PORTING C++ APPLICATIONS TO GPU WITH OPENACC FOR LATTICE QUANTUM CHROMODYNAMICS (QCD)

Peter Boyle\textsuperscript{1}, Kate Clark\textsuperscript{2}, Carleton DeTar\textsuperscript{3}, Meifeng Lin\textsuperscript{4}, Verinder Rana\textsuperscript{4}, Alejandro Vaquero\textsuperscript{3}

\textsuperscript{1}University of Ediburgh
\textsuperscript{2}NVIDIA
\textsuperscript{3}University of Utah
\textsuperscript{4}Brookhaven National Laboratory
1. Lattice QCD

2. Grid C++ Data Parallel Library

3. Porting Grid to GPU

4. Performance

5. Conclusions
LATTICE QCD
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.
The core kernel of lattice QCD is matrix vector multiplications - the so-called \textit{Dslash} operator.

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

- \(x, y\) - regular 4-dimensional grid points.
- \(\gamma_\mu\) - \(4 \times 4\) matrices (fixed).
- \(U_\mu(x)\) - complex SU(3) matrices.
- \(\psi(y)\) - complex 12-component vectors.

The \textit{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 → 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.
Lattice QCD can calculate the properties of the fundamental particles such as the neutron and the proton from first-principles.

Requires enormous amount of computing power: Years of running on leadership-class supercomputers such as Mira and Titan.

Higher computing efficiency and/or more computing power will allow us to reach higher precision and perform more complex calculations.

Example results:

Particle masses calculated from LQCD

Mass differences of isospin multiplets
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.
GRID C++ DATA PARALLEL LIBRARY
Grid\(^1\) 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.

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.

```cpp
// Vectorization
#ifdef GEN
#include "Grid_generic.h"
#endif
#ifdef SSE4
#include "Grid_sse4.h"
#endif
#ifdef (AVX1 || AVX2 || AVX512)
#include "Grid_avx.h"
#endif
#ifdef 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.

```c
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.

```c
#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
```
PORTING GRID TO GPU
PORTING GRID WITH OPENACC

Why OpenACC?

▶ OpenACC, as a directive-based approach, is easy to introduce in any existing code, therefore it should score high in portability.
▶ OpenACC admits compilation for a large number of targets, such as AMD GPUs or even multicore computers.
▶ PGI’s OpenACC implementation supports Universal Virtual Memory (UVM), simplifying data movement and potentially eliminating the need for deep copy.

Naïve Top-Down Approach:

▶ Declare `acc kernels` in the key compute kernels. Did not work due to complicated call structures.
▶ Use of C++ STL container types also complicates things, as not all STL functions have device versions.

```c
#pragma acc kernels default(present)
for(int ss=0; ss<sites; ss++){
    int sU=ss;
    for(int s=0; s<Ls; s++){
        int sF = s+Ls*sU;
        Kernels::DiracOptDhopSite(st,U,st.comm_buf,sF,sU,in,out);}}
```

```
#pragma acc routine seq
template<class Impl>
void WilsonKernels<Impl>::DiracOptDhopSite(StencilImpl &st,DoubledGaugeField &U,
                   std::vector<SiteHalfSpinor,alignedAllocator<SiteHalfSpinor>> &buf,
                   int sF,int sU,const FermionField &in, FermionField &out)
```
Grid Expression Template Engine (P. Boyle): 200 lines of code, self-contained, captures a lot of language features in Grid.

OpenACC + Manual Deep Copy did not work due to the multi-layer, nested data structures.

**PGI’s UVM support came to the rescue!**

```cpp
inline Lattice<obj> & operator= (const obj & splatme) {
    int _osites = this->Osites();
    #pragma acc parallel loop independent copyin(splatme[0:1])
    for(int ss=0; ss<_osites; ss++) {
        _odata[ss] = splatme;
    }
    return *this;
}

template <typename Op, typename T1, typename T2> inline Lattice<obj> & operator=(const LatticeBinaryExpression<Op, T1, T2> expr) {
    int _osites = this->Osites();
    #pragma acc parallel loop independent copyin(expr[0:1])
    for(int ss=0; ss<_osites; ss++) {
        _odata[ss] = eval(ss, expr);
    }
    return *this;
}
```

**With PGI 16.10:**
```
pgc++ -I. -acc -fast -ta=tesla:managed -c++11 -Minfo=accel -O3 main.cc -o gpu.x
```
Interesting as it tests the streaming bandwidth typical for Lattice QCD calculations. Also a component of lattice gauge action calculations.

Custom SU(3) class implementation with OpenACC markup. Used as a data type for the ET engine. [A. Vaquero]

```cpp
//--------------
// suN.h
//--------------
#ifdef _OPENACC
#define OFFLOAD __Pragma("acc routine seq")
#else
#define OFFLOAD __host__ __device__
#endif

template<class vFloat, const int N>
class SuN {
  private:
    vFloat A[N*N];
  public:
    // code omitted ...

    OFFLOAD
    friend inline SuN<vFloat,N> operator+=(SuN<vFloat,N> lhs, const SuN<vFloat,N> &rhs) {
      // matrix add code
    }

    OFFLOAD
    friend inline SuN<vFloat,N> operator*=(SuN<vFloat,N> lhs, const SuN<vFloat,N> &rhs) {
      // matrix multiply code
    }
};
```
Main program is oblivious to the GPU offloading.

```cpp
//          
// main.cc  
//          
typedef SuN<complex<float>,3> Su3f;
int main(int argc, char **argv) {

    std::chrono::high_resolution_clock::time_point start, stop;
    int Nloop=1000;

    // code omitted ....
    int vol = L*L*L*L;
    Grid grid(vol);
    Lattice<Su3f> z(&grid);
    Lattice<Su3f> x(&grid);
    Lattice<Su3f> y(&grid);
    for(int i=0;i<Nloop;i++) {
        z=x*y;
    }
    stop = std::chrono::high_resolution_clock::now();
    double time = (std::chrono::duration_cast<std::chrono::microseconds>(stop - start).count())/Nloop*1000.0;
    double bytes = 3.0*vol*sizeof(Su3f);
    double footprint = 2.0*vol*sizeof(Su3f);
    double flops = 9*(6.0+8.0+8.0)*vol;
    std::cout << lat << "\t" << footprint << "\t" << bytes/time << "\t" << flops/time << std::endl; }
```
Explicit CUDA kernels + managed memory allocator.
Need to decorate all host device functions with __host__ __device__.

```cpp
#ifdef GPU

template<class Expr, class obj> __global__
void ETapply(int N, obj *o_data, Expr Op)
{
    int ss = blockIdx.x;
    o_data[ss] = eval(ss, Op);
}
#endif

template<typename Op, typename T1, typename T2>
inline Lattice<obj> & operator=(
    const LatticeBinaryExpression<Op, T1, T2> expr)
{
    int osites = this->Osites();
    #ifdef GPU
        LatticeBinaryExpression<Op, T1, T2> temp = expr;
        ETapply< decltype(temp), obj > «<_osites, 1»>((int)_osites, this->_odata, temp);
    #else
        for(int ss = 0; ss < osites; ss++){
            _odata[ss] = eval(ss, expr);
        }
    #endif
    return *this;
}
```

See S7716 at 4pm Monday: Ben Barsdell, Kate Clark “Jitify: CUDA C++ RUNTIME COMPIILATION MADE EASY”.

- Front-end to CUDA runtime compiler, nvrtc.
- Eliminates the need to decorate host device functions.

```cpp
template<class obj>
    template<typename Op, typename T1, typename T2>
    Lattice<obj> & Lattice<obj>::operator=(const LatticeBinaryExpression<Op,T1,T2> expr)
    {
        auto _osites = this->Osites();
        auto _odata = this->_odata;

        #ifdef USE_JITIFY
            // set JITIFY_OPTIONS="-include ET.h  -std=c++11"
            parallel_for(policy, 0, _osites, JITIFY_LAMBDA( (_odata,expr), _odata[i]=eval(i,expr); ));
        #else
            for (int i=0; i<_osites; i++) _odata[i] = eval(i,expr);
        #endif

        return *this;
    }
```
PERFORMANCE
Performance on GTX 1080: Summary

- Test platform: NVIDIA GTX 1080. Peak memory bandwidth 288 GB/s.
- Lattice-wide SU(3)xSU(3) streaming bandwidth with OpenACC, CUDA and Jitify.
  - OpenACC: PGI 16.10
  - CUDA: cuda 8.0
  - Jitify: nvrtc, cuda 8.0
- Use of coalesced_ptr in CUDA and Jitify, but not in OpenACC implementation.

---

**Peak Memory Bandwidth**

- **OpenACC**
- **CUDA**
- **Jitify**

![Graph showing peak memory bandwidth for different lattice volumes](image-url)

**Bandwidth [GB/s]**

**Lattice Volume**

- 16x16x16x16
- 24x24x24x24
- 32x32x32x32

---

Example ASCR-funded projects:

1. ECP Application Development for Lattice Gauge Theory (ASCR)
2. LQCD Algorithm and Software Development for HEP (SciDAC ASCR/HEP)
3. LQCD Algorithm and Software Development for NP (SciDAC ASCR/NP)
- [https://github.com/maddyscientist/coalesced_ptr (Kate Clark)](https://github.com/maddyscientist/coalesced_ptr)
- A smart pointer that automatically provides coalesced memory transactions for arrays of arbitrary structures.
- Boosts performance for the CUDA and Jitify implementations dramatically.
- Doesn't work with the OpenACC implementation due to the use of std::complex.

---

**Complex SU(3)xSU(3) Streaming Performance on NVIDIA GTX 1080**

- **CUDA, no coalesced_ptr**
- **CUDA, w/ coalesced_ptr**
- **Jitify, no coalesced_ptr**
- **Jitify, w/ coalesced_ptr**
- **OpenACC, no coalesced_ptr**
Complex SU(3)xSU(3) Streaming Performance on NVIDIA GTX 1080

- CUDA, no coalesced_ptr
- CUDA, w/ coalesced_ptr
- Jitify, no coalesced_ptr
- Jitify, w/ coalesced_ptr
- OpenACC, no coalesced_ptr
- coalesced_ptr works with float data type in OpenACC. [V. Rana]
- Modified U(3)xU(3) test also gets 70% to 80% of peak memory bandwidth.
To get the previous performance, we also need to turn off the default llvm code generator in PGI compiler through `-ta=tesla:managed,nollvm` option, or force inlining with `-Minline`.

![SU(3)xSU(3) streaming test on NVIDIA GTX 1080](image)
CONCLUSIONS
OpenACC in principle provides an easy way to port Grid to GPUs in terms of minimal code changes.

However, the nested data structure and complex data types in Grid makes it challenging.

PGI’s UVM support alleviates the issue of deep copy, but its support for C++ still needs improvement.

We have succeeded in implementing an OpenACC version of SU(3)xSU(3) streaming test which delivers reasonable performance compared to CUDA and JIT.

Memory coalescence is very important for achieving maximum performance on GPUs.

Future work will include a Dslash mini-app that represents the key lattice QCD compute kernel.

Stay tuned!
ACKNOWLEDGMENTS
Partly funded by the US Exascale Computing Project and BNL-SBU Seed Program.

We are thankful for the helpful discussions with the QCD ECP team.

We benefited from the GPU Hackathon at the University of Delaware, where the OpenACC porting was initially started.

We thank Mathias Wagner (NVIDIA) and Mathew Colgrove (NVIDIA), our mentors at UDelHack, who continue to support us after the Hackathon.