EXASCALE COMPUTING PROJECT

Experience with using OpenMP offloading to achieve performance portability for the Grid lattice QCD library

Meifeng Lin (Brookhaven National Laboratory)

Collaborators: Peter Boyle (BNL/U. Edinburgh), Lingda Li (BNL), Kate Clark (NVIDIA), Alejandro Vaquero (U. Utah), Vivek Kale (BNL), Barbara Chapman (BNL/SBU)

P3HPC Forum, September 1-2, 2020

Lattice QCD

- Lattice Quantum Chromodynamics (QCD) is a numerical framework to simulate the strong interactions between quarks and gluons.
- continuous 4D space-time => 4D lattice after discretization
- Physical observables calculated from lattice QCD provide important insights to the QCD theory through comparisons with experimental results, e.g.
  - Internal structures of protons, pions, etc.
  - Bounds for new physics
- Key Algorithm Motifs
  - Markov Chain Monte Carlo
  - Sparse matrix inversions

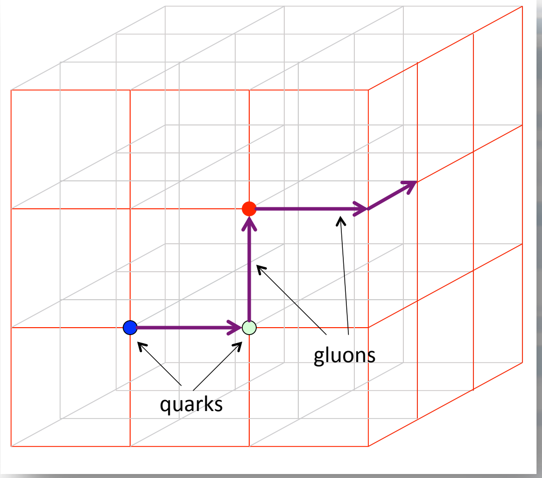
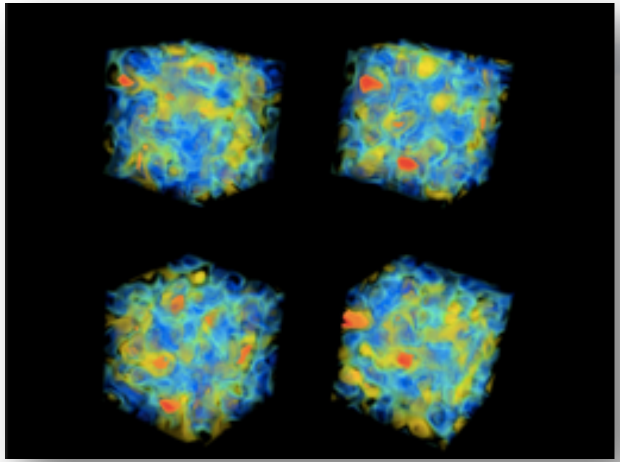


Illustration of a 3D lattice



Visualization of QCD topological charge density. M. McGuigan (BNL)

EXASCALE COMPUTING PROJECT

Computational Kernel

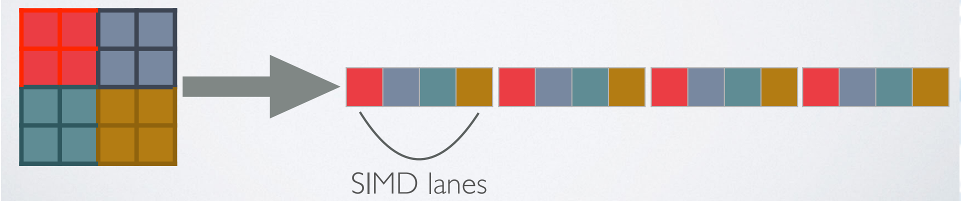
- The core computational kernel of lattice QCD is matrix vector multiplications – the so-called Dslash operator.

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

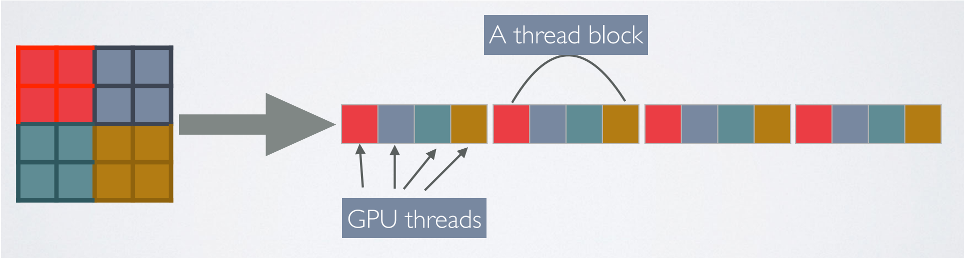
  - $\boldsymbol{x}, \boldsymbol{y}$**:** 4D coordinates
  - $\boldsymbol{\gamma_\mu}$: 4×4 matrices (fixed)
  - $\boldsymbol{U_\mu}(x)$: complex 3×3 matrices, 4 per lattice site (main memory usage)
  - $\boldsymbol{\psi}(\boldsymbol{y})$: complex 12-component vectors, 1 per site (main memory usage)
- <u>Matrix-vector multiplication form is known analytically. No actual matrices are stored.</u>
- Memory requirements per site: $(9\times2\times4+12\times2)\times8=768$ bytes (DP)
- Floating point operations for Wilson Dslash: 1320 flops per site
- **Low arithmetic intensity: 1.7 flops/byte (DP) or 3.4 flops/byte (SP)**

The Grid C++ QCD Library

- Grid[1] is a C++ library for lattice QCD
- Initially designed for SIMD architectures with long SIMD length (Intel Knights Landing, Skylake, etc.).
- Arranges the data layout as if the lattice is divided into virtual “sub-lattices”.
- Each sub-lattice uses one SIMD lane.
- Same data layout can be mapped to GPU architectures
- C++11 (lambda, auto types, etc.)
- Extensive use of templates for high-level abstraction
- Custom expression template engine for performance



Data mapping on SIMD architecture



Data mapping on SIMT architecture

[1] P. Boyle et al., arXiv:1512.0348, https://github.com/paboyle/Grid

Grid's Performance Portable Design

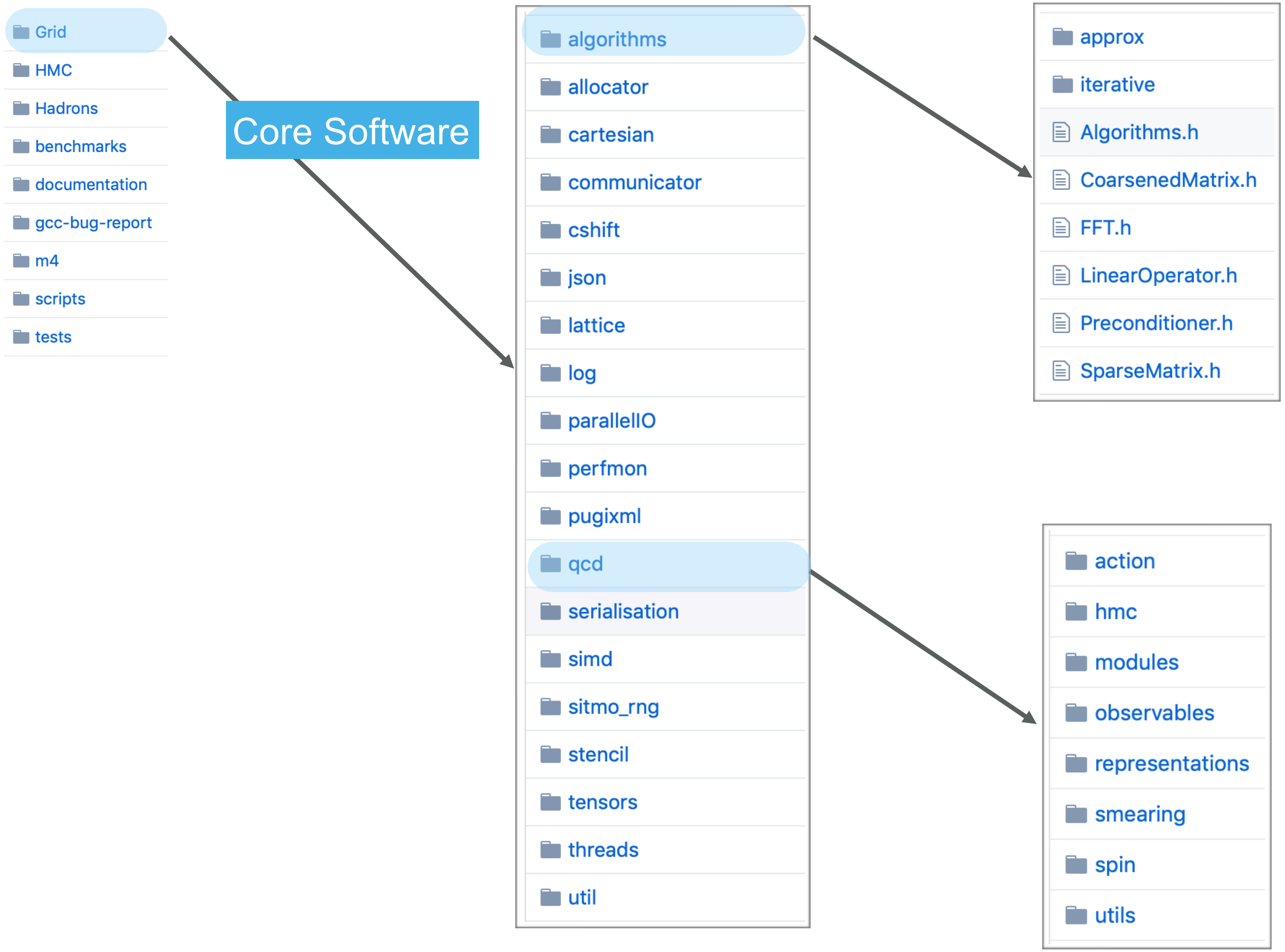
- **Header file with macros to encapsulate architecture-dependent implementations**

```
#ifdef GRID_NVCC
#define accelerator        __host__ __device__
#define accelerator_inline __host__ __device__ inline
#define accelerator_for (…) { //CUDA kernel}

#else
#define strong_inline      __attribute__((always_inline)) inline
#define accelerator
#define accelerator_inline strong_inline
#define accelerator_for(…)  thread_for(…) //for loop with #pragma omp parallel for
```

- **Custom AlignedAlloctor for dynamic memory allocation on different architecture**

```
#ifdef GRID_NVCC
    if ( ptr == (_Tp *) NULL ) auto err = cudaMallocManaged((void **)&ptr,bytes);
#else
  #ifdef HAVE_MM_MALLOC_H
    if ( ptr == (_Tp *) NULL ) ptr = (_Tp *) _mm_malloc(bytes,GRID_ALLOC_ALIGN);
  #else
    …
```

Grid Structure



GridMini

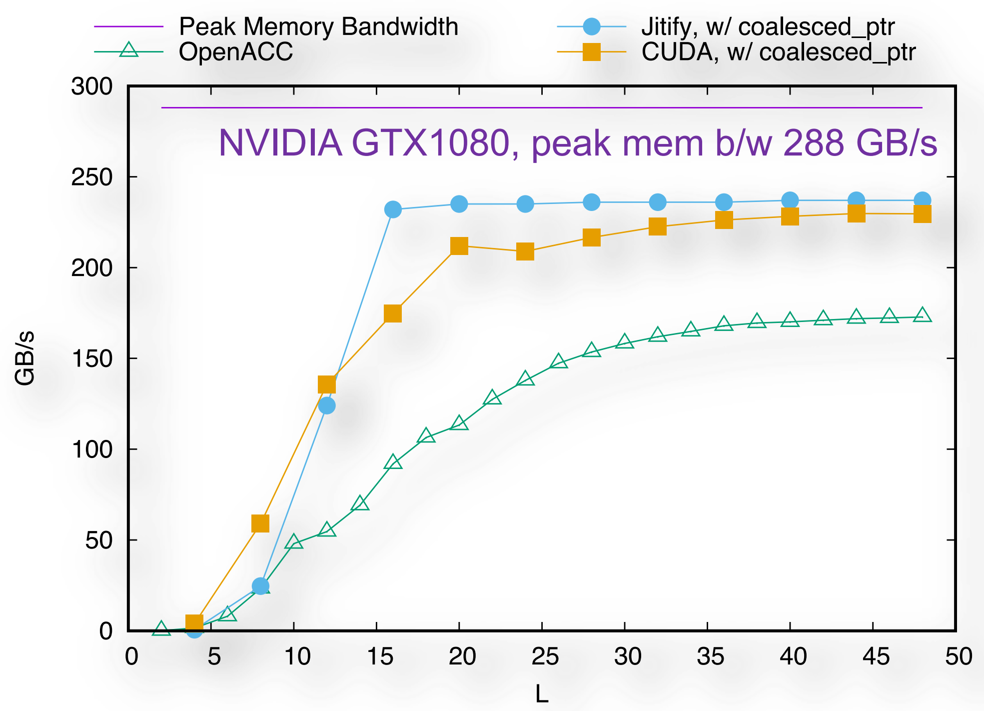
- A substantially reduced version of Grid for experimentation with different programming models.
- Retains same Grid structure: data structures/types, data layout, aligned allocators, macros, …
- Only keeps the high-level components necessary for the benchmarks.
- **SU(3)⨯SU(3) benchmark**: STREAM-like memory bandwidth test
- Important as LQCD is bandwidth bound.

Benchmark_su3

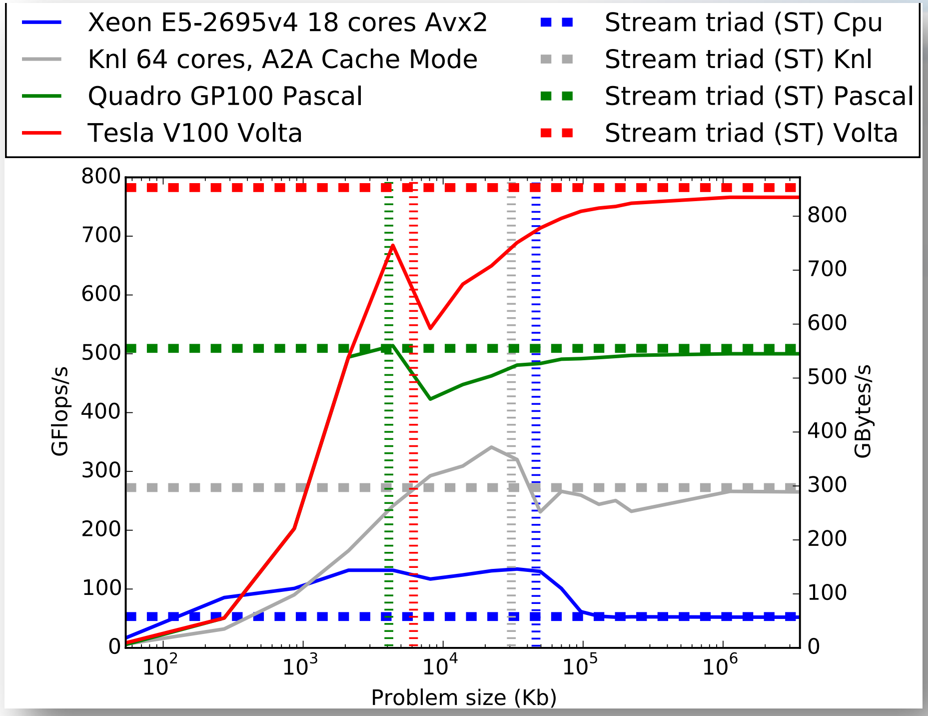
```
LatticeColourMatrix z(&Grid); //Arrays of SU(3)
LatticeColourMatrix x(&Grid); //Arrays of SU(3)
LatticeColourMatrix y(&Grid); //Arrays of SU(3)

double start=usecond();
for(int64_t i=0;i<Nloop;i++){
    z=x*y;
}
double stop=usecond();
double time=(stop-start)/Nloop*1000.0;

double bytes=3*vol*Nc*Nc*sizeof(Complex);
double flops=Nc*Nc*(6+8+8)*vol;
double bandwidth=bytes/time; //GB/s
double Gflops=flops/time;    //0.9 flops/byte SP
```

Previous Work

- Previously[2] explored porting Grid to GPUs using **OpenACC**, **JIT** and **CUDA**.
- Use `coalesced_ptr[3]` to force data coalescencing to get best performance.



- In the **CUDA implementation**, Grid's SIMD data layout can be used to force data coalescencing.
- Same data layout across SIMD and SIMT archs.



[2] Boyle, Peter A., et al. "Performance portability strategies for grid C++ expression templates." EPJ Web of Conferences. Vol. 175. EDP Sciences, 2018. [3] https://github.com/maddyscientist/coalesced_ptr

EXASCALE COMPUTING PROJECT

OpenMP Offloading

- **To add OpenMP offloading (to NVIDIA GPUs) for Benchmark_su3, only two changes required.**
- New macros

```
#elif defined (OMPTARGET)
#define accelerator_inline strong_inline
#define accelerator_for(iterator,num,nsimd, ... )  \
{                                                  \
    _Pragma("omp target teams distribute parallel for”) \
      naked_for(iterator, num, { __VA_ARGS__ }); \
}
```

- Use `cudaMallocManaged` for the memory allocator (for now)

```
#if definded (GRID_NVCC) || defined (OMPTARGET_MANAGED)
    if ( ptr == (_Tp *) NULL ) auto err = cudaMallocManaged((void
**)&ptr,bytes);
```

Issues

- **Deep copy:** explicit data mapping **nontrivial** due to deep nested data structures
  - Managed/Unified memory greatly simplifies things
- **Incorrect results:** output always 0 after offloading
  - Compiler bug related to the use of struct of short vectors as device function return type.

```
struct vec {
  float v[2];
};

inline  vec mult(vec x, vec y){
  vec out;
  out.v[0]=x.v[0]*y.v[0];
  out.v[1]=x.v[1]*y.v[1];
  return out; //causing issue here
}
```

```
vec in1,in2;
vec out[N];
#pragma omp target teams distribute parallel for
map(to:in1,in2) map(from:out[0:N])
for(int n=0;n<N;n++) {
    out[n]=mult(in1,in2);
}
```

Got: 0.000000 0.000000
Expected: 2.000000 2.000000

  - **Fixed as of llvm/12.0.0-git_20200731**

Compiling and Performance

- Compiling with clang++ (**Original**)

```
CXX=clang++
CXXFLAGS=-std=c++14 -g -fopenmp -O3 -fopenmp-targets=nvptx64-nvidia-cuda \
        -lcudart
```

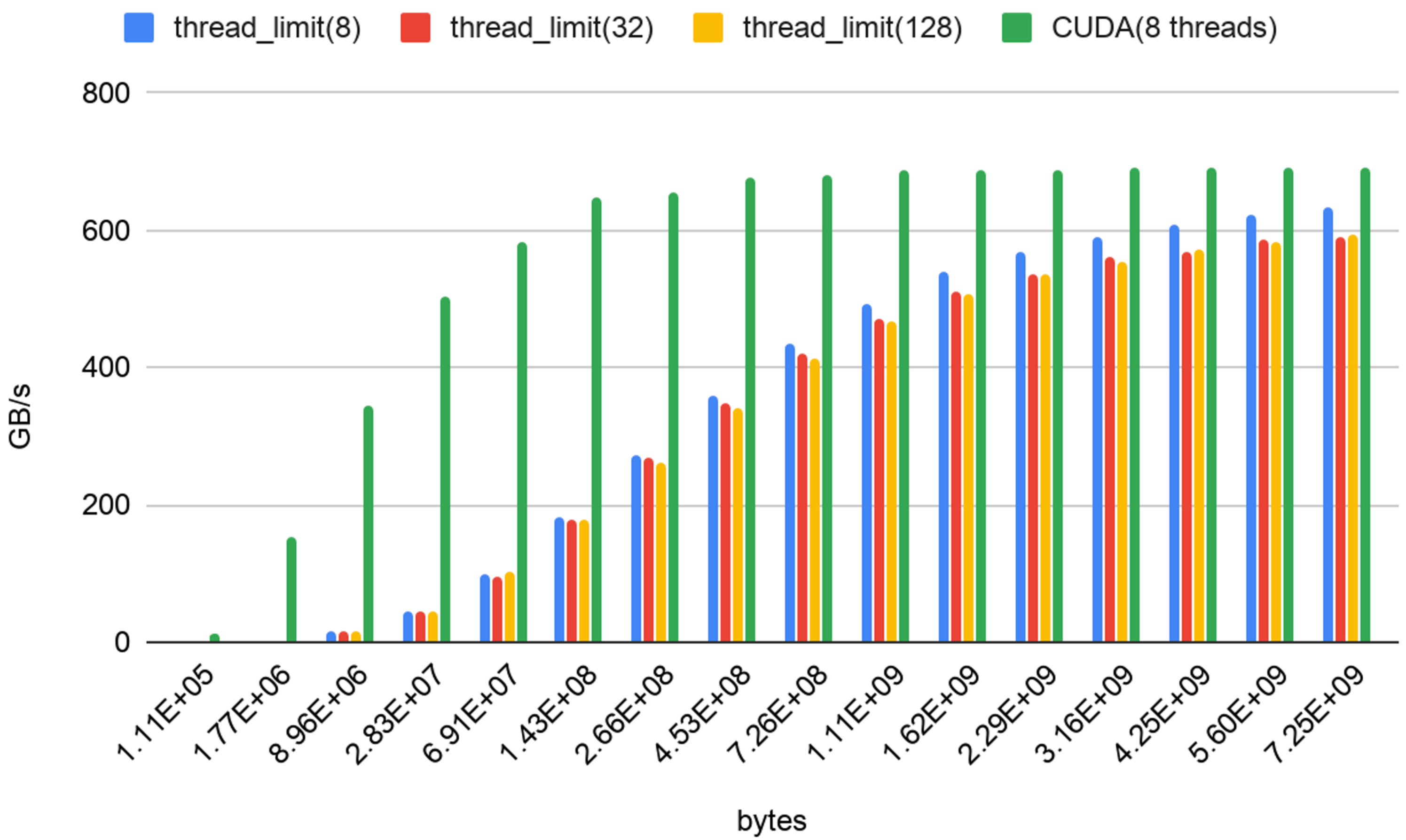
- Performance of Benchmark_su3 maxed out at **125 GB/s** on NVIDIA V100 (Cori-GPU)
  - Peak memory bandwidth should be 900 GB/s
- Learned about `-fopenmp-cuda-mode` at the SOLLVE OpenMP Hackathon*.

```
CXX=clang++
CXXFLAGS=-std=c++14 -g -fopenmp -O3 -fopenmp-targets=nvptx64-nvidia-cuda \
        -lcudart –fopenmp-cuda-mode
```

  - Guarantees to the compiler that the target region is in SPMD mode
  - Performance improved significantly (5X)!

*Special thanks: **Rahul Gayatri (LBNL), Johannes Doerfert (ANL)**

SU(3)xSU(3) Performance Comparison



- With -fopenmp-cuda-mode
- thread_limit(8) seems to be slightly better than 32 or 128.
- CUDA bandwidth plateaus much earlier: **why?**
- With large data, OpenMP version is about 90% of CUDA performance.
- With small data sizes, OpenMP performs much worse: OpenMP version has more overhead?

*benchmark performed on Cori GPU (V100); llvm/12.0.0_git_20200731; cuda/10.1.243

Profiling

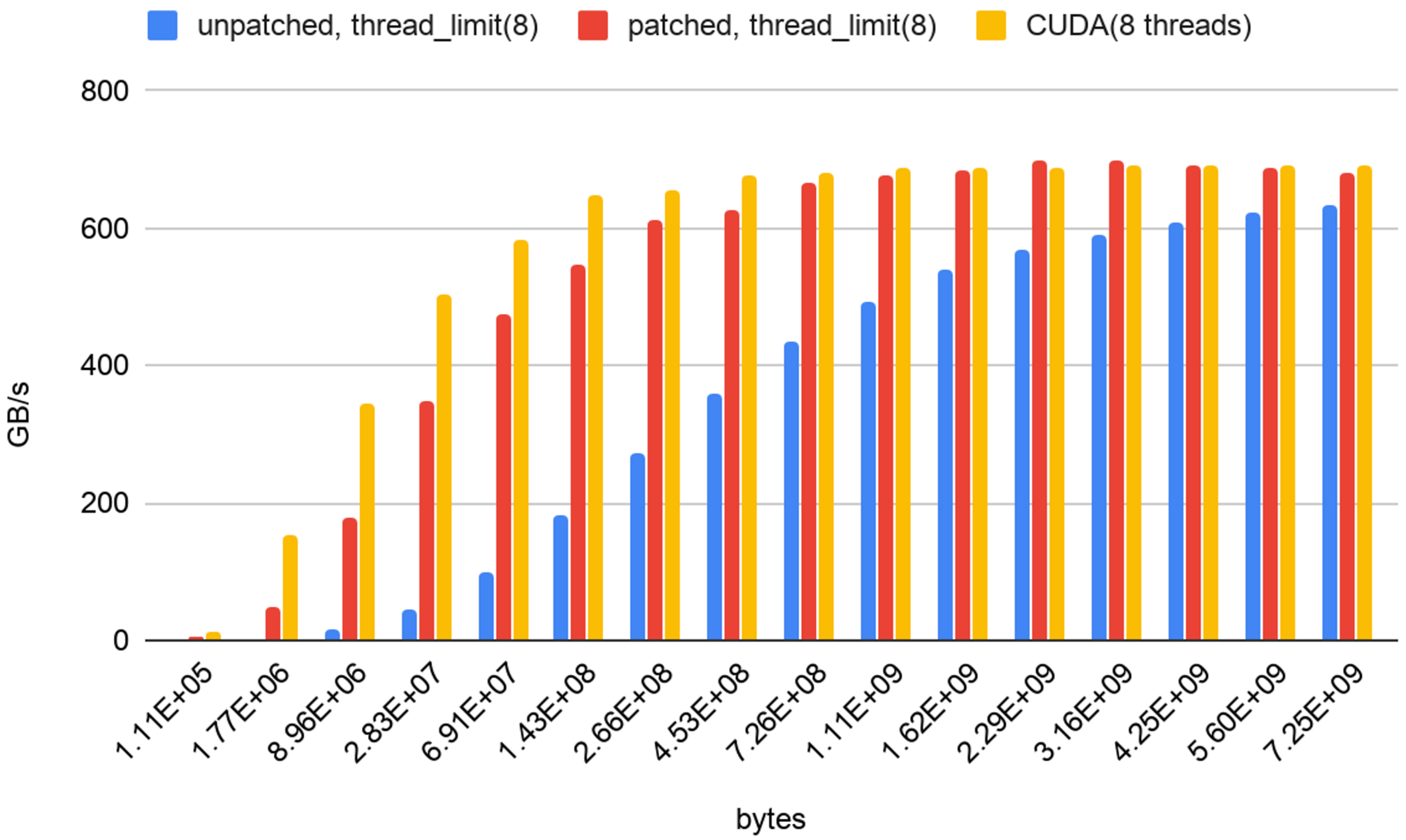
OpenMP

Culprit for poor performance at small memory footprints:
30% time spent on **cuMemAlloc/cuMemFree**

- A known issue of LLVM when many short lived objects with non-overlapping lifetimes are mapped.
- **Solution:** Keep and reuse previously freed memory instead of giving and to CUDA and asking for it again.
  See https://reviews.llvm.org/D81054
- Pool allocation is next to reduce the cost for many consecutive small allocations with overlapping lifetimes.
  See https://reviews.llvm.org/D85274

```
Grid : Message : ==============================================================================
================
Grid : Message : = Benchmarking SU3xSU3  z= x*y
Grid : Message : ==============================================================================
================
Grid : Message :   L            bytes                    GB/s              GFlop/s
Grid : Message : ----------------------------------------------------------
Grid : Message : 32             4.53e+08                 360               165
==21196== Profiling application: ./Benchmark_su3.x -o omp-thread_limit8.nvp --gpu-threads 8
==21196== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   99.25%  868.13ms      1050  826.79us  681.37us  134.76ms  __omp_offloading_34_1af2
3f09__ZN4Grid7LatticeINS_7iScalarINS1_INS_7iMatrixINS_9Grid_simdISt7complexIdENS_12Optimization3vecId
EEEELi3EEEEEEEEaSINS_9BinaryMulIRKSC_SH_EESD_SD_EERSD_RKNS_23LatticeBinaryExpressionIT_T0_T1_EE_l267
                    0.39%  3.3728ms      2101  1.6050us  1.4070us  3.1040us  [CUDA memcpy DtoH]
                    0.36%  3.1669ms      2101  1.5070us  1.2800us  2.0480us  [CUDA memcpy HtoD]
      API calls:   52.08%  893.35ms      2100  425.41us  11.353us  134.78ms  cuMemcpyDtoHAsync
                   16.21%  278.14ms      2100  132.45us  3.8050us  667.31us  cuMemAlloc
                   15.35%  263.32ms         3  87.772ms  79.660us  263.15ms  cudaMallocManaged
                   14.78%  253.49ms      2100  120.71us  3.6810us  10.057ms  cuMemFree
                    0.48%  8.3044ms      2100  3.9540us  2.7470us  19.208us  cuMemcpyHtoDAsync
                    0.47%  8.0046ms      1050  7.6230us  6.7770us  24.284us  cuLaunchKernel
                    0.17%  2.9261ms         1  2.9261ms  2.9261ms  2.9261ms  cuModuleLoadDataEx
                    0.12%  2.0855ms      9455     220ns     120ns  439.55us  cuCtxSetCurrent
                    0.12%  2.0081ms      1050  1.9120us  1.7510us  7.5600us  cuStreamSynchronize
                    0.08%  1.3799ms         1  1.3799ms  1.3799ms  1.3799ms  cuModuleUnload
                    0.05%  816.00us       100  8.1590us     114ns  327.09us  cuDeviceGetAttribute
                    0.03%  590.97us        32  18.467us  1.2510us  320.90us  cuStreamCreate
                    0.02%  377.97us      1050     359ns     297ns  1.2260us  cuFuncGetAttribute
                    0.02%  347.59us         1  347.59us  347.59us  347.59us  cuDeviceTotalMem
                    0.01%  109.15us         1  109.15us  109.15us  109.15us  cuDeviceGetName
                    0.00%  81.811us        32  2.5560us  1.5800us  12.628us  cuStreamDestroy
                    0.00%  21.686us         1  21.686us  21.686us  21.686us  cuMemcpyDtoH
                    0.00%  7.7080us         1  7.7080us  7.7080us  7.7080us  cuDeviceGetPCIBusId
                    0.00%  5.1050us         1  5.1050us  5.1050us  5.1050us  cuMemcpyHtoD
                    0.00%  3.0860us         3  1.0280us     503ns  1.6840us  cuDeviceGet
                    0.00%  2.8190us         1  2.8190us  2.8190us  2.8190us  cuInit
                    0.00%  2.2260us         2  1.1130us     606ns  1.6200us  cuModuleGetGlobal
                    0.00%  2.0120us         4     503ns     142ns     933ns  cuDeviceGetCount
                    0.00%  1.7780us         1  1.7780us  1.7780us  1.7780us  cuDriverGetVersion
                    0.00%  1.0300us         1  1.0300us  1.0300us  1.0300us  cuModuleGetFunction
                    0.00%     698ns         1     698ns     698ns     698ns  cuDevicePrimaryCtxGetSta
te
                    0.00%     555ns         1     555ns     555ns     555ns  cuDevicePrimaryCtxRetain
                    0.00%     537ns         1     537ns     537ns     537ns  cuDevicePrimaryCtxReleas
e
                    0.00%     204ns         1     204ns     204ns     204ns  cuDeviceGetUuid
                    0.00%     189ns         1     189ns     189ns     189ns  cuCtxGetDevice

==21196== Unified Memory profiling result:
Device "Tesla V100-SXM2-16GB (0)"
   Count  Avg Size  Min Size  Max Size  Total Size  Total Time  Name
    7504  34.393KB  4.0000KB  0.9922MB  252.0391MB  42.80000ms  Host To Device
Total CPU Page faults: 1296
```

Patched

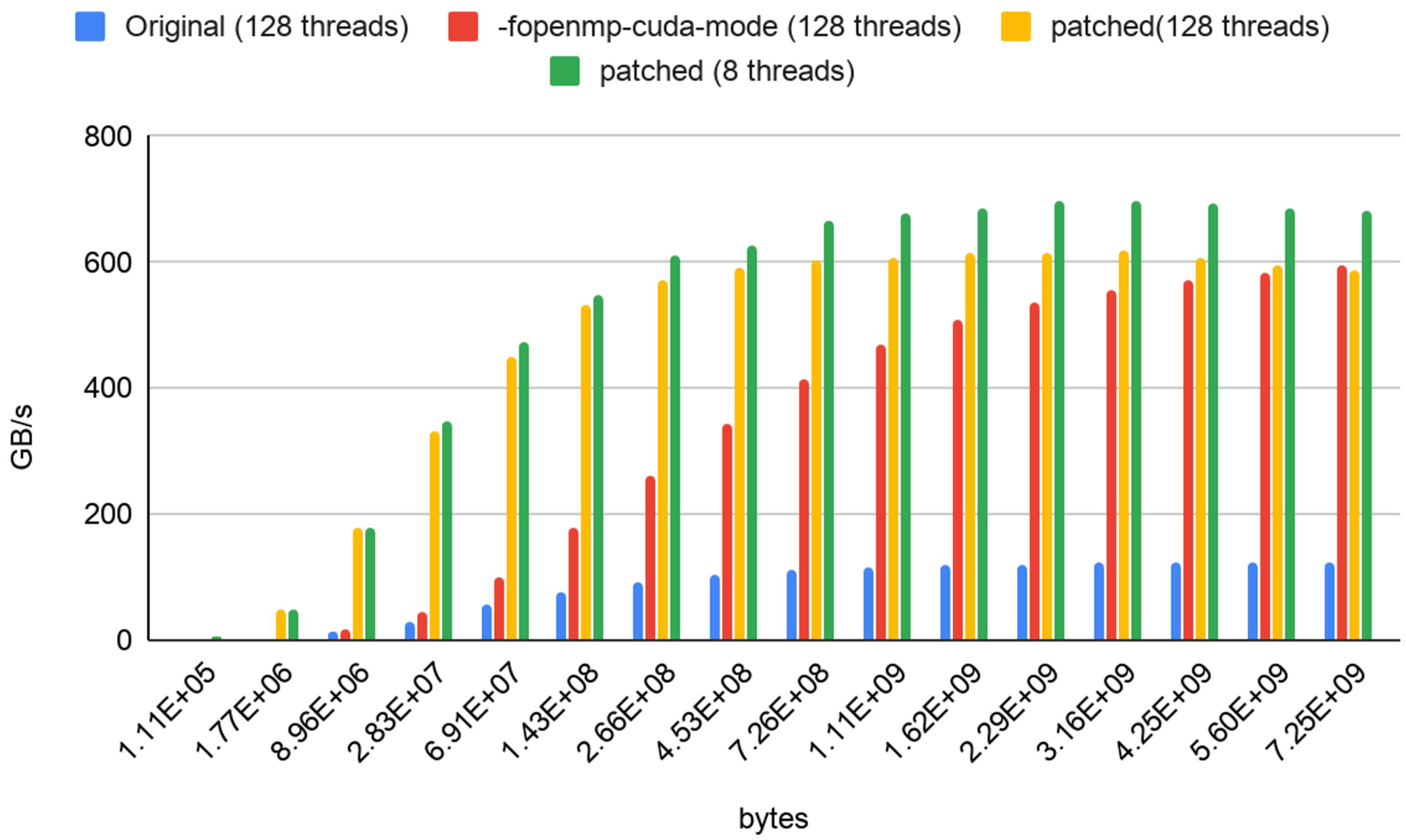


- A patch exists to optimize the memory management in LLVM.
- The patch improves the performance significantly, in particular with small- to medium-sized data footprints.
- The performance is on par with the CUDA version for the most part.
- **The patch has been merged into the LLVM mainline.**

*benchmark performed on Cori GPU (V100); patched version uses llvm/12.0.0_git_20200731-shilei; cuda/10.1.243

EXASCALE COMPUTING PROJECT

Progression of Performance (LLVM12)



- Original gets best performance with 128 threads (also compiler default).
- 8 threads per block seems to be the sweet spot with -fopenmp-cuda-mode, both patched and unpatched.
- Setting **num_teams** together with **thread_limit** doesn't seem to have much effect, since compiler generates num_teams based on # of threads and the loop count, same as the manual num_teams.

*benchmark performed on Cori GPU (V100); patched version uses llvm/12.0.0_git_20200731-shilei; others use llvm/12.0.0_git_20200731; cuda/10.1.243

EXASCALE COMPUTING PROJECT

Summary and Next Steps

- **OpenMP offloading to GPUs seems feasible in GridMini.**
- **LLVM compiler support for omp target has improved a lot!**
- **Communication with the compiler team is very important.**
  - Benefits go both ways.
- **Next Steps:**
  - Make the code truly portable: replace `cudaMallocManaged` with either explicit `target data` clauses or OpenMP's `unified_shared_memory` clause (when supported).
  - Investigate the effects of SIMD vs Scalar (SIMD length=1) data layout
    - right now using Scalar data layout.
  - Move on to the Dslash benchmark: this is our true interest.

Acknowledgments

- ML thanks Johannes Doerfert (ANL) and Rahul Gayatri (LBNL) for their help during the OpenMP Hackathon on August 3-7, 2020.
- Some of the benchmarks used the computing resources at NERSC and OLCF.

This research was supported by the Exascale Computing Project (ECP), Project Number: 17-SC-20-SC, a collaborative effort of two DOE organizations – the 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 – to support the nation's exascale computing imperative. This research used resources of the Oak Ridge Leadership Computing Facility at the Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under contract No. DE-AC05-00OR22725.