Enabling Fortran standard parallelism in GAMESS for accelerated quantum chemistry calculations

Melisa Alkan[i], Buu Q. Pham[i], Jeff R. Hammond[j] and Mark S. Gordon[i]

[i] Department of Chemistry and Ames Laboratory, Iowa State University, Ames, Iowa 50011, USA
[j] NVIDIA Corporation, Helsinki, Porkkalankatu 1, 00180, Finland

Abstract

The performance of Fortran 2008 DO CONCURRENT (DC) relative to OpenACC and OpenMP target offloading (OTO) with different compilers is studied for the GAMESS quantum chemistry application. Specifically, DC and OTO are used to offload the Fock build, which is a computational bottleneck in most quantum chemistry codes, to GPUs. The DC Fock build performance is studied on NVIDIA A100 and V100 accelerators and compared with the OTO versions compiled by the NVIDIA HPC, IBM XL and Cray Fortran compilers. The results show that DC can speed up the Fock build by 3.0x compared with the OTO model. With similar offloading efforts, DC is a compelling programming model for offloading Fortran applications to GPUs.

[i] E-mail: mark@si.chem.msg.iastate.edu

1. Introduction

Quantum chemistry (QC) calculations play an important role in understanding complex chemical phenomena such as reaction mechanisms, chemical reactivity, and many other properties. However, their high computational costs have been a bottleneck of QC applications. Many developments have been accomplished to overcome these challenges over the past two decades, both in terms of new theory development and computer algorithms to take advantage of new hardware[1]. Since most supercomputers and workstations are now equipped with several accelerators on each CPU compute node, many efforts have been invested to made QC codes available for use on these systems. Among these efforts, accelerating QC codes on general purpose Graphics Processing Units (GPGPUs or GPU for short) are becoming particularly prominent.[2–6]

A few popular QC codes have been ported onto GPUs .These include Terachem[2], QUICK[3], and GAMESS/LibCChem,[7] and more are planned to be released[8]. Of the aforementioned QC packages, most utilize the CUDA programming model specialized for NVIDIA GPUs. With other vendors investing in GPUs, it is important to enable QC codes on a variety of GPU architectures. Consequently, many scientific software packages have turned to the directive OpenACC and OpenMP programming model as a portable and productive alternative to CUDA[9–11]. At the same time, a GPU version of the DO CONCURRENT (DC) construct in the Fortran 2008 standard has been enabled by NVIDIA. The present work compares the performance of directive-based programming models, such as OpenMP and OpenACC with that of DC within GAMESS, a popular QC package. A focus here is on offloading the Fock build, a computationally demanding component of most QC methods. The Fock build consists of the computation of four-index two-electron repulsion integrals (ERIs) as well as their digestion into a Fock matrix. Many methods and algorithms have been developed for efficient evaluation of ERIs[12]. For example, in GAMESS, ERIs can be calculated by the Pople-Hehre (PH)[13], McMurchie-Davidson[14], Rys quadrature[15,16], and ERIC[17] methods depending on the specific types of ERIs. Efforts to accelerate all GAMESS ERI packages have been ongoing based on an OpenMP target programming model.[5] In this paper, the GPU offloading focuses on the method containing mixed Pople-Hehre and McMurchie-Davidson recurrence relations.

2. Background

A. GPU Programming Models

The first widely used programming model for GPUs was CUDA C, introduced by NVIDIA in 2007. CUDA Fortran was released by the Portland Group in 2009, thereby facilitating GPU programming directly from Fortran for the first time. OpenACC, released in 2011, provided a set of directives that supported parallel programming across a range of platforms, including GPUs. In 2013, OpenMP also added multi-platform directive support for GPUs. The Fortran 2008 standard added a construct called DO CONCURRENT (DC), which allows application developers to instruct compilers that some form of loop-level parallelism can be exploited. However, implementations were limited to CPU vector and thread parallelism. In 2020, NVIDIA released support for a GPU feature for the DC construct in the NVIDIA HPC Fortran compiler (NVHPC Fortran).

Currently, at least three compilers (Intel, Cray and NVHPC) support DC. Both Intel and Cray Fortran compilers can map DC to vector and thread parallelism in a similar manner to the "`omp` `parallel do simd`” OpenMP directive. The NVHPC Fortran compiler supports DC in a similar manner to the “`acc parallel loop`” OpenACC directive. With NVHPC, DC iterations can be mapped to both CPU and GPU threads and can take advantage of the vector parallelism.

Since GPUs have their memory distinct from the CPU (host) memory, each compiler needs a way to make data accessible to the GPU when generating GPU parallel code. For NVHPC Fortran, allocatable arrays are treated as CUDA managed memory, which is accessible to both CPUs and GPUs on the same node[18]. Furthermore, developers can also explicitly manage the data locality using other mechanisms, such as *omp* and *acc* directives (e.g., “`!$omp shared`” and “`!$omp private`” in OpenMP programming model).

B. Hartree-Fock method

A major goal of QC programs is to solve the Schrodinger equation. An analytic solution for the Schrodinger equation is only available for the simplest systems (e.g., one-electron systems). For all other molecular systems, the Schrodinger equation has to be solved numerically utilizing many approximations. One such fundamental approximation is the Hartree-Fock (HF) method, which can recover 99% of the total energy and is often the first step for QC calculations. Many correlation methods can be developed on top of the HF method to recover the remaining 1% of the energy and to improve the accuracy of the computations. Therefore, accelerating the HF method is an important starting point for wave function-based method development.

In the HF method, the N-body wave function is approximated by a determinant of one-electron functions, called molecular orbitals (MO). Each MO $\{\psi_i(r)\}$ is approximated by a linear combination of atomic orbitals (AO) $\{\phi_\mu(r)\}$:

$$\psi_i(r) = \sum_{\mu}^{K} \phi_\mu(r) C_{\mu i} \tag{1}$$

The MO coefficients $\{C_{\mu i}\}$ are obtained by minimizing the energy of the system, thereby requiring computations of 4-index 2-electron repulsion integrals (ERI), which comprise a bottleneck of the HF method:

$$(\mu\nu|\lambda\sigma) = \iint dr_1 dr_2 \phi_\mu(r_1)\phi_\nu(r_1) r_{12}^{-1} \phi_\lambda(r_2)\phi_\sigma(r_2) \tag{2}$$

In Eq. (2), r_1 and r_2 refer to the coordinates of electrons 1 and 2, respectively. When doing a HF calculation, a user needs to define the geometry of a molecular system as well as a basis set. A basis set contains a set of known Gaussian-like primary functions pre-contracted to form AO basis functions, to which the Fock operator is applied. A contracted basis function $\{\phi_\mu(r)\}$ is defined in eq. (*3*), in which L, $d_{\mu p}$ and $\{\chi_p\}$ are the contraction length, the contraction coefficients and Gaussian-like primitive functions, respectively. R_A and R_P are the centers on which the basis and the primitive functions, respectively, are located.

$$\phi_\mu(r - R_A) = \sum_{p=1}^{L} d_{\mu p} \chi_p(\alpha_{p\mu}, r - R_P) \tag{3}$$

An AO basis function can be classified based on its angular momentum. For example, AOs with angular momenta of 0, 1, 2 and 3 are called s, p, d and f orbitals, respectively. They are formed by contracting corresponding primitive functions. For instance, the $1s$, $2p_x$ and $3d_{xy}$ primitive functions are defined as follows[19]

$$\chi^{1s}(\alpha, r) = \sqrt[4]{\frac{8\alpha^3}{\pi^3}} e^{-\alpha r^2} \tag{4}$$

$$\chi^{2p_x}(\alpha, r) = \sqrt[4]{\frac{128\alpha^5}{\pi^3}} x e^{-\alpha r^2} \tag{5}$$

$$\chi^{3d_{xy}}(\alpha, r) = \sqrt[4]{\frac{2048\alpha^7}{\pi^3}} xy e^{-\alpha r^2} \tag{6}$$

In eqs. (*4*)-(*6*), the numbers 1, 2, 3 in the superscripts $1s$, $2p_x$ and $3d_{xy}$ are AO principle quantum numbers representing the atomic energy levels. The factor $\{\alpha\}$ is the orbital exponent which governs the spatial extent of the orbital. The order of the polynomial factor (e.g., 0, 1 and 2 for the factors 1, x, and xy in the $1s$, $2p_x$ and $3d_{xy}$ functions) corresponds to the orbital angular momentum. The s-type function with a 0th order polynomial factor has 0th order degeneracy; i.e., there is a *single* linearly independent s-type basis function for each AO energy level. The p-type basis function with a 1st order polynomial factor has 1st order degeneracy; i.e., there are *three* linearly independent (degenerate) basis functions (p_x, p_y and p_z) for each AO energy level. Similarly, there are *six* linearly independent (degenerate) d-type basis functions (d_{xx}, d_{yy}, d_{zz}, d_{xy}, d_{xz} and d_{yz}) at each AO energy level.

ERIs of higher angular momentum basis functions have increasing computational complexity. Usually, to achieve meaningful chemical results for the lighter elements in the Periodic Table, it is important to use basis sets that contain up to at least *d* functions. This work focuses on the performance of ERIs containing *d*-type basis functions.

In GAMESS, ERIs that contain d functions are calculated using the SPD and RYS packages. The SPD package is based on mixed Pople-Hehre and McMurchie-Davidson recurrence relations[20]. The RYS package is based on the numerical Rys quadrature method[15,16]. In this paper, the newly developed SPD algorithm and its respective parallelization schemes are discussed in the context of DC.

3. Algorithms

An ERI $(\mu\nu|\lambda\sigma)$ is a linear combination of the primitive ERIs, which are 4-index integrals of primitive (basis) functions, $(\chi_p\chi_q|\chi_r\chi_s)$:

$$(\mu\nu|\lambda\sigma) = \sum_{pqrs}^{L_1,L_2,L_3,L_4} d_{\mu p} d_{\nu q} d_{\lambda r} d_{\sigma s} (\chi_p\chi_q|\chi_r\chi_s) \tag{7}$$

To efficiently compute ERIs over primitive functions, primitive functions which share the same orbital exponents, are grouped into shells. For instance, a d-shell contains all d-type primitive functions used to construct *six* basis functions d_{xx}, d_{yy}, d_{zz}, d_{xy}, d_{yz} and d_{xz}. For some cases, when the orbital exponent of s and p-type functions are the same, they are grouped together and called an L-shell in order to reduce the computational effort.

The default algorithm to evaluate ERIs in GAMESS is to iterate over four shells. All primary integrals in a group of four shells, called a quartet, are computed together facilitating intermediate arrays to be shared among them. Figure 1a shows an MPI/OpenMP implementation of ERIs in GAMESS. The outermost loop iterations *ish* are distributed over MPI ranks, whereas the inner loop iterations *jsh* and *ksh* are fused together using `collapse` clause and parallelized with OpenMP. Both MPI and OpenMP layers use dynamic load balancing schemes[21].

For *d*-shell ERIs, the structure in Figure 1a computes 15 types of quartets including (*sssd*), (*sspd*), ... (*dddd*). These ERIs differ in the count of floating-point operations (FLOP) and the size of intermediates, which can introduce workload imbalance. Such an effect can be reduced by employing dynamic load balance schemes in both work distribution to MPI ranks and OpenMP threads. However, dynamic scheduling is, on the one hand, not available in the OpenMP target; on the other hand, the computational cost and memory differences associated with different ERI types will likely lead to thread divergence that will ruin GPU performance. To take advantage of the massive parallelization offered by the GPUs, the CPU algorithm (Figure 1a) needs to be restructured.

a) OpenMP parallel

```
do ish = 1,nshell
!$omp parallel do
schedule(dynamic)
collapse(2)
 do jsh = 1, ish
  do ksh = 1, ish
   do lsh = 1,ksh
   !screening
   !compute ints
   !digest ints
   enddo
  enddo
 enddo
!$omp end parallel do
enddo
```

b) Example sorting & screening

```
!pre-sorting
!get the #of s,p,d shells
n_s_shl, n_p_shl, n_d_shl
!couple bra,ket of the same
angular momenta
!screening
(ss| - perform screening
|dd) - perform screening
do iijj = 1, (ss|
 do kkll = 1, |dd)
 IDX(nquarts_screened)
 enddo
Enddo
!pass significant ints to GPU
```

c) Example OpenMP kernel

```
!pre-sorting,screening
!$omp target teams
distribute parallel do&
!$omp shared() &
!$omp private()
do iquart = 1, ssdd_quarts
!recover shell index
ish=IDX(s_sh)
jsh=IDX(s_sh)
ksh=IDX(d_sh)
lsh=IDX(d_sh)
 !compute ints
 !digest ints
enddo
!$omp end target teams
distribute parallel do
```

d) Example OpenACC kernel

```
!pre-sorting,screening
!$acc parallel loop&
!$acc copyin() &
!$acc private()
do iquart = 1,
ssdd_quarts
!recover shell index
ish=IDX(s_sh)
jsh=IDX(s_sh)
ksh=IDX(d_sh)
lsh=IDX(d_sh)
 !compute ints
 !digest ints
enddo
```

e) Example DC kernel

```
!pre-sorting,screening

DO CONCURRENT
(iquart=1:ssdd_quarts)&
SHARED() LOCAL()
!recover shell index
ish=IDX(s_sh)
jsh=IDX(s_sh)
ksh=IDX(d_sh)
lsh=IDX(d_sh)
 !compute ints
 !digest ints
enddo
```

Figure 1. (a) MPI/OpenMP implementation of ERIs in GAMESS; (b) quartet sorting and screening for GPU offloading; (c) quartet offloading using OpenMP; and (d) quartet offloading using DC.

Commented [PBQ[L1]: As reviewer suggested, I think we should either remove all arrows

Commented [AM2R1]: done

The code restructuring includes multiple steps as shown in Figure 1b-e. Figure 1b summarizes quartet preparation steps. First, all shells in a given input molecule are sorted into S, P, D groups (e.g., the S group contains only s-type basis functions). Second, these shells are coupled into different pair types, e.g., *ss*, *sp*, ..., *dd*. Next, these pairs are coupled to form quartets of the same types, e.g., (*ss*|*sd*), (ss|pd) up to (*dd*|*dd*). Quartets of each type are first screened using the Schwarz inequality, to avoid computing ERIs that will be smaller than a chosen cutoff[22]. Quartets that have significant values above the chosen cutoff are sent to the GPUs for computation.

In Figure 1c, quartets obtained from preparation steps (Figure 1b) are distributed over teams and then over GPU threads for actual computation using the "`!$omp target teams distribute parallel do`". This is an example of the OpenMP kernels currently developed and implemented in GAMESS. The details of the algorithm are presented in references[6,23].

In Figure 1e, the *DC* equivalent of OpenMP target kernels is obtained by simply replacing the DO-ENDDO construct by the DO CONCURRENT-ENDDO construct. In a similar manner, OpenACC kernels (Figure 1d) can directly be derived from the OpenMP kernels by replacing the "`!$omp target teams distribute parallel do`" by a "`!$acc parallel loop`" directive, and "`!$omp private`" is replaced with "`!$acc private`". The performance of *DC* kernels will be compared, herein, to the baseline OpenMP kernels and their equivalent OpenACC implementations across compilers that currently support GPU offloading.

Commented [PBQ[L3]: If it is not too hard to do, we should add another pseudo code for OpenACC kernel

Commented [AM4R3]: Added

4. Compiling ERI kernels

OpenMP GPU, OpenACC and DC ERI kernels discussed in Section 3 and Figure 1 are compiled using flags and GPUs shown in Table I. In Table I, DC ERI kernels using the nvfortran compiler (NVHPC/22.5) on A100 GPUs are denoted DC-A100; OpenACC kernels using the nvfortran compiler on A100 GPUs are denoted ACC-A100; OpenMP kernels using the nvfortran compiler on the A100 are denoted OMP(nvfortran)-A100; OpenMP kernels using the Cray compiler on the A100 are denoted OMP(cce)-A100. OpenMP kernels on the V100 using the nvfortran and xlf compilers are denoted OMP(nvfortran)-V100 and OMP(xlf)-V100.

Table I. Compiler flags for ERI kernels

Commented [PBQ[L5]: Melisa: could you fill the table?

Commented [AM6R5]: yes

Kernel	**Compiler flags**
DC-A100	-i8 -fast -mcmodel=medium -stdpar=gpu -target=gpu -gpu=cc80 -Minfo=acc -gpu=managed
ACC-A100	-i8 -fast -mcmodel=medium -acc -target=gpu -gpu=cc80 -gpu=managed
OMP(nvf)-A100	-i8 -fast -mcmodel=medium -mp -target=gpu -gpu=cc80 -gpu=managed

OMP(cce)-A100	-O3 -h omp -hnoacc -DSETDEVICEID -sinteger64 -haccel=nvidia80
OMP(xlf)-V100	-qsuffix=f=f90 -qfree=f90 -c -O2 -qsmp=omp -qoffload -qxlf90=autodealloc -q64 -qintsize=8 -qarch=pwr9 -qtune=pwr9 -qflag=W:W -qhalt=W -qspillsize=2500 -qnosave

5. Results

A. Offloading ERI kernels with DC

ERI kernels offloaded by DC are shown in Figure 1e. These kernels were compiled with the flag "`-stdpar=gpu –mp –target=gpu –gpu=cc80 –gpu=managed`" on the Perlmutter[24] supercomputer with A100 GPUs as shown in the second row of Table I. The performance of these ERI kernels was studied by computing the closed shell restricted Hartree-Fock (RHF) energy for clusters of 64 and 128 water molecules with the correlation consistent cc-pVDZ[25] and Pople 6-31G(d)[26] basis sets, both of which are frequently used in QC calculations.

The wall times of the five kernels INT0002, INT0112, INT0102, INT0222, and INT1222 are shown in columns 5 (before optimization) and 6 (after optimization) inTable II. The main optimization for DC kernels is loop unrolling. The numbers 0, 1 and 2 are the angular momentum values of the s, p and d basis functions, respectively. Therefore, the kernel INT0002 contains *three s* and *one d* shells, whereas the kernel INT1222 kernels contains *one p* (or L) and *three d* shells.

Table II. Performance of selected DC kernels before and after compiler-driven optimization. Some speedups are (under N/A) are not reported since the overall timing for the optimized kernel is very small and would not reflect the speedup accurately.

Molecule	Basis Set	Kernel	Number of quartets	Before optimization (s)	After optimization (s)	Speedup
$(H_2O)_{64}$	**6-31G(d)**	INT0002	31,906,009	0.35	0.09	3.88
		INT0112	15,236,334	1.76	0.15	11.70
		INT0102	15,236,334	1.82	0.14	13.00
		INT0222	393,097	0.33	0.03	10.00
		INT1222	172,891	0.39	0.05	7.80
	cc-pVDZ	INT0002	102,844,898	0.72	0.19	3.79
		INT0112	53,286,090	4.71	0.27	17.40
		INT0102	53,286,090	6.14	0.28	21.90
		INT0222	388,113	0.3	0.003	N/A
		INT1222	158,090	0.37	0.005	N/A
$(H_2O)_{128}$	**6-31G(d)**	INT0002	81,844,551	1.0	0.23	4.34
		INT0112	71,173,873	7.73	0.44	17.60
		INT0102	71,173,873	8.47	0.43	19.70
		INT0222	1,637,228	1.34	0.10	13.40
		INT1222	806,825	1.77	0.20	8.85
	cc-pVDZ	INT0002	268,907,047	3.17	0.82	3.87
		INT0112	246,223,301	28.4	1.28	22.2
		INT0102	246,223,301	20.1	1.02	19.7
		INT0222	1,366,689	1.02	0.08	12.80
		INT1222	531,868	1.10	0.11	10.0

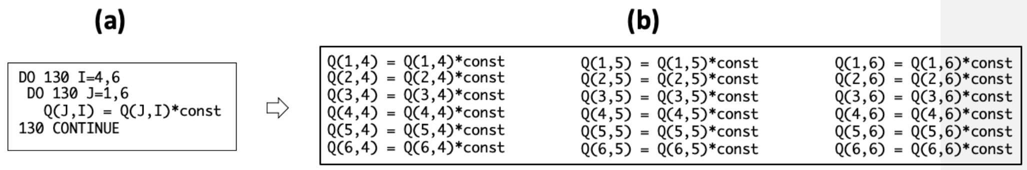
Commented [PBQ[L7]: As pointed out by the reviewer the number in red of this table is not consistent with red numbers in Table IV. I guess this is the result numerical noise between one to another run. I think Melisa can pick one, or averaging a couple of runs for both tables.

Commented [AM8R7]: Did average of 3 runs!

The wall time (see columns 5 and 6, Table II) of a kernel is determined by both the complexity of the kernel and the number of quartets in that kernel. Kernels with higher angular momentum will have higher complexity, which will require higher FLOP counts and larger intermediate arrays. For example, the intermediate array that stores integrals in the kernel INT0002 has the size of *six* double precision elements, whereas the size needed for the same array in the INT1222 kernel is $4 * 6 * 6 * 6 = 864$ double precision elements, where the 4 represents the L (s+p) shell and the 6 represents the d shell. Therefore, the wall time required to process the simplest kernel INT0002 is the smallest.

On the other hand, since the number of quartets in the high and low angular momentum kernels INT0222 and INT1222 are significantly less than that of the intermediate kernels

INT0112, and INT0102 (see column 4, Table II), the wall times required to evaluate the INT0112 and INT0102 are the largest.



```
DO 130 I=4,6
 DO 130 J=1,6
   Q(J,I) = Q(J,I)*const
130 CONTINUE
```

```
Q(1,4) = Q(1,4)*const       Q(1,5) = Q(1,5)*const       Q(1,6) = Q(1,6)*const
Q(2,4) = Q(2,4)*const       Q(2,5) = Q(2,5)*const       Q(2,6) = Q(2,6)*const
Q(3,4) = Q(3,4)*const       Q(3,5) = Q(3,5)*const       Q(3,6) = Q(3,6)*const
Q(4,4) = Q(4,4)*const       Q(4,5) = Q(4,5)*const       Q(4,6) = Q(4,6)*const
Q(5,4) = Q(5,4)*const       Q(5,5) = Q(5,5)*const       Q(5,6) = Q(5,6)*const
Q(6,4) = Q(6,4)*const       Q(6,5) = Q(6,5)*const       Q(6,6) = Q(6,6)*const
```

Figure 2. (a) original code block with CONTINUE label; and (b) manual loop unrolling to achieve performance

The DC ERI kernels are further inspected by using the `-Minfo` compiler flag that can display the compiler invocations. With the additional compiler flag, one can see that kernels using a Fortran `CONTINUE` label (e.g., see the code block in Figure 2a) causes the nvfortran compiler to parallelize loops across CUDA threads as opposed to running them sequentially (see the first row of Table III). The warning message `Loop parallelized across CUDA threads(32) collapse(2)` in the first row of Table III suggests that many individual loops inside the DC region were offloaded to the GPU instead of the desired outcome of having the main DC outer loop spread across the CUDA threads. Offloading loops inside the DC region led to unsatisfactory performance results (Table II, column 5). The structure of these non-optimized code blocks is shown in Figure 2a. Further code optimization restructured these code blocks, specifically, by unrolling the loops according to the compiler suggestions (Table III, top) to lead to optimized code (Figure 2b), thereby achieving significant speedups when compared to the non-optimized version (Table II, columns 6-7). It is an NVIDIA Fortran compiler bug that does not automatically unroll loops containing Fortran 'CONTINUE' statements.

Table III. Output of the -Minfo compiler flag before and after manual loop unrolling

non-optimized kernel message	`Generating NVIDIA GPU code (Loop parallelized across CUDA thread blocks). Loop parallelized across CUDA threads(32) collapse(2)`
optimized kernel message	`Generating NVIDIA GPU code (Loop parallelized across CUDA thread blocks, CUDA threads(128)). Loop run sequentially`

Once the `CONTINUE` labels were removed from the kernel source code and all the loops were manually unrolled (e.g., see the code block in Figure 2b), the `-Minfo` flag output was changed (see the second row of Table III) to reflect the generation of NVIDIA GPU code with the outer DC loop parallelized across CUDA blocks and threads. All other loops inside the DC region will be executed sequentially. Such changes have significantly improved the performance of the DC ERI kernels. Results for these optimized kernels are shown in columns 6 and 7, Table II. The observed speedups range from 3.79x to 27.6x. Due to their small workload, the low (INT0002) and high (INT0222 and INT1222) angular momentum kernels speedups are ~3.8-4.34x and 7.8-13.4x, respectively. The low workload of INT0002 is due to the low complexity of low angular momentum basis functions, whereas the low workload of high angular momentum kernels INT0222 and INT1222 is due to their small numbers of quartets (see column 4, Table II). For large workload kernels INT0112, INT0102, due to huge numbers of quartets, the speedup is greater, ranging from ~17.4-27.6x. This DC optimization will be applied to all 15 d-type ERI kernels in the next sections.

B. Relative performance of DC, OpenMP and OpenAcc offloading

In this section, the performance of ERI kernels offloaded to GPUs using different directive-based programing models, compilers and GPUs is studied. The three programming models are DC, OpenMP and OpenACC. Three compilers are used: nvfortran, cce, and xlf. Two GPUs are used i) the A100 with 40 GB RAM on Perlmutter[24] at the National Energy Research Scientific Computing Center (NERSC), ii) the V100 with 16 GB RAM on Summit[27] at Oak Ridge National Laboatory. The RHF energy calculations for a cluster of 128 water molecules with the correlation consistent basis set cc-pVDZ is used as a test system. For all runs with nvfortran compiler, the number of teams and threads (for OpenMP results) and the number of gangs and number of workers (for OpenACC results) is set by the compiler as it resulted in best performance. For all runs with cray compiler, the number of teams and threads was varied for integral kernels and launch details are reported in Table V. For all runs with xlf compiler, the number of teams is set to 160 and the number of threads is set to 8. All results are summarized in Table IV.

Table IV. Wall time (s) of ERI kernels for RHF energy calculations of a 128-molecule water cluster using different offloading models, compilers and GPUs (1 GPU/calculation).

Kernel	DC, AMD-Milan CPU	DC-A100	ACC-A100	OMP(nvf)-A100	OMP(cce)-A100	OMP(nvf)-V100	OMP(xlf)-V100
INT0002	2.94	0.82	0.70	1.91	1.55	1.54	60.1
INT0112	7.54	1.28	1.77	2.85	3.24	4.35	160.1
INT0102	3.15	1.02	1.13	3.14	3.41	3.63	170.8
INT0222	0.46	0.08	0.006	0.006	0.009	0.20	6.61
INT1222	1.13	0.11	0.51	0.50	0.20	0.58	23.6
Total	15.22	3.31	4.12	8.41	8.41	10.3	421.2

Calculations with DC ERI kernels using the nvfortran compiler (NVHPC/22.5) on A100 GPUs are denoted DC-A100. The results are shown in column 3, Table IV. As discussed in previous sections, for water cluster calculations, the wall time of the kernel INT0112 is the largest (e.g., 1.38 s), whereas other kernels are much smaller and are all below 1.0 (s). The DC-A100 wall times will be used as the reference to compare with other directive-based models, compilers and GPUs.

Calculations with OpenACC kernels using the nvfortran compiler on A100 GPUs are denoted ACC-A100. The results are shown in column 4, Table IV. The optimized DC source code was directly converted to OpenACC with the specification "`acc parallel loop`". The OpenACC wall times for most ERI kernels are almost identical to those of DC except for the INT0112 and INT1222, for which the ACC-A100 is slower. Overall, the speedup of the DC-A100 is about 1.30x relative to the ACC-A100.

Calculations with OpenMP kernels using the nvfortran compiler on the A100 are denoted OMP(nvfortran)-A100. The results are shown in column 5, Table IV. The OpenMP implementation has also benefited from the optimized DC kernels (e.g., removing the `CONTINUE` label and manually unrolling loops). The results show that DC outperforms the OpenMP target offload in all cases. Particularly, for large workload kernels INT0112, INT0102, DC is 2-3.27x faster. Overall, the DC-A100 is 2.67x faster than the OMP(nvfortran)-A100.

Calculations with OpenMP kernels using the Cray compiler on the A100 are denoted OMP(cce)-A100. The results are shown in column 6, Table IV. Tests for this section were performed with the Cray CCE 14.0.1 compiler. The following modules were loaded: cudatoolkit/11.7, craype-accel-nvidia80, craympich/8.1.17, cray-libsci/21.08.1.2, PrgEnv-cray/8.3.3. To successfully execute on the A100 GPUs with CCE compiler, the `simd` clause and the number of teams and threads must be provided. Overall, the DC-A100 is 2.66x faster than OMP(cce)-A100.

Calculations with OpenMP kernels on the V100 using the nvfortran and xlf compilers are denoted OMP(nvfortran)-V100 and OMP(xlf)-V100, respectively. The results are shown in columns 7 and 8 in Table IV. In comparison with the xlf, the nvfortran implementation provides much better performance on the V100. This is because OMP(xlf) performs best when a large number of OpenMP teams and a small number of threads is used resulting in a small number of CUDA grids. Overall, OMP(nvfortran)-V100 is about 40.8x faster than OMP(xlf)-V100. With the same compiler nvfortran and OpenMP model, shifting from V100 (i.e., OMP(nvfortran)-V100) to A100 (i.e., OMP(nvfortran)-A100) can speed up calculations by 1.22x.

In Table IV column 1 additional CPU multicore DC calculations were carried out as a reference. A full Perlmutter node (2 64-core AMD Milan CPUs) was used. Compilation flag for CPU execution of DC was changed to ”`-stdpar=multicore`”. Results reported in Table IV suggest a ~4.60x speedup of DC-GPU code versus DC-CPU code on a single A100 GPU.

As noted above, the optimal performance for OpenMP and OpenACC kernels using nvfortran compiler was achieved without specifying the size of the CUDA grid. Note that DC implementation does not allow for manual specification of the CUDA grid; that is, there is no user control to specify GPU parallelism apriori. Therefore, the CUDA grid being launched for each of the programming models is different and is briefly summarized in Table V.

Table V. Total CUDA grid sizes of ERI kernels for RHF energy calculations of a 128-molecule water cluster using different offloading models and compilers on A100 GPUs (1 GPU/calculation).

Kernel	DC-A100	ACC-A100	OMP(nvf)-A100	OMP(cce)-A100
INT0002	65,535	65,535	3,007,114	159,744
INT0112	65,535	65,535	1,838,231	159,744
INT0102	65,535	65,535	2,667,516	114,688
INT0222	32,690	32,690	32,690	159,744
INT1222	22,693	22,693	22,693	16,384

Nvfortran compiler chooses the number of threads per block to be 128 by default as optimal for NVIDIA-based GPUs. For OpenACC and DC kernels, grid sizes are identical likely due to the fact that DC implementation is based on OpenACC. For integral kernels that require more memory (INT0222, INT1222) the CUDA grid is significantly smaller. For OpenMP-based kernels, while the same INT0222 and INT1222 kernels have small CUDA grids, INT0002, INT0102, and INT0112 kernels have significantly more CUDA thread blocks. In fact, the smaller the kernel (INT0002), the larger the number of CUDA thread blocks launched. This configuration results in most optimal performance for OpenMP-based kernels; the authors hypothesize the choice of the CUDA grid by the compiler depends on the total number of quartets of type INT0002 that are needed to be evaluated. There number of INT0002 type integrals necessary to be computed is larger than the number of INT0222 or INT1222 type integrals on the order of ~1000x due to the fact that there are more *s*-type functions in the basis set. For cray compiler on A100 GPUs, the number of teams and threads were varied to achieve best performance. For instance, for INT0002, INT0112, INT0102, and INT0222, a combination of 832 or 896 teams and 128 and 192 and 128 threads per team yielded optimal results whereas for INT1222 kernel, 128 teams and 128 threads per team resulted in best performance. Hence, due to a significantly smaller CUDA grid launched by the OpenMP kernels using cce compiler, performance of these kernels is worse when compared to OpenMP kernels launched using nvfortran compiler.

In summary, the DC-A100 can be used as a compelling alternative to the directive-based approaches. On the same A100, DC outperforms OpenMP with nvfortran and cce compilers. OpenACC and DC perform similarly. However, while OpenACC is specialized for NVIDIA GPUs, DC is a standard construct in Fortran 2008, which is currently supported by Fortran compilers to achieve CPU parallelism and is rapidly gaining accelerator support by a variety of compilers[28].

C. Application to Chemistry

In this section, the newly developed DC and OpenMP GPU kernels are compared to the existing threaded CPU code in GAMESS. Optimizations discussed in Sections 4A and 4B are applied to all 15 d-type ERI kernels. The full GAMESS application was compiled on the Perlmutter supercomputer with the nvfortran compiler and Cray MPI for both the OpenMP CPU and OpenMP GPU and DC GPU versions. Table VI shows the full Fock build wall times for a cluster of 128 water molecules with the 6-31G(d) basis set for the OpenMP CPU code (column 1), the OpenMP GPU code (column 2), and the DC GPU code (column 3).

Table VI. Wall time to solution (s) of a cluster of 128 water molecules with the 6-31G(d) basis set using *one* A100 GPU.

Model	**OpenMP CPU**	**OpenMP GPU**	**DC GPU**
Fock build	44.0	34.3	11.4

The OpenMP CPU calculation was executed with one MPI rank and 64 threads that effectively utilize the entire 64-core AMD EPYC CPU. The total wall time recorded is 44.0 seconds. Both OpenMP and DC GPU calculations were run with one MPI rank on one A100 GPU. The total Fock build times for the OpenMP and DC GPU are 34.3 and 11.4 (s), respectively. In summary, the DC GPU is 3.0x faster than the OpenMP GPU and 3.89x faster than the corresponding OpenMP CPU. In future implementations, MPI parallelism will be added to both DC and OpenMP kernels to enable multi-GPU execution. Most importantly, the results shown in Table VI demonstrate the practical applicability of DC kernels to accelerate scientific applications such as GAMESS.

5. Concluding remarks

In this work, new integral kernels based on OpenMP and the Fortran 'DO CONCURRENT' (DC) clause were developed in GAMESS for accelerated quantum chemistry calculations. Implementations using DC were compared to directive-based approaches such as OpenMP and OpenACC. While OpenMP remains a portable approach to running GPU-accelerated codes, the performance of standard Fortran parallelism can provide an important alternative for a broad range of legacy quantum chemistry packages. DC can speed up full HF calculations by 3.0x relative to equivalent directive based models. In addition, OpenMP kernels are significantly sped up by employing new nvfortran and cce compilers relative to the mature xlf compiler. Provided small code differences between the DC, OpenMP, and OpenACC models, preprocessor conditions can be employed to enable DC when an NVIDIA GPU is used; other directive-based models can be enabled when other GPUs (e.g., AMD, Intel) are used. It remains an open challenge to create truly portable OpenMP kernels. Currently, one needs to tweak not only the

OpenMP directives for different compilers (e.g., the need to add `simd` for the CCE compiler, varying numbers of teams and threads), but also some modification of the source codes to achieve the best performance. From this perspective, DC could serve as a potential replacement to directive-based approaches as part of the Fortran standard. Nevertheless, OpenMP can provide more directives to support, for instance, data transfer and asynchronicity that might continue outperforming the DC model.

Acknowledgements

The authors thank the Department of Energy GAMESS Exascale Computing Project to the Ames Laboratory, Project Number: 17-SC-20-SC. Ames Laboratory is operated by Iowa State University under Contract No. DE-AC02-07CH11338. The computer time provided by the Oak Ridge Leadership Computing Facility and National Energy Research Scientific Computing Center (NERSC) is gratefully acknowledged. The authors also thank Dr. Luke Roskop for help with compiling and running GAMESS with the CCE compiler suites.

References

(1) Ochsenfeld, C.; Kussmann, J.; Lambrecht, D. S. Linear-Scaling Methods in Quantum Chemistry. In *Reviews in Computational Chemistry*; Reviews in Computational Chemistry; 2007; pp 1–82.

(2) Seritan, S.; Bannwarth, C.; Fales, B. S.; Hohenstein, E. G.; Isborn, C. M.; Kokkila-Schumacher, S. I. L.; Li, X.; Liu, F.; Luehr, N.; Snyder Jr., J. W.; et al. TeraChem: A Graphical Processing Unit-Accelerated Electronic Structure Package for Large-Scale Ab

Initio Molecular Dynamics. *WIREs Comput. Mol. Sci.* **2021**, *11*, e1494.

(3) Miao, Y.; Merz, K. M. J. Acceleration of Electron Repulsion Integral Evaluation on Graphics Processing Units via Use of Recurrence Relations. *J. Chem. Theory Comput.* **2013**, *9*, 965–976.

(4) Kwack, J.; Bertoni, C.; Pham, B.; Larkin, J. Performance of the RI-MP2 Fortran Kernel of GAMESS on GPUs via Directive-Based Offloading with Math Libraries. In *WACCPD*; Wienke, S., Bhalachandra, S., Eds.; Springer International Publishing: Cham, 2020; pp 91–113.

(5) Pham, B. Q.; Carrington, L.; Tiwari, A.; Leang, S. S.; Alkan, M.; Bertoni, C.; Datta, D.; Sattasathuchana, T.; Xu, P.; Gordon, M. S. Porting Fragmentation Methods to GPUs Using an OpenMP API: Offloading the Resolution-of-the-Identity Second-Order Møller–Plesset Perturbation Method. *J. Chem. Phys.* **2023**, *158*, 164115.

(6) Pham, B. Q.; Alkan, M.; Gordon, M. S. Porting Fragmentation Methods to Graphical Processing Units Using an OpenMP Application Programming Interface: Offloading the Fock Build for Low Angular Momentum Functions. *J. Chem. Theory Comput.* **2023**, *19*, 2213–2221.

(7) 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.; et al. Recent Developments in the General Atomic and Molecular Electronic Structure System. *J. Chem. Phys.* **2020**, *152*, 154102.

(8) Kowalski, K.; Bair, R.; Bauman, N. P.; Boschen, J. S.; Bylaska, E. J.; Daily, J.; de Jong,

W. A.; Dunning, T. J.; Govind, N.; Harrison, R. J.; et al. From NWChem to NWChemEx: Evolving with the Computational Chemistry Landscape. *Chem. Rev.* **2021**, *121*, 4962–4998.

(9) Deslippe, J.; Samsonidze, G.; Strubbe, D. A.; Jain, M.; Cohen, M. L.; Louie, S. G. BerkeleyGW: A Massively Parallel Computer Package for the Calculation of the Quasiparticle and Optical Properties of Materials and Nanostructures. *Comput. Phys. Commun.* **2012**, *183*, 1269–1289.

(10) Ku, S.; Chang, C. S.; Hager, R.; Churchill, R. M.; Tynan, G. R.; Cziegler, I.; Greenwald, M.; Hughes, J.; Parker, S. E.; Adams, M. F.; et al. A Fast Low-to-High Confinement Mode Bifurcation Dynamics in the Boundary-Plasma Gyrokinetic Code XGC1. *Phys. Plasmas* **2018**, *25*, 56107.

(11) Ishihara, T.; Gotoh, T.; Kaneda, Y. Study of High–Reynolds Number Isotropic Turbulence by Direct Numerical Simulation. *Annu. Rev. Fluid Mech.* **2008**, *41*, 165–180.

(12) Gill, P. M. W. Molecular Integrals Over Gaussian Basis Functions; Sabin, J. R., Zerner, M. C. B. T.-A. in Q. C., Eds.; Academic Press, 1994; Vol. 25, pp 141–205.

(13) Pople, J. A.; Hehre, W. J. Computation of Electron Repulsion Integrals Involving Contracted Gaussian Basis Functions. *J. Comput. Phys.* **1978**, *27*, 161–168.

(14) McMurchie, L. E.; Davidson, E. R. One- and Two-Electron Integrals over Cartesian Gaussian Functions. *J. Comput. Phys.* **1978**, *26*, 218–231.

(15) Rys, J.; Dupuis, M.; King, H. F. Computation of Electron Repulsion Integrals Using the Rys Quadrature Method. *J. Comput. Chem.* **1983**, *4*, 154–157.

(16) King, H. F.; Dupuis, M. Numerical Integration Using Rys Polynomials. *J. Comput. Phys.* **1976**, *21*, 144–165.

(17) Fletcher, G. D. Recursion Formula for Electron Repulsion Integrals over Hermite Polynomials. *Int. J. Quantum Chem.* **2006**, *106*, 355–360.

(18) Harris, M. Unified Memory for CUDA Beginners https://developer.nvidia.com/blog/unified-memory-cuda-beginners/.

(19) Szabó, A.; Ostlund, N. S. Modern Quantum Chemistry : Introduction to Advanced Electronic Structure Theory. 1996.

(20) Ishimura, K.; Nagase, S. A New Algorithm of Two-Electron Repulsion Integral Calculations: A Combination of Pople-Hehre and McMurchie-Davidson Methods. *Theor. Chem. Acc.* **2008**, *120*, 185–189.

(21) 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&Reg; Xeon Phi&Trade; Processor. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*; SC '17; ACM: New York, NY, USA, 2017; pp 39:1--39:12.

(22) Horn, H.; Weiß, H.; Háser, M.; Ehrig, M.; Ahlrichs, R. Prescreening of Two-Electron Integral Derivatives in SCF Gradient and Hessian Calculations. *J. Comput. Chem.* **1991**, *12*, 1058–1064.

(23) Chapman, B.; Pham, B.; Yang, C.; Daley, C.; Bertoni, C.; Kulkarni, D.; Oryspayev, D.; D'Azevedo, E.; Doerfert, J.; Zhou, K.; et al. Outcomes of OpenMP Hackathon: OpenMP

Application Experiences with the Offloading Model (Part II) BT - OpenMP: Enabling Massive Node-Level Parallelism; McIntosh-Smith, S., de Supinski, B. R., Klinkenberg, J., Eds.; Springer International Publishing: Cham, 2021; pp 81–95.

(24) Perlmutter https://www.nersc.gov/systems/perlmutter/.

(25) Dunning, T. H. Gaussian Basis Sets for Use in Correlated Molecular Calculations. I. The Atoms Boron through Neon and Hydrogen. *J. Chem. Phys.* **1989**, *90*, 1007–1023.

(26) Ditchfield, R.; Hehre, W. J.; Pople, J. A. Self-Consistent Molecular-Orbital Methods. IX. An Extended Gaussian-Type Basis for Molecular-Orbital Studies of Organic Molecules. *J. Chem. Phys.* **2003**, *54*, 724–728.

(27) Summit user guide https://docs.olcf.ornl.gov/systems/summit_user_guide.html#nvidia-tesla-v100.

(28) Fortran compilers that support "DO CONCURRENT" https://www.intel.com/content/www/us/en/develop/documentation/fortran-compiler-oneapi-dev-guide-and-reference/top/language-reference/a-to-z-reference/c-to-d/do-concurrent.html. b) https://releases.llvm.org/12.0.0/tools/flang/docs/DoConcurrent.html c) https://support.hpe.com/hpesc/public/docDisplay?docId=a00115296en_us&page=Fortran_Command-line_Options.html d) https://developer.nvidia.com/blog/accelerating-fortran-do-concurrent-with-gpus-and-the-nvidia-hpc-sdk/ e) https://fortran-lang.discourse.group/t/gsoc22-accelerating-fortran-do-concurrent-in-gcc/3269.