Scaling the Hartree-Fock Matrix Build on Summit

Giuseppe M. J. Barca
Research School of Computer Science
Australian National University
Canberra, Australia
giuseppe.barca@anu.edu.au

David L. Poole
Departmet of Chemistry and Ames Laboratory
Iowa State University
Ames, IA, United States
davpoole@iastate.edu

Jorge L. Galvez Vallejo
Department of Chemistry and Ames Laboratory
Iowa State University
Ames, IA, United States
jg4@iastate.edu

Melisa Alkan
Department of Chemistry and Ames Laboratory
Iowa State University
Ames, IA, United States
alkan@iastate.edu

Colleen Bertoni
Leadership Computing Facility
Argonne National Laboratory
Lemont, IL, United States
bertoni@anl.gov

Alistair P. Rendell
College of Science and Engineering
Flinders University
Adelaide, Australia
alistair.rendell@flinders.edu.au

Mark S. Gordon
Department of Chemistry and Ames Laboratory
Iowa State University
Ames, IA, United States
mgordon@iastate.edu

***Abstract*—Usage of Graphics Processing Units (GPU) has become strategic for simulating the chemistry of large molecular systems, with the majority of top supercomputers utilizing GPUs as their main source of computational horsepower. In this paper, a new fragmentation-based Hartree-Fock matrix build algorithm designed for scaling on many-GPU architectures is presented. The new algorithm uses a novel dynamic load balancing scheme based on a binned shell-pair container to distribute batches of significant shell quartets with the same code path to different GPUs. This maximizes computational throughput and load balancing, and eliminates GPU thread divergence due to integral screening. Additionally, the code uses a novel Fock digestion algorithm to contract electron repulsion integrals into the Fock matrix, which exploits all forms of permutational symmetry and eliminates thread synchronization requirements.The implementation demonstrates excellent scalability on the Summit computer, achieving good strong scaling performance up to 4096 nodes, and linear weak scaling up to 612 nodes.**

***Index Terms*—GPU, Hartree-Fock, Summit**

I. Introduction

The Hartree-Fock (HF) method is a central pillar of quantum chemistry (QC) calculations [1]. Nearly all other QC methods rely on HF as a starting point or building block. The computational focus, and the primary bottleneck, of the HF process is the evaluation of the many complex two-electron repulsion integrals (ERIs) and the multiplication of these integrals by the density matrix to form the Fock matrix. The number of ERIs scales as $\mathcal{O}(N^4)$, where N – the number of basis functions – is proportional to the number of atoms. The number of basis functions grows rapidly with the size of the molecular system of interest, rendering the evaluation and processing of the corresponding integrals computationally challenging. Therefore, creative algorithms are needed in order to apply HF and related methods to large molecular systems that require exascale computing to be feasible [2].

One such (very successful) type of method is fragmentation, in which the system of interest can be divided into multiple fragments in such a way that each fragment calculation can be performed on a separate node (or group of nodes) while still retaining acceptable accuracy [3].

```
1  E_HF = 0;
2  forall fragments and fragment pairs do
3      Guess initial D matrix;
4      Compute H^core;
5      F = H^core;
6      repeat
7          for ERI batches {ab|cd} do
8              Compute ERI (αβ|γδ) ∈ {ab|cd};
9              F_αβ += Σ_γδ D_γδ [(αβ|γδ) − 1/2 (αδ|γβ)];
10         end
11         Diagonalize F and obtain new D;
12     until converged;
13     E_HF += 1/2 Σ_αβ (H^core_αβ + F_αβ);
14 end
15 Subtract fragment energies from E_HF;
```

Algorithm 1: A fragmentation-based HF algorithm.

Pseudocode for a fragmentation-based HF algorithm is shown in Alg. 1. Here we assumed usage of a second order many-body expansion, which involves only up to fragment-

SC20, November 9-19, 2020, Is Everywhere We Are
978-1-7281-9998-6/20/$31.00 ©2020 IEEE

pair interactions (see Section II B). For each fragment and fragment pair the algorithm performs an iterative HF calculation. The most computation-intense stages are the evaluations of the ERIs, and combining them with the density matrix $\boldsymbol{D}$, to form the elements of the Fock matrix $\boldsymbol{F}$ (lines 8-9). The latter stage (line 9) is called the digestion of the ERIs. Both stages scale as the number of integrals, *i.e.* as $\mathcal{O}(N^4)$ with system size. Although ERI screening techniques have been developed that reduce such scaling to $\mathcal{O}(N^2)$ asymptotically with system size, the scaling prefactor is so large that the ERIs cannot be stored in main memory, and they must be recomputed in batches at each SCF iteration and digested on the fly in order to avoid I/O overhead.

In this work, an algorithm which performs a scalable and efficient build of the Fock matrix, for fragmentation-based HF methods, on distributed heterogeneous architectures with many GPUs is presented. Manifold GPU-accelerated codes have been developed for the Hartree-Fock method [4]–[14], although very few run on multiple GPUs [7]. The novel contributions in this work are:

- A double-stream digestion algorithm that uses full symmetry and at the same time eliminates explicit GPU thread synchronization.
- A novel dynamic load balancing scheme for achieving high parallel efficiency in the integral computation and digestion, both across threads of the same GPU and across multiple GPUs.
- Integration of the screening of the integrals within the dispatch of shell quartets. This systematic screening circumvents the use of any conditional statement in the device code and eliminates GPU thread divergence and related underutilization.
- An integral code based on optimized recursive approaches that minimize the computational cost per class of integrals specifically for the GPU architecture. This allows optimal usage of the complex GPU memory hierarchy, thereby maximizing performance.
- Code optimized for the computation of each class is generated in the form of CUDA kernels. These highly optimized kernels are one of the keys for the efficient computation of the integrals on the GPU and they account for over 20,000 lines of CUDA code.

Section II describes required notation (II-A), and introduces the many-body expansions (II-B) and the Hartree-Fock method (II-C). In Section III, based on state-of-the-art GPU implementations of the Fock build, we discuss the unresolved algorithmic challenges for achieving high scalability and performance. Sections IV and V present our scalable HF algorithm and discuss how it addresses the above-mentioned challenges, while Section VI examines the effect of various optimizations on its performance. Finally, Section VII presents an overview of the Summit supercomputer at Oak Ridge National Laboratory and scaling results of our code on this platform.

All code was written in C++, and uses the Message Passing Interface (MPI) for distributed-memory parallelization and CUDA C as a GPU port path. All timings are in seconds.

II. Background

This section contains notation and background on the HF computation that is needed to understand the implementation presented in Sections IV and V.

A. Basis functions, shells and integrals

Quantum chemical calculations are performed by approximating the wave function (solution to the Schrödinger equation which is unfortunately not directly solvable), which describes the electronic behavior, with a combination of contracted Gaussian basis functions.

A contracted Gaussian basis function (CGF)

$$|\alpha) = \sum_{i=1}^{K_A} \varphi^{\boldsymbol{a}}_{\alpha,i}(\boldsymbol{r}), \tag{1}$$

is a sum of primitive Gaussian functions (PGFs)

$$\varphi^{\boldsymbol{a}}_{\alpha,i}(\boldsymbol{r}) \equiv |\alpha]_i = D^i_{\mathrm{A}}(x-A_x)^{a_x}(y-A_y)^{a_y}(z-A_z)^{a_z}e^{-\lambda_i|\boldsymbol{r}-\boldsymbol{A}|^2} \tag{2}$$

K_{A} is the degree of contraction and is typically between 1 and 20. A primitive is defined by its contraction coefficient D^i_{A}, exponent λ_i, atomic center $\boldsymbol{A} = (A_x, A_y, A_z)$, angular momentum vector $\boldsymbol{a} = (a_x, a_y, a_z)$ and total angular momentum $a = a_x + a_y + a_z$. For conciseness, we will usually suppress the primitive index i.

CGFs are classified in different types based on the total angular momentum value a of their primitives: we will use only s-, p- and d-type CGFs which have $a = 0$, $a = 1$ and $a = 2$, respectively. Depending on the atomic elements within the molecule, a variable number of CGFs – from 1 to over 50 – is placed at each atomic center. This also implies that the number of basis functions N grows linearly with the number of atoms.

For computational convenience Gaussian functions are grouped into shells. A primitive shell, indicated with the symbol $|a]$, is a set of PGFs sharing the total angular momentum a, the same exponent λ_i and center $\mathbf{A}$. Similarly, a contracted shell $|a)$ is a set of CGFs sharing the same PGFs and total angular momentum. There are $(a+1)(a+2)/2$ functions in a shell with total angular momentum a. For example, a contracted p-type shell $|a = 1) \equiv |1)$ is a set of three CGFs $\{p_x, p_y, p_z\}$ with the same PGFs and with angular momentum vectors $(1, 0, 0)$, $(0, 1, 0)$, and $(0, 0, 1)$, respectively.

Shells are then coupled to form shell pairs, which model the one-electron density. A contracted (primitive) shell pair is the set of CGF (PGF) pairs obtained by the tensor product $|ab) = |a) \otimes |b)$ ($|ab] = |a] \otimes |b]$). For example, a $|11)$ shell pair is the set of 9 CGFs pairs

$\{p_xp_x, p_xp_y, \ldots, p_zp_y, p_zp_z\}$ obtained by the tensor product of two $|1)$ shells.

Finally, primitive and contracted shell pairs are further coupled into shell quartets $|abcd] = |ab] \otimes |cd]$ and $|abcd) = |ab) \otimes |cd)$. These are used to model the Coulomb interaction between two electrons via the following two-electron repulsion integrals of CGFs

$$(\alpha\beta|\gamma\delta) = \sum_{i}^{K_A}\sum_{j}^{K_B}\sum_{k}^{K_C}\sum_{l}^{K_D} [\alpha\beta|\gamma\delta]_{ijkl} \tag{3}$$

which are sums of primitive ERIs

$$[\alpha\beta|\gamma\delta]_{ijkl} \equiv [\alpha\beta|\gamma\delta] = \iint \varphi^{\boldsymbol{a}}_{\alpha,i}(\boldsymbol{r}_1)\varphi^{\boldsymbol{b}}_{\beta,j}(\boldsymbol{r}_1)\,\frac{1}{|\boldsymbol{r}_1-\boldsymbol{r}_2|}\,\varphi^{\boldsymbol{c}}_{\gamma,k}(\boldsymbol{r}_2)\varphi^{\boldsymbol{d}}_{\delta,l}(\boldsymbol{r}_2)d\boldsymbol{r}_1d\boldsymbol{r}_2 \tag{4}$$

As for shells, total angular momentum values are used to indicate contracted and primitive classes of integrals $(ab|cd)$ and $[ab|cd]$, respectively. For example, $(22|10)$ denotes a class of $6 \times 6 \times 3 \times 1 = 108$ contracted integrals that arise from two d shells, a p shell and an s shell.

Since the ERI couple four basis functions, their number grows as $\mathcal{O}(N^4)$. Even for modest molecular sizes, this poses a very significant computational challenge. For example, using the pcSeg0 basis set [15] for a 200-water cluster results in 2600 basis functions and, formally, in $\mathcal{O}(10^{13})$ ERIs. In this study, we will deal with systems with over 9000 water molecules.

B. *Many body expansion*

A many-body expansion (MBE) decomposes the energy of the system into a set of fragment energies and corrections involving pairwise and higher-order fragment interactions. The general MBE for a set of non-covalently bonded clusters is

$$E = \sum_{I}^{N_F} E_I + \sum_{I<J}^{N_F} \Delta E_{IJ} + \sum_{I<J<K}^{N_F} \Delta E_{IJK} + ... \tag{5}$$

where, E_I represents the energy of fragment I, the parameter N_F is the number of fragments, and the different ΔE_x represent the correction for the n-body interaction. Lower order interactions are subtracted from the higher order ones to avoid double counting of the energies, e.g. $\Delta E_{IJ} = E_{IJ} - E_I - E_J$.

It is visible that MBEs are an embarrassingly parallel approach, allowing quantum chemistry to target large systems with reasonable accuracy. MBE expansions truncated at a certain degree have been shown to provide accurate results. In this work, a simple MBE expansion truncated at the second level (fragment-pair interactions) is used to showcase a scalable fragmentation approach for the accelerated Hartree-Fock algorithm.

C. *Restricted Hartree-Fock*

The Restricted Hartree-Fock (RHF) method solves the following generalized matrix eigenvalue problem

$$\mathbf{FC} = \mathbf{SC}\epsilon \tag{6}$$

for a molecule whose electrons are all paired, where the molecular orbital (MO) coefficient matrix $\mathbf{C}$ contains unknown coefficients that describe the MOs in a chosen one-electron atomic orbital basis $\{\phi^{\boldsymbol{a}}_{\alpha}(\mathbf{r})\}$ with N CGFs. The elements of the $\mathbf{S}$ matrix are the atomic orbital overlaps, while ϵ is a diagonal matrix of orbital energies, and $\mathbf{F}$ is the Fock matrix with elements

$$F_{\alpha\beta} = H^{core}_{\alpha\beta} + \sum_{\gamma\delta}^{N} D_{\gamma\delta}\left[(\alpha\beta|\gamma\delta) - \frac{1}{2}(\alpha\delta|\gamma\beta)\right] \tag{7}$$

The $H^{core}_{\alpha\beta}$ terms account for the electronic kinetic energy and the electronic-nuclear attraction, while $D_{\gamma\delta}$ are elements of density matrix defined as

$$D_{\alpha\beta} = 2\sum_{i}^{N_e/2} C_{\alpha i}C_{\beta i} \tag{8}$$

where N_e is the number of electrons.

The Fock matrix elements in Eq. (7) depend on the MO coefficients – namely the solution to the problem – via the density matrix elements. Thus, Eq. (6) is solved using an iterative scheme called the Self-Consistent Field (SCF) procedure. During each SCF iteration a new Fock matrix is formed, and, as mentioned in Section I, this involves recomputing and digesting the ERIs on the fly. Because of the large number of ERIs, which scales as $\mathcal{O}(N^4)$ with system size, efficient implementations rely on the use of upper bounds in order to avoid the evaluation of numerically insignificant integrals, that is integrals whose magnitude is smaller than a user-defined accuracy threshold τ. One of the most successful ERI screening approaches uses the following Cauchy-Schwartz inequality [16]

$$|(\alpha\beta|\gamma\delta)| \leq I_{ab}I_{cd} \qquad \forall(\alpha\beta|\gamma\delta) \in (ab|cd) \tag{9}$$

where $I_{ab} = \max_{|\alpha\beta)\in|ab)} (\alpha\beta|\alpha\beta)^{\frac{1}{2}}$. Since Eq. (9) applies to all the integrals $(\alpha\beta|\gamma\delta)$ within a given class $(ab|cd)$, for a given accuracy τ the computation of the entire class can be skipped if $I_{ab}I_{cd} \leq \tau$.

Once the ERIs are computed and digested, the generalized eigenvalue problem in Eq. (6) is solved to yield a new MO coefficient matrix $\mathbf{C}$, which is used to form the associated density matrix (Eq. (8)), and hence to form the Fock matrix in the next iteration. The procedure is repeated until the Fock matrix is converged.

III. Algorithmic challenges and related work

Designing an efficient distributed and GPU-accelerated implementation of Alg. 1 involves several computational challenges.

The first algorithmic challenge is to balance two kinds of intrinsically heterogeneous workloads across parallel execution units. Fragments and fragment pairs are systems with different numbers of atoms, therefore carrying heterogeneous computational costs. For each fragment or fragment pair, the computational bottleneck of the HF method is the evaluation of ERIs. The most efficient algorithms for the evaluation of ERIs compute the integrals in classes. Since integral classes contain very dissimilar numbers of integrals, utilization of these approaches involves uneven workloads.

```
1  for α = 1, ..., N do
2      for β = α, ..., N do
3          p = α(α + 1)/2 + β;
4          for γ = 1, ..., N do
5              for δ = γ, ..., N do
6                  q = γ(γ + 1)/2 + δ;
7                  if p ≤ q then
8                      Compute (αβ|γδ);
9                      F_αβ ← D_γδ(αβ|γδ);
10                     F_γδ ← D_αβ(αβ|γδ);
11                     F_αγ ← D_αγ(αβ|γδ);
12                     F_αδ ← D_αδ(αβ|γδ);
13                     F_βγ ← D_βγ(αβ|γδ);
14                     F_βδ ← D_βδ(αβ|γδ);
15                 end
16             end
17         end
18     end
19 end
```

Algorithm 2: Naïve ERI digestion algorithm.

A second challenge is associated with the full exploitation of the following permutational symmetry rules for the ERIs:

$$\begin{aligned}(\alpha\beta|\gamma\delta) &= (\alpha\beta|\delta\gamma) = (\beta\alpha|\gamma\delta) = (\beta\alpha|\delta\gamma) \\ &= (\gamma\delta|\alpha\beta) = (\gamma\delta|\beta\alpha) = (\delta\gamma|\alpha\beta) = (\delta\gamma|\beta\alpha) \quad (10)\end{aligned}$$

Using these equalities leads to up to eight-fold savings in the number of integrals. However, this symmetry is often either partially or fully neglected in existing GPU codes to avoid potential significant thread synchronization during the digestion of the ERIs [4]–[8]. In fact, if only symmetry-unique integrals are computed, then each integral must contribute to the formation of up to six elements of the Fock matrix, as shown in Alg. 2.

Parallelization over the α, β, γ, δ loops in Alg. 2 leads to potential race conditions as each $(\alpha\beta|\gamma\delta)$ can contribute to identical elements of the Fock matrix. For example, all integrals with the same α and β indices, and different γ and δ will be assigned to different threads but will contribute to the same Fock matrix element $F_{\alpha\beta}$. Previous attempts to exploit full symmetry and eliminate race conditions used either a combination of mutual exclusion objects (mutexes) and locks [9] or atomic operations [10], both incurring high synchronization costs, thereby impinging on the GPU parallel performance and code scalability.

The third challenge is associated with the screening of ERIs. As discussed in Section II-C, in order to reduce the computational effort, it is essential to screen out numerically insignificant integrals using, for example, the Cauchy-Schwartz inequality. Previous implementations all assigned different GPU threads to either the computation of single integrals or single integral classes. In these approaches, after a coarse-grained screening stage, integrals were more finely screened directly on the GPU [4]–[10], leading to thread-divergent execution paths and GPU resources underutilization (threads idle because of insignificant integrals).

A fourth challenge is related to the minimization of the FLOP cost for the evaluation of the ERIs. Efficient algorithms evaluate ERIs in classes via recursive approaches. This is because integrals within the same class share a large number of recursive intermediates. The pursuit of an optimal strategy to form the integrals within a given $(ab|cd)$ class using the minimum number of recursive intermediates leads to NP optimization (tree-search) problems. In order to devise efficient algorithms for ERI evaluations, one must solve such tree-search problems and generate optimal codes that compute all the integrals within each different integral class using only the smallest set of recursive intermediates [17].

The fifth and last algorithmic challenge concerns the maximization of usage of the computational capabilities of each GPU for the evaluation of the ERIs, which is the bottleneck of the HF. For extracting maximum performance from GPUs, an efficient use of the complex memory hierarchy must be implemented. More specifically, this involves: i) enforcing coalesced and 128-byte aligned (cached) global memory transactions, ii) maximizing usage of shared memory (user-controlled cache), with coalesced memory transactions while at the same time minimizing banks conflicts, iii) optimizing register usage.

The algorithm described in the following Sections aims to address all of these challenges, thereby leading to scalable code with an optimal parallel efficiency.

IV. Load Balancing Fragment Calculations

In order to address the load balancing issue arising from heterogeneous fragment and fragment pair sizes, the algorithm uses a multilayer parallel scheme to distribute fragments to different nodes and GPUs. The code starts at the highest parallel level by accepting the coordinates of each of the monomers as an input. From these input monomers, a fragment queue is generated on each rank. This fragment queue stores each fragment and each unique fragment pair, sorted by the number of atoms within the fragment or pair, with the largest molecular system occurring first.

After the fragment queue is generated, the MPI ranks are divided into MPI groups via communicators. One MPI rank is set as the “super-master” rank, existing within its own MPI group. The rest of the MPI ranks are divided

into MPI groups. Each MPI group has a master rank and a number of slave ranks associated with it, with the number of overall ranks per group being equivalent to the number of GPUs per group, a variable which is set at run time.

Once the MPI groups are generated, the fragments are distributed using a dynamic load balancing master-slave scheme, with the super-master serving as the master process and the masters of each other MPI group serving as the "slave" processes. A fragment, represented by an integer value, is sent from the super-master to the master of a worker MPI group. The master of that MPI group, in turn, broadcasts the fragment integer value to the slave processes in its own group. Finally, the MPI group performs the HF calculation on the fragment. Once the fragment calculation is complete, the master of the MPI group sends a message to the super-master informing of fragment completion, at which point it receives another fragment from the super-master. This process is repeated until all fragments have been computed.

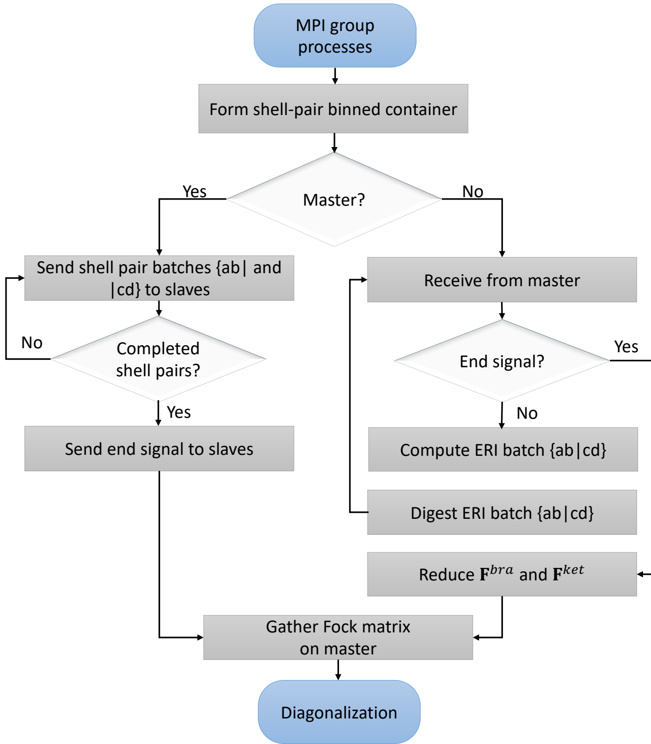

Fig. 1. The accelerated Fock build algorithm.

V. The Accelerated Fock Build Algorithm

Figure 1 shows the overarching computational scheme of our RHF algorithm. Each MPI process within the MPI group assigned to a given fragment or pair is attached to a different GPU. The GPUs managed by an MPI group can be on a single node or on multiple nodes. For our current implementation, we target NVIDIA V100 GPUs and use CUDA as a programming model.

The algorithm for each MPI group begins by pairing the contracted shells $|a)$ in the basis set into shell pairs $|ab)$. The permutational symmetry relationship $|ab) = |ba)$ is exploited to form only symmetry unique shell pairs. These are further screened using traditional shell pair screening [17] to select only the $\mathcal{O}(N)$ numerically significant shell pairs out of their $\mathcal{O}(N^2)$ total number.

While being formed, significant shell pairs are classified by type t_{ab} and size s_{ab}. Shell pairs with the same type have the same total angular momenta a and b, and the same contraction degree $K_{AB} = K_A K_B$. The type index t_{ab} is represented as an integer between 1 and the number of shell pair types n_T. The shell pair size s_{ab} is defined as

$$s_{ab} = \begin{cases} \text{int}\left(-\log_{10} I_{ab}\right) & I_{ab} \leq 1 \\ 0 & I_{ab} > 1 \end{cases} \tag{11}$$

with int() being the integer part function, and I_{ab} is the Cauchy-Schwartz bound factor. Based on the traditional Cauchy-Schwartz upper bound, Eq. (11) enables to screen out numerically insignificant integrals using the relationship

$$|(\alpha\beta|\gamma\delta)| \geq \tau \leftrightarrow s_{ab} + s_{cd} \leq -\log_{10}(\tau) \tag{12}$$

where the notation $\leftrightarrow$ indicates necessity and sufficiency, and τ is a positive user-defined accuracy threshold smaller than one. We used $\tau = 10^{-10}$.

Significant shell pairs with the same size and type are gathered into batches, indicated with the notation $|ab\}$, and suitably inserted in the data structure shown in Fig. 2, which is a "binned" shell-pair-batch container. A batch can contain up to N_{ab}^{max} shell pairs, where N_{ab}^{max} is a parameter chosen at run time.

As shown in Fig. 2, each (t_{ab}, s_{ab}) bin can contain multiple batches $|ab\}$, each identified by a local batch index g_{ab}. This provides a bijection whereby each (t_{ab}, s_{ab}, g_{ab}) triple maps to one and only $|ab\}$ and vice versa.

Once each process has formed the shell-pair binned container, the computation of two-electron integrals is parallelized over the GPUs associated with the MPI group by using a master-slave dynamic load balancing algorithm. The master process distributes pairs of batches of "bra" and "ket" shell pairs ($\{ab|$,$|cd\}$) to the slave processes, each attached to a different GPU. The assignment of batch pairs is performed with minimum overhead via only the two triplets of integers (t_{ab}, s_{ab}, g_{ab}) and (t_{cd}, s_{cd}, g_{cd}).

The implementation fully exploits the eight-fold permutational symmetry of the ERIs, dispatching only the symmetry-unique batch pairs.

The integral class screening is also performed at the batch-pair indices distribution stage. For two given batch pair index triplets (t_{ab}, s_{ab}, g_{ab}) and (t_{cd}, s_{cd}, g_{cd}), we select from the binned container only sizes s_{ab} and s_{cd} that satisfy Eq. (12). This systematic screening circumvents the use of any conditional statement in the device code and eliminates GPU thread divergence and underutilization due to insignificant integral classes.

In order to achieve an optimal workload balance both across GPU threads spawned on the same device and among different GPUs, shell pair batches are dispatched according to a greedy algorithm such that: i) integral classes are computed in a decreasing computational cost order, so that, for a large-enough system, all GPUs work at the same time on the same class type (with uniform angular momentum and degree of contraction); ii) only integral classes with the same computational cost are computed on a single GPU.

Once a slave process receives the (t_{ab}, s_{ab}, g_{ab}) and (t_{cd}, s_{cd}, g_{cd}) triplets, it transfers the corresponding $\{ab|$ and $|cd\}$ batches data from the host memory to that of its attached GPU. Therefore, the GPU computes the batch $\{ab|cd\}$ of all the integrals classes arising from the shell quartets given by the $\{ab| \otimes |cd\}$ product. In the spirit of Ufimtsev's 1T1CI mapping [5], [6], each GPU thread is assigned to the calculation of a single contracted class. This ensures that all the threads spawned on the GPU treat integral classes with exactly the same computational cost with perfect load balance.

Once computed, the ERIs in the $\{ab|cd\}$ batch are digested directly on the GPU using an algorithm that adopts two Fock matrices $\mathbf{F}^{bra}$ and $\mathbf{F}^{ket}$. More details on the ERI evaluation and their digestion are presented in Sections V-A and V-B, respectively.

Since the ERIs are formed and digested on the GPU, the algorithm minimizes the host-device data transfer, which is a well-known bottleneck for heterogeneous architectures.

The algorithm is iterated until all the significant bra-ket batch pairs are distributed and the master sends an end signal to all the slaves. Therefore, the slaves reduce their $\mathbf{F}^{bra}$ and $\mathbf{F}^{ket}$ matrices into a single partial Fock matrix on the GPU. The partial Fock matrices from each slave's GPU are then transferred to the slave's host memory and gathered on the master where they are reduced to form the full Fock matrix $\mathbf{F}$.

A. Evaluation of two-electron integrals

Once data for the $\{ab|$ and $|cd\}$ batches is transferred from the host to the GPU memory, $N_{ab}N_{cd}$ threads are spawned on the GPU, where N_{ab} and N_{cd} are the number of shell pairs $(ab|$ and $(cd|$ within batches $\{ab|$ and $|cd\}$, respectively. Each thread is assigned to the evaluation of an entire contracted integral class $(ab|cd) \in \{ab|cd\}$: as discussed later in this Section, this allows to minimize the computational cost associated with each integral class by using, for each thread, established recursive approaches [17]–[19].

The parallelization is performed by assigning to each thread block a single bra shell-pair $(ab| \in \{ab|$, and a BLOCKDIM number of kets $|cd) \in |cd\}$, where BLOCKDIM is the size of a thread block.

Pseudocode for a GPU integral computation kernel is shown in Alg. 3. A number of arrays (pointers) are passed to the kernel, each containing data for contracted

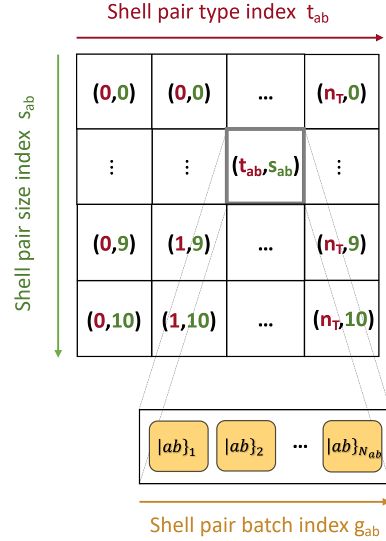

Fig. 2. The shell-pair batch binned container.

shell pairs $(ab|$ (A_t^{gl}, AB_t^{gl}) or primitive shell pairs $[ab|$ $(P_t^{gl}, U_P^{gl}, \zeta^{gl})$ within batch $\{ab|$, as well as data for $|cd)$ (C_t^{gl}, CD_t^{gl}) and $|cd]$ $(Q_t^{gl}, U_Q^{gl}, \eta^{gl})$ in batch $|cd\}$. Definitions for the various parameters are discussed in the caption of Alg. 3. The superscript "gl" emphasizes that the arrays reside in the global memory of the GPU, while for conciseness we used the subscript $t = \{x, y, z\}$.

All threads within a given block access data for the same $(ab|$ and its corresponding $[ab|$, which is stored in shared memory (line 6-13) to achieve minimal latency and higher bandwidth. Since all threads in a warp access exactly the same $(ab|$ and $[ab|$ data, reads are performed by a single-thread broadcast avoiding any memory bank conflicts.

Threads within a block access data for different $|cd)$ and all the primitives $|cd]$ associated with it. This is too much data to be stored in shared memory, hence it must be read from global memory. For efficiency, the global memory arrays are structured so that threads within the same warp can retrieve their data in a single coalesced memory transaction. For example, C_t^{gl} stores contiguously data for different $|cd)$, which is accessed by contiguous threads within the same warp. Furthermore, based on the BLOCKSIZE and N_{cd} values, the arrays are padded to ensure 128-byte aligned access, therefore maximizing bandwidth utilization.

On line 17 and 22, loops for contracting over $|cd]$ and $[ab|$, respectively, are started. The innermost loop is chosen to be over $[ab|$ to grant most frequent reads from the shared memory arrays, as opposed to the $|cd]$ data that resides in global memory. From line 27 data from the bra and ket primitive pairs is combined to form the ERIs. Each thread uses a variation of the Head-Gordon-Pople (HGP) algorithm [20], that we developed and optimized for GPU, for the evaluation of the integrals within a contracted given class $(ab|cd)$. For each of the primitive quartets $[ab|cd]$ in a

```
1  __global__ void kernel_a_b_c_d(
     K_AB, A_t^gl[], AB_t^gl[], P_t^gl[], U_P^gl[], ζ^gl[],
     K_CD, C_t^gl[], CD_t^gl[], Q_t^gl[], U_Q^gl[], η^gl[], ERIs^gl[])
2  {
3  gthreadIdx = blockIdx.x * BLOCKSIZE + threadIdx.x;
4  compute ab_offset, cd_offset;
5  compute cd_threadIdx, ERI_offset;
6  __shared__ A_t^sh = A_t^gl[ab_offset+blockIdx.x];
7  __shared__ AB_t^sh = AB_t^gl[ab_offset+blockIdx.x];
8  __shared__ U_P^sh[K_AB], P_t^sh[K_AB], ζ^sh[K_AB];
9  if threadIdx.x < K_AB then
10 |   U_P^sh[threadIdx.x] = U_P^gl[ab_offset*K_AB+threadIdx.x];
11 |   P_t^sh[threadIdx.x] = P_t^gl[ab_offset*K_AB+threadIdx.x];
12 |   ζ^sh[threadIdx.x] = ζ_t^gl[ab_offset*K_AB+threadIdx.x];
13 end
14 __syncthreads();
15 C_t = C_t^gl[gthreadIdx];
16 CD_t = CD_t^gl[gthreadIdx];
17 for k_cd = 1,...,K_CD do
18 |   U_Q = U_Q^gl[cd_offset+cd_threadIdx+N_cd * k_cd];
19 |   Q_t = Q_t^gl[cd_offset+cd_threadIdx+N_cd * k_cd];
20 |   η = η^gl[cd_offset+cd_threadIdx+N_cd * k_cd];
21 |   QC_t = Q_t − C_t;
22 |   for k_ab = 1,...,K_AB do
23 |   |   U_P = U_P^sh[k_ab];
24 |   |   P_t = P_t^sh[k_ab];
25 |   |   ζ = ζ^sh[k_ab];
26 |   |   PA_t = P_t − A_t^sh;
27 |   |   [0]^(m) ← U_P, U_Q, P_t, Q_t, η, ζ;
28 |   |   [e0|f0]^(m) ← [0]^(m), QC_t, PA_t, ζ, η; (VRRs)
29 |   |   (e0|f0)^(m) += [e0|f0]^(m);
30 |   end
31 end
32 (ab|cd) ← (e0|f0)^(0), AB_t^sh, CD_t; (HRRs)
33 stride = N_ab N_cd + padding;
34 for (αβ|γδ)_i ∈ (ab|cd) do
35 |   ERIs^gl[ERI_offset+i*stride] = (αβ|γδ)_i;
36 end
37 }
```

Algorithm 3: A generic GPU integral kernel. The superscript "gl" emphasizes that the arrays reside in the global memory of the GPU, while $t = \{x, y, z\}$. For a given contracted shell pair $(ab|$: $AB_t = A_t - B_t$, with $\boldsymbol{A} = (A_x, A_y, A_z)$ being the geometric centre of shell $|a)$. For a given primitive shell pair $[ab|_{ij}$ contributing to a contracted shell pair $(ab|$: $\zeta = \lambda_i + \lambda_j$, $P_t = (\lambda_i A_t + \lambda_j B_t)/\zeta$, $U_P = (\sqrt{\pi}/\zeta)^{3/2} e^{-(|\boldsymbol{A}-\boldsymbol{B}|^2)\lambda_i\lambda_j/\zeta}$. Analogously, for a given primitive shell pair $[cd|_{kl}$ contributing to a contracted shell pair $(cd|$: $\eta = \lambda_k + \lambda_l$, $Q_t = (\lambda_k C_t + \lambda_l D_t)/\eta$, $U_Q = (\sqrt{\pi}/\eta)^{3/2} e^{-(|\boldsymbol{C}-\boldsymbol{D}|^2)\lambda_k\lambda_l/\eta}$.

significant class, the algorithm begins by computing some "fundamental generalized integrals", indicated with the $[0]^{(m)}$ notation. For each given quartet $[ab|cd]$ all $[0]^{(m)}$ with $m \in [0, \ldots, L]$, where $L = a + b + c + d$, must be computed. In order to be computed, the $[0]^{(m)}$ quantities require geometric (atom positions) and basis-set-specific factors (P_t, Q_t, U_P, U_Q, ζ and η), as well as the evaluation of the following generalized Boys function

$$F_m(T) = \int_0^1 t^{2m} \exp\left(-t^2 T\right) dt \tag{13}$$

with $T = \eta\zeta\,|\boldsymbol{P} - \boldsymbol{Q}|^2/(\eta + \zeta)$.

The computation of the generalized Boys function with $m = L$ is done via a modified cubic Chebyshev interpolation developed by Gill *et al.* [21], requiring only 14 FLOPs. The interpolation coefficients are stored in a lookup table which is transferred from the host memory to the constant memory of each GPU at the beginning of the calculation. The table contains 72 MB of data, and its transfer time is negligible. The remaining $F_m(T)$ functions with $m \in [0, L)$ are computed using downward recurrence [21].

Once the fundamental integrals are computed, they are stored in the SM registers. Therefore, on lines 28-29, starting from the $[0]^{(m)}$ quantities each thread uses the Obara-Saika vertical recurrence relations (VRRs) [22], [23] to compute a number of intermediate primitive classes of the kind $[e0|f0]$, with $e \leq a + b$ and $f \leq c + d$, which are some of the ingredients required for the computation of the desired contracted integral class $(ab|cd)$.

As they are formed, the $[e0|f0]$ intermediates are contracted on the fly into $(e0|f0)$ using Eq. (3).

Finally, on line 32, horizontal recurrence relations (HRRs) [17] are used to transform the contracted $(e0|f0)$ classes into the desired $(ab|cd)$. Since the HRRs rely directly on contracted intermediates for building angular momentum on two out of the four Gaussian centers, their usage greatly reduces the FLOP count [17].

All the integrals within a given $(ab|cd)$ class share all the $[0]^{(m)}$ and a potentially large number of recursive intermediates. In order to minimize the number of recursive intermediates required by the HGP algorithm, a heuristic approach in tandem with the sieve method [24] was used to solve the necessary NP tree search problem.

On lines 34-36 in Alg. 3 the integrals are stored in the global memory array ERI^{gl}. This array is padded and structured so that the stores are performed in one aligned and coalesced memory transaction by each warp.

Since the optimal recursive strategy is different for each class, a code generator was written that implements the optimal solution of our tree search for each integral class. Therefore, code optimized for the computation of each class is generated in the form of CUDA kernels. These highly optimized kernels are one of the keys for the efficient computation of the integrals on the GPU and they account for over 20,000 lines of pre-generated CUDA code. Once a batch-pair is received, based on the class type a scheduler launches the corresponding kernel on the GPU.

B. Double-stream digestion

Once computed, a batch $\{ab|cd\}$ of ERIs is stored in the GPU main memory. To perform the digestion of the

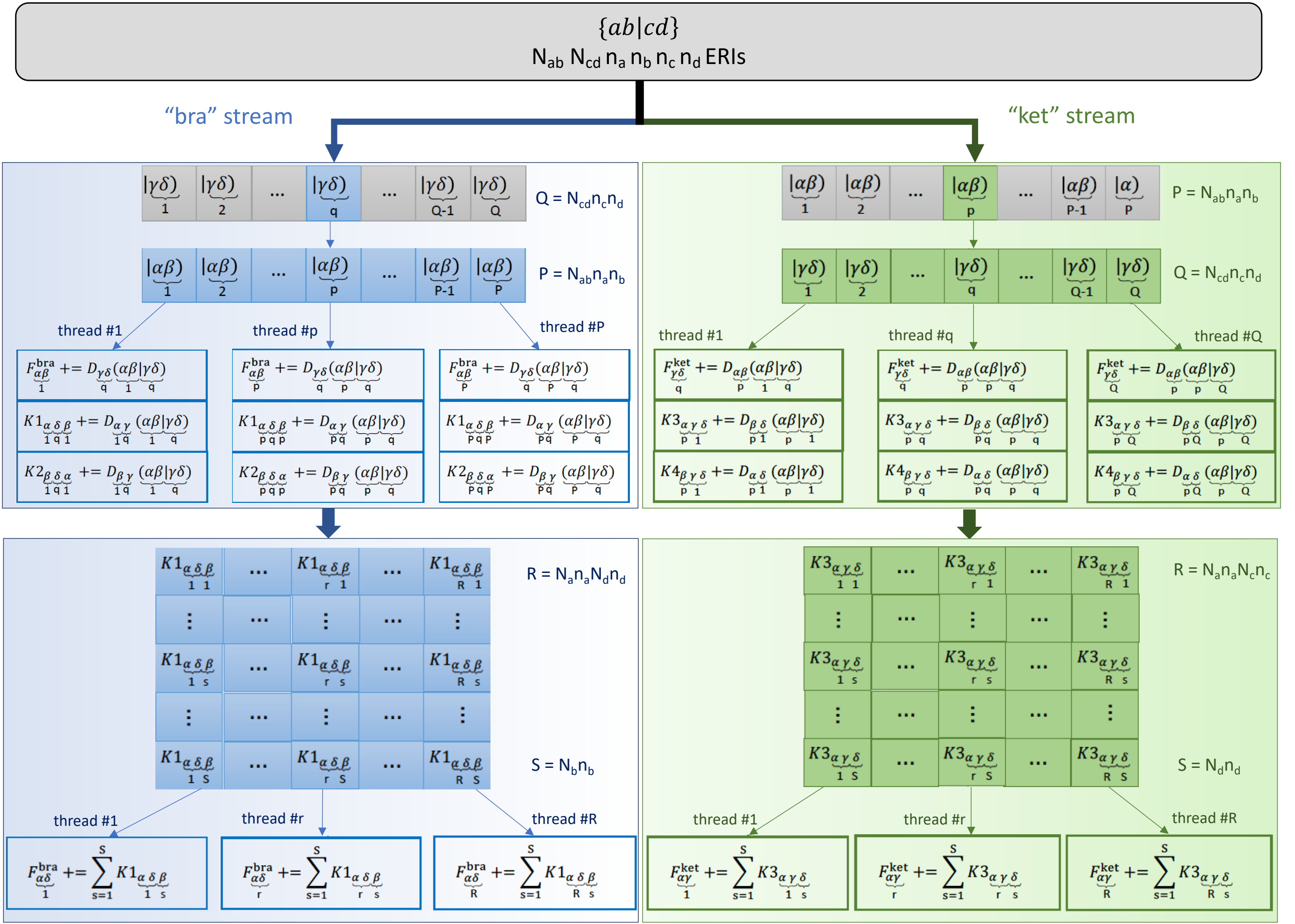

Fig. 3. The double-stream digestion algorithm. The algorithm uses two CUDA streams that execute in parallel on the GPU. The "bra" stream loops over the basis-function pairs $|\gamma\delta)$, which are associated with the $q \in [1, \ldots, \mathrm{Q}]$ index. For each $|\gamma\delta)$, $\mathrm{P} = \mathrm{N_{ab}n_an_b}$ GPU threads are launched, each mapped to a unique $|\alpha\beta)$ and to a corresponding $p \in [1, \ldots, P]$ index. Each thread performs a complete digestion of their assigned $(\alpha\beta|\gamma\delta)$ integrals into the $F^{\mathrm{bra}}_{\alpha\beta}$ elements, and a partial digestion into the $K1_{\alpha\delta\beta}$ and $K2_{\beta\delta\alpha}$ buffer elements. Once the execution of this kernel has completed, a new kernel is launched with $\mathrm{R} = \mathrm{N_an_aN_dn_d}$ threads. Each thread performs the reduction of $K1_{\alpha\delta\beta}$ into $F^{\mathrm{bra}}_{\alpha\delta}$ for a different $\alpha\delta$ index pair by summing over β, which is associated with index $s \in [1, \ldots, S]$. In the "ket" stream the roles of $|\gamma\delta)$ and $|\alpha\beta)$ pairs are swapped.

$\{ab|cd\}$ integrals, we adopt the double stream algorithm shown in Fig. 3.

In order to store the digestion intermediates, the algorithm uses a "bra" and a "ket" Fock matrix, $\mathbf{F}^{\mathrm{bra}}$ and $\mathbf{F}^{\mathrm{ket}}$, and four exchange 3D buffers **K1**, **K2**, **K3**, **K4**. While the Fock matrices scale as N^2 in memory, the four exchange buffers scale as $\mathcal{O}(1)$; the **K1** and **K2** both with dimensions $(\mathrm{N_dn_d}) \times (\mathrm{N_{ab}n_an_b})$, and **K3** and **K4** with dimensions $(\mathrm{N_an_a}) \times (\mathrm{N_{cd}n_cn_c})$ and $(\mathrm{N_bn_b}) \times (\mathrm{N_{cd}n_cn_d})$, respectively. The $\mathrm{N_{ab}}$ and $\mathrm{N_a}$ parameters are the number of shell pairs and the number of unique shells $|a)$ in a batch $|ab\}$, while $\mathrm{n_a}$ is the number of basis functions within shell $|a)$.

As shown in Fig. 3, the algorithm uses two CUDA streams that execute in parallel on the GPU. The "bra" stream loops over the $\mathrm{N_{cd}n_cn_c}$ basis-function pairs $|\gamma\delta)$ within batch $|cd\}$. For each $|\gamma\delta)$ we launch $\mathrm{N_{ab}n_an_b}$ GPU threads. Each thread is mapped to a different basis-function pair $|\alpha\beta)$ within the "bra" shell pair batch $|ab\}$. Each thread performs a complete digestion of their assigned $(\alpha\beta|\gamma\delta)$ integrals into the $F^{\mathrm{bra}}_{\alpha\beta}$ elements, and a partial digestion of the ERIs into the $K1_{\alpha\delta\beta}$ and $K2_{\beta\delta\alpha}$ buffer elements.

Then $\mathrm{N_an_aN_dn_d}$ threads are spawned on the GPU, each performing the reduction of $K1_{\alpha\delta\beta}$ into $F^{\mathrm{bra}}_{\alpha\delta}$ elements for a different $\alpha\delta$ index pair by summing over the third dimension of **K1** (associated with index β). Similarly (not shown in Fig. 3), $\mathrm{N_bn_bN_dn_d}$ threads are launched, each reducing $K2_{\beta\delta\alpha}$ into a $F^{\mathrm{bra}}_{\beta\delta}$ element for a different $\beta\delta$ index pair. This is a synchronization-free process as each thread writes to a different element of $\mathbf{F}^{\mathrm{bra}}$, **K1** and **K2**. In parallel, the "ket" stream performs an analogous algorithm

where the roles of the bra batch $|ab\}$ and the ket batch $|cd\}$ are inverted.

Our double-stream digestion minimizes redundant computation by fully exploiting symmetry and eludes explicit thread synchronization altogether.

VI. Performance optimization

In the fragmentation-based HF algorithm, the vast majority of the execution time is spent computing and digesting the ERIs. For this reason, most of our optimizations were devoted to maximizing the performance of these computational stages on the GPU.

The NVIDIA profiling tools were used to analyze the ERI kernels' performance. Profiling data revealed that the host-device data transfer had a minor impact (3-5%) on the total computation time, therefore we focused on improving the computational performance of the kernels.

Using nvprof, we conducted more detailed profiler analyses that unveiled low global memory load and write efficiencies. For example, the $(ps|ps)$ kernel, which accounts for most of the ERI computation time for a 150-water cluster when using the pcSeg0 basis set, showed only a 29% and 25% global memory load and write efficiency, respectively. This could be tracked down to the fact that the global memory arrays which were passed to the ERI kernels in Alg. 3, were not structured for 128-byte aligned and coalesced access. For example, each thread within a given thread block read Q_t data as $Q_t^{gl}[\text{cd_offset} + k_{cd} + \text{threadIdx.x} * K_{CD}]$. This caused each thread in a warp to read from global memory with a K_{CD} stride, resulting often in uncoalesced reads. Additionally, the $[ab|$ primitives pair data was not loaded in shared memory but read from global memory as for $|cd]$ primitives, with the ordering of the k_{ab} and k_{cd} loops being inverted compared to that in Alg. 3. Furthermore, the $ERIs^{gl}$ array was not structured to enable 128-byte aligned and coalesced writes. Each thread would write all the integrals within the class at contiguous locations, resulting for all kernels except those for $(ss|ss)$ classes into uncoalesced and potentially misaligned writes.

The profiling results led us to optimize memory transactions, enforcing coalesced and aligned global memory loads and writes, and increasing the use of shared memory, to yield the algorithm described in Section V-A.

Table I shows timings for the evaluation of the ERIs before and after memory transaction optimizations by using a different maximum number of registers per thread for a maximum shell pair batch size $N_{ab}^{max} = 1920$. The results in Table I were obtained on a 150-water cluster using the pcSeg0 basis set (1950 basis functions). The maximum number of registers per thread was set at compile time through the nvcc "-maxregcount" flag.

The optimized kernels show a maximum speedup of 4.6× against the unoptimized version for 160 registers. This significant amelioration is indicative of more efficient memory transactions, where, for example, for the $(ps|ps)$ kernel the global memory load and write efficiencies improved to 67.78% and 99.5%, respectively.

Max registers	Unopt time (s)	Opt time (s)	Max occupancy
32	7.535	1.636	92%
40	7.363	1.626	66%
80	6.991	1.591	33%
160	6.821	1.561	28%

TABLE I
Effect of memory optimizations on execution time for various numbers of maximum registers per thread and maximum achieved warp occupancy. The maximum shell pair batch size is $N_{ab}^{max} = 1920$.

BLOCKSIZE	Evaluation time (s)	
	$N_{ab}^{max} = 2560$	$N_{ab}^{max} = 1920$
32	1.626	1.619
64	1.605	1.575
128	1.577	1.561
256	1.599	1.577

TABLE II
Effect of thread block size (BLOCKSIZE) on ERI evaluation time.

Our profiling study also showed that the performance of the optimized kernels is mainly limited by a large number of registers used on a per-thread basis (up to 160), which results in a maximum achieved warp occupancy of 28%. In order to enhance occupancy, we systematically decreased the maximum number of registers. However, as shown in Table I, although decreasing the number of registers does improve the achieved occupancy, it also results in longer execution times. The worsening of performance is due to an increasing register spilling as the register count is lowered, which causes a very high local memory traffic overhead. The register spilling effect is so strong that even if the 32-register configuration yields a 92% maximum occupancy, the execution is up to $\sim 10\%$ slower than when using 160 registers.

Table II shows timings for the evaluation of the ERIs using the optimizations discussed so far and 160 registers per thread (on the same molecular system) as the thread block size is varied. Changes in the thread block size lead to minimal time differences for both $N_{ab}^{max} = 2560$ and $N_{ab}^{max} = 1920$. However, as ERIs need to be calculated at every step of an SCF calculation even minimal execution time differences can lead to appreciable delays for the MBE based calculations, therefore we used a block size of 128.

Finally, an optimization with respect to the maximum batch size N_{ab}^{max} was performed. Figure 4 shows the Fock build time for different N_{ab}^{max} and for systems composed of 30 (blue line), 40 (yellow line) and 50 glycine (red line) units, each poly-glycine $(C_2H_3NO)_n$ bearing $7n+3$ atoms,

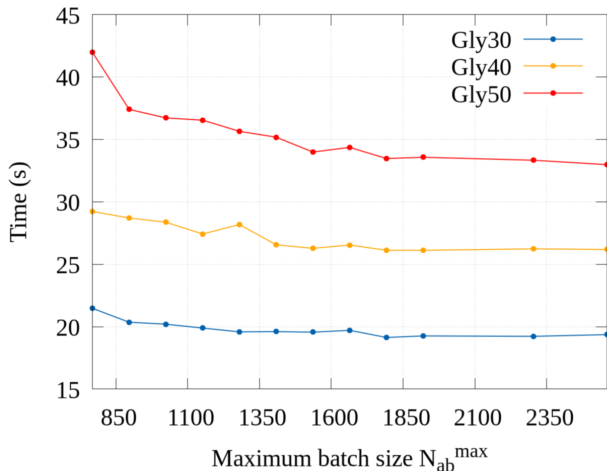

Fig. 4. Single-GPU Fock build timings for different choices of maximum shell pair sizes N_{ab}^{max} for polyglycine $(C_2H_3NO)_n$ molecules with 30, 40 and 50 monomer units.

where n is the number of monomers.

For smaller problem sizes, e.g. 30 glycine units, the dependence on the batch size is not as drastic, although a decrease in execution time is observed as the batch size grows. However, for the largest problem size (50 glycine units) the calculation with batch size 2560 was 1.27× faster than when a batch size of 768 was adopted. This is because larger systems generate more shell pairs, thereby filling larger batches which enable better GPU usage. Heuristically, we observed that a 1920 batch size yields optimal or near-optimal performance for most (large) molecular systems.

VII. Results

A. The Oak Ridge Summit System

All results presented in this work were obtained on the Summit system at the Oak Ridge National Laboratory. Summit is a 200-petaFLOP supercomputer with 4,608 compute nodes. Each of the nodes consists of 2 IBM Power9 CPUs and 6 NVIDIA Volta V100 GPUs. Each Power9 processor is connected to three V100 GPUs via NVIDIA's dual NVLink interconnect with a peak bandwidth of 25 GB/s per link, resulting in 100 GB/s of peak bidirectional bandwidth between the CPU and GPU. Each Summit node contains two 22 SIMD multi-core (SMC) Power9 processors with a total of 512GB of DDR4 memory. Each V100 has a 6MB L2 cache and 16 GB HMB2 memory with a peak bandwidth of 900 GB/s. The basic building block of a V100 is a streaming multiprocessor (SM), which consists of 32 double-precision and 64 single-precision CUDA cores. With 80 SMs, each V100 can achieve 7.8 TFLOP/s of double-precision floating-point performance.

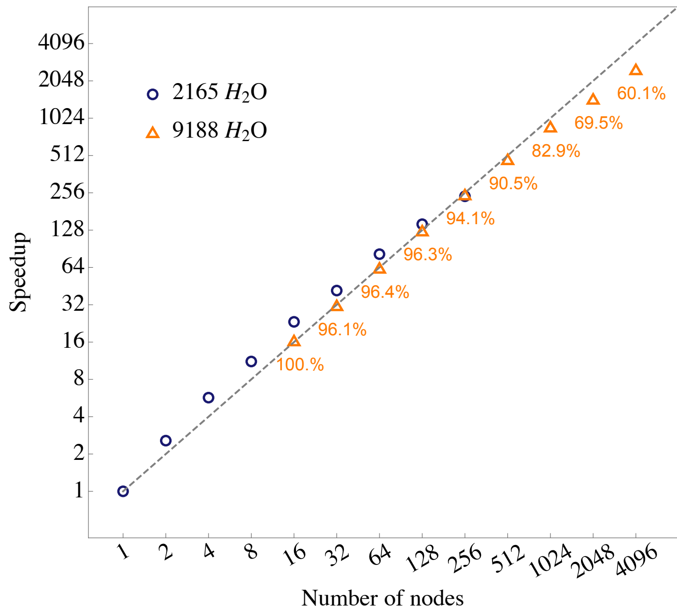

Fig. 5. Strong scaling (log-log scale) of the fragmentation-based Fock build code for two different water clusters from one up to 4096 nodes on Summit (89% of the machine). The numbers labeled in orange are the parallel efficiencies of the 9188 H_2O calculations normalized to the 16-node performance.

B. Strong scaling

Figure 5 shows the speedup of the Fock build code while running on an increasing number of Summit nodes to execute calculations on 2165-H_2O (6495-atom) and 9188-H_2O (27,564-atom) clusters using the pcSeg0 basis set, yielding a total of 28,145 and 119,444 basis functions, respectively.

In all calculations each MPI group was mapped to a single node, thereby using all of the 6 GPUs available on it.

The 2165-water cluster was split into 33 fragments with ∼195 atoms each, yielding 561 fragment pairs. Using this molecular system, the HF was strong scaled from one up to 256 nodes. The code shows a slightly superlinear strong scaling from one up to 128 nodes. At 256 nodes (1536 GPUs) the code shows a 93.8% parallel efficiency, with a 240.0× speedup with respect to the execution on one node. The 256-node calculation took 26 seconds.

The 9188-water cluster was split into 115 fragments with ∼250 atoms each, yielding 6670 fragment pairs. The 16-node calculation was used as a performance baseline as calculations on a lower number of nodes could not be executed to completion due to a 2-hour resource allocation limit on Summit. The code shows a good strong scaling performance, with a $> 80\%$ parallel efficiency up to 1024 nodes or 6144 GPUs.

The largest calculation - using 4096 nodes (24,576 V100 GPUs) - shows a 60.1% parallel efficiency, with a 154× speedup with respect to the 16-node run, or a 2463× speedup with respect to the extrapolated single node run

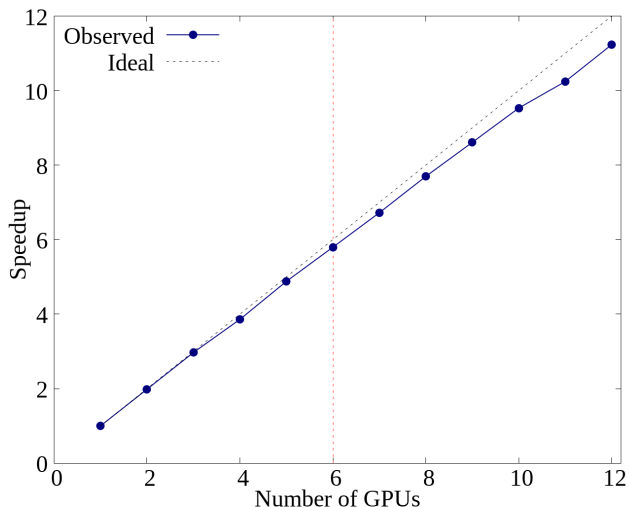

Fig. 6. Strong scaling of single-MPI-group executions with respect to the number of GPUs. Calculations were performed on a 200-water cluster (600 atoms) using the pcSeg0 basis set (2600 basis functions). The red dashed line indicates the number of GPUs used for results in Fig. 5.

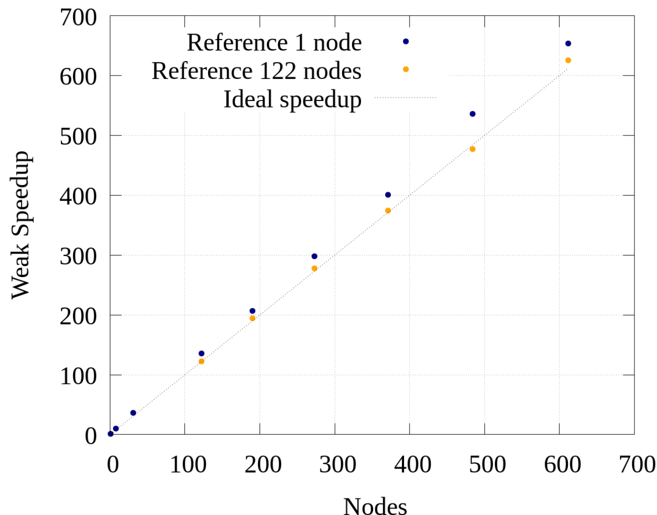

Fig. 7. Weak scaling of Fock build code from 1 to 612 nodes on Summit. Blue dots represents node weak speedups with respect to the single node execution. Yellow dots represent weak speedups with a 122-node calculation as a reference.

time. The calculation took 20.5 seconds. The performance deterioration for this calculation arises not because of increased communication between the MPI processes, but because the number of nodes becomes too large compared to the number of fragment pairs in the chosen molecular system, with each fragment pair calculation running in less than 20.5 seconds. In such circumstances, each node receives only a few (∼1.6) fragment pairs and the parallel work load becomes fragile and easily unbalanced.

Figure 6 shows the speedup of single-MPI-group executions with respect to the number of GPUs. The code used only one MPI group to perform the Fock build for a large fragment with 200 water molecules. The implementation shows an excellent scalability with respect to the number of GPUs, with the 12-GPU calculation yielding a 94% parallel efficiency with respect to the single-GPU execution. The 6-GPU configuration, which is the one adopted to obtain all the results in Fig. 5, has a ∼97% parallel efficiency.

In order to provide a baseline performance measure, we compared the execution time of our code with the QUICK GPU code [10] when using a single GPU (QUICK does not currently support multi-GPU execution). Our code showed a 2.4× and a 3.8× speedup with respect to QUICK, when executing the HF algorithm for a 200-water cluster and an 80-glycine system using the pcSeg0 basis set, respectively. We also compared our single-GPU execution time with the widely used GAMESS computational quantum chemistry package [25]–[27] running in parallel on the 21 cores of one Power9 CPU. The calculation yielded a 62.4× speedup.

C. *Weak scaling*

Figure 7 shows the weak scaling of Fock build code from one to 612 nodes on Summit. In order to obtain the weak scaling data we timed the Fock build for calculations on water cluster systems with an increasing number N_F of equal size fragments. Since we use a second order MBE the total computational workload is proportional to the square of the number of basis functions; thus, increasing the number of nodes as the square of the system size will result in an approximately constant workload across nodes. Two reference systems were chosen - one using a 72-water cluster on a single node, and one using a 925-water cluster on 122 nodes. The weak scaling efficiency and speedup can be computed using the following equations:

$$\text{efficiency}_n = \frac{\text{runtime}_r}{\text{runtime}_n}, \ \text{speedup}_n = \text{efficiency}_n * n, \quad (14)$$

where r refers to the reference system used and n is the number of nodes used.

The Fock build code achieves excellent weak scaling results. With the one-node system as a reference, the code displays superlinear scaling with the weak-scaling speedups. The largest system, with 2086 water molecules, reaches a speedup of ∼653× at 612 nodes. Using the 122-node system as a reference, the code demonstrates linear scaling, with slightly superlinear speedups for nearly every calculation involved. At 612 nodes, the weak-scaling speedup reaches ∼625×. The one case where speedup is not higher than ideal is at 484 nodes, where the speedup is ∼476×.

VIII. Conclusions

In this work, a new fragmentation-based Fock build algorithm for many-GPU architectures was presented. The

new algorithm implemented an number of innovations compared to previous GPU-based HF matrix build algorithms. Among these, a novel dynamic balancing scheme that allows to achieve high parallel efficiency in the computation and digestion of the ERIs both across threads of the same GPU and across multiple GPUs. The scheme also integrates the screening of the integrals within the dispatch of shell quartets, thereby eliminating thread divergence due to numerically insignificant integral classes. The Fock digestion step is performed with a double-stream algorithm that both takes advantage fully of permutational symmetry, and maximizes GPU throughput by completely eliminating thread synchronization requirements.

Benchmarks show that the new Fock build code demonstrates very good performance on one of the fastest supercomputers in the world, Summit. Strong-scaling calculations on a 2165-water cluster demonstrate nearly superlinear scaling up to 256 Summit nodes (or 1536 V100 GPUs), with superlinear scaling achieved for up to 128 nodes (768 V100 GPUs). Strong scaling calculations on a 9188-water cluster demonstrate good scaling up to 89% of entire Summit machine, with a 60.1% efficiency on 4096 Summit nodes (24,576 V100 GPUs). Finally, weak-scaling calculations on various-sized water clusters up to 612 nodes showed efficiencies of over 100% and more-than-ideal speedups (up to 650×) for nearly all weak-scaling calculations performed. Single-GPU benchmarks against the QUICK quantum chemistry code running on GPU show speedups between 2.4× and 3.8×, while showing a speedup of 62.4× against the GAMESS software package running in parallel on the 21 cores of a single P9 CPU.

Overall, the new Fock build algorithm performs very well on Summit.

IX. Acknowledgements

This research was supported by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration. C. Bertoni was supported by the Argonne Leadership Computing Facility, which is a U.S. Department of Energy Office of Science User Facility operated under contract DE-AC02-06CH11357.

References

[1] A. Szabo and N. S. Ostlund, *Modern quantum chemistry: introduction to advanced electronic structure theory.* Mineola, N.Y: Dover Publications, 1996.

[2] M. S. Gordon, G. Barca, S. S. Leang, D. Poole, A. P. Rendell, J. L. Galvez Vallejo, and B. Westheimer, "Novel computer architectures and quantum chemistry," *The Journal of Physical Chemistry A*, 05 2020. [Online]. Available: https://doi.org/10.1021/acs.jpca.0c02249

[3] M. S. Gordon, D. G. Fedorov, S. R. Pruitt, and L. V. Slipchenko, "Fragmentation Methods: A Route to Accurate Calculations on Large Systems," *Chemical Reviews*, vol. 112, no. 1, pp. 632–672, jan 2012. [Online]. Available: https://doi.org/10.1021/cr200093j

[4] K. Yasuda, "Two-electron integral evaluation on the graphics processor unit," *Journal of Computational Chemistry*, vol. 29, no. 3, pp. 334–342, feb 2008. [Online]. Available: http://doi.wiley.com/10.1002/jcc.20779

[5] I. S. Ufimtsev and T. J. Martínez, "Quantum Chemistry on Graphical Processing Units. 1. Strategies for Two-Electron Integral Evaluation," *Journal of Chemical Theory and Computation*, vol. 4, no. 2, pp. 222–231, feb 2008. [Online]. Available: https://pubs.acs.org/doi/10.1021/ct700268q

[6] I. S. Ufimtsev and T. J. Martinez, "Quantum Chemistry on Graphical Processing Units. 2. Direct Self-Consistent-Field Implementation," *Journal of Chemical Theory and Computation*, vol. 5, no. 4, pp. 1004–1015, apr 2009. [Online]. Available: https://pubs.acs.org/doi/10.1021/ct800526s

[7] G. Shi, V. Kindratenko, I. Ufimtsev, and T. Martinez, "Direct self-consistent field computations on GPU clusters," in *Proceedings of the 2010 IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2010*, 2010, pp. 1–8.

[8] N. Luehr, I. S. Ufimtsev, and T. J. Martínez, "Dynamic Precision for Electron Repulsion Integral Evaluation on Graphical Processing Units (GPUs)," *Journal of Chemical Theory and Computation*, vol. 7, no. 4, pp. 949–954, apr 2011. [Online]. Available: https://pubs.acs.org/doi/10.1021/ct100701w

[9] A. Asadchev and M. S. Gordon, "New Multithreaded Hybrid CPU/GPU Approach to Hartree–Fock," *Journal of Chemical Theory and Computation*, vol. 8, no. 11, pp. 4166–4176, nov 2012. [Online]. Available: https://pubs.acs.org/doi/10.1021/ct300526w

[10] Y. Miao and K. M. Merz, "Acceleration of Electron Repulsion Integral Evaluation on Graphics Processing Units via Use of Recurrence Relations," *Journal of Chemical Theory and Computation*, vol. 9, no. 2, pp. 965–976, feb 2013. [Online]. Available: https://pubs.acs.org/doi/10.1021/ct300754n

[11] T. Yoshikawa and H. Nakai, "Linear-scaling self-consistent field calculations based on divide-and-conquer method using resolution-of-identity approximation on graphical processing units," *Journal of Computational Chemistry*, vol. 36, no. 3, pp. 164–170, jan 2015. [Online]. Available: http://doi.wiley.com/10.1002/jcc.23782

[12] K. Yasuda and H. Maruoka, "Efficient calculation of two-electron integrals for high angular basis functions," *International Journal of Quantum Chemistry*, vol. 114, no. 9, pp. 543–552, may 2014. [Online]. Available: http://doi.wiley.com/10.1002/qua.24607

[13] S. Choi, O.-K. Kwon, J. Kim, and W. Y. Kim, "Performance of heterogeneous computing with graphics processing unit and many integrated core for hartree potential calculations on a numerical grid," *Journal of Computational Chemistry*, vol. 37, no. 24, pp. 2193–2201, sep 2016. [Online]. Available: http://doi.wiley.com/10.1002/jcc.24443

[14] J. Kalinowski, F. Wennmohs, and F. Neese, "Arbitrary Angular Momentum Electron Repulsion Integrals with Graphical Processing Units: Application to the Resolution of Identity Hartree–Fock Method," *Journal of Chemical Theory and Computation*, vol. 13, no. 7, pp. 3160–3170, jul 2017. [Online]. Available: https://pubs.acs.org/doi/10.1021/acs.jctc.7b00030

[15] F. Jensen, "Segmented contracted basis sets optimized for nuclear magnetic shielding," *Journal of Chemical Theory and Computation*, vol. 11, no. 1, pp. 132–138, 01 2015. [Online]. Available: https://doi.org/10.1021/ct5009526

[16] J. L. Whitten, "Coulombic potential energy integrals and approximations," *The Journal of Chemical Physics*, vol. 58, no. 10, pp. 4496–4501, 1973. [Online]. Available: https://doi.org/10.1063/1.1679012

[17] P. M. Gill, “Molecular integrals over gaussian basis functions,” ser. Advances in Quantum Chemistry, J. R. Sabin and M. C. Zerner, Eds. Academic Press, 1994, vol. 25, pp. 141 – 205. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0065327608600192

[18] G. M. J. Barca and P. M. W. Gill, “Two-electron integrals over gaussian geminals,” *Journal of Chemical Theory and Computation*, vol. 12, no. 10, pp. 4915–4924, 10 2016. [Online]. Available: https://doi.org/10.1021/acs.jctc.6b00770

[19] G. M. J. Barca and P.-F. Loos, “Three- and four-electron integrals involving gaussian geminals: Fundamental integrals, upper bounds, and recurrence relations,” *The Journal of Chemical Physics*, vol. 147, no. 2, p. 024103, 2017. [Online]. Available: https://doi.org/10.1063/1.4991733

[20] M. Head-Gordon and J. A. Pople, “A method for two-electron gaussian integral and integral derivative evaluation using recurrence relations,” *The Journal of Chemical Physics*, vol. 89, no. 9, pp. 5777–5786, 1988. [Online]. Available: https://doi.org/10.1063/1.455553

[21] P. M. Gill, B. G. Johnson, and J. A. Pople, “Two-electron repulsion integrals over Gaussian s functions,” *International Journal of Quantum Chemistry*, vol. 40, no. 6, pp. 745–752, dec 1991. [Online]. Available: http://doi.wiley.com/10.1002/qua.560400604

[22] S. Obara and A. Saika, “Efficient recursive computation of molecular integrals over cartesian gaussian functions,” *The Journal of Chemical Physics*, vol. 84, no. 7, pp. 3963–3974, 1986. [Online]. Available: https://doi.org/10.1063/1.450106

[23] ——, “General recurrence formulas for molecular integrals over cartesian gaussian functions,” *The Journal of Chemical Physics*, vol. 89, no. 3, pp. 1540–1559, 1988. [Online]. Available: https://doi.org/10.1063/1.455717

[24] P. M. W. Gill, M. Head-Gordon, and J. A. Pople, “An efficient algorithm for the generation of two-electron repulsion integrals over gaussian basis functions,” *International Journal of Quantum Chemistry*, vol. 36, no. S23, pp. 269–280, jun 1989. [Online]. Available: http://doi.wiley.com/10.1002/qua.560360831

[25] M. W. Schmidt, K. K. Baldridge, J. A. Boatz, S. T. Elbert, M. S. Gordon, J. H. Jensen, S. Koseki, N. Matsunaga, K. A. Nguyen, S. Su, T. L. Windus, M. Dupuis, and J. A. Montgomery, “General atomic and molecular electronic structure system,” *Journal of Computational Chemistry*, vol. 14, no. 11, pp. 1347–1363, 1993. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/jcc.540141112

[26] M. S. Gordon and M. W. Schmidt, “Chapter 41 - advances in electronic structure theory: Gamess a decade later,” in *Theory and Applications of Computational Chemistry*, C. E. Dykstra, G. Frenking, K. S. Kim, and G. E. Scuseria, Eds. Amsterdam: Elsevier, 2005, pp. 1167 – 1189.

[27] G. M. J. Barca, C. Bertoni, L. Carrington, D. Datta, N. De Silva, J. E. Deustua, D. G. Fedorov, J. R. Gour, A. O. Gunina, E. Guidez, T. Harville, S. Irle, J. Ivanic, K. Kowalski, S. S. Leang, H. Li, W. Li, J. J. Lutz, I. Magoulas, J. Mato, V. Mironov, H. Nakata, B. Q. Pham, P. Piecuch, D. Poole, S. R. Pruitt, A. P. Rendell, L. B. Roskop, K. Ruedenberg, T. Sattasathuchana, M. W. Schmidt, J. Shen, L. Slipchenko, M. Sosonkina, V. Sundriyal, A. Tiwari, J. L. Galvez Vallejo, B. Westheimer, M. Włoch, P. Xu, F. Zahariev, and M. S. Gordon, “Recent developments in the general atomic and molecular electronic structure system,” *The Journal of Chemical Physics*, vol. 152, no. 15, p. 154102, 2020. [Online]. Available: https://doi.org/10.1063/5.0005188

Appendix: Artifact Description/Artifact Evaluation

SUMMARY OF THE EXPERIMENTS REPORTED

All calculation and scalability benchmarks for code were run on the Oak Ridge Summit system. The code was built using gcc/6.4.0, cuda/9.2.148 and IBM spectrum MPI 10.3.1.2. The code was run using the corresponding libraries.

We ran strong scaling benchmarks for the Fock build code from 1 up to 4096 nodes of Summit. For each node, we used all the 6 NVIDIA Volta V100 GPUs available. We also ran benchmarks for strong scaling the code with respect to the number of GPUs within a single MPI group as described in section VII-B of the manuscript.

We ran weak scaling benchmarks for the Fock build code from 1 up to 612 nodes of Summit.

We ran several optimization benchmarks, which are described in detail in Section VI of the manuscript.

ARTIFACT AVAILABILITY

Software Artifact Availability: Some author-created software artifacts are NOT maintained in a public repository or are NOT available under an OSI-approved license.

Hardware Artifact Availability: There are no author-created hardware artifacts.

Data Artifact Availability: There are no author-created data artifacts.

Proprietary Artifacts: None of the associated artifacts, author-created or otherwise, are proprietary.

Author-Created or Modified Artifacts:

```
Persistent ID:
↪  https://zenodo.org/record/3877610\#.XtrR0i2r2\_s
Artifact name: SC20 Summit Fock Build
Citation of artifact: 10.5281/zenodo.3877610
```

BASELINE EXPERIMENTAL SETUP, AND MODIFICATIONS MADE FOR THE PAPER

Relevant hardware details: Summit system

Operating systems and versions: Linux Red Hat 7.6

Compilers and versions: gcc/6.4.0

Applications and versions: nvprof/9.2.148

Libraries and versions: cuda/9.2.148 and IBM spectrum MPI 10.3.1.2

ARTIFACT EVALUATION

Verification and validation studies: All benchmarks were performed at least in triplicates. The accuracy and exactness of the code were verified by computing the Hartree-Fock energy of the systems involved using the GAMESS program and cross-checking agreement at very high precision (nanoHartree level). All integrals were also cross-checked against reference values obtained GAMESS with an agreement within 10^{-12}. Finally, the code uses Ctest to automatically verify at build time the agreement with benchmark suite of high precision reference energies and two-electron integral values.

Accuracy and precision of timings: As above mentioned, all benchmarks were performed at least in triplicates using C++ high-precision timers.

Used manufactured solutions or spectral properties: N/A

Quantified the sensitivity of results to initial conditions and/or parameters of the computational environment: During our benchmarks and validation studies, we occasionally observed variations in the timings always of the order of microseconds. All our results are in seconds and hence not affected by this kind of computational environment oscillations. We did not experience hardware faults during our large runs.

Controls, statistics, or other steps taken to make the measurements and analyses robust to variability and unknowns in the system. N/A