SERI: High-Throughput Streaming Acceleration of Electron Repulsion Integral Computation in Quantum Chemistry using HBM-based FPGAs

Philip Stachura*, Guanyu Li*, Xin Wu†‡, Christian Plessl†‡, Zhenman Fang*
*Simon Fraser University, Burnaby, BC, Canada †Paderborn Center for Parallel Computing,
‡Department of Computer Science, Paderborn University, Warburger Str. 100, 33098 Paderborn, Germany
Email: {philip_stachura, guanyu_li, zhenman}@sfu.ca, {xin.wu, christian.plessl}@uni-paderborn.de

***Abstract*—The computation of electron repulsion integrals (ERIs) is a key component for quantum chemical methods. The intensive computation and bandwidth demand for ERI evaluation presents a significant challenge for quantum-mechanics-based atomistic simulations with hybrid density functional theory: due to the tens of trillions of ERI computations in each time step, practical applications are usually limited to thousands of atoms.**

In this work, we propose SERI, a high-throughput streaming accelerator for ERI computation on HBM-based FPGAs. In contrast to prior buffer-based designs, SERI proposes a novel streaming architecture to address the on-chip buffer limitation and the floorplanning challenge, and leverages the high-bandwidth memory to overcome the bandwidth bottleneck in prior designs. Moreover, to meet the varying computation, bandwidth, and floorplanning requirements between the 55 canonical quartet classes in ERI calculation, we design an automation tool, together with an accurate performance model, to automatically customize the architecture and floorplanning strategy for each canonical quartet class to maximize their throughput. Our performance evaluation on the AMD/Xilinx Alveo U280 FPGA board shows that, SERI achieves an average speedup of 9.80x over the previous best-performing FPGA design, a 3.21x speedup over a 64-core AMD EPYC 7713 CPU, and a 15.64x speedup over an Nvidia A40 GPU. It reaches a peak throughput of 23.8 GERIS (10^9 ERIs per second) on one Alveo U280 FPGA. SERI will be released soon at https://github.com/SFU-HiAccel/SERI.

I. Introduction

Quantum chemical methods [1], [2] determine the accuracy and predictive power of atomistic simulations based on quantum mechanics, known as *ab initio* molecular dynamics (AIMD) [3]. The hybrid density functional theory (DFT) [4] based AIMD can accurately model reactive and complex systems, which are generally inaccessible by means of classical molecular dynamics [5]. The electron repulsion integral (ERI) describes the repulsion between the densities of two electrons and the computational complexity may scale as $O(n^4)$ [1], where n is the number of electrons in the system. Moreover, tens of trillions of ERIs may have to be computed in each time step in practical simulations. Thus, the ERI computations limit the hybrid DFT-based AIMD simulations, even parallelized on supercomputers with 12000 CPU cores, to systems about one thousand atoms [6], [7].

Motivated by many applications of the AIMD simulation, dedicated libraries for the ERI computations are developed on CPUs [8]–[12], GPUs [13]–[18], and FPGAs [19], [20] for high-performance computing. Because of intrinsic data dependencies induced by multi-dimensional recurrence relations (RRs) and complex loop structures for different combinations of quantum angular momenta, the ERI computations on CPUs and GPUs suffer from inefficient vectorization [10] and underutilized parallelization [17]. In contrast, the computation and memory system on FPGAs can be customized to match the diverse data layouts and loop structures, which are essential for performant ERI computations. For large ERI classes, the latest best performing FPGA design [20] exhibits better performance and energy efficiency than the libint library [8], which is widely deployed for the ERI computations in many atomistic simulation packages [21], [22]. However, its throughput [20] is bound by 1) insufficient off-chip memory bandwidth for large quartet classes, 2) limited parallelism for small quartet classes, and 3) the overhead in its buffer-based dataflow design.

To address the off-chip bandwidth limitation in computing large ERI classes, we decide to leverage the high-bandwidth memory (HBM) on modern datacenter FPGAs, more specifically, the AMD/Xilinx Alveo U280 FPGA board. However, the distinct characteristics of Alveo FPGAs compared to that of Intel/Altera Stratix 10 FPGAs used in the prior best performing study [20] introduce new challenges for our accelerator design: 1) there are fewer on-chip memory blocks, which makes the prior buffer-based design [20] infeasible on Alveo U280; 2) there is less floating-point computing capability in DSPs, which requires a more efficient accelerator architecture; and 3) Alveo FPGAs use a multi-die design, which is more challenging for the timing closure due to die crossings.

In this paper, we present SERI, a high-throughput streaming accelerator for ERI computation on HBM-based FPGAs. The streaming architecture has the following benefits: 1) it significantly reduces the on-chip memory usage; 2) it is more friendly for floorplanning to achieve a better timing closure; and 3) it avoids the multi-cycle overhead in buffer-based dataflows to begin each iteration. To realize the streaming architecture, we segment the ERI computation into multiple stages and implement each stage as a separate FPGA kernel. The intermediates are streamed between the computation kernels using kernel-to-kernel streaming. Moreover, to address different buffer partition requirements inside two neighboring kernels—i.e., recurrence relations and Gaussian quadrature stages in ERI computation—we introduce a novel buffer permutation kernel between them. This kernel takes care of the buffer layout change without affecting the dataflow throughput and avoids excessive buffer partitioning, which would require a huge number of on-chip memory banks; and it is connected with the original two neighboring kernels using streams.

To implement a more efficient streaming architecture with lower DSP computing capabilities, we apply two major optimizations. First, we scale down the parallelism factor in non-bottlenecking dataflow stages to save DSP usage. Second, we play a balance between DSPs and LUTs to achieve a higher computation throughput. Moreover, for small ERI classes with

small loop trip counts, we further explore the parallelism among the inputs by duplicating the kernels to achieve a higher throughput. Regarding the timing closure optimizations, our streaming architecture enables us to bind smaller FPGA kernels onto each die and pipeline the long wires (especially die-crossing wires) between FPGA kernels.

Finally, since there are 55 canonical ERI quartet classes and each class may have a different computation, memory, and floorplanning demand, we also develop an automation tool, together with an accurate performance model, to automatically generate the best performing HLS design with a corresponding floorplanning strategy for each quartet class on a given FPGA.

Experimental results on the Alveo U280 FPGA demonstrate that SERI reaches a peak throughput of 23.8 GERIS (10^9 ERIs per second) for large quartet classes on one Alveo U280 FPGA. Compared to the prior best-performing FPGA design [20], on average, SERI achieves a speedup of 9.80x and a performance/watt improvement of 14.25x. Compared to state-of-the-art libint library [8] on a 64-core AMD EPYC 7713 CPU, it achieves a speedup of 3.21x and a performance/watt improvement of 15.47x. Compared to state-of-the-art libintx library [18] on an Nvidia A40 GPU, it achieves a speedup of 15.64x and a performance/watt improvement of 30.03x.

In summary, this paper makes the following contributions:

1. A high-throughput streaming architecture to accelerate ERI computations on resource-constrained HBM-based FPGAs, covering all *canonical* quartet classes up to f orbital.
2. A design automation tool, which incorporates all computation, memory, and floorplanning optimizations, to generate the best performing design for each ERI class.
3. Superior performance and performance/watt over prior best performing FPGA design and state-of-the-art CPU library.

II. Background on ERI Computation

An ERI describes the repulsion between the densities of two electrons [1], [23], one at $\mathbf{r}_1$ and the other at $\mathbf{r}_2$

$$\iint \mathrm{d}\mathbf{r}_1\,\mathrm{d}\mathbf{r}_2 \frac{g_{\mathbf{a},\mathbf{A},\alpha}(\mathbf{r}_1) g_{\mathbf{b},\mathbf{B},\beta}(\mathbf{r}_1) g_{\mathbf{c},\mathbf{C},\gamma}(\mathbf{r}_2) g_{\mathbf{d},\mathbf{D},\delta}(\mathbf{r}_2)}{|\mathbf{r}_1 - \mathbf{r}_2|} \quad (1)$$

where $g_{\mathbf{a},\mathbf{A},\alpha}(\mathbf{r}_1)$ is a normalized Cartesian Gaussian-type orbital (GTO) centered at $\mathbf{A} = [A_x, A_y, A_z]$ with orbital exponent α and angular momentum $L_a = a_x + a_y + a_z$. The 4 GTOs may locate at 4 different atomic centers. Hence, the ERI computations may exhibit quartic scaling complexity.

The most commonly used orbitals for AIMD are s, p, d, and f for $L = 0$, 1, 2, and 3, respectively [3]. Each orbital consists of $n_g = (L+1)(L+2)/2$ Cartesian GTOs, e.g., 10 GTOs for f orbital. Conventionally, $[ab|cd]$ denotes an ERI quartet class, which includes a set of integrals for all combinations of the involved GTOs, e.g., an $[ff|ff]$ class contains 10,000 ERIs. Furthermore, an $[ab|cd]$ quartet class is uniquely defined as *canonical* class, if it satisfies

$$L_a \geq L_b, L_c \geq L_d, \text{ and } n_{g_a} n_{g_b} \geq n_{g_c} n_{g_d}. \quad (2)$$

There are 55 *canonical* $[ab|cd]$ classes for up to the f orbital.

Many algorithms for the ERI computation were devised in the past decades [23]–[28]. The Rys quadrature algorithm [25] is chosen in this work for the FPGA streaming architecture design. It requires less intermediates for computing an $[ab|cd]$ quartet class, and thus can facilitate the inter-kernel communication. Furthermore, the Rys quadrature is known to be numerically stable for the ERI classes involving higher angular momentum, and one can take advantage of the single-precision floating-point operations on FPGAs.

The Rys quadrature algorithm is briefly described in Fig. 1. Interested audience are referred to the original paper [25].

Input: G_{abcd}, R_{rys}
Output: $[ab|cd]_{n\text{-bit}}$, ϵ
for each $[ab|cd]$ quartet **do**
  #1 Preparation: build $\mathbf{B}$ and $\mathbf{C}$ as coefficients for RRs
  #2 Recurrence relations (RRs):
  for $\xi \in \{x, y, z\}$, $\mu \in [1, n_{\text{rys}}]$, $l \in [1, L_d]$ **do**
    for $k \in [1, L_c + L_d]$, $j \in [1, L_b]$, $i \in [1, L_a + L_b]$ **do**
      compute $I(i, j, k, l, \mu, \xi)$ via RRs in Eqs. (3)
    end for
  end for
  #3 Gaussian quadrature:
  for $d \in [1, n_{g_d}]$, $c \in [1, n_{g_c}]$, $b \in [1, n_{g_b}]$, $a \in [1, n_{g_a}]$ **do**
    compute all $[ab|cd]$ integrals via Eq. (4)
  end for
  find b_{max}
  $\epsilon \leftarrow b_{\text{max}} \cdot (2^{n-1} - 1)^{-1}$
  #4 Compress-store: compute $[ab|cd]_{n\text{-bit}}$ via Eq. (5)
end for

Fig. 1: Pseudocode for computing the $[ab|cd]$ quartets.

For an $[ab|cd]$ quartet class, the Rys algorithm requires 4 GTOs (G_{abcd}), including atomic coordinates and orbital exponents, and the roots and weights of the Rys polynomials (R_{rys}) as inputs from the host. Then, the following stages are performed for the ERI computation.

Preparation stage: Two small auxiliary arrays $\mathbf{B} \in \mathbb{R}^{3 \times n_{\text{rys}}}$ and $\mathbf{C} \in \mathbb{R}^{6 \times n_{\text{rys}}}$ are built by using the inputs, where n_{rys} denotes the order of the Rys polynomials. $\mathbf{B}$ and $\mathbf{C}$ are used as coefficients for recurrence relations in the next stage.

Recurrence relations stage: The intermediates are arranged in a 6-dimensional array $I(i, j, k, l, \mu, \xi)$ and computed via 4 multi-dimensional recurrence relations (RRs)

$$I(i,j,k,l,\mu,\xi) \xleftarrow{\text{horizontal RR}} I(i,j,k+l,0,\mu,\xi) \quad (3a)$$

$$\xleftarrow{\text{horizontal RR}} I(i+j,0,k+l,0,\mu,\xi) \quad (3b)$$

$$\xleftarrow{\text{vertical RR}} I(i+j,0,0,0,\mu,\xi) \quad (3c)$$

$$\xleftarrow{\text{vertical RR}} I(0,0,0,0,\mu,\xi) \quad (3d)$$

starting from $I(0, 0, 0, 0, \mu, \xi)$ [1]. The indices i, j, k, and l are for the orbitals a, b, c, and d, respectively. μ enumerates the orders of the Rys polynomial and ξ represents x, y, and z-axis. The vertical RRs increase the angular momenta for a and b, while the horizontal RRs shift them from a and b to orbitals c and d, respectively. In the dimensions of orbitals there are read-after-write data dependencies for these RRs.

Gaussian quadrature stage: All ERIs of an $[ab|cd]$ quartet class are computed by Gaussian quadrature

[1] Due to complexity only high level RRs are shown here. Complete formulas can be found in the original paper [25].

$$[ab|cd] = \sum_{\mu=1}^{n_{\text{rys}}} w_\mu I_{\mu,x} I_{\mu,y} I_{\mu,z} \quad (4)$$

where w_μ is the weights of the Rys polynomial and $I_{\mu,x}$, $I_{\mu,y}$, and $I_{\mu,z}$ are shorthand for the corresponding $I(i,j,k,l,\mu,\xi)$ in the x, y, and z-axis.

Compress-store stage: Tens of trillions of ERIs are usually required in real-world AIMD simulations. To reduce such high memory demand, an ERI compression algorithm [29] is seamlessly integrated into the ERI computation.

After computing the $[ab|cd]$ quartet, the maximum absolute integral, denoted as b_{max}, needs to be found. Then, the floating-point ERIs can be represented as n-bit signed integers for ϵ, a "quantum value" for this quartet

$$[ab|cd]_{n\text{-bit}} = \texttt{ANINT}([ab|cd]/\epsilon), \quad (5)$$

where the function ANINT returns the nearest integer of its argument and the divide operation is avoided by multiplying ϵ^{-1} in our design. The $[ab|cd]_{n\text{-bit}}$ array, together with ϵ, form the outputs of the ERI computation. The maximum absolute error for the compression is bound by $\epsilon/2$ for a quartet.

III. Motivation and Challenges

A. Prior Studies and Limitations

The previous best performing FPGA design for ERI computation [20] implemented a buffer-based architecture on the Intel Stratix 10 GX 2800 FPGA to customize its on-chip memory layout and corresponding loop structures to exploit data parallelism and pipeline parallelism. In addition, it employed ERI compression to reduce the data volume and thus reduce the off-chip bandwidth requirement. It achieved a peak throughput of 11.2 GERIS. However, it has several limitations.

1. **Off-chip memory bandwidth bottleneck:** When computing large quartet classes, its overall throughput is limited by the off-chip memory bandwidth, even after the ERI compression technique is employed.
2. **Limited computation parallelism:** When computing small quartet classes with small loop trip counts, its overall throughput is limited by the amount of computation parallelism, since the design was only parallelized via the loops in the recurrence relations and Gaussian quadrature stages.
3. **Overhead in buffer-based dataflow:** Its buffer-based dataflow implementation not only consumes a large amount of on-chip memory banks, but also incurs a multi-cycle (10 cycles in [20]) overhead to begin each dataflow iteration.

B. New Challenges on AMD/Xilinx HBM-based FPGA

To address the off-chip bandwidth bottleneck for computing large quartet classes, we decide to leverage the high-bandwidth memory on modern datacenter FPGAs. Specifically, we select the AMD/Xilinx Alveo U280 FPGA board, which features an HBM with up to 460 GB/s off-chip bandwidth. However, it also presents a number of new challenges for our accelerator design due to distinct characteristics of AMD/Xilinx Alveo FPGAs compared to that of Intel/Altera Stratix 10 FPGAs.

Challenge 1: Fewer on-chip memory blocks. The ERI calculation requires a significant amount of data to be passed between calculation stages. To achieve a high throughput, a large on-chip memory bandwidth is required, especially for a buffer-based design. Unfortunately, compared to the Stratix 10 GX 2800 board, Alveo U280 is a smaller board, which provides fewer on-chip memory blocks (bandwidth).

TABLE I: Memory block comparison.

	M20K	BRAM18K	URAM
Capacity	20 Kb	18 Kb	288 Kb
Port max bit-width	40	36	72 x2 (TDP)

TABLE II: Available memory block bandwidth comparison.

	Stratix 10 GX 2800	Alveo U280	Alveo U280 (dynamic region)
Mem blocks	M20K	BRAM + (URAM)	BRAM + (URAM)
Num banks	11,721	4,032 + (960)	3,552 + (960)
Net bit-width	475K	145K + (69K)	127K + (69K)

Shown in Table I, the BRAM18K memory block on the Alveo U280 is roughly equivalent to the M20K memory block on the Stratix 10. Table II compares the total number of memory banks and aggregated block memory port-width (i.e., on-chip memory bandwidth) between the two boards. For the aggregated port bit-width, the Alveo U280 is 3.3x lower considering BRAMs, or 2.2x lower considering URAM as well, compared to the Stratix 10 board.

Due to the significantly fewer on-chip memory blocks and lower on-chip memory bandwidth, a straightforward porting of the buffer-based dataflow design [20] onto the Alveo U280 board is no longer feasible and a new architecture is needed.

Challenge 2: Lower floating-point computing capability in DSPs. Furthermore, the Stratix 10 GX 2800 features hardened DSP blocks, which supports $5,760$ simultaneous single-precision floating-point multiplications (fmuls). However, for the Alveo U280, the default floating-point multiplication implementation requires 3 DSP slices, resulting in a capacity of $9,024/3 = 3,008$ fmuls, only 52% of that in the Stratix 10 board. This requires a more efficient architecture with less computing resource available.

Challenge 3: More challenging timing closure. Unlike the Stratix 10 GX 2800 board that uses a monolithic die for its programmable logic region, the Alveo U280 is comprised of multiple dies called super logic regions (SLRs). These dies are stitched together using super long lines (SLLs). However, in addition to limited SLL connections between the dies, there is also a timing penalty for wires which cross the boundaries. Thus, it is difficult to place & route large kernels across multiple SLRs, often resulting in timing closure failure or poor achieved frequencies. This is especially more challenging for buffer-based designs that would result in more SLR crossings.

IV. SERI Design and Implementation

A. Overall Architecture and Novelties

SERI addresses the previous design limitations and new challenges with the following novelties.

A streaming architecture on an HBM-based FPGA. To address the challenge of insufficient on-chip memory blocks and the overhead in buffer-based dataflow design, we propose a streaming architecture design by avoiding buffers wherever

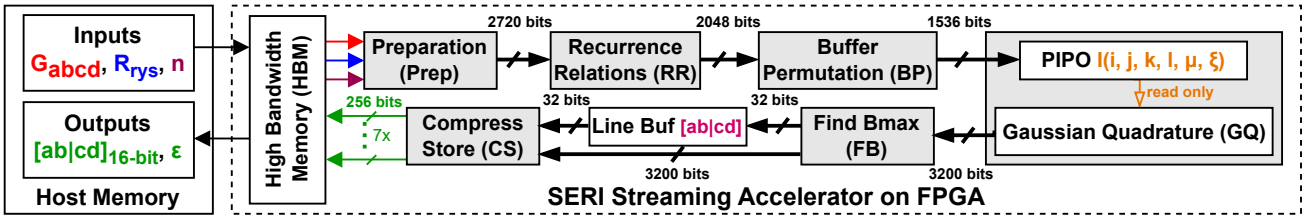

Fig. 2: Design of our SERI streaming FPGA accelerator; stream bit-widths shown for $[ff|ff]$, the largest ERI quartet class.

possible in favour of streams. Moreover, the streaming architecture also reduces the connectivity complexity between different dataflow stages, resulting in less routing congestion and better timing closure. Fig. 2 shows the overall design of SERI's streaming architecture for the largest $[ff|ff]$ class.

After inputs are loaded from host memory onto the FPGA-side HBM—HBM is leveraged to address the off-chip memory bandwidth bottleneck—the FPGA is invoked to compute n number of quartets for a given molecule. Each stage of the ERI computation shown in Fig. 1 is implemented as a separate Vitis HLS kernel, which are connected by kernel-to-kernel streaming connections. The major buffer used in the design is in the GQ (Gaussian quadrature) kernel, which is implemented as a Ping-Pong buffer (PIPO) of the 6-dimensional intermediates array (I) to enable uninterrupted processing of the data. Additionally, as ERI values are streamed through the find $b_{\max}$ stage, they are stored temporarily in a line buffer (implemented using a large-depth FIFO) until $b_{\max}$ is calculated and the compress-store (CS) stage can start processing them. The outputs of the CS stage—which include the compressed 16-bit ERIs along with the corresponding ϵ values needed for decompression—are written consecutively to the HBM through multiple concurrent ports to avoid the off-chip bandwidth bottleneck.

Computation efficiency optimizations. To address the limited computation parallelism issue in small to medium sized quartets, we explore the input parallelism of the design by duplicating multiple copies for different inputs. To address the limited DSP computing capability challenge, we maximize resource reuse in non-bottleneck dataflow kernels to reduce their DSP usage, and trade-off DSPs for LUTs where possible.

Buffer permutation support. To further reduce the number of on-chip memory blocks required by the major 6-dimensional intermediates buffer used in the GQ stage, we introduce a novel buffer permutation (BP) stage between the recurrence relations (RR) and GQ stages. The BP stage is designed to automatically change the layout of the intermediates buffer to meet different buffer partitioning requirements from both RR and GQ stages using the minimum number of on-chip memory banks, while not affecting the dataflow throughput.

Design automation. Finally, we develop a design automation tool, together with an accurate performance model, to automatically optimize the flexible SERI architecture for each of the 55 canonical quartet classes, which have varying computation, memory, and floorplanning requirements. Moreover, to address the timing closure difficulties on the Alveo U280, SERI automatically places each kernel within an SLR to avoid routing through SLLs, and automatically splits a large kernel into two smaller copies to fit into an SLR. Inter-kernel communication is done through (pipelined) streaming connections, allowing for a large amount of data to be continuously transferred over the die boundaries without any performance overhead.

B. Design of Each Streaming Stage

1) Preparation (Prep): The prep stage receives two sets of primary input arguments, 4 GTOs (G_{abcd}) and the roots and weights of the Rys polynomial (R_{rys}), plus n, the number of quartets for a given molecule. This stage is always the smallest relative to the others, allowing it to be unrolled when needed. As such its latency can be tuned to match that of the slowest stage to allow for more resource reuse and leave resources for more complex downstream stages.

2) Recurrence Relations (RR): This stage processes the 6-fold loops over the dimensions $\xi\mu lkji$ as defined in Fig. 1. The k, j, i dimensions are always fully unrolled, with the remaining ξ, μ, and l dimensions being flattened and pipelined. Depending on the resource availability, further flexible unrolling by a RR unroll factor (U_{rr}) can be applied, which is controlled by our design automation tool. The base number of cycles for an iteration to complete (when $U_{rr} = 1$) is defined as $n_{Brr} = 3 \times n_{\text{rys}} \times (L_d + 1)$. The actual number of cycles with additional flexible unrolling becomes $n_{rr} = \text{ceil}(n_{Brr}/U_{rr})$.

The output stream bit-width is defined by $U_{rr} \times (L_c + 1) \times (L_b + 1) \times (L_a + 1) \times 32$ and produces a value each cycle.

A design parameter `RR_SPLIT` is introduced, which allows dividing the RR computation into multiple kernels to avoid a high degree of clustering that impacts the timing closure.

3) Gaussian Quadrature (GQ): This stage computes all ERIs for an $[ab|cd]$ quartet using Eq. (4). The dimensions a and b as defined in Fig. 1, μ and ξ as defined in Eq. (4), are always fully unrolled, with the remaining c and d being flattened and pipelined. The access to $I_{\mu,\xi}$, where $\mu \in [1, n_{\text{rys}}]$ and $\xi \in \{x, y, z\}$, presents irregular patterns and does not match the order in which $I_{\mu,\xi}$ are produced in the RR stage. Therefore, a PIPO with depth=2 (i.e., double buffer) is used to overlap the buffer loading and GQ calculation time.

For large quartet classes, this stage quickly becomes bottlenecked by the available resources (primarily DSPs) within a single SLR. To solve this issue, the design parameter `GQ_SPLIT` is introduced, which allows the GQ calculations to be divided into 2 separate HLS kernels GQ-A and GQ-B. Each kernel requires its own complete copy of the $I(i, j, k, l, \mu, \xi)$ buffer to accommodate the complex access pattern. As GQ-A receives data from the RR, in addition to storing it in its local PIPO, it also relays the data to GQ-B through kernel-to-kernel

streams. Both GQ-A and GQ-B then produce outputs directly to the find $b_{\max}$ kernel.

Additionally, to optimize for timing closure, the ξ dimension of $I(i, j, k, l, \mu, \xi)$ is manually unrolled into multiple arrays. This encourages Vivado to spread them across the board, significantly reducing congestion and improving timing closure.

The number of cycles to complete an iteration of GQ is defined as $n_{gq} = n_{g_c} \times n_{g_d}$. The outputs of the GQ kernel(s) are continuously streamed to the next stage with a net bit-width of $n_{g_b} \times n_{g_a} \times n_{\text{rys}} \times 3 \times 32$.

For smaller quartet classes, while there are enough FPGA resources to increase the compute parallelism, there are insufficient trip counts in the c and d dimensions (e.g., $c = d = 1$ in the $[ff|ss]$ quartet) for further unrolling. Instead, SERI increases the input parallelism of the design by placing multiple copies of this entire accelerator (all stages) on the FPGA to work for multiple input molecules. The input parallelism of the design is controlled by the design parameter `INP_PAR`.

4) Buffer Permutation (BP): For the intermediates buffer $I(i, j, k, l, \mu, \xi)$, as explained in Sections IV-B2 and IV-B3, in each cycle, while RR concurrently writes to the i, j, k dimensions (and potentially l, μ, and ξ dimensions with $U_{rr} > 1$), GQ concurrently reads from the i, j, μ, and ξ dimensions. To satisfy both stages, the buffer has to be almost completely partitioned along all dimensions. This is too costly for on-chip memory consumption and routing complexity, especially for medium to large quartet classes. The huge bit-width access and low depth do not allow an effective use of BRAMs/URAMs, while using LUTRAMs for such a large buffer fails to achieve timing closure due to high congestion levels.

Therefore, we introduce a BP stage to reduce the buffer partition and routing complexity of the PIPO between the RR and GQ stages. This BP stage uses an array of FIFOs to re-arrange the order of data to allow both RR and GQ to emit and consume data in their ideal fashion with a relatively small resource overhead. Since it is added in parallel with the other stages in the dataflow, the dataflow throughput is not affected as long as its latency is not larger than the other stages (i.e., RR and GQ stages). Further details on its design and implementation are discussed in Section IV-C.

The design parameter `BP_BYPASS` is introduced, which will remove the BP stage and have RR feeding directly into GQ when it is enabled. This is used for quartets that end in $|ss]$ as the relevant k and l dimensions are always 1. It means that $I(i, j, 1, 1, \mu, \xi)$ will be fully partitioned anyway, removing the need for buffer permutation.

5) Find b_{max} (FB): To compress the ERIs, the $b_{\max}$ value needs to be found for the given iteration's data. This FB stage compares the ERIs that are being streamed from the GQ step and passes them to an external line buffer (implemented using a large-depth FIFO), so that they can be buffered until all ERIs have been checked and $b_{\max}$ has been found. The $b_{\max}$ value is then sent to the compress-store stage.

As find $b_{\max}$ just passes through values from the previous stage, the output stream bit-width for the ERIs is the same ($n_{g_b} \times n_{g_a} \times n_{\text{rys}} \times 3 \times 32$), and the stream to send $b_{\max}$ is 32 bits. The number of cycles to complete an iteration is also the same as that of GQ: $n_{fb} = n_{gq} = n_{g_c} \times n_{g_d}$.

6) Compress-Store (CS): Once the CS stage receives a $b_{\max}$ value, it begins to drain the external line buffer and compress the 32-bit floats into 16 bits as given in Eq. (5). The resulting compressed ERIs are written continuously to the HBM over a number of 256-bit AXI ports via burst transactions. The design parameter `NUM_ERIS_PORTS` controls the number of AXI ports used and is balanced by the design automation tool to ensure that sufficient off-chip bandwidth is provided.

The 32-bit ϵ value is also sent through these AXI ports in the spare bandwidth, since the available bandwidth of the combined AXI port-width always slightly exceeds the number of bits needed to be written in each cycle. As the CS stage stores values at the same rate it receives them, the number of cycles to complete a dataflow iteration is the same as the preceding stages: $n_{cs} = n_{fb} = n_{gq} = n_{g_c} \times n_{g_d}$.

C. Buffer Permutation

When a computing stage S uses a buffer B, it has buffer partitioning requirements. Conceptually, the buffer B can be divided into two sets of dimensions: 1) a set of fully partitioned dimensions, P, and 2) a set of non-partitioned dimensions, D, representing the buffer depth of each partition. Note a partially partitioned dimension can always be decomposed into a fully partitioned dimension and a non-partitioned dimension.

If two stages S_1 and S_2 access the same buffer, each views the buffer from its own perspective as P_1, D_1 and P_2, D_2. For example, for the RR stage, $P_1 = \{i, j, k\}$ and $D_1 = \{l, \mu, \xi\}$; for simplicity of writing, we assume $U_{rr} = 1$ and it works similarly for $U_{rr} > 1$. While for the GQ stage, $P_2 = \{i, j, \mu, \xi\}$ and $D_2 = \{k, l\}$. Unfortunately, this causes a conflict as $P_1 \neq P_2$. The naive approach of resolving this conflict is to partition the buffer such that $P_{\text{buffer}} = P_1 \cup P_2 = \{i, j, k, \mu, \xi\}$, resulting in excessive usage of on-chip memory blocks and complex routing, as explained in Section IV-B4.

Instead, we propose a novel generic buffer permutation (BP) stage to address this issue. Before explaining how BP works in detail, we first define the following dimension sets.

$$PP = P_1 \cap P_2 = \{i, j\} \quad \text{always partitioned} \tag{6}$$
$$DD = D_1 \cap D_2 = \{l\} \quad \text{always depth} \tag{7}$$
$$DP = D_1 - DD = \{\mu, \xi\} \quad \text{depth to partitioned} \tag{8}$$
$$PD = P_1 - PP = \{k\} \quad \text{partitioned to depth} \tag{9}$$

We also define the length function of a dimension set, e.g., $\text{len}(P)$ or $\text{len}(D)$, to be the total number of elements in those dimensions, i.e., width of a P set or depth of a D set.

The buffer permutation flow is illustrated in Fig. 3, which is composed of N FIFOs, with the following FIFO properties.

$$\text{in\_width} = \text{len}(P_1) = \text{len}(\{i, j, k\}) \tag{10}$$
$$\text{out\_width} = \text{len}(PP) = \text{len}(\{i, j\}) \tag{11}$$
$$\begin{aligned} \text{in\_depth} &= 2 \times \text{len}(DD) \times \lceil \text{len}(DP)/N \rceil \\ &= 2 \times \text{len}(\{l\}) \times \lceil \text{len}(\{\mu, \xi\})/N \rceil \end{aligned} \tag{12}$$

Basically, it reads consecutively from RR buffer's (actually implemented in streams) fully partitioned dimensions ($P1 =$

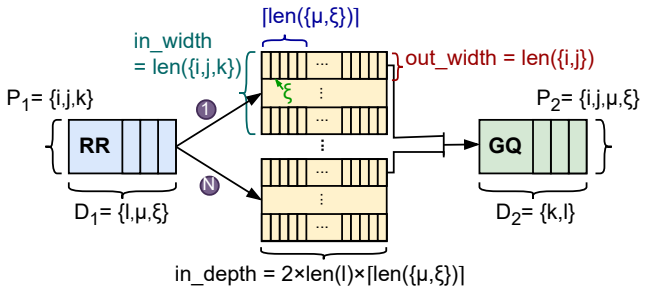

Fig. 3: Buffer permutation flow diagram.

$\{i, j, k\}$) and stores those values in the N FIFOs one after another in sequence. While this FIFO write phase can be coded such that it breaks from the loop earlier, decreasing the number of cycles, SERI opts for a different approach. It trades slightly longer trip count for significantly simpler control logic, alleviating congestion. As such, the FIFO write phase takes $T_W = \text{len}(DD) \times \lceil \text{len}(DP)/N \rceil \times N = \text{len}(\{l\}) \times \lceil \text{len}(\{\mu, \xi\})/N \rceil \times N$ cycles to complete.

Meanwhile, the FIFO read phase concurrently reads from all N FIFOs the $PP = \{i, j\}$ dimensions each cycle, and places the values in the appropriate index of GQ's buffer so that its fully partitioned dimensions become $P_2 = \{i, j, \mu, \xi\}$; the GQ stage has a PIPO buffer. This FIFO read will take $T_R = \text{len}(DD) \times \lceil \text{len}(DP)/N \rceil \times \text{len}(PD) = \text{len}(l) \times \lceil \text{len}(\{\mu, \xi\})/N \rceil \times \text{len}(k))$ cycles to complete. Note the FIFO depth is sized with twice the needed depth (Eq. (12)), so that both FIFO write and read phases can run in parallel in a dataflow fashion without stalling.

The total number of cycles the BP stage takes is defined as:

$$\begin{aligned} n_{bp} &= \max(T_W, T_R) \\ &= \max(\text{len}(l) \times \lceil \text{len}(\{\mu, \xi\})/N \rceil) \times N, \\ &\qquad \text{len}(l) \times \lceil \text{len}(\{\mu, \xi\})/N \rceil \times \text{len}(k)) \\ &= \max((L_d + 1) \times \lceil n_{\text{rys}} \times 3/N \rceil \times N, \\ &\qquad (L_d + 1) \times \lceil n_{\text{rys}} \times 3/N \rceil \times (L_c + 1)) \end{aligned} \tag{13}$$

The number of FIFOs, N, is controlled by the design parameter `BP_FIFO_COUNT`. Due to the optimization necessary to achieve reliable timing closure in the GQ stage (as discussed in Section IV-B3), N needs to be a multiple of 3.

D. Overall Performance Model

The overall throughput of our design is characterized by the achieved frequency (f_{MHz}), number of ERIs per quartet (n_{ERIQ}), and the longest number of cycles it takes among all the stages to complete calculations in the dataflow. The number of cycles it takes to complete each stage is defined as n_{rr}, n_{bp}, n_{gq}, n_{fb}, and n_{cs} in Sections IV-B and IV-C, corresponding to the RR, BP, GQ, FB, and CS stages, respectively. Note $n_{cs} = n_{fb} = n_{gq}$. The preparation stage does not have a separate count as it can be flexibly scaled to any number of cycles. The input parallelism of the design will increase the throughput of our accelerator by a factor of `INP_PAR`. Therefore, the overall throughput GERIS (10^9 ERIs per second) is calculated as:

$$\text{GERIS} = \frac{\texttt{INP\_PAR} \cdot f_{\text{MHz}} \cdot n_{\text{ERIQ}}}{10^3 \cdot \max(n_{rr}, n_{bp}, n_{gq})} \tag{14}$$

E. Design Automation Tool

In order to facilitate the design of kernels that span a large range of 55 canonical quartet classes, a design automation tool is created to automatically adjust the design parameters of the flexible architecture to scale the design. Given the target quartet class $[ab|cd]$ and desired input parallelism `INP_PAR` (optional), it will produce a tailored FPGA design and build the bitstream. The automation tool selects the canonical quartets as they have the largest angular momenta for the a and b orbitals, which results in the highest degree of consistent compute parallelism, optimizing for the highest throughput.

Our tool automatically chooses the following parameters `RR_SPLIT`, `GQ_SPLIT`, `BP_BYPASS`, `BP_FIFO_COUNT`, and `NUM_ERIS_PORTS`, as defined in Fig. 4, Eqs. (15) and (16). The tool manages the selection of these parameters for a specified quartet class and ensures the correct version of each kernel is synthesised and connected properly. Lastly, when a user does not specify the input parallelism `INP_PAR`, based on the resource utilization of `INP_PAR`=1, we choose the maximum `INP_PAR` that passes the timing closure.

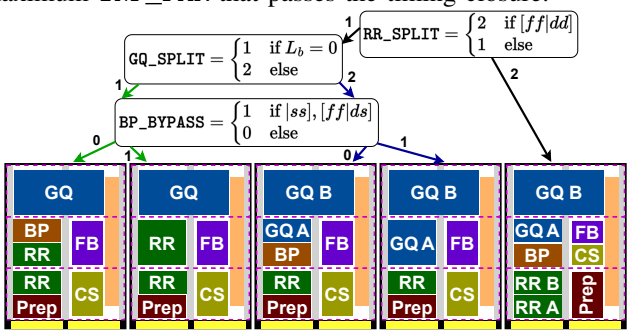

Fig. 4: Automatic floorplanning configurations.

$$\texttt{BP\_FIFO\_COUNT} = \begin{cases} 6 & \text{if } [fp|fd], [fd|fd] \\ 3 & \text{else} \end{cases} \tag{15}$$

$$\texttt{NUM\_ERIS\_PORTS} = \text{ceil}(n_{g_a} n_{g_b}/16) \tag{16}$$

Moreover, our automation tool also varies the floorplanning given the parameters to achieve better timing closure. It will assign kernels to SLR or slot regions based on the `RR_SPLIT`, `GQ_SPLIT`, and `BP_BYPASS` parameters as shown in Fig. 4. This is necessary not only because the kernel topology will change due to the design parameters, but also because the amount of relative FPGA resources that each stage consumes changes among quartet classes. For example, the RR stage takes less resources than GQ for large quartet classes; however, for smaller quartet classes they are closer to each other.

Our floorplanning configurations are shown in Fig. 4. The floorplan is designed to minimize large stream connection crossings and follows the overall pattern that data starts from the bottom left, goes up and back down in a clockwise fashion and ends back at the HBM on the bottom right. Some floorplan configurations have multiple regions for the same stage, such as RR. These get utilized when the input parallelism goes above 1, and kernel instances are distributed evenly between those regions. If there is only a single slot for a kernel type, all instances of it get placed in the same slot or SLR region when input parallelism increases.

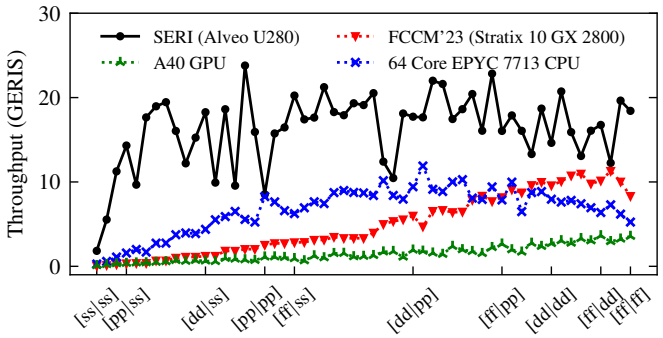

Fig. 5: Throughput comparison between SERI, best previous FPGA design, libint running on 64-core CPU, and libinx running on an Nvidia A40 GPU.

V. Results and Analysis

A. Experimental Setup

SERI is synthesized and built with Vitis HLS & Vivado 2023.2 and evaluated on an Alveo U280 FPGA with C++ host code and Xilinx runtime (XRT) version 2.16. The Benchmark molecular system consists of 16 sites, each possesses s, p, d, and f orbitals, arranged on a cubic $4 \times 2 \times 2$ lattice, with a lattice parameter of 1 Å. To accurately measure some very fast running kernels with a high input parallelism, multiple benchmark molecules are calculated in a single shot. The execution time is measured for the FPGA kernel run only and the power consumption is measured using the XRT `xbutil` command. Results are collected from the average of five runs.

On the AMD EPYC 7713 CPU (64 cores), the ERI computation is performed using the libint library [8] (version 2.7.2) parallelized with MPI (Open MPI, version 4.1.1) on all 64 cores for a runtime of about 10 seconds. At the same time, the power consumption of this CPU socket is measured by using RAPL counters [30]. The CPU benchmark is compiled with GCC 11.2.0 using `-O3 -march=znver3`.

On the Nvidia A40 GPU, the libintx library [18] is used, with power consumption collected via `nvidia-smi`.

Both CPU and GPU libraries internally use double precision. The numerical results produced by SERI are compared with the libint computation. The maximum absolute errors are about 10^{-7} – 10^{-5} Hartree (in an acceptable range), mainly caused by the 16-bit ERI compression.

All of the these experiments were carried out on the Noctua 1 & 2 supercomputers [31].

B. Overall Performance and Performance/Watt Comparison

Fig. 5 compares the throughput between SERI, the previous best performing FPGA design (FCCM'23 [20]), a 64-core AMD EPYC 7713 CPU running libint, and an Nvidia A40 GPU running libintx. SERI outperforms them for all quartets and achieves an average speedup of 9.80x against [20] on the Intel Stratix 10 GX 2800 FPGA, 3.21x compared to the 64-core CPU, and 15.64x compared to the A40 GPU. The reasons for our substantial improvement over the previous best performing FPGA design [20] are explained in Sections III-A and IV-A. The variance in SERI's achieved throughput for different quartets is mainly caused by the achieved input parallelism factor and clock frequency, as shown in Fig. 6.

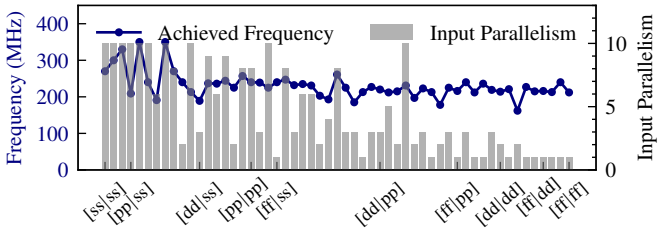

Fig. 6: Frequency and input parallelism factor achieved for each quartet class in SERI FPGA builds.

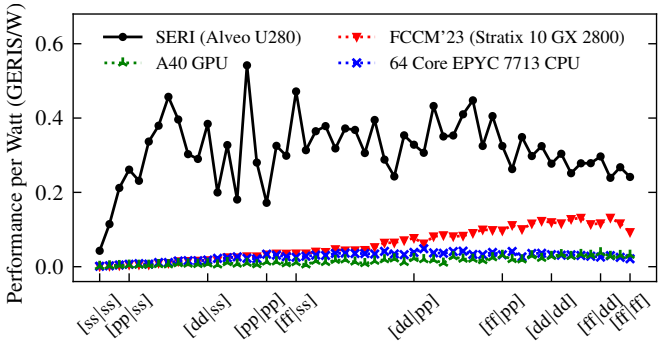

Fig. 7: Energy efficiency comparison between SERI, best previous FPGA design, libint running on 64-core CPU, and libinx running on an Nvidia A40 GPU.

Fig. 7 compares their energy efficiency. SERI achieves significantly better performance/watt, on average, 14.25x better than the prior best FPGA design [20], 15.47x better over the 64-core CPU, and 30.03x better over the A40 GPU.

C. Performance Model Accuracy and Breakdown

Fig. 8 compares the predicted throughput using our performance model against the actual achieved throughput for all the 55 canonical quartet classes, implemented with an input parallelism of 1. For the majority of quartet classes, our predicted throughput well matches with the actual throughput. For a few occasional cases, such as the $[ff|ss]$ quartet class, there is a relatively larger variance. This is because their pipeline loop trip counts are very low, and there is one clock cycle overhead in the design (most likely caused by HBM data access), which now represents a bigger percentage.

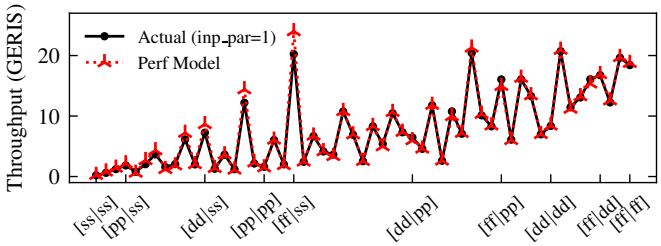

Fig. 8: Actual throughput vs. model predicted throughput using input parallelism = 1 for all canonical quartet classes.

Table III lists latencies of each streaming dataflow stage—including n_{rr} (latency of recurrence relations stage), n_{bp} (latency of buffer permutation stage), and n_{gq} (same latency for

TABLE III: Latency of each stage for sample quartet classes.

Quartet	n_{ERIQ}	n_{rr}	n_{bp}	n_{gq}
$[pp\|ss]$	9	1	-	1
$[dd\|dp]$	216	6	3	6
$[fd\|dp]$	1080	15	18	17
$[fd\|fd]$	3600	54	54	60

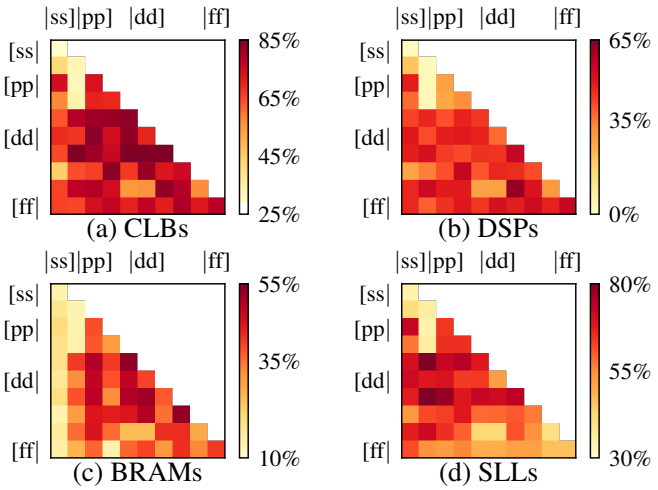

Fig. 9: Post place-and-route resource utilization.

Gaussian quadrature, find $b_{\max}$, and compress store stages)—for sample quartet classes in different ranges. As shown, our design automation can automatically choose various design parameters to balance the latencies between dataflow stages.

D. Resource Utilization

Fig. 9 shows the final post place-and-route resource utilization for each of the 55 canonical quartet classes, with the highest input parallelism designs that passed timing closure. Overall, most quartet classes utilize a significant amount of FPGA resources, especially the CLB (configurable logic block) and SLL resources, which usually limit the timing closure. Note that a larger quartet class does not always consume more resources, due to the different input parallelism factors applied for each quartet, as shown in Fig. 6.

Due to our streaming architecture design, the BRAM utilization has been significantly reduced, and does not exceed 53% of our smaller Alveo U280 FPGA board. For $|ss]$ quartets, BRAM utilization is very low, since the intermediates array is small enough to be fully partitioned, and thus the buffer permutation stage is bypassed. Due to our more efficient architecture design, we manage to use less than 65% DSPs of Alveo U280 FPGA to achieve an average of 9.80x speedup over prior best performing FPGA design [20].

Regarding the SLL utilization, quartet classes that have a larger bit-width-to-compute ratio and a higher input parallelism have a larger SLL utilization. Therefore, the SLL utilization is lower for $[ff|$ quartet classes compared to others, since: 1) they have more computation to do per quartet, 2) they use a lower stream-width to transfer data between streaming stages as they have more cycles to do the data transfer (as long as it can be hidden by the computation), and 3) they have a lower input parallelism.

VI. Related Work

Recent developments for the ERI computations in quantum chemistry mostly target fixed computer architectures, e.g., CPUs [8], [9], [12] and GPUs [16], [17]. As explained in Introduction, ERI computations on CPUs and GPUs usually suffer from inefficient vectorization [10] and underutilized parallelization [17]. A quantitative comparison to state-of-the-art CPU and GPU libraries is presented in Section V-B.

Many studies have demonstrated that classical molecular dynamics simulations can benefit from the flexible configurability of FPGAs [32]–[35]. On the other hand, the FPGA-accelerated AIMD simulations are largely unexplored. An early work from Kindratenko et al. reported the computation of $[ss|ss]$ on FPGAs using the SRC MAP C compiler [19]. Recently, the ERI computation for generic $[ab|cd]$ classes up to f orbital was performed by Wu et al. [20], for which we have a quantitative comparison to in Section V-B.

VII. Conclusion and Future Work

In this work, we have presented SERI, a high-throughput streaming accelerator for the ERI computation in quantum chemistry using HBM-based FPGAs. This new streaming design overcomes several drawbacks in the prior best performing FPGA kernels on the Intel Stratix 10 GX 2800: 1) the on-chip memory consumption is reduced dramatically by means of streaming intermediates between the computation kernels; 2) the multi-cycle overhead for each dataflow iteration in the earlier buffer-based design is eliminated, and 3) better timing closure and floorplanning are also achieved across multiple SLRs. Furthermore, to optimize the diverse computation, memory, and floorplanning requirements, we have developed an automation tool for designing individual FPGA kernels for all 55 canonical $[ab|cd]$ quartet classes to achieve high-throughput for the ERI computations. Our evaluation for a synthetic molecular system on the AMD/Xilinx Alveo U280 FPGA demonstrates that, SERI achieves an average speedup of 9.80x and an average performance/watt improvement of 14.25x than the earlier buffer-based designs on Intel Stratix 10 GX 2800 FPGAs. Compared to the libint library on the 64-core AMD EPYC 7713 CPU, SERI achieves an average speedup of 3.21x and an average performance/watt improvement of 15.47x. In future work, we plan to explore large AIMD simulations with multi-FPGAs, and integrate with existing AIMD simulation software.

Acknowledgements

We acknowledge the partial support from NSERC Discovery Grant RGPIN-2019-04613, DGECR-2019-00120, Alliance Grant ALLRP-552042-2020; CFI John R. Evans Leaders Fund and BC Knowledge Development Fund. We also thank the computing time provided to us on the high-performance computers Noctua 1 & 2 at the NHR Center PC^2. These are funded by the German Federal Ministry of Education and Research and the state governments participating on the basis of the resolutions of the GWK for the national high-performance computing at universities (www.nhr-verein.de/unsere-partner).

REFERENCES

[1] J. A. Pople, "Nobel Lecture: Quantum chemical models," *Reviews of Modern Physics*, vol. 71, pp. 1267–1274, Oct 1999.

[2] W. Kohn, "Nobel Lecture: Electronic structure of matter – wave functions and density functionals," *Reviews of Modern Physics*, vol. 71, pp. 1253–1266, Oct 1999.

[3] J. Hutter, M. Iannuzzi, and T. D. Kühne, "Ab initio molecular dynamics: A guide to applications," in *Comprehensive computational chemistry (First Edition)*, M. Yáñez and R. J. Boyd, Eds. Oxford: Elsevier, 2024, pp. 493–517.

[4] A. D. Becke, "A new mixing of Hartree–Fock and local density-functional theories," *The Journal of Chemical Physics*, vol. 98, no. 2, pp. 1372–1377, 01 1993.

[5] R. Iftimie, P. Minary, and M. E. Tuckerman, "Ab initio molecular dynamics: Concepts, recent developments, and future trends," *Proceedings of the National Academy of Sciences*, vol. 102, no. 19, pp. 6654–6659, 2005.

[6] S. V. Levchenko, X. Ren, J. Wieferink, R. Johanni, P. Rinke, V. Blum, and M. Scheffler, "Hybrid functionals for large periodic systems in an all-electron, numeric atom-centered basis framework," *Computer Physics Communications*, vol. 192, pp. 60–69, 2015.

[7] M. Redies, G. Michalicek, J. Bouaziz, C. Terboven, M. S. Müller, S. Blügel, and D. Wortmann, "Fast all-electron hybrid functionals and their application to rare-earth iron garnets," *Frontiers in Materials*, vol. 9, 2022.

[8] E. F. Valeev, "Libint: A library for the evaluation of molecular integrals of many-body operators over Gaussian functions," http://libint.valeyev.net, 2022, version 2.7.2.

[9] Q. Sun, "Libcint: An efficient general integral library for Gaussian basis functions," *Journal of Computational Chemistry*, vol. 36, pp. 1664–1671, 2015.

[10] B. P. Pritchard and E. Chow, "Horizontal vectorization of electron repulsion integrals," *Journal of Computational Chemistry*, vol. 37, no. 28, pp. 2537–2546, 2016.

[11] H. Huang and E. Chow, "Accelerating quantum chemistry with vectorized and batched integrals," in *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*, 2018, pp. 529–542.

[12] F. Neese, "The SHARK integral generation and digestion system," *Journal of Computational Chemistry*, vol. 44, no. 3, pp. 381–396, 2023.

[13] 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, 2008.

[14] A. Asadchev, V. Allada, J. Felder, B. M. Bode, M. S. Gordon, and T. L. Windus, "Uncontracted Rys quadrature implementation of up to G functions on graphical processing units," *Journal of Chemical Theory and Computation*, vol. 6, no. 3, pp. 696–704, 2010.

[15] J. Kussmann and C. Ochsenfeld, "Employing OpenCL to accelerate ab initio calculations on graphics processing units," *Journal of Chemical Theory and Computation*, vol. 13, no. 6, pp. 2712–2716, 2017.

[16] Y. Tian, B. Suo, Y. Ma, and Z. Jin, "Optimizing two-electron repulsion integral calculations with McMurchie–Davidson method on graphic processing unit," *The Journal of Chemical Physics*, vol. 155, no. 3, p. 034112, 07 2021.

[17] G. M. B. Jorge Luis Galvez Vallejo and M. S. Gordon, "High-performance GPU-accelerated evaluation of electron repulsion integrals," *Molecular Physics*, vol. 121, no. 9-10, p. e2112987, 2023.

[18] A. Asadchev and E. F. Valeev, "Libintx: A library for accelerated evaluation of molecular integrals of many-body operators over Gaussian atomic orbitals." https://github.com/ValeevGroup/libintx, 2023, experimental version.

[19] V. Kindratenko, I. Ufimtsev, and T. Martínez, "Evaluation of two-electron repulsion integrals over Gaussian basis functions on SRC-6 reconfigurable computer," https://users.ncsa.illinois.edu/kindr/papers/rssi08_paper2.pdf, 2008.

[20] X. Wu, T. Kenter, R. Schade, T. D. Kühne, and C. Plessl, "Computing and compressing electron repulsion integrals on FPGAs," in *2023 IEEE 31st Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. Los Alamitos, CA, USA: IEEE Computer Society, May 2023, pp. 162–173.

[21] D. G. A. Smith, L. A. Burns, A. C. Simmonett, R. M. Parrish, M. C. Schieber, R. Galvelis, P. Kraus, H. Kruse, R. Di Remigio, A. Alenaizan, A. M. James, S. Lehtola, J. P. Misiewicz, M. Scheurer, R. A. Shaw, J. B. Schriber, Y. Xie, Z. L. Glick, D. A. Sirianni, J. S. O'Brien, J. M. Waldrop, A. Kumar, E. G. Hohenstein, B. P. Pritchard, B. R. Brooks, I. Schaefer, Henry F., A. Y. Sokolov, K. Patkowski, I. DePrince, A. Eugene, U. Bozkaya, R. A. King, F. A. Evangelista, J. M. Turney, T. D. Crawford, and C. D. Sherrill, "PSI4 1.4: Open-source software for high-throughput quantum chemistry," *The Journal of Chemical Physics*, vol. 152, no. 18, p. 184108, 05 2020.

[22] T. D. Kühne, M. Iannuzzi, M. Del Ben, V. V. Rybkin, P. Seewald, F. Stein, T. Laino, R. Z. Khaliullin, O. Schütt, F. Schiffmann, D. Golze, J. Wilhelm, S. Chulkov, M. H. Bani-Hashemian, V. Weber, U. Borštnik, M. Taillefumier, A. S. Jakobovits, A. Lazzaro, H. Pabst, T. Müller, R. Schade, M. Guidon, S. Andermatt, N. Holmberg, G. K. Schenter, A. Hehn, A. Bussy, F. Belleflamme, G. Tabacchi, A. Glöß, M. Lass, I. Bethune, C. J. Mundy, C. Plessl, M. Watkins, J. VandeVondele, M. Krack, and J. Hutter, "CP2K: An electronic structure and molecular dynamics software package - quickstep: Efficient and accurate electronic structure calculations," *The Journal of Chemical Physics*, vol. 152, no. 19, p. 194103, 05 2020.

[23] T. Helgaker, P. Jørgensen, and J. Olsen, *Molecular Integral Evaluation*. John Wiley & Sons, Ltd, 2000, ch. 9, pp. 336–432.

[24] M. Dupuis, J. Rys, and H. F. King, "Evaluation of molecular integrals over Gaussian basis functions," *The Journal of Chemical Physics*, vol. 65, no. 1, pp. 111–116, 07 1976.

[25] J. Rys, M. Dupuis, and H. F. King, "Computation of electron repulsion integrals using the Rys quadrature method," *Journal of Computational Chemistry*, vol. 4, no. 2, pp. 154–157, 1983.

[26] 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, 04 1986.

[27] 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, 11 1988.

[28] L. E. McMurchie and E. R. Davidson, "One- and two-electron integrals over Cartesian Gaussian functions," *Journal of Computational Physics*, vol. 26, no. 2, pp. 218–231, 1978.

[29] M. Guidon, F. Schiffmann, J. Hutter, and J. VandeVondele, "Ab initio molecular dynamics using hybrid density functionals," *The Journal of Chemical Physics*, vol. 128, no. 21, p. 214104, 06 2008.

[30] The Linux Kernel Development Community, "Power capping framework," https://www.kernel.org/doc/html/latest/power/powercap/powercap.html, 2024, [Accessed: March 27th, 2024].

[31] C. Bauer, T. Kenter, M. Lass, L. Mazur, M. Meyer, H. Nitsche, H. Riebler, R. Schade, M. Schwarz, N. Winnwa, A. Wiens, X. Wu, C. Plessl, and J. Simon, "Noctua 2 supercomputer," *Journal of large-scale research facilities*, vol. 9, 2024.

[32] Y. Gu, T. VanCourt, and M. Herbordt, "Accelerating molecular dynamics simulations with configurable circuits," in *International Conference on Field Programmable Logic and Applications, 2005.*, 2005, pp. 475–480.

[33] D. Jones, J. E. Allen, Y. Yang, W. F. Drew Bennett, M. Gokhale, N. Moshiri, and T. S. Rosing, "Accelerators for classical molecular dynamics simulations of biomolecules," *Journal of Chemical Theory and Computation*, vol. 18, no. 7, pp. 4047–4069, 2022, pMID: 35710099.

[34] C. Yang, T. Geng, T. Wang, R. Patel, Q. Xiong, A. Sanaullah, C. Wu, J. Sheng, C. Lin, V. Sachdeva, W. Sherman, and M. Herbordt, "Fully integrated FPGA molecular dynamics simulations," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '19. New York, NY, USA: Association for Computing Machinery, 2019.

[35] C. Wu, T. Geng, A. Guo, S. Bandara, P. Haghi, C. Liu, A. Li, and M. Herbordt, "FASDA: An FPGA-aided, scalable and distributed accelerator for range-limited molecular dynamics," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '23. New York, NY, USA: Association for Computing Machinery, 2023.