Performance Portability Strategies for Grid C++ Expression Templates

Peter A. Boyle[1], *M.A.* Clark[2], *Carleton* DeTar[3], *Meifeng* Lin[4,*], *Verinder* Rana[4], and *Alejandro* Vaquero Avilés-Casco[3]

[1]*Higgs Centre for Theoretical Physics, School of Physics & Astronomy, University of Edinburgh, EH9 3FD, UK*
[2]*NVIDIA Corporation, Santa Clara, CA 95050, USA*
[3]*Department of Physics and Astronomy, University of Utah, Salt Lake City, UT 84112, USA*
[4]*Computational Science Initiative, Brookhaven National Laboratory, Upton, New York 11973, USA*

Abstract. One of the key requirements for the Lattice QCD Application Development as part of the US Exascale Computing Project is performance portability across multiple architectures. Using the Grid C++ expression template as a starting point, we report on the progress made with regards to the Grid GPU offloading strategies. We present both the successes and issues encountered in using CUDA, OpenACC and Just-In-Time compilation. Experimentation and performance on GPUs with a SU(3)×SU(3) streaming test will be reported. We will also report on the challenges of using current OpenMP 4.x for GPU offloading in the same code.

1 Introduction

The USQCD Collaboration and its partners[1] are engaged in the application development (AD) efforts [1] as part of the US Exascale Computing Project [2] to redesign a software stack for lattice QCD simulations for the anticipated exascale computing systems becoming available in a few years. While we do not know the exact configurations of the exascale systems, the architectures are expected to be diverse, and consist of heterogeneity, complex memory hierarchies and multiple levels of parallelism. One of the key requirements for the USQCD exascale data-parallel applications programming interface (API) is *performance portability*. That is, the ability to run the same code on different architectures without losing too much performance. The Grid [3] C++ data-parallel library is being considered as one of the candidate frameworks on top of which the data-parallel API will be developed. Therefore an investigation of the ease of portability in Grid has been performed.

Grid is a data-parallel library for lattice QCD. It uses C++11 features and has its own expression template engine. Grid has been optimized for CPU-based systems, including Intel Xeon CPUs, Intel Knights Corner and Knights Landing processors, and the IBM BG/Q systems. Its use of expression template engine and complex data structures makes it challenging to port to GPUs. The initial effort to

---

[*]Speaker, email: mlin@bnl.gov
[1]As of Lattice 2017, our partners included University of Edinburgh, University of Illinois, NVIDIA and Stony Brook University.

port Grid to GPUs started in summer 2016 when a subset of the authors implemented a GPU version of the stripped-down Grid expression template engine using a simple CUDA kernel [4]. Since then, the QCD ECP AD team has been exploring more extensively different portability approaches for Grid. Since the whole Grid library is quite complicated and has tens of thousands of lines of code, we again started out with a simple prototype code that contains the Grid expression template engine for proof of concept, and develop examples to include more realistic use cases in QCD simulations. This paper summarizes the findings and roadblocks in the various approaches.

Our main considerations are *ease of use* and *performance*:

- Ease of use. The portability requirement of the USQCD exascale software API means we do not want to rewrite the whole library in order to port it to GPUs. Instead, we want to reuse as much code as we can, and the ideal portability technique would require only a few trivial modifications.

- Performance. We aim to develop software optimized for the next generation of supercomputers, capable of computing at the exascale level. The portable code should perform well in all next-generation platforms, in order to stay competitive.

To meet both of these two requirements can be complicated, and compromises may be needed. Code with high portability can be accepted if the performance loss is minor. A small rewrite of a few key parts of the library might be acceptable. Hence we must look for a method that will achieve the most balanced result. To this end, we have studied OpenACC, OpenMP, Just-In-Time compilation and CUDA as part of our performance portability research, which will be described in Section 2. Following a proof-of-concept test in Section 3, we show the performance of a lattice-wide SU(3)×SU(3) test in Sections 4 and 5, first with a scalar data layout and then with a vector data layout. Section 6 contains the summary and outlook of the project.

2 Technical Approaches

2.1 OpenACC

OpenACC is a standard of compiler directives that allows offloading computationally intensive tasks to an accelerator. In C/C++ code, OpenACC statements are introduced as

```
#pragma acc ...
```

where the . . . should be replaced with one of the multiple OpenACC directives available. These statements indicate parallelizable regions or data movement from/to the accelerator, and the compiler generates the necessary compute and data movement kernels based on the directives. In this respect, OpenACC is quite similar to the familiar OpenMP shared-memory programming model. However, to offload a compute region to the GPU using OpenACC, we also need to be concerned about data movement, whereas in the OpenMP threading model, the programmer is mostly concerned about how to distribute the threads.

OpenACC loops can be divided into three levels: gangs, workers and vectors. A gang represents a group of workers, and a worker a group of vectors. They exist to accommodate different degrees of parallelism existing in a GPU (there is a grid of blocks, and each block contains several threads). They can also be mapped to multicore vector architectures. Users can also choose to use descriptive directives such as `kernels` which will give the compiler the freedom to analyze the code and generate compute kernels as it sees fit. For a more detailed introduction to OpenACC, refer to [5].

OpenACC, as a directive-based approach, is easy to introduce in any existing code, and satisfies our "ease of use" requirement. It also allows compilation for a large number of targets, such as NVIDIA GPUs, AMD GPUs and multicore architectures, making it a highly portable approach.

However, the generated kernels are dependent on the compiler, giving users limited control over performance. In addition, while C code is easy to port to GPUs using OpenACC, porting C++ code is often non-trivial due to complicated data structures that would make it hard to manage the data movement between CPU and GPU without the support of deep copy [5]. PGI compiler's support for the unified virtual memory (UVM) greatly alleviates the difficulty with the data management for C++ code, but other issues such as the lack of support for C++ standard template library (STL) persist.

2.2 OpenMP 4.5

Recently in OpenMP 4.5, the concept of fork-join parallelism has been extended to accelerator devices. Computationally intensive code can now be offloaded onto the accelerators thanks to this feature. If multiple accelerators are connected to the host, OpenMP allows one to take advantage of them by specifying their device IDs. To execute on the device, OpenMP uses the `#pragma omp target` construct, which is similar to the OpenACC `#pragma acc parallel` construct. The `#pragma omp teams` construct starts a team of threads and begins parallel execution. The data directives in OpenMP are given by the `map` clause which further gets decorated using `to`, `from`, `tofrom` which essentially are similar to `copyin`, `copyout`, `copy` in OpenACC. In addition, the concepts of `gang`, `worker`, `vector` in OpenACC correspond to `teams`, `distribute parallel for`, `distribute simd` in OpenMP.

OpenMP is similar to OpenACC in the sense that they are both relying on the compiler to generate the compute kernels. However, OpenMP faces more challenges than OpenACC for C++ code due to the lack of UVM support in the current compiler implementations [6, 7].

2.3 CUDA

CUDA is NVIDIA's API for GPU programming. Since it is vendor-specific, it does not provide true portability. Given that NVIDIA's GPUs are the main-stream GPU architecture for high performance computing, we consider this an acceptable approach. It is also necessary to write some CUDA kernels, which will cause some code branching and potentially enlarge the code size significantly. In addition, all functions called by the CUDA kernels need to be decorated with `__device__` qualifier and with `__host__` as well if it is also compiled for CPU. That may become too tedious for a large code base[2]. However, it is a mature GPU programming model that offers good performance on NVIDIA GPUs, and its C++ support has been steadily improving. This approach is also interesting because Grid has important areas of code such as the stencil operators that would also need to be offloaded to the GPUs and may be simple to do with CUDA.

2.4 Just-In-Time (JIT) Compilation and Jitify

An alternative to using CUDA and compiler directives is to use the CUDA runtime compiler `nvrtc` to perform "Just-In-Time" (JIT) compilation on demand. Since `nvrtc` is a pure GPU compiler, it is not as constrained as the off-line heterogeneous compiler, and can be forced to interpret all encountered functions as `__device__` functions, eliminating the need to decorate the functions `__host__` `__device__` necessary for explicit CUDA programming.

Jitify [8] is a new C++ header library developed at NVIDIA that provides a simple front-end to `nvrtc`. It uses C++11 features to present simple single lambda-style launch syntax, and can choose GPU or CPU as runtime execution policy, where CPU code is compiled off-line and GPU code is compiled during runtime. Jitify removes all CUDA-specific extensions from user code, but the user

---

[2]Future versions of CUDA may allow for `__host__` `__device__` namespaces, which can greatly simplify this approach.

can still access them if needed. However, its just-in-time compilation nature makes it hard to detect problems before program execution. In addition, all kernel functions need to be known in the header files.

3 Porting the Grid Expression Template Engine

In this section we demonstrate the proofs of concept of using the above approaches in a simple stripped-down example code. Listing 1 shows the key components of the CPU version of the stripped-down expression template in Grid, where the main computation occurs in the highlighted `for` loop.

```
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;
  }
};
```

Listing 1. Grid expression template

The expression template engine will allow for evaluations of arbitrary expressions such as the ones shown below, where `v1, v2` and `v3` are lattice-wide arrays of the `double` type:

```
Lattice<double> v1(&grid);
Lattice<double> v2(&grid);
Lattice<double> v3(&grid);
v1=1.0;
v2=2.0;
v3=v1+v2;
v3=v1+v2+v1*v2;
```

Listing 2. Example composite expressions.

For each of the approaches, the code changes needed to get the expressions to be evaluated on the GPU are shown in Table 1. All of our implementations assume UVM support, either through the `cudaMallocManaged` allocator for CUDA and Jitify, or through `-ta=managed` compile flag for OpenACC with the PGI compiler. The OpenMP implementation did not work due to its lack of deep-copy or UVM support.

4 SU(3)×SU(3) With Scalar Data Layout

In this section we demonstrate implementations using CUDA, OpenACC and Jitify of an SU(3)xSU(3) streaming test. The starting point is a templated $N \times N$ matrix class which is then used as the data type for the expression template, i.e. replacing `double` in Listing 2 with `Su3f` (for single-precision complex 3×3 matrices).

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

Listing 3. SU(3)×SU(3) test code snippet.

| 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);
``` |

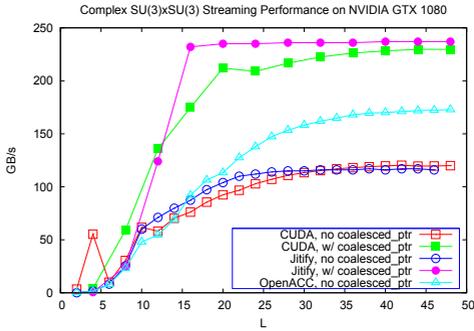**Table 1.** Comparison of compute kernels in different GPU programming models.

In this näive implementation, the lattice-wide arrays of complex 3×3 matrices are not vectorized, unlike in the actual Grid implementation where vectorized layout is built-in (see Section 5). This Array of Structure (AoS) data order limits its achievable performance on the GPU due to the lack of memory coalescence. To boost performance we used the `coalesced_ptr` class [9] that automatically break up AoS into AoSoAoS order to ensure coalesced thread access.

We performed our tests on a workstation with NVIDIA GTX 1080 and quad-core Intel i7 CPU. The maximum bandwidth on the GTX 1080 is 288 GB/s, with maximum sustainable bandwidth of around 240 GB/s. Shown in Figure 1 are the bandwidth results with our CUDA, Jitify and OpenACC implementations. Without the use of the `coalesced_ptr` class, CUDA and Jitify could obtain only about half of the maximum sustainable bandwidth (~ 120 GB/s), while OpenACC outperforms both of them. When `coalesced_ptr` is used, both CUDA and Jitify saturated the memory bandwidth, with CUDA under-performing Jitify at large memory sizes and outperforming at small memory sizes, likely due to the different thread and block number choices in these two approaches. OpenACC with `coalesced_ptr` did not work with complex data types due to PGI compiler bugs, but with real data type (`float`), OpenACC with `coalesced_ptr` was also able to deliver reasonable memory bandwidth, as shown in Figure 2.
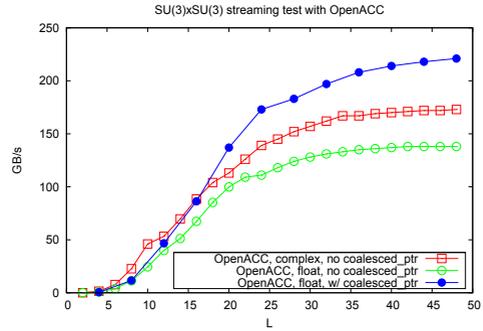
5 SU(3)×SU(3) With Vector Data Layout

While using the `coalesced_ptr` class allowed us to achieve very good bandwidth in our SU(3)×SU(3) tests, it is interesting to see if we can obtain coalesced access with Grid's native vector data layout that is optimized for the SIMD processors. To achieve efficient vectorization Grid splits the `Lattice` object on each compute node into N_{sub} virtual nodes, where N_{sub} is equal to the SIMD length of the processors divided by the floating point precision. The splitting of the `Lattice` object in SIMD lanes occurs at the level of the basic data types such as `complex` or `float`. Any higher-level data types built on these fundamental data types will be vectorized as well.

On the CPU, parallelization on SIMD lanes is straightforward because the bulk points in different SIMD lanes are independent from each other and can be computed at the same time. On the GPU,

Figure 1. Complex SU(3)×SU(3) streaming bandwidth on NVIDIA GTX 1080 for implementations with OpenACC, CUDA and Jitify.

Figure 2. Comparison of OpenACC SU(3)×SU(3) streaming bandwidth with and without coalesced pointer.

this Array-of-Structures-of-Vectors/Arrays (AoSoV, or AoSoA) data layout allows for an efficient mapping to GPU threads. For example, each thread block can process one or several vectors, and the threads can operate on different SIMD lanes of each vector. This approach should achieve coalesced access if the threads in a warp are forced to read data in consecutive memory.

The main obstacle in this approach is the fact that our `Lattice` object hosts vectorized data types. In the particular case of our SU(3)×SU(3) test, each element of the matrix is actually a vector of real or complex numbers corresponding to different virtual nodes. If we apply the expression template as before without any changes, each GPU thread will operate on the vector objects, rather than one element of the SIMD vector, which is the complete opposite of what we would want. Instead, we need to extract the scalar data from the vector data types before the GPU computation, and then merge them back to form the vector objects after, as shown below:

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

$$
\begin{pmatrix} a_1^{11} & \cdots \\ \vdots & \ddots \end{pmatrix} \quad \begin{pmatrix} a_2^{11} & \cdots \\ \vdots & \ddots \end{pmatrix}
$$
Thread 1   Thread 2

$$
\begin{pmatrix} a_3^{11} & \cdots \\ \vdots & \ddots \end{pmatrix} \quad \begin{pmatrix} a_4^{11} & \cdots \\ \vdots & \ddots \end{pmatrix}
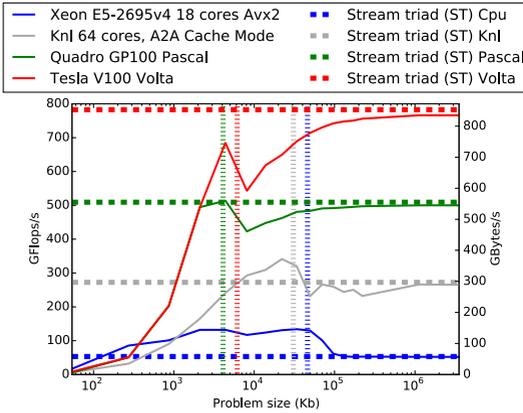$$
Thread 3   Thread 4

Grid provides functions to perform the extraction/insertion of scalarized objects from/into vectorized ones. In terms of the expression template engine, we implemented the extract and merge functions so that each GPU thread processes one SIMD lane and each block gets the whole vector.

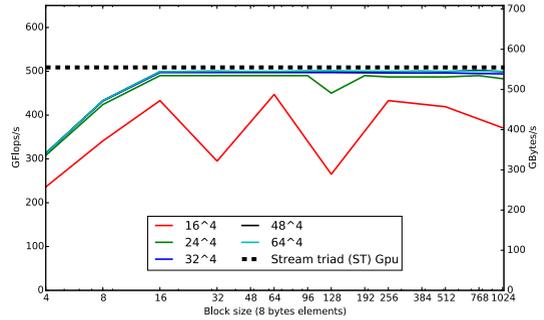The relevant code snippets are shown in Listing 4.

```
//Make each thread {eval} one element of the vector, extracted through {extractS}.
   auto sD = extractS<obj,sObj>(arg._odata[ss], tIdx);
   ...
//Put the results back to form the vector again after {evalS}.
   auto sD = evalS(blockIdx.x,Op,threadIdx.x);
   mergeS(_odata[blockIdx.x], sD, threadIdx.x);
```

Listing 4. Extract and merge the vector objects in Grid.

Figure 3. Performance of the miniapp in several architectures. The vertical lines indicate the cache size, the horizontal ones show the memory bandwidth obtained in a stream triad test. In all cases we can easily saturate the memory bandwidth of the machine.

Figure 4. Performance of the miniapp as a function of the vector length in a Quadro GP100. The vector length is measured in terms of single precision complex numbers. For a vector of 16 complex numbers (twice the size of a KNL vector), performance hits the maximum.

With the same code we could run the SU(3)×SU(3) tests on several different platforms, including the Intel Xeon E5-2695v4 "Broadwell" processor at 2.1 GHz, the Intel "Knights Landing" Xeon Phi(TM) CPU 7230 1.30GHz [3], the NVIDIA Quadro GP 100 and Tesla V100 GPUs, the results of which are shown in Figure 3. The left axis shows the measured floating point performance, while the right axis shows the measured bandwidth. The horizontal dashed lines indicate the plateaued results from STREAM Triad tests on the same platforms, and the vertical lines indicate the cache sizes of these architectures. The cache effects are evident, and the fact that our results follow closely the STREAM triad results shows that our implementation indeed achieves the maximum memory bandwidth possible.

The GPU performance may depend on the vector length of the data types, or in GPU language, the block size in the kernel call, as shown in Figure 4. The performance plateaus with a block size of 16 threads, or equivalently a vector length of 16 single-precision complex numbers. The large vector length required to achieve the best performance is not desirable as it puts more constraints on the lattice dimensions and vectorization choices. A solution is to create a two-dimensional block and assign several vectors per block. The *x* coordinate of the thread index would run through the SIMD lanes, as before, but now the *y* coordinate would run over the different vectors inside the block. This has not been implemented yet, but will be in the near future.

6 Summary and Outlook

We have investigated three approaches for performance portability in the context of Grid expression templates: OpenACC and/or OpenMP directives, CUDA and Jitify. First we demonstrated how we could offload the Grid expression template engine to GPUs in a simple test case. Then we did some performance studies in the SU(3)×SU(3) streaming test, first with a naive data layout, then with Grid's vector data layout.

OpenACC, in combination with PGI compiler's UVM support, can be used to offload Grid's expression template engine, and also delivers reasonable performance in our SU(3)xSU(3) streaming

---

[3]Booted in all-to-all cache mode which renders lower bandwidth than the quad cache mode.

test. However, there are a number of compiler issues that potentially limit our ability to use OpenACC in a large code base.

Jitify eliminates the need to decorate the device functions, has a simple user interface and can deliver good performance if used with coalesced pointer class. However, it is a relatively new JIT library and still under active development. Its usability in a large code still needs to be tested.

Using CUDA to offload requires all the host functions also called in a GPU kernel to be decorated as `__host__ __device__`, which may become tedious and error-prone in a large code base. Another issue we encounter repeatedly is that using `std::complex` would cause the compiler to complain, likely due to the fact that the operators in std::complex are not decorated as `__host__ __device__` functions. This is a universal issue with the GPU offloading of the C++ STL. But in the example we have tested, CUDA shows as a viable option both in terms of ease of programming and performance, and is our programming model of choice for porting Grid to the GPU given the current state of the art.

We have also demonstrated that Grid's SIMD-friendly vector data layout can be easily mapped to the GPU execution model and achieve good performance. To eliminate the need for a large vector length in order to achieve the best GPU performance, a two-dimensional thread block mapping may be necessary, which will be implemented in the future. We are in the process of porting the Dslash operator in Grid to GPUs, in which some routines that are outside of the Grid expression template engine, such as the Cshift operations, may require additional attention.

Acknowledgments

References

[1] C. DeTar, *Lattice QCD Application Development within the US DOE Exascale Computation Project*, in *Proceedings, 35th International Symposium on Lattice Field Theory (Lattice2017): Granada, Spain*, to appear in EPJ Web Conf.

[2] https://exascaleproject.org/

[3] P.A. Boyle, G. Cossu, A. Yamaguchi, A. Portelli, PoS **LATTICE2015**, 023 (2016)

[4] P.A. Boyle, PoS **LATTICE2016**, 013 (2017), `1702.00208`

[5] R. Farber, ed., *Parallel Programming with OpenACC*, 1st edn. (Morgan Kaufmann, Boston, 2017)

[6] I. Karlin, T. Scogland, A.C. Jacob, S.F. Antao, G.T. Bercea, C. Bertolli, B.R. de Supinski, E.W. Draeger, A.E. Eichenberger, J. Glosli et al., *Early Experiences Porting Three Applications to OpenMP 4.5* (Springer International Publishing, Cham, 2016), pp. 281–292

[7] *OpenMP 4.5 IBM November 2015 Hackathon: Current Status and Lessons Learned*, https://codesign.llnl.gov/pdfs/2015-11-IBM-OpenMP-Hackathon-outbrief-final.pdf

[8] https://github.com/NVIDIA/jitify

[9] K. Clark, https://github.com/maddyscientist/coalesced_ptr