to be offloaded to the GPU.

```
template <typename Op, typename T1,typename T2> inline Lattice<obj> & operator=(const
   LatticeBinaryExpression<Op,T1,T2> expr)
{
  int _osites=this->Osites();
  for(int ss=0;ss<_osites;ss++){
    _odata[ss] = eval(ss,expr);
  }
  return *this;
}
```

DIFFERENT APPROACHES STUDIED

OpenACC/OpenMP

- Pros: Directives-based approach; Easy to add to existing code; Portable across different platforms.
- Cons: Lack of deep-copy support; Use in C++ code non-trivial; Dependent on compiler; Developer has little control.

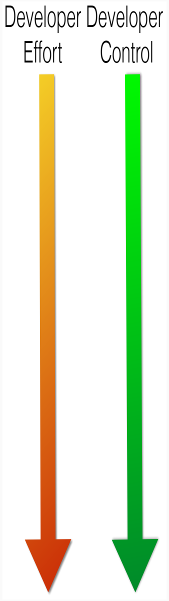
Just-In-Time: Jitify

- New JIT header library being developed at NVIDIA.

  See GTC2017 talk - Ben Barsdell, Kate Clark "Jitify: CUDA C++ Runtime Compilation Made Easy"
- Pros: No need for CUDA extensions (though available). CPU and GPU execution policies can be present simultaneously.
- Cons: Runtime compilation. Kernel functions need to be given in header files.

CUDA

- Pros: Mature programming model for NVIDIA GPUs. C++ support is steadily improving. Easy to control for performance.
- Cons: Need to write some CUDA kernels; Some code branching unavoidable. Supports NVIDIA GPUs only. Need to declare all host device functions.



OPENMP/OPENACC OFFLOAD COMPILER SUPPORT (CIRCA 2016)

- OpenMP:
  - GCC: v6.1 has full C/C++ support for OpenMP 4.5.
  - Intel: v16 has support for OpenMP 4.0.
  - Cray: supports OpenMP 4.0
  - Clang/LLVM: v3.8 supports some OpenMP 4.0 and 4.5
  - ...
- OpenACC:
  - PGI (NVIDIA)
  - Cray
  - GCC 6.1
  - Research compilers: OpenUH (U of Houston), OpenARC (ORNL)
- Targets/Architectures (to be) supported:
  - AMD and NVIDIA GPUs
  - Intel MICs and Xeons
  - IBM Power
  - ARM
  - FPGA
  - ...

COMPARISON OF DIFFERENT IMPLEMENTATIONS

Kernel

OpenACC

```
#pragma acc parallel loop independent copyin(expr[0:1])
 for(int ss=0;ss<_osites;ss++){
    _odata[ss] = eval(ss,expr);
 }
```

OpenMP

```
#pragma omp target device(0) map(to: expr) map(tofrom:_odata[0:_osites])
  {
  #pragma omp teams distribute parallel for
     {
     for (int i=0; i<_osites; i++)
        _odata[ss] = eval(ss,expr);
     }
   }
```

Jitify

```
parallel_for(policy, 0, _osites,
              JITIFY_LAMBDA( (_odata,expr),
              _odata[i]=eval(i,expr); ));
```

CUDA

```
 template<class Expr, class obj> __global__
  void ETapply(int N,obj *_odata,Expr Op)
  {
    int ss = blockIdx.x;
    _odata[ss]=eval(ss,Op);
  }
 LatticeBinaryExpression<Op,T1,T2> temp = expr;
 ETapply< decltype(temp), obj > <<<_osites,1>>>((int)_osites,this->_odata,temp);
```

OTHER CODE CHANGES/COMPARISONS

- OpenACC
  - Need to specify device routines with `#pragma acc routine`. Defined in OFFLOAD.
  - Need PGI's Unified Virtual Memory (UVM) support for data management.
  - Choose target at compile time

    ```
    [GPU] pgc++ -acc -ta=tesla:managed --c++11  -O3 main.cc -o gpu.x
    [CPU] pgc++ -acc -ta=multicore --c++11-O3 main.cc -o cpu.x
    ```
- OpenMP
  - Similar to OpenACC, but no compiler UVM support yet. So code is not working yet.
- Jitify
  - Use managed memory allocator for UVM support. Execution policy defined in main program.

    ```
    static const Location ExecutionSpaces[] = DEVICE;
    policy = ExecutionPolicy(location);
    ```
- CUDA
  - Customized allocator: aligned allocator for CPUs, managed allocator for GPUs.

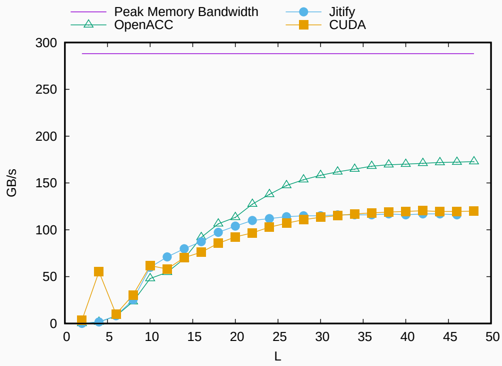
    ```
    #ifdef GPU
        cudaMallocManaged((void **)&ptr, __n*sizeof(_Tp));
    #elif defined(AVX512)
        ptr = (pointer) _mm_malloc(__n*sizeof(_Tp), 64); //changes with the target architecture
    #elif ...
    ```
  - OFFLOAD macro needed for functions on both host and device

    ```
    #ifdef __NVCC__
    #define OFFLOAD __host__ __device__
    #elif defined (_OPENACC)
    #define OFFLOAD _Pragma("acc routine seq")
    #else
    #define OFFLOAD
    #endif
    ```

SU(3)xSU(3) STREAMING TEST

```
Lattice<Su3f> z(&grid);
Lattice<Su3f> x(&grid);
Lattice<Su3f> y(&grid);
for(int i=0;i<Nloop;i++) {
      z=x*y;
}
```

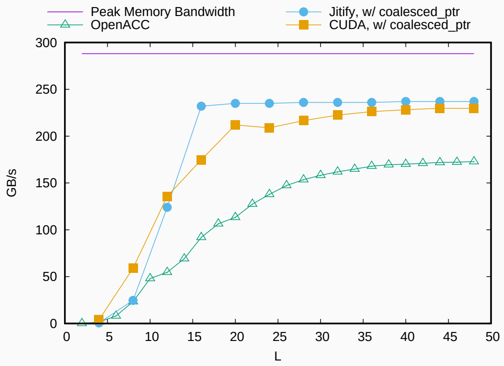
- Performance comparison with default setting (no tuning of thread/block numbers). NVIDIA GTX 1080 (Pascal Gaming Card)



SU(3)xSU(3) STREAMING TEST

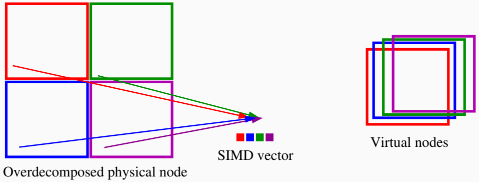
```
Lattice<Su3f> z(&grid);
Lattice<Su3f> x(&grid);
Lattice<Su3f> y(&grid);
for(int i=0;i<Nloop;i++) {
      z=x*y;
}
```

- Performance comparison with default setting (no tuning of thread/block numbers). NVIDIA GTX 1080 (Pascal Gaming Card)

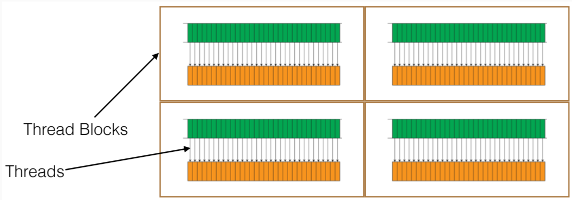


MAPPING SIMD DATA LAYOUT ONTO GPUS

- Poor performance due to lack of memory coalescing with the AoS data layout
- Can be overcome by using a `coalesced_ptr` class (K. Clark)
    - Transforms AoS into AoSoAoS
    - Performance boost by a factor of 2
- Grid's native SIMD vector layout can be used to ensure coalescence without coalesced_ptr.



- Each GPU thread within a thread block processes one element of the vector. Thread blocks map to outer sites.



CODE SNIPPETS

- Since the top-level data structures are of vector types, some “hacking” is needed to make different threads process different elements of the vector.
- Make each thread **eval** one element of the vector, extracted through **extractS**.

```
//C++14 and CUDA 9 needed to make this work
 template<class obj> OFFLOAD inline auto evalS(const unsigned int ss, const Lattice<obj> arg,
      const int tIdx) //-> decltype(typename obj::scalar_object)
 {
   typedef typename obj::scalar_object sObj;
   auto sD = extractS<obj,sObj>(arg._odata[ss], tIdx);
   return (sObj) sD;
 }
```

- Put the results back to form the vector again after **eval**.

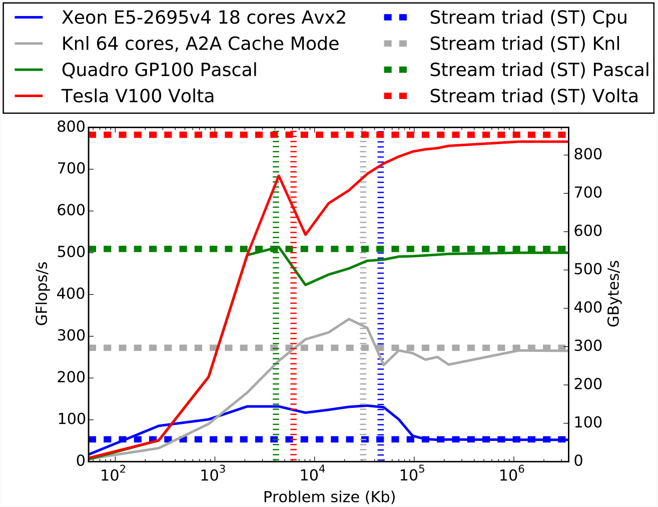
```
template<class Expr, class obj> __global__
void ETapply(int N,obj *_odata,Expr Op)
{
  if (blockIdx.x < N) {
    typedef typename obj::scalar_object sObj;
    auto sD = evalS(blockIdx.x,Op,threadIdx.x);
    mergeS(_odata[blockIdx.x], sD, threadIdx.x);
  }
}
```

- Outer sites become thread blocks; inner sites become threads.

```
ETapply<decltype(temp), obj> < < <_osites,_isites> > > ((int)_osites,this->_odata,temp);
```

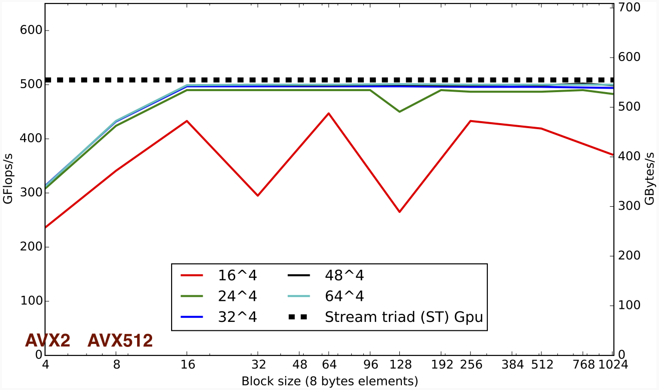
SU(3)xSU(3) PERFORMANCE

Same code. Performance saturates STREAM Triad results on multiple architectures.



BLOCK SIZE DEPENDENCE

- How big do we need to make the blocks?
- Twice the AVX512 width (1024 bits) already saturates the performance.



Tests on NVIDIA Quadro GP100

SUMMARY

SUMMARY

- LQCD's regular grid structure is great for parallelization.
- The diverse hardware architectures that LQCD software has been optimized for result in significant division of code bases.
- OpenMP is becoming more and more important for LQCD with the increasing on-node parallelism.
- With the pursuit performance portability, OpenMP may give us a way to "Grand Unification".

What works for us:

- Simple (nested) `parallel for` loops nowadays indispensable.
- API calls provide more flexibility

What may work for us:

- `simd`: some recent work show that directive-guided compiler vectorization can give good performance.
- `task`: used to overlap communication and computation.

What we still need:

- Better support for C++ GPU offloading.
- Complex c++ programming requires deep copy or UVM support for accelerator offloading.