Performance Portability Experiments with the Grid C++ Lattice QCD Library

Alejandro Vaquero

University of Utah

August 22th, 2017

DOE Exascale Software project, with:

Peter Boyle, University of Edinburg
Kate Clark, NVIDIA
Carleton Detar, University of Utah
Meifeng Lin, BNL
Verinder Rana, BNL

Motivation

Exascale Computing

- US DOE planning to bring Exascale capable machines by 2021-2023
- The architecture is expected to be complex
  - Heterogeneity
  - Complex memory hierarchy
  - Multiple levels of parallelism
- USQCD and its partners are working on a new scientific software stack that will be *Exascale-ready* under the **Exascale Computing Project** (ECP)

ECP requirements

- **Efficiency**
  - The code must exploit the multiple levels of parallelism of the exascale architectures, as well as its heterogenous nature
  - Communications are expected to become a serious bottleneck and might deserve special treatment
- **Flexibility and ease of use**
  - The code must be flexible enough so the different algorithms required in our physics calculations can be implemented in a simple way
  - Support for several layers of abstraction that simplify the development
- **Performance Portability**
  - The code should be portable with minimal modifications, while preserving competitive performance

The Grid library

Data parallel C++ mathematical object library

P. A. Boyle, G. Cossu, A. Yamaguchi, A. Portelli

`https://www.github.com/paboyle/Grid`

- Adapted for Quantum Field Theories
- Abstracts away the complexities of the target machine
  - **Efficient:** Offers a high level interface that exploits all levels of parallelism

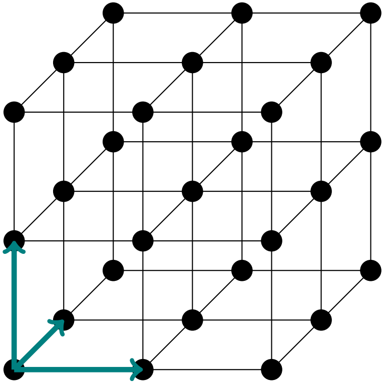
    $$\text{MPI} \otimes \text{OpenMP} \otimes \text{SIMD}$$

  - **Flexible:** Uses C++ classes to wrap SIMD intrinsics
    - Object operations are defined independently of vector length or layout
    - Vector class operator directly translated into intrinsic code
    - Performance not impacted by the high-level interface
    - Current supported SIMD instruction sets: SSE4, AVX, AVX2, AVX512, IMCI, ARM Neon, QPX
  - **Performance portable:** Easily hits maximum memory bandwidth in supported platforms
    - Different architectures are supported through redefinitions of the SIMD intrinsics
    - Unfortunately this approach won't work for Gpus

Indeed **GPU** support is missing from the list

Data layout in QCD (and computer simulated QFTs)

- Quantum Chromo Dynamics (QCD) is our current theory to describe the strong force
- QCD is simulated in supercomputers in a $4D$ lattice
  - Each site of the lattice carries complex vectors of 3 or 12 components
  - Each link of the lattice carries a SU(3) matrix (3×3 complex matrix)
  - There are D independent links per site ($D =$ dimension)
  - (Anti)Periodic boundary conditions are enforced on each dimension
- Grid implements natively the required datatypes (complex matrices, vectors...)



SIMD parallelism in Grid

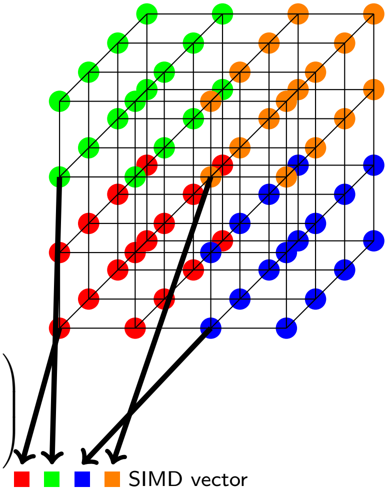
- Split the lattice in virtual nodes (outer sites)
- Each simd lane deals with a virtual node (inner sites)

Advantages
Easily to parallelize

Disadvantages
Permutations/rotations to keep the right boundary conditions

- Vector datatypes appear at the lowest level (AoSoV)

$$\begin{pmatrix} a_1^{11}a_2^{11}a_3^{11}a_4^{11} & a_1^{12}a_2^{12}a_3^{12}a_4^{12} & a_1^{13}a_2^{13}a_3^{13}a_4^{13} \\ a_1^{21}a_2^{21}a_3^{21}a_4^{21} & a_1^{22}a_2^{22}a_3^{22}a_4^{22} & a_1^{23}a_2^{23}a_3^{23}a_4^{23} \\ a_1^{31}a_2^{31}a_3^{31}a_4^{31} & a_1^{32}a_2^{32}a_3^{32}a_4^{22} & a_1^{33}a_2^{33}a_3^{33}a_4^{33} \end{pmatrix}$$



Expression Templates in Grid

- Using C++11 templates we can abstract complex datatypes and treat them as numbers in an expression

```
Grid grid(vol);           // Creates a cartesian grid of points
Lattice<Su3f> x(&grid);   // Creates objects that populate the cartesian
Lattice<Su3f> y(&grid);   // grid with SuNf objects (= SU3 matrix single
Lattice<Su3f> z(&grid);   // precision)

z = x*(y+x) - y*y;        // Expression to be evaluated
```

- Achieved through a recursive evaluation of expressions
- Sample code for the miniapp used for portability tests:

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

template <typename Op, typename T1, typename T2>
auto inline eval (const unsigned int ss, const LatticeBinaryExpression<Op,T1,T2> &expr)
-> decltype(expr.Op.func(eval(ss,expr.arg1),eval(ss,expr.arg2))) {
  return expr.Op.func(eval(ss,expr.arg1),eval(ss,expr.arg2));
}

template<class obj>
inline obj& eval(const unsigned int ss, const Lattice<obj> arg) {
  return arg._odata[ss];
}
```

Porting Grid to GPUs

- Porting to GPUs quite challenging
  - **C++11** code strongly based on **STL**, which is not well supported in GPUs
  - Deep templating makes **deep copy** (or **unified memory**) a requirement
  - Achieving **coalesced access** with the **AoSoV** layout can be complicated
- Several possibilities

OpenACC/OpenMP

- Directive-based, unifies code
- Easy to adapt existing code
- Portable
- Lacks deep copy support
- Use in C++ code non-trivial
- Compiler dependent
- Developer has little control

Jitify

- CUDA extensions not needed
- CPU-GPU execution policies can be present simultaneously
- Runtime compilation might affect performance
- Kernel functions must be supplied in header files

www.github.com/NVIDIA/jitify

CUDA

- Most mature approach
- C++ support improving
- Developer has full control
- Need to write CUDA kernels, code branching
- Only NVIDIA GPU support
- Must add __host__ __device__ decorations

Porting Grid to Gpus

Kernel examples

OpenACC

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

OpenMP

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

Jitify

```
static const Location ExecutionSpaces[] = DEVICE;
policy = ExecutionPolicy(location)
...
parallel_for(policy, 0, this->oSites(),
  JITIFY_LAMBDA( (_odata, expr), _odata[i]=eval(ss,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> <<<nBs,nTs>>>(this->oSites(),this->_odata,temp);
```

Porting Grid to Gpus

- Gpu compilers won't give coalesced access in a AoSoV layout
  - Usually they assign a whole vector to a thread, and performance decreases due to register spilling
  - Performance can be improved with a coalesced wrapper that dynamically transforms the layout
- But Grid's native layout can be effectively used by the gpu giving coalesced access
  - Assign osites $\longrightarrow$ blocks, isites $\longrightarrow$ threads
- Requires some modifications to make the Expression Templates work properly

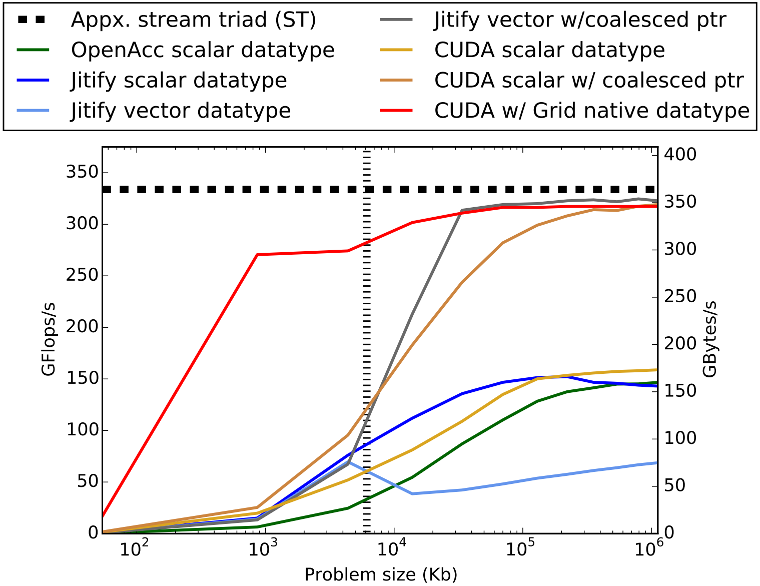
$$\begin{pmatrix} a_1^{11} a_2^{11} a_3^{11} a_4^{11} & a_1^{12} a_2^{12} a_3^{12} a_4^{12} & a_1^{13} a_2^{13} a_3^{13} a_4^{13} \\ a_1^{21} a_2^{21} a_3^{21} a_4^{21} & a_1^{22} a_2^{22} a_3^{22} a_4^{22} & a_1^{23} a_2^{23} a_3^{23} a_4^{23} \\ a_1^{31} a_2^{31} a_3^{31} a_4^{31} & a_1^{32} a_2^{32} a_3^{32} a_4^{22} & a_1^{33} a_2^{33} a_3^{33} a_4^{33} \end{pmatrix} \longrightarrow \begin{matrix} \underset{\text{Thread 1}}{\begin{pmatrix} a_1^{11} & \dots \\ \vdots & \ddots \end{pmatrix}} & \underset{\text{Thread 2}}{\begin{pmatrix} a_2^{11} & \dots \\ \vdots & \ddots \end{pmatrix}} \\ \underset{\text{Thread 3}}{\begin{pmatrix} a_3^{11} & \dots \\ \vdots & \ddots \end{pmatrix}} & \underset{\text{Thread 4}}{\begin{pmatrix} a_4^{11} & \dots \\ \vdots & \ddots \end{pmatrix}} \end{matrix}$$

Each **osite** is a vectorized matrix, each **isite** access a particular simd lane of the vectorized matrix

Porting Grid to Gpus

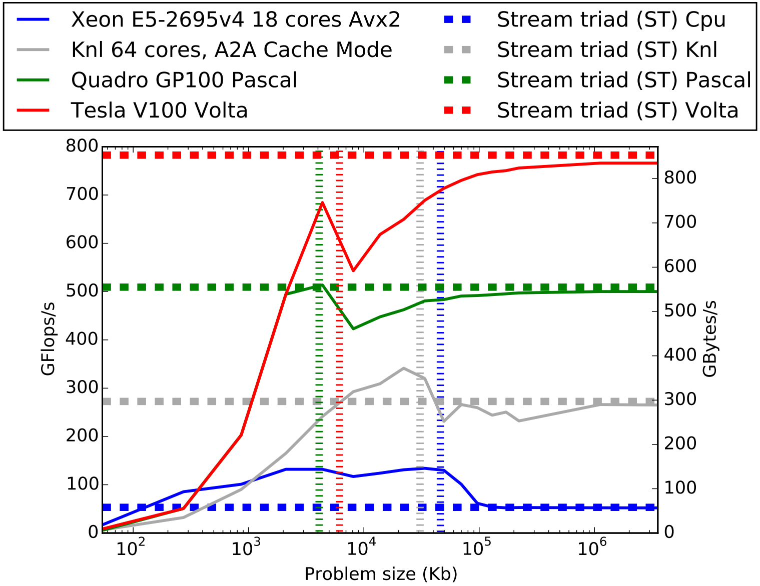
```
template<class Expr, class vObj> __global__ void ETapply(obj *_odata,Expr Op) {
  typedef typename vObj::scalar_object sObj;
  auto sD = evalS(blockIdx.x, Op, threadIdx.x);
  mergeS(_odata[blockIdx.x], sD, threadIdx.x);
}

template <typename Op, typename T1,typename T2>
inline Lattice<vObj> & operator=(const LatticeBinaryExpression<Op,T1,T2> expr) {
#ifdef __NVCC__
  ETapply<decltype(expr), vObj> <<<this->oSites(), this->iSites()>>> (this->_odata, expr);
#else
  // CPU code goes here...
#endif
  return *this;
}

template <typename Op, typename T1, typename T2>
auto inline evalS (const unsigned int ss, const LatticeBinaryExpression<Op,T1,T2> &expr, const int tIdx)
-> decltype(expr.Op.func(evalS(ss,expr.arg1,tIdx), evalS(ss,expr.arg2,tIdx))) {
  return expr.Op.func(evalS(ss,expr.arg1,tIdx), eval(ss,expr.arg2,tIdx));
}

template<class vObj> inline auto evalS(const unsigned int ss, const Lattice<vObj> arg, const int tIdx) {
  auto sD = extractS<vObj, sObj>(arg._odata[ss], tIdx);
  return (sObj) sD;
}
```

Porting Grid to Gpus



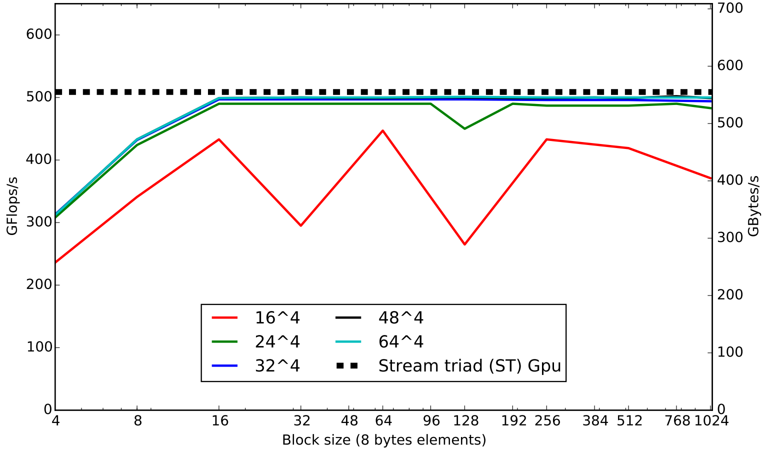
Porting Grid to Gpus

- Comparison P100, V100, KNL. CUDA + Grid Native datatypes



Porting Grid to Gpus

- Results with native datatypes almost independent of vector length
- Can be improved if we assign several vectors per block



Summary Grid

Main difficulties

- By design Grid easily create many temporary objects (constructor calls) during any operation
- The coalesced pointer also requires calls to constructors to operate
- In most implementations constructors are not easily called from device code
    - OpenACC completely forbids this
    - CUDA is more relaxed, but we still find segfaults
    - Jifity directly uses the GNU compiler and seemed to be the most flexible approach
- Unified memory is a must for our implementation
- **OpenACC** offers easy portability, but it's not mature enough and performance is not good
- **Jitify** must place kernels and datatypes in header files, which can be cumbersome
- **CUDA** with native Grid datatypes
    - Superb performance
    - No need to change the layout
    - The **nvcc** compiler might struggle with the C++
    - What do we do with the STL?
- Other members of the team are trying other approaches. For example, NIM (see Xian-Yong Jin's talk Wed. 2:35PM)

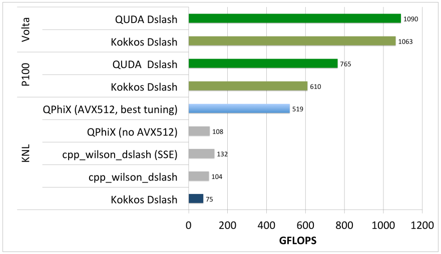
Next steps

- Integrate Grid's ET (not our stripped down version), tough C++ test for the compilers
- Implement a simple linear operator $w = Mv$ (dslash)

Other approaches: Kokkos (B. Joo, T. Kurth, J. Deslippe and K. Clark)

- Single Right Hand Side (SRHS) Case:
- Naive Implementation: No explicit vectorization over lattice sites
  - Not much vectorization opportunity for compiler
  - 3×3 and 3×4 complex matrices, short trip count loops
- Expect performance similar to legacy codes on KNL
- Lack of vectorization causes low performance on KNL
- GPU Performance is very good
  - 80% of QUDA Library on P100
  - No vectorization needed due to SIMT programming model
  - Each thread works on single site, in scalar mode

Single RHS Dslash Kernel

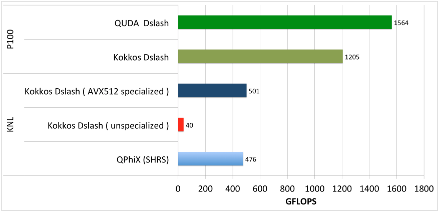


V100 results from NVIDIA, P100 result from SummitDev (OLCF), KNL results from Cori (NERSC). KNL nodes have 68 cores, but in some cases (QPhiX AVX512) only 64 may have been used, for load balance reasons. Lattice volume was 32^4 sites: B. Joo (JLab) T. Kurth and J.Deslippe (NERSC), K. Clark (NVIDIA)

Other approaches: Kokkos (B. Joo, T. Kurth, J. Deslippe and K. Clark)

- Multi Right Hand Side (MRHS) Case:
- Potential to vectorize over right hand sides and reuse gauge field
- Using regular Kokkos complex, performance was low
  - Still problems with vectorization
- Specializing Vector Type and using some AVX512 intrinsics gave a major boost (navy blue bar)
  - $12\times$ speedup over unspecialized
  - Comparable to QPhiX SRHS code
- GPU performance was really good, specially on Volta
  - Lower latency, unoptimized code runs better
  - $4\times$ speedup in int32 indexing operations
- **Code is performance portable**

Multi RHS Dslash Kernel



V100 results from NVIDIA, P100 result from SummitDev (OLCF), KNL results from Cori (NERSC). KNL nodes have 68 cores, but in some cases (QPhiX AVX512) only 64 may have been used, for load balance reasons. The lattice volume was $16^3 \times 32$ sites, with 8 RHS for KNL and 16 for GPU (natural H/W length) B. Joo (JLab) T. Kurth and J.Deslippe (NERSC), K. Clark (NVIDIA)

Thank you

Thank you for your attention

Acknowledgements

- We thank Mathias Wagner (NVIDIA) for his help during the recent hackathon
- B. Joo thanks NERSC, for travel support and a summer Associate appointment for the work involving Kokos, for which we also gratefully acknowledge use of computer resources at NERSC (Cori), OLCF (SummitDev) and NVIDIA (P100 and V100 devices).
- This research was supported by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of two U.S. Department of Energy organizations (Office of Science and the National Nuclear Security Administration) responsible for the planning and preparation of a capable exascale ecosystem, including software, applications, hardware, advanced system engineering and early testbed platforms, in support of the nation's exascale computing imperative.
- Part of this research was carried out at the Brookhaven Hackathon 2017. Brookhaven Hackathon is a collaboration between Brookhaven National Laboratory, University of Delaware, Stony Brook University, and the Oak Ridge Leadership Computing Facility at the Oak Ridge National Laboratory, and used resources of Brookhaven National Laboratory.