Outcomes of OpenMP Hackathon: OpenMP Application Experiences with the Offloading Model[⋆] (Part II)

Barbara Chapman[5], Buu Pham[1], Charlene Yang[2], Christopher Daley[3], Colleen Bertoni[4], Dhruva Kulkarni[3], Dossay Oryspayev[5], Ed D'Azevedo[6], Johannes Doerfert[4], Keren Zhou[7], Kiran Ravikumar[8], Mark Gordon[1], Mauro Del Ben[3], Meifeng Lin[5], Melisa Alkan[1], Michael Kruse[4], Oscar Hernandez[6], P.K. Yeung[8], Paul Lin[3], Peng Xu[1], Swaroop Pophale[6], Tosaporn Sattasathuchana[1], Vivek Kale[5], William Huhn[4], and Yun (Helen) He[3]

[1] Iowa State University, Ames, IA
{buupq,alkan,pxu,tsatta,mgordon}@iastate.edu
[2] NVIDIA Corporation, Santa Clara, CA
charleney@nvidia.com
[3] Lawrence Berkeley National Laboratory, Berkeley, CA
{csdaley,dkulkarni,mdelben,paullin,yhe}@lbl.gov
[4] Argonne National Laboratory, Lemont, IL
{jdoerfert,mkruse,whuhn,bertoni}@anl.gov
[5] Brookhaven National Laboratory, Upton, NY
{doryspaye,mlin, vkale}@bnl.gov
[6] Oak Ridge National Laboratory, Oak Ridge, TN
{dazevedoef,oscar,pophaless}@ornl.gov
[7] Rice University, Houston, TX
keren.zhou@rice.edu
[8] Georgia Institute of Technology, Atlanta, GA
kiran.r@gatech.edu, pk.yeung@ae.gatech.edu

Abstract. This paper reports on experiences gained and practices adopted when using the latest features of OpenMP to port a variety of HPC applications and mini-apps based on different computational motifs (BerkeleyGW, WDMApp/XGC, GAMESS, GESTS, and GridMini) to accelerator-based, leadership-class, high-performance supercomputer systems at the Department of Energy. As recent enhancements to OpenMP become available in implementations, there is a need to share the results of experimentation with them in order to better understand their behavior in practice, to identify pitfalls, and to learn how they can be effectively deployed in scientific codes. Additionally, we identify best practices from these experiences that we can share with the rest of the OpenMP community.

Keywords: OpenMP · Device Offload · Application Experiences.

[⋆] Supported by Exascale Computing Project (ECP) OpenMP Hackathon hosted by SOLLVE and NERSC [29].

Keywords: OpenMP · Device Offload · Application Experiences.

1 Introduction

In this paper we continue the exploration of OpenMP usage in HPC applications (GAMESS, GESTS, and GridMini) in Section 2. We conclude in Section 3 and provide acknowledgments in Section 4.

2 Application Experiences

2.1 GAMESS

2.1.1 Application Overview GAMESS is a general electronic structure software package comprising of a variety of quantum mechanical (QM) methods [6]. GAMESS is primarily written in Fortran and parallelized using both pure MPI [14, 12] and hybrid MPI/OpenMP [23, 24]. A high-performance C++/CUDA library, namely LibCChem, has been recently developed to accelerate GAMESS on GPUs. Alternatively, GAMESS Fortran is directly offloaded to GPUs using OpenMP. In this section, offloading strategies of the Hartree-Fock (HF) method, which is an essential step of *ab initio* methods, will be discussed.

2.1.2 Application Motif A HF computation requires i) evaluation of N^4 electron repulsion integrals (ERIs) as shown in eq. (1), ii) formation of N^2-element Fock matrix, iii) Fock matrix diagonalization for eigen energies and vectors. Here, N is the system size, usually represented by the number of Gaussian basis functions $\phi_\mu(r)$. In GAMESS, a basis function can be characterized by a few parameters, including the so-called angular momentum, which is an integer starting from 0 upwards, used here for sorting ERIs.

$$(\mu\nu|\lambda\sigma) = \iint dr_1\, dr_2 \phi_\mu^*(r_1)\phi_\nu(r_1){r_{12}}^{-1}\phi_\lambda^*(r_2)\phi_\sigma(r_2), \tag{1}$$

To provide the optimal performance for ERI evaluation, three different integral algorithms have been implemented in GAMESS, including a) Rotated-axis [25, 28, 21, 1], b) Electron repulsion Integral Calculator (ERIC) [13], and c) Rys quadrature [19, 11, 27]. Depending on the characteristics of basis functions, different ERI algorithms are selected at runtime (Figure 1a).

In this work, we focus on i) offloading the Rys quadrature kernel from full GAMESS package, ii) examining OpenMP offloading compiler support using a Fortran mini-app of rotated-axis integral code, and iii) analyzing performance of a C++ integral kernel from GAMESS-LibCChem. The first two efforts were carried out on Ascent/Summit, while the latter was performed on NERSC's CoriGPU.

2.1.3 OpenMP parallelization strategy In the OpenMP HF implementation targeting CPUs [2, 22], there is a large workload imbalance between threads. This imbalance is handled by using dynamic loop scheduling and loop collapsing (c.f. lines 2 and 3, Figure 1a). To adapt this code to the SIMT architecture in GPUs, the ERI codes were restructured based on their optimal algorithms as discussed in [4]. However, this code can be improved by sorting the integrals with respect to basis function angular momentum and the inherent permutational symmetry of the integrals into different classes (e.g. R 1112, R 1121 in Figure 1c). In this study, a particular class, the 1121-kernel is presented in Figure 1d and for further optimization see Figure 1e. The main bottleneck of the 1121-Rys kernel is to contract calculated ERIs (line 6 of Figure 1d) with density for six Fock matrix elements using atomic operations (lines $12-14$). To reduce such synchronization overhead, in Figure 1e, ERIs are evaluated in chunk (lines $4-7$), which are then contracted with density to update the Fock matrix in a separated GPU parallel region (lines $10-17$). Data exchange between quartet evaluation and Fock update are managed by the target data directive (lines 1 and 18).

2.1.4 Results Relative timing of the new (Figure 1e) and the original (Figure 1d) offloading schemes for the 1121-kernel is studied using water cluster of $16-128$ molecules. The cc-pVDZ basis set is used for all water clusters introducing $54-570$K quartets for computation and Fock digestion. The 1121-kernel wall time of $(H_2O)_{64}$ is recorded while varying the number of quartets computed concurrently, which is NSIZE$*80*128$, the number of teams (NTEAMS) and threads per team (NTHREAD). The optimal wall time is usually achieved with a small number of threads, large number of team and medium value of chunk size (Fig. 2).

Table 1. Wall time (s) and speedup of new offloading implementation (Figure 1e). relative to the original one (Figure 1d)

	NQUART	Scheme 1d	Scheme 1e	Speedup
$(H_2O)_{16}$	$54,274$	0.30	0.41	0.73
$(H_2O)_{32}$	$263,075$	0.73	0.51	1.43
$(H_2O)_{64}$	$1,256,691$	2.94	1.00	2.94
$(H_2O)_{80}$	$2,095,639$	4.82	1.75	2.75
$(H_2O)_{96}$	$3,118,724$	6.97	2.56	2.72
$(H_2O)_{112}$	$4,322,306$	9.49	3.69	2.57
$(H_2O)_{128}$	$5,727,489$	12.40	4.80	2.58

Stacking optimal series, i.e., those contain minimum wall time data point, showing that the 1121-kernel can be evaluated in 1.00 (s) with medium chunk size NSIZE 80, NTEAMS 160, and NTHREAD 8. In comparison with the a)

```
do ish=1,nshell
!$omp parallel do
 schedule(dynamic) collapse(2)
 do jsh=1,ish
 do ksh=1,ish
  do lsh=1,ksh
!choose ERI package
    if(do_sp) call sp
    if(do_spd) call spd
    if(do_eric) call eric
    if(do_rys) call rys
  enddo
 enddo
 enddo
!$omp end parallel do
enddo
```

b)

```
!$omp target teams distribute
 parallel do
do isp=1,n_sp
    call sp(isp)
enddo
!$omp target teams distribute
 parallel do
do ispd=1,n_spd
    call spd(ispd)
enddo
!$omp target teams distribute
 parallel do
do ieric=1,n_eric
    call eric(ieric)
enddo
!$omp target teams distribute
 parallel do
do irys=1,n_rys
    call rys(irys)
enddo
```

c)

```
! computing each ERI class
 using Rys
!$omp target teams distribute
 parallel do
do i1112=1,n_1112
    call r_1112(i1112)
enddo
!$omp target teams distribute
 parallel do
do i1121=1,n_1121
    call r_1121(i1121)
enddo
!$omp target teams distribute
 parallel do
do i1122=1,n_1122
    call r_1122(i1122)
enddo
 ...
```

d)

```
!$omp target teams distribute
 &
!$omp parallel do
do i=1,n_1121
! evaluating quartet i-th
call r_1121_int(quart(i))
! fock matrix contraction
do k=1,6
!$omp atomic
fock(k)=fock(k) &
          + quart(i)*den(k)
enddo
enddo
```

e)

```
!$omp target data map(alloc:
 quart (ichunkstart:ichunkend))
do ichunk = 1, nchunk_1121
! evaluating in chunk
!$omp target teams distribute
 parallel do
do i=ichunkstart,ichunkend
 call r_1121_int_mod(
  quart(ichunkstart:ichunkend))
enddo
!Fock matrix contraction
!$omp target teams distribute
 parallel do
 do i=ichunkstart,ichunkend
  do k=1,6
!$omp atomic
fock(k)=fock(k) &
               +quart(i)*den(k)
  enddo
 enddo
enddo
!$omp end target data
```

f)

```
subroutine eri_mini
do ish=1,nshell
 do jsh=1,ish
 do ksh=1,ish
  do lsh=1,ksh
    call shellquart
    call fock_mat
  enddo
 enddo
 enddo
enddo
end subroutine eri_mini
subroutine shellquart
    call setcrd(rmat,p,t,q)
    call r12loop(r12,r34)
    call fmt(fgrid,xgrid)
    call jtype(gmat,rmat,p,t,q,
     r12,r34 fgrid,xgrid)
end subroutine shellquart
```

Fig. 1. (a) CPU OpenMP implementation , (b) refactoring ERI codes based on optimal algorithm, (c) further sorting for the Rys quadrature algorithm, (d) detail of 1121-Rys kernel; and (e) separation of ERI evaluation and digestion, f) Fortran mini-app ERI code and its SHELLQUART kernel.

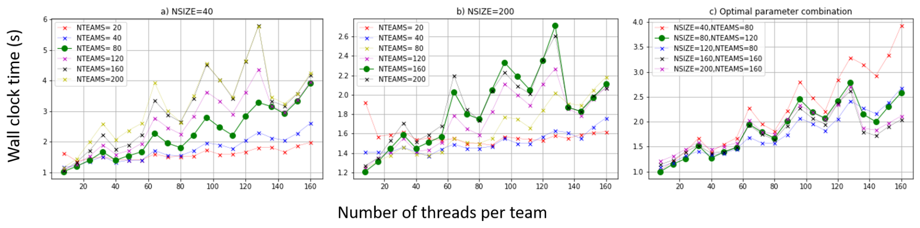

Fig. 2. a,b) variation of the 1121-kernel wall time with respect to NTEAMS, NTHREAD and NSIZE. Optimal series for each chunk size is in bold green; c) stacking of optimal series extracted from various chunk sizes.

original algorithm, separating atomic updates introduce a speedup of $2.5x$ (Table 1).

2.1.5 Fortran mini-app A Fortran mini-app was extracted from the rotated-axis algorithm (Figure 1f) to be portable to a variety of computer clusters and explore GPU compilers from different vendors. This Fortran mini-app has been compiled on various hardware and compilers (Table 2). The explicit interface is not required for the offloading region for IBM compiler, while it was needed for NVIDIA's HPC SDK 21.2 compiler. However this issue was resolved with HPC SDK 21.3. Despite the fact that the ERI code was well modularized and worked well on CPU, using compilers for offloading to GPU with the automatic inlining option (e.g., using `-qinline`) did not show noticeable performance improvements. On the other hand, manually inlining implementation greatly improves the performance. A noteworthy observation was made that a runtime out-of-memory on the device was encountered, which was resolved by moving some subroutine arguments to modules (e.g., STCRD, R12LOOP, FMT, and JTYPE).

Table 2. Summary of various compilers used for the mini-app.

Compiler	Systems	Declare target for external subroutine	Compiler flags
IBM	Ascent	No clause	`-qsmp=omp -qoffload -O2`
NVIDIA HPC SDK 21.2	CoriGPU	Explicit interface	`-Mextend -tp=skylake -Mcuda=cc70 -ta=tesla:cc70 -fast -mp=gpu`

2.1.6 C++-mini-app A C++ mini-app was extracted from LibCChem and ported to GPU using CUDA and OpenMP. The kernel was wrapped into a Google Benchmark [15] application that is configured using CMake [20]. Google Benchmark can adjust the number of iterations depending on single kernel execution time and measurement noise to output a reliable result. CMake and preprocessor provide flags handling of the compiler required for C++, CUDA, or OpenMP in a single source directory with multiple build directories, one for each compiler.

Table 3. Mini-Mini-App performance results in seconds.

Language	Compiler	Variant	CPU Kernel	GPU Kernel
CUDA	nvcc		2003.0	43.0
CUDA	nvcc	localmem	1934.0	50.8
OpenMP	clang		2657.0	54.3
OpenMP	CCE		2023.0	75.7
OpenMP	gcc		5885.0	2054.9
OpenMP	nvc		*error*	

For the Cori GPU system, Table 3 shows the Google Benchmark CPU time per iteration (including waiting for results to arrive in the CPU) and the GPU kernel-only time as measured by NVIDIA's nvprof. The CUDA version has a "localmem" variant which uses temporary `__shared__` arrays containing copies of the working set of a single block, but otherwise calls the same (inlined) kernel. The OpenMP source does not have an equivalent to the localmem variant because required data-initialization of team-local memory in OpenMP are incompatible with clang's SPMD-mode and would cause a major performance penalty. For OpenMP, Clang performed the best, slightly behind the CUDA version, followed by CCE. The execution time when compiled with gcc was not competitive. NVIDIA's HPC SDK 21.3 compiler (formerly PGI) either failed with a compiler error or produced a crashing executable (`-O0` or `-O1`).

2.1.7 Challenges and lessons learned To adapt the GPU SIMT model, the Hartree-Fock code was restructured so that integrals of the same class are computed concurrently. The bottleneck was found to be atomic updates in the Fock matrix contraction (Figure 1d), which were further optimized by using them in a separate target region(Figure 1e), in which data exchange between parallel regions are retained on GPU and governed by the target data directive. The kernel performance was also found to vary strongly with respect to the number of teams, threads per team and amount of data loaded to GPU for computation. The results show that utilizing NTEAMS=160 and NTHREAD=8 to be processed at a time yields desirable performance. The rotated-axis mini-app was offloaded with the basic target constructs and tested on various systems, and shown that "out-of-memory" runtime error can be resolved by using modules. The C++ mini-app was ported to GPU using CUDA and OpenMP, with the kernel wrapped into a Google Benchmark application configured using CMake. It was found that `gcc`-compiled OpenMP kernel did not show competitive timing. NVIDIA's HPC SDK 21.3 compiler either failed with a compiler error or produced a crashing executable.

2.2 GESTS

2.2.1 Application Overview GESTS (GPUs for Extreme Scale Turbulence Simulations) is a pseudo-spectral Direct Numerical Simulation (DNS) code used to study the fundamental behavior of turbulent flows [17]. The presence of disorderly fluctuations over a wide range of scales in three-dimensional (3D) space and time poses stringent resolution requirements [30], especially if localized events of high intensity [31] must be captured accurately. However, large scale pseudo-spectral simulations are dominated by communication costs, which become an even greater burden versus computation when codes are ported to heterogeneous platforms whose principal strengths are in computational speed.

In recent work on Summit, we addressed these challenges by developing a batched asynchronous algorithm [26] to enable overlapping computations, data copies and network communication using CUDA, which enabled problem sizes as large as 6 trillion grid points.

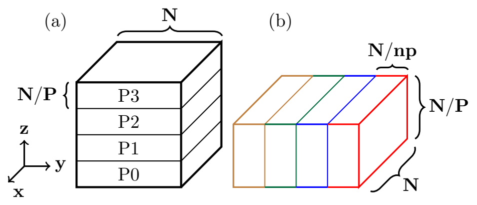

Fig. 3. Left/(a): Decomposition of an N^3 solution domain among P MPI processes into *slabs* of data of size $N \times N \times N/P$. Right/(b): Further decomposition of a slab into np smaller sub-volumes, each of size $N \times N/np \times N/P$.

Here we discuss the development of a portable implementation using advanced asynchronous OpenMP features on the GPUs, in order to enable even larger problem sizes using newer exascale architectures like Frontier.

2.2.2 Application Motif We focus on three-dimensional Fast Fourier Transforms (3D FFT) which are crucial to the GEST code. To benefit from the architecture of emerging platforms with large CPU memory and multiple accelerators we use a one-dimensional (*slabs*) domain decomposition as shown in Figure 3a. This helps reduce communication costs as fewer MPI processes are involved [26] although point-to-point message sizes are larger. Within each plane in a slab, 1D FFTs in two directions (here x and y) are performed readily using highly optimized GPU libraries (cuFFT or rocFFT), while the FFT in the third (z) direction requires an all-to-all global transpose that re-partitions the data into, say, $x-z$ planes. However if N is very large (up to 18,432 in [26]) a complete slab may not fit into the smaller GPU memory. We address this by dividing each slab into np smaller *sub-volumes*, as in Figure 3b. In effect, batches of data formed from the sub-volumes are copied to the GPU, computed on, and copied back; while operations on different portions may overlap with one another. For example, as the code proceeds from left to right, GPU computation on a sub-volume colored in blue, host-to-device copy for another in red (or device-to-host in green) and non-blocking all-to-all on another in brown can occur asynchronously.

2.2.3 OpenMP implementation strategy In the "batched" scheme described above, when FFTs in y need to be computed, a sub-volume of data consisting of $N \times N/P$ lines of size N/np need to be copied. Essentially, for each value of y (between 1 to N) and z (between 1 to N/P) a copy needs to be performed for N/np elements in the innermost dimension (x) of length N. Efficient strided data transfers between the CPU and GPU are thus important. Simple approaches such as packing on the host prior to transfer, or performing multiple copies one line at a time are inefficient, because of an extra data-reordering operation on the CPU and the overhead of numerous smaller copies respectively [26]. Instead, we make use of two different approaches depending on the complexity of the strided memory accesses.

For simple strided copies where strides are small and in the innermost dimension only, we can use `omp_target_memcpy_rect` which copies a specified subvolume inside a larger array on the host to a smaller buffer on the device or vice versa. This OpenMP 4.5 routine is similar to `cudaMemcpy2d` but asynchronous execution will be supported only in OpenMP 5.1. We are using OpenMP tasks as a workaround.

(a) `zero-copy`

```
1  TARGET ENTER DATA MAP(to:d_buf)
    DEPEND(IN:indep) DEPEND(OUT:tdep)
    NOWAIT
2
3  TARGET TEAMS DISTRIBUTE
    PARALLEL DO COLLAPSE(4)
    DEPEND(INOUT:tdep) NOWAIT
    IS_DEVICE_PTR(h_buf)
4  do yg=1,nt
5     do z=1,mz
6        do y1=1,my
7           do x=1,N
8              y = my*(yg-1)+y1
9              d_buf(x,y,z) = h_buf(x,z,y1,yg)
10          end do
11       end do
12    end do
13 end do
14 END TARGET TEAMS DISTRIBUTE
    PARALLEL DO
15
16 TARGET EXIT DATA MAP(from:d_buf)
    DEPEND(IN:tdep)
    DEPEND(OUT:outdep) NOWAIT
```

(b) `detach`

```
1  TASK DEPEND(out:var) DETACH(event)      }
2     TARGET DATA USE_DEVICE_PTR(a)        }
3        FFTExecute (a, forward, stream)   }
4        FFTExecute (a, inverse, stream)   } Ⓐ
5     END TARGET DATA                      }
6     cudaStreamAddCallback (stream,       }
         ptr_cp, C_LOC(event), 0)          }
7  END TASK                                }
8
9  TARGET UPDATE TO(b)                     } Ⓑ
    DEPEND(inout:b) NOWAIT                 }
10
11 TARGET TEAMS DISTRIBUTE                 }
    DEPEND(IN:var) NOWAIT                  } Ⓒ
12    a(:, :, :) = a(:, :, :)/N            }
13 END TARGET TEAMS DISTRIBUTE             }
```

```
1  subroutine callback (stream, status, event)
2     call omp_fulfill_event(f_event)
3  end subroutine callback
```

Fig. 4. (a) Asynchronous OpenMP implementation of the `zero-copy` kernel for unpacking data from the pinned host array (`h_buf`) to the device array (`d_buf`). Here for an N^3 problem $my = mz = N/nt$ where nt is MPI process count. (b) Interoperability between non-blocking FFT libraries and OpenMP *tasks* using `DETACH` while ensuring correct asynchronous execution.

For more complex stride patterns, like those in unpacking operations where strided read and write memory access are required to transpose data in the second and third dimension as shown in line 9 of Figure 4a, a `zero-copy` kernel [3] is appealing. In this approach, GPU threads are used to initiate many small transfers between pinned memory on the host and the device memory. The array on the host is made device accessible using the `IS_DEVICE_PTR` clause. However, since using GPU compute resources for data copies may slow down other computations, we use the `zero-copy` approach only when complex stride patterns are involved.

In OpenMP, asynchronous execution can be achieved using the `TASK` clause for work on the host, `NOWAIT` for device kernels and data copies, and `DEPEND` to enforce the necessary synchronization between different *tasks*. However, when non-blocking libraries such as cuFFT or rocFFT are called from inside an OpenMP task, the desired asynchronism breaks down. Figure 4b illustrates the issue via a

1D FFT code fragment, in which Task A calls the non-blocking libraries to compute the transforms, Task B performs a host-to-device data copy, whereas Task C multiplies the result by a scalar. It is important to note here that the FFT library function is called from the host but device arrays need to be passed in to it. This is achieved using the `USE_DEVICE_PTR` clause which tells the OpenMP runtime to pass the device pointer of the array, to the library call. In OpenMP 5.1 usage of the `USE_DEVICE_PTR` with a Fortran pointer is depreciated, but the `USE_DEVICE_ADDR` clause can be used equivalently instead. Without the highlighted gray lines, the host thread that is executing task A will launch the FFT kernels to the GPU. Since these library calls are non-blocking, control will return immediately to the host thread, which proceeds to end the task. As a result, the device kernels launched may not have completed, or even started running, when Task A is considered "complete". The subsequent release of dependency between Tasks A and C allows the latter to start prematurely, leading to incorrect answers.

Correct execution can be ensured using the OpenMP 5.0 `DETACH` clause, with `cudaStreamAddCallback`, as shown in Figure 4b. Now as the host thread launches the FFTs, it introduces a callback function (where `ptr_cb` is a pointer to it) into the stream in which the FFTs are executing. The host thread then detaches itself from task A to proceed with other operations. Once the FFTs finish executing on the device, the callback function is invoked which "fulfills" the `event` and completes task A, releasing the dependency and thus allowing Task C to execute correctly with the intended data. We also understand that OpenMP `interop` clause introduced in the 5.1 standard can help overcome this issue as well. However, so far we have chosen to use `DETACH` as it is part of the 5.0 standard and is expected to be supported by the compilers earlier.

2.2.4 Summary and future work We have briefly addressed some key challenges encountered in developing a portable implementation of extreme scale 3D FFTs using OpenMP to target GPUs. Efficient strided data copies are performed using `Zero-copy` kernels and `omp_target_memcpy_rect`. Although full compiler support for it is not yet available, the OpenMP 5.0 feature `DETACH` is expected to resolve an issue of interoperability between non-blocking GPU library calls and OpenMP tasks. Future work will include testing the `DETACH` approach and using it to develop a batched asynchronous 3D FFT code (and eventually pseudo-spectral simulation of turbulence) capable of problem sizes beyond that recently achieved on Summit. Timing data over a range of problem sizes will be reported separately when available.

2.3 GridMini

2.3.1 Application Overview Lattice Quantum Chromodynamics (LQCD) [16] is a computational framework that allows scientists to simulate strong interactions between the subatomic particles called quarks and gluons. LQCD provide crucial theoretical input to nuclear and high energy physics research, but its high computational demand limits the precision of the numerical results it can obtain. LQCD software has been written and optimized for many different computer architectures, including many/multi-core CPUs [9] and NVIDIA GPUs [10], to access as many computing resources as possible. Recently there has also been significant effort getting some of the major LQCD code bases to run on Intel and AMD GPUs. Portable programming models and frameworks such as Kokkos, HIP and SyCL have been investigated [18, 7], and implementations for production-grade software are under way. Here we evaluate use of OpenMP as a portable programming model for Grid [8], a new lattice QCD library written in modern C++. Since Grid is a fairly large library, with multi-level abstractions, we use a mini-app based on Grid, `GridMini` [9], to evaluate several OpenMP features that are needed to support LQCD computing, including the `target` directive and associated data management clauses.

2.3.2 Application Motif LQCD is a cartesian-grid based application, with a four-dimensional hypercubic mesh representing space and time. Each grid point represents a quark field variable, while the links between grid points approximate the gluon field variables. The main computational algorithm in lattice QCD is Markov Chain Monte Carlo simulations that are used to generate ensembles of background gluon fields. These gluon field ensembles are then used to perform measurement calculations which then lead to physical results. In both the Monte Carlo ensemble generation and measurement calculations, high-dimensional complex sparse matrix inversions are needed, which are usually done through iterative linear solvers such as conjugate gradient (CG). In CG, the key computational kernel is high-dimensional matrix-vector multiplication, the so-called Dslash operator in LQCD. There are several variants of the discrete Dslash operator depending on the discretization schemes used, but in modern lattice QCD simulations, all of them are very large sparse matrices, on or larger than the order of $10^{10} \times 10^{10}$. The arithmetic intensity for the Dslash operator is about 1.7 flops per byte in double precision. Since we use red-black preconditioning for CG, the arithmetic intensity is even lower, reduced to 0.85 flops per byte. Therefore LQCD computation is highly memory bandwidth bound, and the on-node performance of LQCD code depends on achieving as much memory bandwidth as possible on the given architecture. Grid has been highly optimized for many-core and multi-core CPUs with efficient SIMD vectorization, so our work focuses on performance and portability on the GPUs.

2.3.3 OpenMP parallelization strategy Grid and GridMini support different architectures at the low level through C++ preprocessor macros, which may invoke different implementations. Since LQCD parallelization is mostly done to the `for` loops that iterate through lattice sites, an `accelerator_for` macro is defined, along with function attribute macros that may expand to different architecture-dependent definitions. Different implementations are enabled

[9] https://github.com/meifeng/GridMini

through macros passed through the compiler flag `-D`. The OpenMP paralleization for CPUs uses the standard `omp parallel for` directive, while for accelerator offloading, `omp target` directives are used. A relevant code snippet is shown in Listing 1.1.

```
#define naked_for(i,num,...) for ( uint64_t i=0;i<num;i++) { __VA_ARGS__ } ;
#define accelerator_inline __attribute__((always_inline)) inline
#ifdef OMPTARGET
#define accelerator_for(iterator,num,nsimd, ... )  \
{ _Pragma("omp target teams distribute parallel for num_teams(nteams) thread_limit(gpu_threads)") \
        naked_for(iterator, num, { __VA_ARGS__ }); }
#elif defined (GRID_OMP)
#define accelerator_for(iterator,num,nsimd, ... )   _Pragma("omp parallel for") naked_for(iterator, num, { __VA_ARGS__ });
#endif

//other code omitted
accelerator_for(ss,me.size(),1,{
                me[ss] = eval(ss,expr);
               })
```

Listing 1.1. C++ macros that define the loop-level computation in GridMini.

A more tricky issue is the memory management, as Grid uses deeply nested data structures. In the CUDA implementation, `cudaMallocManaged` is used as the default dynamic memory allocator, so it is unnecessary to perform manual data management. Previously [5], we successfully used `cudaMallocManaged` together with OpenMP target offloading. But since it is CUDA specific, the code cannot run on other GPU architectures. Recently we have successfully replaced `cudaMallocManaged` with manual data management through OpenMP `map` clauses, but in order to do that, we have to explicitly expose the raw data pointer. An example of this is shown in Listing 1.2.

```
auto xv=x.View(); auto yv=y.View(); auto zv=z.View(); //x,y,z are arrays of SU(3) matrices
#pragma omp declare mapper(decltype(xv) x) map(x._odata[0:x.size()]) map(x)
extern uint32_t gpu_threads;
#pragma omp target enter data map(alloc:zv) map(to:xv) map(to:yv)
#pragma omp target teams distribute parallel for thread_limit(gpu_threads)
for(int64_t s=0;s<vol;s++) {
    zv[s]=xv[s]*yv[s];
}
#pragma omp target exit data map (from:zv) map (delete:yv) map (delete:xv)
```

Listing 1.2. Manual data mapping in GridMini.

2.3.4 Results We use the SU(3)×SU(3) benchmark (main computation is shown in Listing 1.2) to evaluate the GPU memory bandwidth, as this is highly indicative of the performance we can achieve on the GPUs since our application is memory-bandwidth bound. We compiled our code with LLVM/Clang++ built from the main LLVM repository on 01/17/2021, with the compiling options `std=c++14-g-fopenmp-fopenmp-cuda-mode-O3-fopenmp-targets=nvptx64-nvidia-cuda`. We used `gcc/8.3.0` and `cuda/11.0.3`. The results for achieved GPU memory bandwidth of the NVIDIA V100 GPU on the NERSC's CoriGPU system as a function of the memory footprint are shown in Figure 5, where we compare four different implementations. `llvm map` refers to the implementation with manual `map` clauses and `malloc` memory allocator. `llvm managed` uses `cudaMallocManaged` without any manual data mapping. `llvm map+managed` allocates memory with `cudaMallocManaged`, but also uses `map` to do data copying.

`nvcc managed` is the reference CUDA implementation with `cudaMallocManaged`, compiled with the `nvcc` compiler.

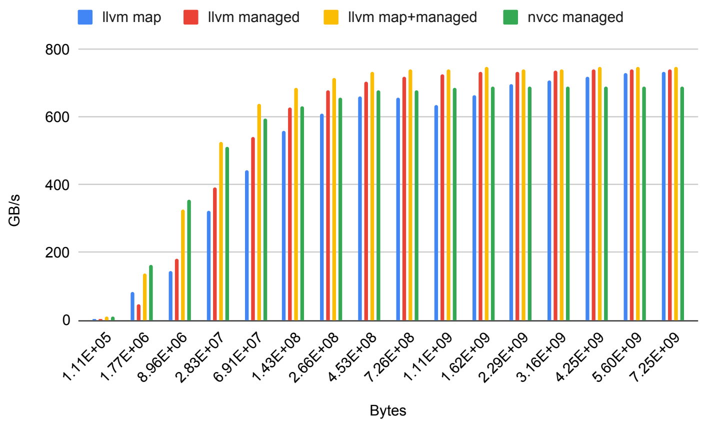

Fig. 5. Measured GPU memory bandwidth on NVIDIA V100 (Cori) with the SU(3)×SU(3) benchmark.

We find that the `llvm map` implementation generally performs a little worse than the `llvm managed` version. But if we combine the managed memory allocator with manual data mapping, in the `llvm map+managed` version, we obtain the best performance, which even outperforms the native CUDA implementation. In all these tests, 8 GPU threads per block is used, which gives the best performance compared to 16, 32 and up to 512 threads per block. This is probably due to the fact that the current data layout does not guarantee data coalescing.

3 Conclusions

The five applications presented in the paper(s) have different complexity and computational motifs. As seen in the BerkeleyGW application, some optimizations required line-level profiling information. It is therefore desirable for all OpenMP compilers to provide accurate symbolic debugging information without impeding compiler optimizations. For the WDMApp/XGC application, the main challenge was to tune a multi-level loop nest using OpenMP target offload constructs. Finding concurrency in applications and exploiting it using fine-grained parallelism is important for achieving good performance. With different vendor implementations, it becomes necessary for applications to be aware of equivalent OpenMP directives that may not be equally performant. For example, a parallel for construct was converted into a loop construct because this provided better performance with the NVIDIA OpenMP compiler.

The lessons learnt by the GAMESS team during the hackathon were the need to reduce overhead in atomic operations using chunks, strategies for target data and offloading blocks of code, and selecting the optimal number of threads per team. The GESTS application discussed the some of the current challenges they are facing regarding the portable implementation of the extreme-scale 3D FFTs, a Fortran code, using OpenMP, and reported on efficient strided data copies. The detach clause is used to address the problem of synchronizing an OpenMP kernel that uses the depend clause with a prior asynchronous CUDA call. The GridMini application team has reported on their SU(3)$\times$SU(3) benchmark in order to evaluate the GPU memory bandwidth, since the application is memory-bandwidth bound. They found that cudaMallocManaged allocators can be replaced with OpenMP unstructured maps for local host storage and that the use of cudaMallocManaged with OpenMP gave the best performance.

The more successful application teams had mini-apps to experiment with before porting the actual application. One major advantage of this approach is isolation of experimental changes for easy debugging and reproducibility. Having compiler experts at hand to help with ports is beneficial to applications, especially while resolving issues that appear in full-scale application runs but are not reproducible in mini-apps. Most applications also reported issues between OpenMP and vendor math libraries. It would be beneficial for applications if there were prepackaged compatible math libraries with all OpenMP compilers. Most of the applications were successfully able to use the OpenMP offload API as well as see speedup, which is very encouraging for OpenMP adoption by applications.

4 Acknowledgement

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.

This research used resources of the Oak Ridge Leadership Computing Facility, which is a DOE Office of Science User Facility supported under Contract DE-AC05-00OR22725.

This research used resources of the National Energy Research Scientific Computing Center (NERSC), a U.S. Department of Energy Office of Science User Facility located at Lawrence Berkeley National Laboratory, operated under Contract No. DE-AC02-05CH11231.

This work was supported by the Argonne Leadership Computing Facility, which is a DOE Office of Science User Facility supported under Contract DE-AC02-06CH11357.

We'd like to thank all other mentors who volunteered their time and expertise during the hackathon.

References

1. A new algorithm of two-electron repulsion integral calculations: a combination of Pople–Hehre and McMurchie–Davidson methods (2008). https://doi.org/https://doi.org/10.1007/s00214-007-0295-5
2. Alexeev, Y., Kendall, R.A., Gordon, M.S.: The distributed data scf. Computer Physics Communications **143**(1), 69–82 (2002)
3. Appelhans, D.: Tricks, Tips, and Timings: The Data Movement Strategies You Need to Know. In: GPU Technology Conference (2018)
4. Bak, S., Bertoni, C., Boehm, S., Budiardja, R., Chapman, B., Doerfert, J., Earl, C., Eisenbach, M., Elwasif, W., Finkel, H., Hernandez, O., Huber, J., Iwasaki, S., Kale, V., Kent, P.R.C., Kwack, J., Lin, M., Luszczek, P., Luo, Y., Pham, B., Pophale, S., Ravikumar, K., Sarkar, V., Scogland, T., Tian, S.: Openmp application experiences: Porting to accelerated nodes. summitted
5. Bak, S., et al.: Openmp application experiences: Porting to accelerated nodes. submitted to Parallel Computing (2020)
6. Barca, G.M.J., Bertoni, C., Carrington, L., Datta, D., De Silva, N., Deustua, J.E., Fedorov, D.G., Gour, J.R., Gunina, A.O., Guidez, E., Harville, T., Irle, S., Ivanic, J., Kowalski, K., Leang, S.S., Li, H., Li, W., Lutz, J.J., Magoulas, I., Mato, J., Mironov, V., Nakata, H., Pham, B.Q., Piecuch, P., Poole, D., Pruitt, S.R., Rendell, A.P., Roskop, L.B., Ruedenberg, K., Sattasathuchana, T., Schmidt, M.W., Shen, J., Slipchenko, L., Sosonkina, M., Sundriyal, V., Tiwari, A., Galvez Vallejo, J.L., Westheimer, B., Włoch, M., Xu, P., Zahariev, F., Gordon, M.S.: Recent developments in the general atomic and molecular electronic structure system. The Journal of Chemical Physics **152**(15), 154102 (2020)
7. Bi, Y.J., Xiao, Y., Gong, M., Guo, W.Y., Sun, P., Xu, S., Yang, Y.B.: Lattice qcd package gwu-code and quda with hip. arXiv preprint arXiv:2001.05706 (2020)
8. Boyle, P., Yamaguchi, A., Cossu, G., Portelli, A.: Grid: A next generation data parallel c++ qcd library. arXiv preprint arXiv:1512.03487 (2015)
9. Boyle, P.A.: Machines and algorithms. arXiv preprint arXiv:1702.00208 (2017)
10. Clark, M.A., Babich, R., Barros, K., Brower, R.C., Rebbi, C.: Solving Lattice QCD systems of equations using mixed precision solvers on GPUs. Comput. Phys. Commun. **181**, 1517–1528 (2010)
11. Dupuis, M., Rys, J., King, H.F.: Evaluation of molecular integrals over gaussian basis functions. The Journal of Chemical Physics **65**(1), 111–116 (1976)
12. Fedorov, D.G., Olson, R.M., Kitaura, K., Gordon, M.S., Koseki, S.: A new hierarchical parallelization scheme: Generalized distributed data interface (gddi), and an application to the fragment molecular orbital method (fmo). Journal of Computational Chemistry **25**(6), 872–880 (2004)
13. Fletcher, G.D.: Recursion formula for electron repulsion integrals over hermite polynomials. International Journal of Quantum Chemistry **106**(2), 355–360 (2006)
14. Fletcher, G.D., Schmidt, M.W., Bode, B.M., Gordon, M.S.: The distributed data interface in gamess. Computer Physics Communications **128**(1), 190–200 (2000)
15. Google: Google benchmark – a microbenchmark support library. `https://github.com/google/benchmark`
16. Gupta, R.: Introduction to lattice QCD: Course. In: Les Houches Summer School in Theoretical Physics, Session 68: Probing the Standard Model of Particle Interactions (7 1997)
17. Ishihara, T., Gotoh, T., Kaneda, Y.: Study of high Reynolds number isotropic turbulence by direct numerical simulations. Annu. Rev. Fluid Mech. **41**, 165–180 (2009)

18. Joó, B., Kurth, T., Clark, M.A., Kim, J., Trott, C.R., Ibanez, D., Sunderland, D., Deslippe, J.: Performance portability of a wilson dslash stencil operator mini-app using kokkos and sycl. In: 2019 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC). pp. 14–25. IEEE (2019)
19. King, H.F., Dupuis, M.: Numerical integration using rys polynomials. Journal of Computational Physics **21**(2), 144–165 (1976)
20. Kitware: Cmake. https://cmake.org/
21. McMurchie, L.E., Davidson, E.R.: One- and two-electron integrals over cartesian gaussian functions. Journal of Computational Physics **26**(2), 218–231 (1978)
22. Mironov, V., Alexeev, Y., Keipert, K., D'mello, M., Moskovsky, A., Gordon, M.S.: An efficient mpi/openmp parallelization of the hartree-fock method for the second generation of intel® xeon phi™ processor. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis. SC '17, Association for Computing Machinery, New York, NY, USA (2017)
23. Mironov, V., Moskovsky, A., D'Mello, M., Alexeev, Y.: An efficient mpi/openmp parallelization of the hartree–fock–roothaan method for the first generation of intel® xeon phi™ processor architecture. The International Journal of High Performance Computing Applications **33**(1), 212–224 (2019)
24. Pham, B.Q., Gordon, M.S.: Hybrid distributed/shared memory model for the ri-mp2 method in the fragment molecular orbital framework. Journal of Chemical Theory and Computation **15**(10), 5252–5258 (2019)
25. Pople, J.A., Hehre, W.J.: Computation of electron repulsion integrals involving contracted gaussian basis functions. Journal of Computational Physics **27**(2), 161–168 (1978)
26. Ravikumar, K., Appelhans, D., Yeung, P.K.: GPU acceleration of extreme scale pseudo-spectral simulations of turbulence using asynchronism. In: Proceedings of The International Conference for High Performance Computing, Networking and Storage Analysis (SC'19), Denver, CO, USA. ACM, New York, NY, USA,
27. Rys, J., Dupuis, M., King, H.F.: Computation of electron repulsion integrals using the rys quadrature method. Journal of Computational Chemistry **4**(2), 154–157 (1983)
28. Schlegel, H.B.: An efficient algorithm for calculating ab initio energy gradients using s, p cartesian gaussians. The Journal of Chemical Physics **77**(7), 3676–3681 (1982)
29. SOLLVE and NERSC: January 2021 ECP OpenMP Hackathon by SOLLVE and NERSC (2021 [Online]), the event happened on 22, 27, 28, 29 Jan 2021. Accessed 7 Apr 2021. https://sites.google.com/view/ecpomphackjan2021
30. Yeung, P.K., Sreenivasan, K.R., Pope, S.B.: Effects of finite spatial and temporal resolution on extreme events in direct numerical simulations of incompressible isotropic turbulence. Phys. Rev. Fluids **3**, 064603 (2018)
31. Yeung, P.K., Zhai, X.M., Sreenivasan, K.R.: Extreme events in computational turbulence. Proc. Nat. Acad. Sci. **112**, 12633–12638 (2015)