SORCERI: Streaming Overlay Acceleration for Highly Contracted Electron Repulsion Integral Computations in Quantum Chemistry

Philip Stachura
Simon Fraser University
Burnaby, BC, Canada
pstachur@sfu.ca

Xin Wu
Paderborn Center for Parallel Computing
Department of Computer Science
Paderborn University
Paderborn, Germany
xin.wu@uni-paderborn.de

Christian Plessl
Paderborn Center for Parallel Computing
Department of Computer Science
Paderborn University
Paderborn, Germany
christian.plessl@uni-paderborn.de

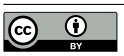
Zhenman Fang
Simon Fraser University
Burnaby, BC, Canada
University of Minnesota Twin Cities
Minneapolis, MN, United States
zhenman@sfu.ca

Abstract

The computation of highly contracted electron repulsion integrals (ERIs) is essential to achieve quantum accuracy in atomistic simulations based on quantum mechanics. Its growing computational demands make energy efficiency a critical concern. Recent studies demonstrate FPGAs' superior performance and energy efficiency for computing primitive ERIs, but the computation of highly contracted ERIs introduces significant algorithmic complexity and new design challenges for FPGA acceleration.

In this work, we present SORCERI, the first streaming overlay acceleration for highly contracted ERI computations on FPGAs. SORCERI introduces a novel streaming Rys computing unit to calculate roots and weights of Rys polynomials on-chip, and a streaming contraction unit for the contraction of primitive ERIs. This shifts the design bottleneck from limited CPU-FPGA communication bandwidth to available FPGA computation resources. To address practical deployment challenges for a large number of quartet classes, we design three streaming overlays, together with an efficient memory transpose optimization, to cover the 21 most commonly used quartet classes in realistic atomistic simulations. To address the new computation constraints, we use flexible calculation stages with a free-running streaming architecture to achieve high DSP utilization and good timing closure.

Experiments demonstrate that SORCERI achieves an average 5.96x, 1.99x, and 1.16x better performance per watt than libint on a 64-core AMD EPYC 7713 CPU, libintx on an Nvidia A40 GPU, and SERI, the prior best-performing FPGA design for primitive ERIs. Furthermore, SORCERI reaches a peak throughput of 44.11 GERIS (10^9 ERIs per second) that is 1.52x, 1.13x, and 1.93x greater than libint, libintx and SERI, respectively. SORCERI will be released soon at https://github.com/SFU-HiAccel/SORCERI.



FPGA '26, Seaside, CA, USA

ACM ISBN 979-8-4007-2079-6/2026/02
https://doi.org/10.1145/3748173.3779198

CCS Concepts

• **Hardware → Hardware accelerators**; • **Computer systems organization → Data flow architectures**; **Reconfigurable computing**; **High-level language**; • **Applied computing → Chemistry**.

Keywords

Electron Repulsion Integrals, Quantum Chemistry, Atomistic Simulation, Overlay Architecture, FPGA Acceleration

ACM Reference Format:
Philip Stachura, Xin Wu, Christian Plessl, and Zhenman Fang. 2026. SORCERI: Streaming Overlay Acceleration for Highly Contracted Electron Repulsion Integral Computations in Quantum Chemistry. In *Proceedings of the 2026 ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA '26), February 22–24, 2026, Seaside, CA, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3748173.3779198

1 Introduction

Computation of highly contracted electron repulsion integrals (ERIs) in quantum chemistry is a performance-critical kernel for accurate *ab initio* molecular dynamics (AIMD) simulations that provide critical information for drug discovery, industrial catalysis, and semiconductor properties [10]. Recent advances in high-performance computing (HPC) have enabled AIMD to reach "quantum accuracy" for large-scale biomolecular simulations [27] and material modeling [7]. However, power consumption has become a critical limiting factor for HPC clusters, with top HPC systems exceeding 30 MW (megawatts) [30]. Thus, energy-efficient acceleration for contracted ERI computation is essential for the sustainability of accurate AIMD simulations in HPC.

FPGAs have proven to be performant and energy-efficient accelerators in machine learning and scientific computing, making them increasingly deployed in data centers [4–6, 20, 23]. The advantages of FPGA-accelerated ERI computation using single primitive Gaussian-type orbitals (GTOs) have been demonstrated in recent studies [26, 33]. SERI [26], the current best-performing HBM-based FPGA accelerator for primitive ERI computation, reaches a peak

throughput of 23.8 GERIS (10^9 ERIs per second) on one AMD/Xilinx Alveo U280 FPGA and shows superior energy efficiency over the libint library [31]—which is widely used for the ERI computation in popular atomistic simulation packages [16, 25]—on a 64-core AMD EPYC 7713 CPU and libintx [2] on an Nvidia A40 GPU.

However, accurate approximation of a Slater-type orbital (STO) [24] requires contracted GTOs, which are linear combination of least-squares-fitted primitive GTOs [12, 13]. In general, one contracted GTO requires ≥ 4 primitive GTOs for accurate results. Hence, the computation of contracted ERIs involves not only evaluating tens of trillions of primitive GTOs, but also "contraction" of these primitive ERIs. This high algorithmic complexity presents new design challenges for FPGA acceleration. To the best of our knowledge, no attempts have been made in this direction yet.

In this paper, we present SORCERI, the first practical streaming overlay acceleration for computing highly contracted ERIs on FPGAs. SORCERI integrates all computation stages for contracted ERIs, based on the widely used Rys quadrature algorithm [8, 21], directly into an FPGA design. It starts with the computation of the roots and weights of the Rys polynomials using a novel streaming unit. This is followed by a set of computation stages for the necessary primitive ERIs, which are replicated to maximize throughput, and a streaming contraction unit for the intermediate contracted ERIs. Then, the ERIs pass through the final horizontal recurrence relations using a flexible transpose operation. All these computation units are seamlessly integrated into a single FPGA accelerator, thus shifting the bottleneck from limited CPU-FPGA communication bandwidth to available FPGA computation resources. This also enables full floating-point precision results for increased accuracy.

Our experimental evaluation of SORCERI for the contracted ERI computation shows its high energy efficiency: on average, SORCERI on an AMD/Xilinx Alveo U280 FPGA achieves 5.96x greater performance per watt than the state-of-the-art CPU library libint [31] on a 64-core AMD EPYC 7713 CPU, 1.99x greater than the state-of-the-art GPU library libintx [3] on an Nvida A40 GPU, and 1.16x greater than the prior best-performing FPGA design SERI [26] on the same FPGA. For absolute performance, SORCERI achieves a peak throughput of 44.11 GERIS (10^9 ERIs per second) on an Alveo U280 FPGA, which is 1.52x, 1.13x, and 1.93x greater than the peaks of libint, libintx and SERI, respectively. On average, speedups of 1.50x over libint, 0.96x over libintx, and 1.57x against SERI are obtained for the 21 common ERI classes, ranging from $(ss|ss)$ to $(dd|dd)$, required for realistic AIMD simulations. SORCERI is the first FPGA accelerator design deployable in end-to-end quantum-mechanics-based atomistic simulation packages.

In summary, this paper makes the following contributions:

1. The first practical streaming architecture for highly contracted ERI computation with seamless integration of all stages into a single compute resource-constrained FPGA.
2. A versatile set of three FPGA overlays that compute all 21 canonical ERI classes up to d orbitals, without requiring quartet class specific implementations.
3. High energy efficiency of up to 0.58 GERIS/Watt and high-throughput of up to 44.11 GERIS, without resorting to ERI compression, and thus preserving full floating-point accuracy throughout the entire computation.

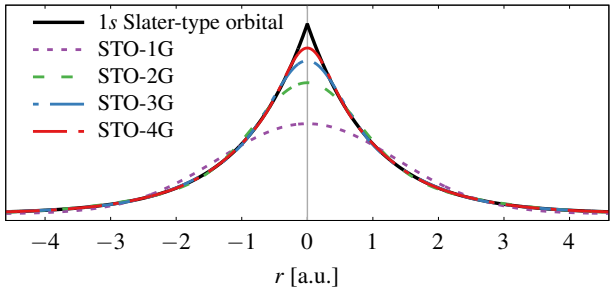


Figure 1: 1s Slater-type orbital (STO) of a hydrogen atom (solid black line) is more accurately represented by contracted Gaussian-type orbitals (STO-nG) of higher contraction n.

2 Background on Contracted ERI Computation

In quantum chemistry, an STO centered at $\mathbf{A} = [A_x, A_y, A_z]$ with angular momentum L_a is approximated by a contracted GTO $\phi_{\mathbf{A},L_a}(\mathbf{r})$, which is a linear combination of multiple primitive GTOs

$$\phi_{\mathbf{A},L_a}(\mathbf{r}) = \sum_{i=1}^{n_a} C_i \, g_{\mathbf{A},L_a,\alpha_i}(\mathbf{r}) \tag{1}$$

where $\mathbf{r} = [x, y, z]$ are the Cartesian coordinates of an electron and n_a is the degree of contraction for $\phi_{\mathbf{A},L_a}(\mathbf{r})$ [12]. The contraction coefficients $\{C_i\}$ and Gaussian exponents $\{\alpha_i\}$ differ among the primitive GTOs $g_{\mathbf{A},L_a,\alpha_i}(\mathbf{r})$ that constitute the contracted GTO.

Fig. 1 illustrates the 1s STO of a hydrogen atom along with the primitive GTO (STO-1G) and several contracted GTOs (STO-nG), where n is the degree of contraction. Higher degrees of contraction yield more accurate representations of atomic orbitals. Typically, $n \geq 4$ is needed in AIMD simulations. In addition to the s orbital, commonly used contracted GTOs include the p and d orbitals, all of which are covered in this work.

A contracted ERI is a 6-dimensional integration, involving 4 contracted GTOs usually at 4 different atomic centers for two electrons, one at $\mathbf{r}_1$ and the other at $\mathbf{r}_2$

$$\iint \mathrm{d}\mathbf{r}_1 \, \mathrm{d}\mathbf{r}_2 \frac{\phi_{\mathbf{A},L_a}(\mathbf{r}_1)\phi_{\mathbf{B},L_b}(\mathbf{r}_1)\phi_{\mathbf{C},L_c}(\mathbf{r}_2)\phi_{\mathbf{D},L_d}(\mathbf{r}_2)}{|\mathbf{r}_1 - \mathbf{r}_2|} \tag{2}$$

Conventionally, a contracted ERI quartet class is denoted as $(ab|cd)$, and a primitive class is denoted as $[ij|kl]$. Due to its 8-fold permutation symmetry, a total of 21 most widely used canonical $(ab|cd)$ classes, from $(ss|ss)$ to $(dd|dd)$, are considered in this work.

Because each contracted GTO is a linear combination of multiple primitive GTOs with different Gaussian exponents, $(ab|cd)$ can be expanded as a 4-fold sum of all primitive ERIs, each denoted by $[ij|kl]$, and multiplied by the corresponding contraction coefficients

$$(ab|cd) = \sum_{i=1}^{n_a}\sum_{j=1}^{n_b}\sum_{k=1}^{n_c}\sum_{l=1}^{n_d} C_i C_j C_k C_l [ij|kl] \tag{3}$$

However, the overall transformation in Eq. (3) does not work efficiently for the contracted ERI computation, due to the formidable computational complexity $O(n^4)$, where n is the degree of contraction. To facilitate the calculation of contracted ERIs, the widely used Rys quadrature algorithm [8, 21] incorporates the contraction

stage into the recurrence relations to enable more data reuse for intermediate ERIs [9].

Algorithm 1 briefly describes the Rys quadrature algorithm. Interested readers are referred to the original papers [8, 9, 21].

Input: Basis set
Output: $(ab|cd)$

for each $(ab|cd)$ quartet **do**
  for each $[ij|kl]$ primitive quartet **do**
    #1 Rys polynomials: compute roots $\{t_\mu\}$ and weights $\{w_\mu\}$
    #2 Auxiliary coefficients: build auxiliary **B** and **C** matrices
    #3 Vertical Recurrence Relations (VRRs):
    for $\xi \in \{x, y, z\}$, $\mu \in [1, n_{\text{Rys}}]$ **do**
      for $e \in [0, L_a + L_b]$, $f \in [0, L_c + L_d]$ **do**
        compute $I(e, 0, f, 0, \mu, \xi)$ via Eqs. (4)
      end for
    end for
    #4 Gaussian Quadrature (GQ):
    for $\xi \in \{x, y, z\}$, $\mu \in [1, n_{\text{Rys}}]$ **do**
      compute intermediate primitive $[e0|f0]$ via Eq. (5)
    end for
  end for
  #5 Contraction:
  for $i \in [1, n_a]$, $j \in [1, n_b]$, $k \in [1, n_c]$, $l \in [1, n_d]$ **do**
    compute full-contracted $(e0|f0)$ via Eq. (3)
  end for
  #6 Horizontal Recurrence Relations (HRRs):
  for $\xi \in \{x, y, z\}$, $b \in [0, L_b]$, $d \in [0, L_d]$ **do**
    compute final contracted $(ab|cd)$ via Eqs. (6)
  end for
end for

Alg. 1 Pseudocode for computing contracted $(ab|cd)$ quartets.

A basis set from the host is provided as input for computing contracted ERIs. For an $(ab|cd)$ quartet class, the basis set contains 4 contracted GTOs centered on **A**, **B**, **C**, and **D**. Then, the following steps are performed for the contracted ERI computation.

Rys polynomials: Computation of roots $\{t_\mu\}$ and weights $\{w_\mu\}$ of the Rys polynomials, which was pre-computed on host in prior studies [26, 33], is completely integrated on the FPGA in SORCERI to minimize CPU-FPGA communication.

Auxiliary coefficients: Two auxiliary arrays **B** and **C** are built, after $\{t_\mu\}$ and $\{w_\mu\}$ of the Rys polynomial are available.

Vertical Recurrence Relations (VRRs): The intermediate integrals $I(e, 0, f, 0, \mu, \xi)$, where $e \in [0, L_a + L_b]$ and $f \in [0, L_c + L_d]$, are generated from $I(0, 0, 0, 0, \mu, \xi)$ and auxiliary arrays using two 4-dimensional VRRs

$$\begin{aligned} I(e+1,0,f,0,\mu,\xi) = {} & eB_{2,\mu}\, I(e-1,0,f,0,\mu,\xi) \\ & + fB_{1,\mu}\, I(e,0,f-1,0,\mu,\xi) \\ & + C_{\xi,\mu}\, I(e,0,f,0,\mu,\xi) \end{aligned} \tag{4a}$$

$$\begin{aligned} I(e,0,f+1,0,\mu,\xi) = {} & fB_{3,\mu}\, I(e,0,f-1,0,\mu,\xi) \\ & + eB_{1,\mu}\, I(e-1,0,f,0,\mu,\xi) \\ & + C_{2\xi,\mu}\, I(e,0,f,0,\mu,\xi) \end{aligned} \tag{4b}$$

Gaussian Quadrature (GQ): Compute the primitive ERI quartets $[e0|f0]$ as intermediate integrals via

$$[e0|f0] = \sum_{\mu=1}^{n_{\text{Rys}}} w_\mu \left[\prod_{\xi \in \{x,y,z\}} I(e_\xi, 0, f_\xi, 0, \mu, \xi) \right] \tag{5}$$

where n_{Rys} is the order of the Rys polynomial.

Contraction: The 4-fold summation in Eq. (3) is used to combine primitive ERIs to compute $(e0|f0)$ with the exception that the contraction coefficients are applied in the VRRs stage to reduce multiplications. The contraction stage is vital for computing the contracted ERIs and is implemented for the first time in this work.

Horizontal Recurrence Relations (HRRs): The final contracted ERIs $(ab|cd)$ are obtained from intermediate integrals $(e0|f0)$ via two HRRs by transferring angular momentum from e and f to b and d, respectively

$$(ab|c0) = (a+1, b-1|c0) + (A_\xi - B_\xi)\,(a, b-1|c0) \tag{6a}$$

$$(ab|cd) = (ab|c+1, d-1) + (C_\xi - D_\xi)\,(ab|c, d-1) \tag{6b}$$

where $\xi \in \{x, y, z\}$, requiring HRRs to be applied for all ξ.

The lossy ERI compression [11] used in previous FPGA designs [26, 33] has been removed in SORCERI to ensure full floating-point accuracy in the contracted ERI computation. By integrating all computation stages into a single FPGA design, SORCERI eliminates the CPU-FPGA communication bottleneck in prior work.

3 Motivation and Challenges

3.1 Limitations of Existing Work

The prior best-performing FPGA accelerator for ERI computation, SERI, focused primarily on calculating primitive ERIs using a streaming architecture [26]. This design segmented computations into multiple Vitis HLS kernels connected via kernel-to-kernel streams. Although effective, it faced a number of limitations that prevent deployment in practical AIMD simulations.

Quartet Class Specific Kernels. The dedicated implementations required for each canonical quartet class necessitate either a multi-FPGA deployment with 21 devices to cover the most commonly used quartet classes, or frequent FPGA bitstream reconfigurations which outweigh the accelerator's benefits.

Reduced Accuracy from Lossy ERI Compression. Even with multiple HBM channels, bandwidth limitations require compressing ERIs from 32-bit floats to 16-bit values, negatively impacting the accuracy of the computed results.

Host-FPGA Bandwidth Limitations. Although HBM provided superior internal bandwidth to store computed ERIs, transferring the results back to the host for subsequent computations is constrained by the PCIe bandwidth, negating the FPGA's performance advantage.

3.2 Opportunities and Challenges

To address these limitations, we design FPGA overlays capable of efficiently computing ERIs across multiple quartet classes and extend the computations from primitive ERIs to contracted ERIs. By computing contracted ERIs directly on the FPGA, we alleviate PCIe bandwidth constraints, eliminating the need for result compression and thus improving accuracy. Incorporating additional computation stages on-chip, starting directly from the basis set, further reduces the required host-FPGA bandwidth. However, transitioning to such an overlay architecture introduces several key new challenges:

3.2.1 Diverse Computational Requirements. Quartet classes vary significantly in required compute resources and calculation patterns

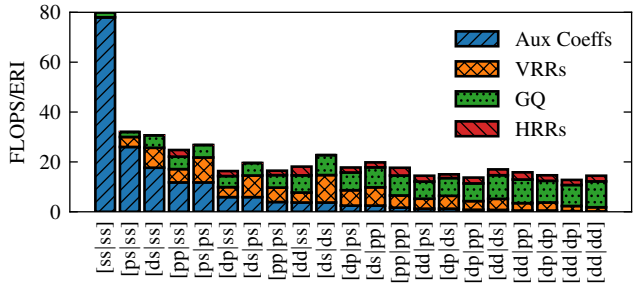


Figure 2: Comparison of FLOPS per ERI for each quartet class.

within each computation stage which makes creating overlays difficult. Fig. 2 shows the number of floating-point operations required to compute a single ERI for each quartet class. As shown, not only do some quartet classes require vastly more operations per ERI than others, but the division also varies a lot between stages.

3.2.2 Complex Rys Polynomial Computations. To support ERI computation from the basis set as input, the roots and weights of the Rys polynomial must be computed on the FPGA (instead of on the CPU as in previous work). These calculations are based on equations which cannot be solved analytically. Instead, the Rys polynomial roots and weights are solved numerically using a set of piecewise approximations with each segment involving multiple polynomial approximations in combination with additional calculations and non-linear functions. The piecewise cases are segmented based on both the order of the Rys polynomial (μ) and the value of the input (x). This creates a large tree of unique calculation paths where each node can differ greatly from others, making it challenging to create a common calculation unit.

3.2.3 Memory Access Pattern. The Gaussian quadrature (GQ) and horizontal recurrence relations (HRR) stages require significant intermediate storage with orthogonal memory partitioning requirements. Existing buffer permutation methods lack sufficient flexibility for general overlays, necessitating more adaptive buffer strategies. An adaptive transposition mechanism is necessary to handle this complexity without sacrificing throughput.

3.2.4 Floating-point Accumulation Constraints. The contraction stage requires efficient accumulation of floating-point results. AMD/Xilinx UltraScale+ FPGAs do not natively support floating-point accumulation with a pipeline initiation interval (II) of 1. This, along with a varying distance between values that need to be accumulated among quartet classes, makes implementing a versatile accumulation stage challenging.

4 SORCERI Design and Implementation

SORCERI is designed to compute highly contracted ERIs. This allows it to solve the CPU-FPGA communication bottleneck from previous designs by contracting multiple primitive ERIs on the FPGA before transferring the results back to the host. This requires new computation stages that are implemented in this work for the first time. **SORCERI presents the following novelties.**

Computation of contracted ERIs directly from basis set. Placing more calculation stages on chip solves both the input bandwidth bottleneck for small quartet classes and the output bandwidth bottleneck for large quartet classes. After loading the input basis set, which is negligible in size compared to the number of contracted ERIs that will be calculated, we introduce a dispatch stage. For each given combination of contracted basis functions, it iterates and distributes all the primitive GTO information necessary to compute a given contracted ERI. The primitive ERIs then need to be contracted and finalized in the new contraction and HRR stages, which reduces the number of outputs that need to be stored, alleviating the bandwidth bottleneck.

Overlay support. To avoid dynamic FPGA reconfiguration overhead during runtime, we reduce the number of unique implementations to cover $(ss|ss)$ to $(dd|dd)$ from 21, down to only 3. We design overlays which are capable of calculating multiple quartet classes in a single implementation to enable deployment of the accelerator in real-world use cases. This is done by designing stages with flexible architectures that support runtime flexibility in managing intermediate values and effectively balancing compute utilization between stages and quartet classes.

Rys polynomial computation. To process the primitive ERIs from only the GTO data, we add a new Rys computation unit to solve for the roots and weights of the Rys polynomial. In prior designs these were pre-computed on the host, due to the complexity of many non-linear operations and multiple high order polynomials. To address this, we develop a streaming unit which decomposes the calculation into phases and allows for flexible parallelization of polynomial computation to perform the calculations in a performant and resource efficient manner.

Efficient buffer transpose. To enable the flexibility required to support multiple quartet classes in a single overlay for the GQ and HRR stages, we implement a flexible and efficient buffer transpose. While previous work [26] had solved the orthogonal buffer access requirements in some stages of ERI computation using a buffer permutation stage, that design does not provide the necessary runtime flexibility required for our overlays. Our transpose implementation provides a more generic solution that remains performant for varying sized inputs with the same hardware implementation.

Interleaved loopback accumulation. To maintain II=1 throughput while accumulating values we design a streaming accumulation unit that tracks multiple interleaved independent sums with a loopback mechanism. By interleaving the sums, the adder's latency is hidden, enabling continuous operation with a runtime-configurable number of concurrent sums.

SORCERI splits the 21 most commonly used canonical quartet classes into three overlays in order to make it easier to balance the varying computational demands of different quartet classes. Each computation stage is implemented as a Vitis HLS kernel and are connected to each other using kernel-to-kernel streams as shown in Fig. 3. Unless the stage has an internal buffer, it is implemented using free-running pipelines to allow for continuous and high-throughput processing with less fanout and simpler pipeline control logic. Stages that do contain an internal buffer are implemented using PIPOs (ping-pong buffers) in a dataflow fashion, such that the throughput is not affected by data reads/writes and data can be processed every cycle. SORCERI takes the basis set and a list of basis

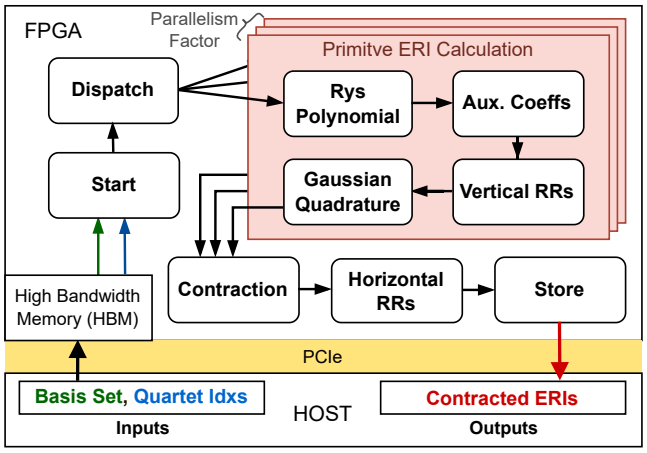


Figure 3: SORCERI streaming architecture diagram.

function indexes for the quartets to solve for as input. It outputs contracted ERIs which are stored directly into host memory.

The design is scaled to maximize performance for the given resources through a parallelism factor, which increases the number of instances of the primitive ERI calculation kernels.

Each overlay is manually floor-planned with kernel to super logic region (SLR) mappings to provide a layout that balances resource utilization across SLRs while minimizing SLR crossings to achieve good timing closure with a high parallelism factor.

4.1 Design Summary

The overlays are divided by the orders of the Rys polynomial they support (μ). Since the Rys roots and weights calculation differs the most between different μ values, it presents the greatest advantage in decreasing the scope of which values the implemented hardware needs to calculate. A summary of the quartet classes is shown in Table 1: for each quartet class, it shows the corresponding μ value and the number of ERIs per quartet. The number of GTOs in a basis set will increase the number of primitive quartets to solve, but will always be contracted to the same final number of ERIs per quartet for a given class.

Table 1: List of all 21 supported canonical quartet classes and their corresponding order of Rys polynomial (μ) and number of ERIs for a single quartet.

Class	μ	# ERIs	Class	μ	# ERIs	Class	μ	# ERIs
$(ss\|ss)$	1	1	$(pp\|ps)$	2	27	$(dp\|ds)$	3	108
$(ps\|ss)$	1	3	$(dd\|ss)$	3	36	$(dp\|pp)$	3	162
$(ds\|ss)$	2	6	$(ds\|ds)$	3	36	$(dd\|ds)$	4	216
$(pp\|ss)$	2	9	$(dp\|ps)$	3	54	$(dd\|pp)$	4	324
$(ps\|ps)$	2	9	$(ds\|pp)$	3	54	$(dp\|dp)$	4	324
$(dp\|ss)$	2	18	$(pp\|pp)$	3	81	$(dd\|dp)$	4	648
$(ds\|ps)$	2	18	$(dd\|ps)$	3	108	$(dd\|dd)$	5	1296

The first overlay supports $\mu = 1, 2$. These represent the smallest quartet classes that have a low number of resulting ERIs per quartet. The next overlay supports $\mu = 3$. And the last one, $\mu = 4, 5$, covers the largest quartet classes, and also has the largest number of ERIs produced for each quartet.

4.1.1 Overlay $\mu = 1, 2$. This overlay covers the smallest quartet classes which produce the fewest number of ERIs. The largest of the set, $(pp|ps)$, has 27 ERIs per quartet. In order to maximize its throughput, the calculation of quartets in all stages is pipelined with II=1 to produce all ERIs for a given quartet every cycle. This allows implementations for most of the stages to be relatively simple. However, the Gaussian quadrature stage faces some challenges with managing the varying indexing patterns between its intermediate inputs and primitive ERI outputs, which causes additional routing congestion and will be discussed in Section 4.3.

4.1.2 Overlay $\mu = 3$. The approximate number of FLOPS/ERI is roughly similar for all medium to large quartet classes. Therefore, we balance the computation so that an overlay produces roughly the same number of ERIs per cycle for all quartet classes. Since this overlay, as well as the $\mu = 4, 5$ overlay, have a larger number of ERIs per quartet, we allow flexibility with how many cycles a single quartet takes, which enables us to better balance the performance across quartet classes.

To help manage varying and large classes, the design splits the ERIs in a given quartet into batches, where it will calculate between 12-18 primitive ERIs in a batch, which are then processed by the contraction and HRR stages. The quartet classes with $\mu = 3$ will require between 3-10 batches to compute a quartet. Since the smallest number is 3, stages like Rys polynomial and coefficient calculation have a relaxed II=3 to allow for more resource reuse.

4.1.3 Overlay $\mu = 4, 5$. Like the $\mu = 3$ overlay, the $\mu = 4, 5$ overlay divides the ERIs in a quartet into multiple batches which will be solved over multiple cycles.

In the Gaussian quadrature (GQ) stage, due to the larger quartet classes, the intermediates array is too large to be fully partitioned. In prior work, a specialized buffer permutation stage was used to perform a partial transpose from the fully partitioned dimensions that are produced from the previous stage to the orthogonal ones needed for GQ calculation. Unfortunately, the buffer permutation layer lacks the flexibility to support multiple quartet classes while remaining performant for each quartet class individually. As such, new diagonal buffer transpose phases are added to accomplish this goal which are described in detail in Section 4.3.

4.2 Rys Polynomial Roots and Weights

The Rys polynomial roots and weights are solved numerically using a set of multiple polynomial approximations, in combination with additional calculations and non-linear functions. The polynomials to solve, along with the surrounding calculations vary based on the order of the Rys polynomial (μ) and the value of the input (x). This creates a large tree of calculations where the number of unique calculation paths, each containing multiple different high-degree polynomials (up to degree 17), makes implementing an FPGA module to solve these efficiently in a pipelined manner difficult.

While larger quartet classes have higher order Rys polynomials, they also have more cycles between two pipeline iterations to compute a given ERI, which provides an opportunity for resource reuse within the unit. In contrast, for the $\mu = 1, 2$ overlay, all quartet

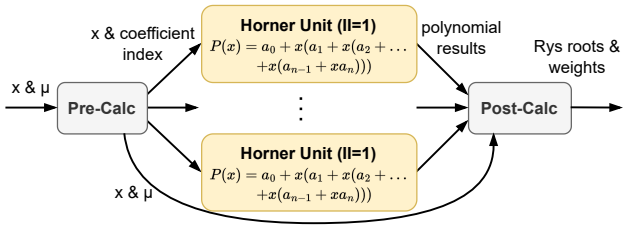


Figure 4: Rys polynomial calculation unit flexible architecture diagram.

classes need to be calculated with pipeline II=1. This requires the architecture to be flexible enough to provide an efficient trade-off between pipeline II and resource utilization.

To do this with a single calculation module would not provide the required resource efficiency needed for this accelerator, which is why we design an architecture that can be generated to provide tuned hardware implementation variants to cover the $\mu = 1, 2$, $\mu = 3$, and $\mu = 4, 5$ overlays.

The Rys calculation unit views the roots and weights computation as follows. It extracts all Horner polynomial patterns from the calculations and stores all the coefficients in a single large array. It then separates out the calculation into 3 distinct phases as shown in Fig. 4: 1) pre-calculation, 2) Horner polynomial calculation, and 3) post-calculation. This separation is possible because in the Rys polynomial calculation algorithm, no inputs to a Horner polynomial expansion ever depend on another Horner polynomial expansion. The pre-calculation phase takes the input, x, and based on its value, identifies which Horner polynomials should be solved. It also calculates the inputs to the polynomials which can vary. It then sends the inputs, along the indices of the coefficients for the Horner polynomials down to a set of streams. Each stream is connected to a Horner polynomial calculation unit which is fully unrolled and pipelined to compute a high-degree Horner polynomial with II=1. The last stage collects the results from the Horner calculation units, and proceeds with the remaining calculations. Both the pre- and post-calculation stages are tuned to minimize inefficient operations such as divisions, exponentials, and square roots.

The number of internal Horner polynomial calculation units can be adjusted to meet the overall throughput requirements. The pre- and post-calculation stages can also be scaled by adjusting their II to allow for operator resource reuse. Additionally, all these stages are implemented using free-running pipelines.

For the $\mu = 1, 2$ overlay, an overall pipeline II=1 is required, which necessitates 3 Horner calculation units. The Rys calculation for $\mu = 3$ overlay needs to have an II=3, requiring 2 Horner calculation units to process the 6 Horner expansions for a given input. And lastly, the $\mu = 4, 5$ overlay's unit needs an overall II=14 requiring only 1 Horner calculation unit to process the 14 Horner expansions for a given input.

Table 2 summarizes the throughput target and the number of parallel Horner calculation units required for each μ-overlay. The overall target II for the Rys calculation stage is balanced to be the maximum allowed without slowing down the throughput of downstream stages. The number of internal II=1 Horner units is then chosen to achieve the overall target II given the maximum number of Horner expressions needed to solve the Rys polynomial approximation for the given values of μ.

Table 2: Summary of the number of internal Horner units needed to satisfy throughput requirements in the Rys polynomial calculation unit for each overlay

Overlay	Target II	Horner expansions	Horner units
$\mu = 1, 2$	1	3	3
$\mu = 3$	3	6	2
$\mu = 4, 5$	14	14	1

4.3 Efficient Buffer Transpose

In both the GQ (Gaussian quadrature) and HRR (horizontal recurrence relations) stages, the fully partitioned output dimensions of one calculation stage differs completely from the partitioning needed by the next. In the case of GQ, the VRR (vertical recurrence relations) stage produces $|e| \times |f| \times |\mu| \times |\xi|$ values in parallel accessible $|e| \times |f|$ blocks over $|\mu| \times |\xi|$ iterations. In contrast, the GQ needs parallel access to all elements from the μ and ξ dimensions over $|e| \times |f|$ iterations. In the case of HRR, the first phase produces values across the a and b dimensions over $|c| \times |d|$ cycles, and the next phase consumes values from the c and d dimensions over $|a| \times |b|$ iterations.

While previous work presented a buffer permutation layer capable of doing this operation without performance impact in a dataflow environment, it does not stay performant when the size of the dimensions changes without generating new hardware. This makes it unsuitable for use in an overlay.

To solve this, we present an efficient transpose technique that takes advantage of being in a latency-insensitive environment to allow accesses across orthogonally partitioned dimensions without affecting the throughput.

For simplicity, assume that each buffer has one reader and one writer, and each port can access one element in the buffer in a given cycle. Let us take an array of size $P \times D$, where P is the size of the fully partitioned dimensions, and D is the size of depth-only dimensions. As shown in Fig. 5, the traditional layout for sequential access would be to have P buffers, each with D elements, where each buffer is assigned a static P dimension index and contains all the D elements for that index. This is helpful because most of the time the index for a given producer or consumer is static and known at compile-time, allowing an accessor to connect to only one port on each buffer. This makes it resource efficient and easy to place and route. However, if we want to access all elements along the D dimension for a given P index, that would take D cycles because all those elements are in the same buffer.

Instead of storing rows horizontally and columns vertically among the buffers, we store them diagonally (elements in the same color as shown in Fig. 5). This allows any row or column to be accessible in a single cycle. As seen in Fig. 5, in the traditional scheme, it is impossible to access the column of $(i, 0)$ elements in the same cycle because they are all stored in the same buffer. With the diagonal partitioning scheme, not only can all $(i, 0)$ elements

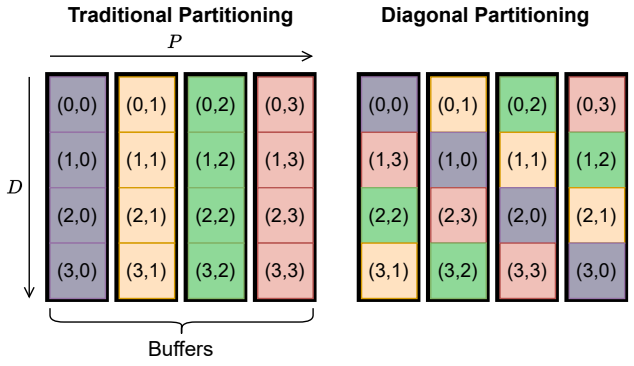


Figure 5: Traditional vs diagonal partitioning buffer element layout.

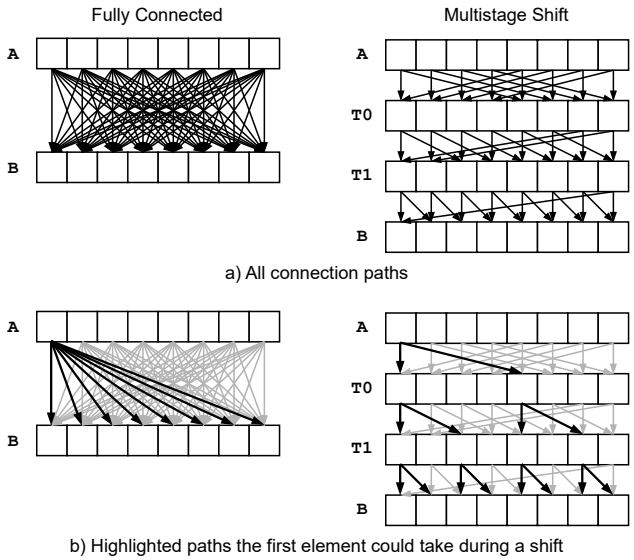


Figure 6: Element rotation scheme with lower congestion and resource utilization.

be accessed in a single cycle by reading the diagonals, but also we still maintain single cycle access to the rows.

The added complexity is that it is no longer known at compile-time which buffer a given reader needs to access, since its desired element could be in any buffer depending on what index it is reading. This requires the ports to be fully connected, which uses too many resources and is very difficult, if not impossible, to place and route especially for meaningfully sized dimensions.

We solve this by applying a progressive multistage element shift to rotate values when reading, as shown in Fig. 6 (right half). This reduces fanout, the size of the multiplexers connecting ports, and the number of connections. This enables practical implementation for large buffer sizes. The only limitation is that the implemented buffer sizes need to be square and the dimensions must be a power of 2. This is needed in order to allow for efficient modulo operations in the shifting stages.

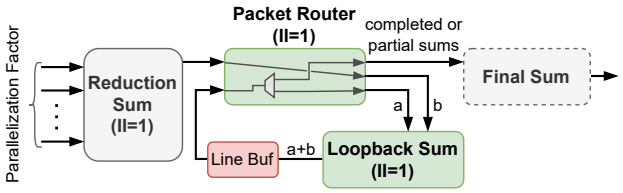


Figure 7: Contraction unit architecture diagram.

When reading using the diagonal partitioning scheme, we first load elements from each buffer into a temporary array. Then, through a series of temporary arrays, we apply a shift to rotate the row back to its original order. By reducing the number of cross-connections and removing the need for a fully connected layer, we allow for resource efficient II=1 orthogonal row/column access.

Although the implemented capacity of the buffer may need to be larger than the used space, it is not necessary that the stored data be square or a power of 2. The complete transpose will be performed in $\max(N, M)$ cycles, where N and M are the dimensions of the used space, not the implemented capacity.

4.4 Contraction

The contraction stage is responsible for accumulating all of the produced primitive ERIs into a single set of contracted ERIs. This is done by accumulating the set of ERIs that is produced every cycle from all the primitive ERI producing kernels as determined by the parallelism factor. This creates a challenge as UltraScale+ FPGAs do not natively support II=1 floating-point accumulation due to the RAW (read-after-write) dependency.

An additional complexity posed by medium to large-sized quartet classes is that it may take more than one cycle to produce all the ERIs for a single quartet. This creates a distance between sets of values to be accumulated, which varies among quartet classes.

To implement an accumulation unit capable of performing this addition without interruption between quartets, we design a streaming contraction unit shown in Fig. 7. This unit is comprised entirely of flush-able pipelines, with all except the packet router, which has complex control logic, being free-running. The packet router gets information during runtime on how many sequential values should be accumulated together, along with the distance between values that need to be accumulated. It then uses a loopback mechanism where it can send pairs of packets to be summed that will come back to it after a hardware loopback latency delay. When a value loops-back, it can either be directed into the loopback sum again for further accumulation, or, if accumulation is complete, be directed out to the final stage.

This loopback delay creates two cases that need to be handled: 1) when the accumulation dependency distance is less than the loopback delay, and 2) when the accumulation dependency distance is greater or equal to the loopback delay.

For case 1, since the addition of the loopback does not occur fast enough to accumulate directly, we instead create a set of partial sums. It is important to note that the number of partial sums needs to be divisible by the target accumulation dependency distance. These partial sums are fed through the loopback in an interleaved

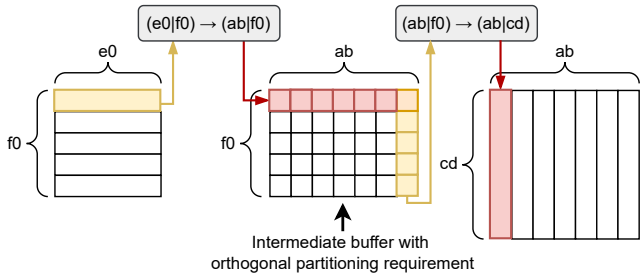


Figure 8: HRR steps input and output partitioning requirements.

fashion allowing the overall system to still run at an II=1. Once they are ready, they are sent into a final sum stage, where, in a separate pipeline, they are combined from partial into complete sums. Since the final sum stage only adds values and does not perform a running accumulation, we do not have a RAW dependency issue.

For case 2, since the accumulation dependency distance is sufficiently large, the results of an accumulation will be ready when the next value needs to be added to that running sum, allowing us to satisfy the RAW dependency without partial sums. We just need to add a line buffer to prevent deadlocks for larger distances to hold values to wait for their next accumulation. In this case the final sum stage can be bypassed, and in the case of $\mu = 4, 5$, can be removed entirely, since the accumulation dependency distance for all the large quartet classes is greater than the loopback delay.

The packet router enables the dynamic management of ERIs based on the given quartet class known only at runtime.

4.5 Horizontal Recurrence Relations

The HRR computation involves transforming contracted ERIs $(e0|f0)$ through intermediate steps into the final $(ab|cd)$ integrals. This involves two main transformations: $(e0|f0) \rightarrow (ab|f0)$ and $(ab|f0) \rightarrow (ab|cd)$ as illustrated in Fig. 8. Each transformation requires recurrence relations that step through angular momentum components, necessitating access patterns orthogonal to the data production order. For example, producing $(ab|f0)$ involves parallel computation across $e0$ dimensions iterated over $f0$, while the next step requires parallel access across $f0$ iterated over ab.

For high angular momenta, the size of these intermediate ERI sets, i.e., $(ab|f0)$, becomes too large for full partitioning. We apply the diagonal buffer partitioning scheme with a PIPO to efficiently manage the required data transpose between these HRR stages without performance degradation.

Additionally, the sets e, f, a, b, c, and d, each represent a set of 3-D integer vectors, i.e., (l_x, l_y, l_z), such that $l_x+l_y+l_z = L$. The sets a, b, c, and d are defined by the angular momenta (L) of the given orbital for that component. The e and f sets are the union of sets for each angular momentum value in the interval of $[L_a, L_a+1, \ldots, L_a+L_b]$ and $[L_c, L_c+1, \ldots, L_c+L_d]$ for e and f respectively. Storing these naively in multi-dimensional arrays leads to significant space underutilization, as only specific combinations are valid. To address this issue, we employ linearized indexing. Each valid 3-D vector within a set is mapped to a unique 1-D index. The recurrence relations, which involve operations like adding $(1, 0, 0)$, $(0, 1, 0)$, or $(0, 0, 1)$ to an index vector, are implemented using pre-computed lookup functions. These functions take the linearized index of an input vector and return the linearized index corresponding to the result.

This combination of a diagonal buffer transpose for managing large intermediates and linearized indexing with lookup functions for sparse vector access allows for an efficient implementation of the HRR stages, managing the overall 12-D calculation in a compact and efficient manner.

5 Results and Analysis

In this section, we first present the experimental setup. Then we evaluate SORCERI against the state-of-the-art CPU and GPU libraries, libint [31] and libintx [3], performing contracted ERI computation, and SERI, the prior best-performing FPGA design, which only solves primitive ERIs. This comparison is evaluated in terms of energy efficiency as well as throughput. Finally, we present the resource utilization breakdown and clock frequency of our SORCERI design on the FPGA.

5.1 Experimental Setup

To enable a comparison to SERI we use the performance metric GERIS (10^9 ERIs per second) in terms of primitive ERI equivalents. For evaluating energy efficiency, GERIS/W (GERIS per watt) is used. The evaluation of SORCERI includes the transfer of resulting ERIs back to the host memory, while the SERI and libintx evaluations exclude this step which means their end-to-end throughput should be lower than reported. Additionally, the GPU libintx library is based on a new formalism [19] of the McMurchie-Davidson algorithm [17] that does not require the calculation of Rys roots and weights. A summary of the comparison is shown in Table 3.

Table 3: Comparison of stages performed between benchmarked libint (CPU), libintx (GPU), SERI, and SORCERI implementations.

	libint	libintx	SERI	SORCERI
Start from basis set	✓	✓	✗	✓
Rys roots & weights	-	-	✗	✓
Primitive ERIs	✓	✓	✓	✓
Contraction	✓	✓	✗	✓
Compression	✗	✗	✓	✗
Result ERIs in host mem	-	✗	✗	✓

The benchmark molecular system is a 32 site 4 × 4 × 2 cubic lattice with a lattice parameter of 1 Å. Synthetic STO-4G, STO-5G, and STO-6G basis sets are used for benchmarking since highly contracted GTOs are the focus of this work.

SORCERI is evaluated on an AMD/Xilinx Alveo U280 FPGA board. The design is synthesized and built with Vitis & Vivado 2023.2 with C++ host code using the XRT APIs. Execution time is measured from the start to finish of the FPGA kernel execution, which includes the data transfer of results directly into host memory, and power consumption is measured using the XRT APIs.

The AMD EPYC 7713 CPU (64 cores) is evaluated using the libint library [31] (version 2.7.2) parallelized with MPI (Open MPI, version 4.1.1) and complied with GCC 11.2.0 using `-O3 -march=znver3`.

Socket power consumption is measured using RAPL counters during execution [29].

The Nvidia A40 GPU (CC=8.6) is evaluated using the libintx library [3] compiled with nvcc (CUDA 11.8). The throughput and power consumption are measured for the CUDA kernel execution only, excluding CPU-GPU data transfers.

SERI was also implemented on an AMD/Xilinx Alveo U280 FPGA and its performance data for the 21 bitstreams that cover quartet classes up to $(dd|dd)$ are taken from the published results [26].

5.2 Performance Gain Breakdown

To better attribute SORCERI's performance improvements, we perform an ablation study where each optimization is toggled *on* vs. *off*. Table 4 summarizes the end-to-end speedup (range) of each optimization across the supported quartet classes. Speedups vary by quartet class and are not strictly additive due to bottleneck shifts and interactions. For staged element rotation, without this optimization, the design is unable to place-and-route due to congestion.

Table 4: Ablation study summary: end-to-end speedup from toggling each optimization individually.

Optimization	Speedup
Sec 4.2 On-chip Rys & contraction (removing host–FPGA bottleneck)	~8x–22x
Sec 4.3 Efficient buffer transpose:	
Diagonal partitioning	~15x–16x
Staged element rotation	*enables routing*
Sec 4.4 Contraction loopback II=1 accumulate	~3x
Sec 4.5 New horizontal recurrence relations:	
Algorithmic benefit	~1x–1.45x
Architecture optimizations	~2x–36x

5.3 Overall Comparison

5.3.1 Energy Efficiency Comparison. For energy efficiency, Fig. 9 shows that SORCERI achieves an average 5.96x better performance per watt against libint on EPYC 7713 64-core CPU, 1.99x better against libintx on A40 GPU, and 1.16x better against SERI. Overall, SORCERI offers superior energy efficiency compared to the CPU, and is on par with the prior best-performing primitive ERI FPGA design despite performing more calculation stages. SORCERI, being able to achieve up to 0.58 GERIS/W, also has the low absolute power consumption ranging between 61-77 W compared to 241-246 W of libint on the CPU, 120-163 W of libintx on the GPU.

As seen in Fig. 9, when comparing energy efficiency between highly contracted basis sets, different degrees of contraction (i.e. STO-4G/5G/6G) have nearly the same performance (per watt) in terms of primitive ERIs. This is because the majority of the calculations, specifically Rys, VRR, and GQ, are dedicated to solving for the large number of primitive ERI components.

5.3.2 Throughput Comparison. To simplify the throughput performance comparison, we average the results between the STO-4G, STO-5G, and STO-6G contraction degrees for each quartet class. We note that SERI does not have separate performance for different degrees of contraction since it only computes primitive ERIs.

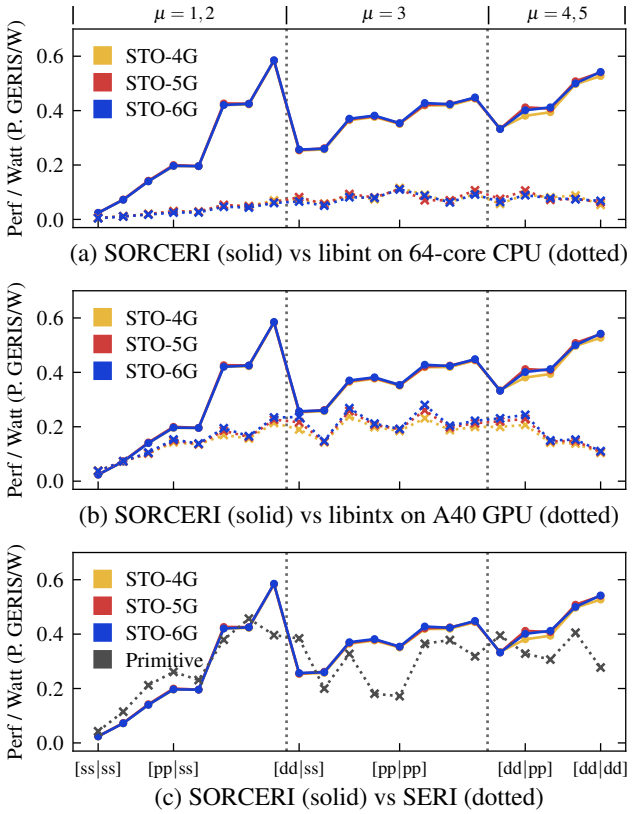


Figure 9: Energy efficiency comparison in primitive ERI equivalents per watt between SORCERI, libint on a 64-core CPU, libintx on an A40 GPU, and best previous FPGA design. Curves correspond to contraction degrees STO-4G/5G/6G.

Fig. 10 shows that SORCERI achieves an average 1.50x speedup against the 64-core CPU, 0.96x speedup against the A40 GPU, and 1.57x speedup against SERI. Please note the PCIe data transfers are excluded in the SERI and libintx benchmark on the Nvidia A40 GPU, while SORCERI includes the transfer of result ERIs back to host memory. This demonstrates that our design is competitive in the majority of mid-to-large quartet classes and achieves significant speedups for smaller quartet classes compared to the CPU, while performing on par with libintx on GPU. Compared to SERI, SORCERI outperforms for most quartet classes despite integrating additional computation stages for contracted ERIs.

As a note, SERI performs worse relative to the CPU and GPU in this comparison than in the results of the SERI paper [26], because we are comparing SERI's primitive ERI performance to contracted ERI performance (though in terms of primitive equivalents). As highlighted in Alg. 1, contracted ERIs can be computed more efficiently since the HRRs are moved out of the primitive inner loop.

All overlays show a trend of increased performance for larger quartet classes in their set, as ERIs per quartet increases. The slight drop in performance for $(pp|pp)$ occurs because that quartet class has one batch which is only partially filled since the number of ERIs per quartet is not evenly divisible by the supported batch sizes, causing one cycle to perform fewer calculations.

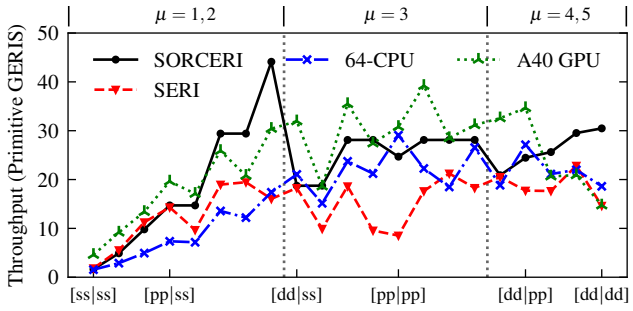


Figure 10: Average throughput comparison between SORCERI, libint on 64-core CPU, libintx on an A40 GPU, and best previous FPGA design.

5.4 Resource Utilization and Frequency

Table 5: Post place-and-route resource utilization and achieved frequency of each overlay on an Alveo U280 FPGA.

Overlay	Par	DSP	LUT	FF	BRAM	Freq
$\mu = 1, 2$	8	7549 (84%)	946K (73%)	1.2M (45%)	487.5 (24%)	208 MHz
$\mu = 3$	7	6634 (74%)	974K (75%)	1.3M (50%)	481.5 (24%)	227 MHz
$\mu = 4, 5$	7	5898 (65%)	944K (72%)	1.2M (45%)	635.5 (32%)	213 MHz

Table 5 shows the post place and route resource utilization and achieved frequency for each of the overlays. The $\mu = 1, 2$ overlay is able to achieve a higher degree of parallelism due to lower LUT utilization per instance compared to the other two. Our streaming and free-running pipeline design enables SORCERI to achieve a high resource utilization of up to 84% of DSPs, while still achieving more than 200 MHz clock frequency with Vitis HLS designs.

Table 6: Post place-and-route percentage of FPGA resource utilization for each calculation stage and overlay.

Overlay:	$\mu = 1, 2$ (Par 8)			$\mu = 3$ (Par 7)			$\mu = 4, 5$ (Par 7)		
Stage	DSP	LUT	BRAM	DSP	LUT	BRAM	DSP	LUT	BRAM
Dispatch	2%	4%	~0%	1%	3%	~0%	1%	3%	~0%
Rys Poly.	26%	26%	8%	17%	14%	6%	9%	10%	4%
Aux. Coeffs	10%	7%	0%	6%	4%	0%	6%	4%	0%
VRRs	16%	8%	0%	22%	12%	0%	16%	10%	0%
GQ	23%	10%	0%	23%	23%	0%	30%	27%	9%
Contraction	5%	5%	0%	3%	3%	2%	2%	2%	~0%
HRRs	1%	1%	0%	1%	5%	3%	1%	4%	3%
(Others)	1%	12%	16%	1%	11%	13%	~0%	12%	16%

Table 6 shows the resource utilization breakdown between stages across all three overlays. The Others category includes components such as FIFOs in between kernels, smaller stages such as Start and Store, and the static region. The overlay for the largest quartets, $\mu = 4, 5$, has the highest utilization of Gaussian quadrature, with significantly fewer resources spent on earlier stages such as Rys polynomial and auxiliary coefficient calculations. For $\mu = 3$, the VRR stage has a higher utilization compared to the other overlays, consistent with the expectations from Fig. 2. Lastly, $\mu = 1, 2$ has very high utilization for the Rys polynomial and auxiliary coefficient stages. Since each overlay balances trade-offs to accommodate multiple quartet classes, the proportional split does not match any single quartet exactly. However, the observed resource allocations do follow the expected trends in the compute demand across the implemented classes.

6 Related Work

Since ERI computation plays a critical role in accurate quantum chemistry simulations, dedicated libraries have been developed for CPUs [14, 19, 28, 31], GPUs [1–3, 22, 32], and FPGAs[26, 33]. However, ERI computations require large intermediates and have irregular memory access, limiting performance and energy efficiency on CPUs and GPUs. Quantitative comparisons to CPU and GPU libraries are presented in Section 5.3. Prior FPGA implementations have been limited in functionality, targeting either only *s* orbital [15] or the primitive ERIs [26, 33]. A recent work [18] leverages Versal AI engines on AMD/Xilinx VCK5000 card to accelerate primitive ERI computations for classes up to $[hh|hh]$ and demonstrates energy efficiency comparable to SERI on an Alveo U280 FPGA, but does not support contracted ERI computations.

7 Conclusion

In this work, we presented SORCERI, the first practical FPGA accelerator for performant and energy-efficient computation of highly contracted ERIs in quantum chemistry. By integrating all computation stages into a single FPGA design, SORCERI eliminates the prior bottleneck of CPU-FPGA communication while preserving full floating-point accuracy in the computation of the 21 commonly used ERI classes, covering the *s*, *p*, and *d* orbitals that are generally required in realistic AIMD simulations. The overlay enables calculation of those ERI classes with only 3 instead of 21 unique bitstreams needed by previous designs. Our evaluation demonstrates that SORCERI has a peak throughput of 44.11 GERIS (10^9 ERIs per second) and delivers 5.96x, 1.99x, and 1.16x better energy efficiency over libint on AMD EPYC 7713 CPU (64-core), libintx on Nvidia A40 GPU, and SERI, the previous best FPGA kernel for primitive ERIs on an Alveo U280 FPGA, respectively. Since SORCERI exhibits high throughput and excellent performance per watt for the computation of highly contracted ERIs, it paves the way for energy-efficient, large-scale quantum-mechanics-based atomistic simulations on FPGA-equipped HPC clusters.

Acknowledgements

We acknowledge the partial support from NSERC Discovery Grant RGPIN-2019-04613, DGECR-2019-00120, and CFI John R. Evans Leaders Fund. We also thank the computing time provided to us on the high-performance computer Noctua 2 at the NHR Center PC2. This center is jointly supported by the Federal Ministry of Research, Technology and Space and the state governments participating in the National High-Performance Computing (NHR) joint funding program (www.nhr-verein.de/en/our-partners).

References

[1] Melisa Alkan, Buu Q. Pham, Daniel Del Angel Cruz, Jeff R. Hammond, Taylor A. Barnes, and Mark S. Gordon. 2024. LibERI–A Portable and Performant Multi-GPU Accelerated Library for Electron Repulsion Integrals via OpenMP Offloading and Standard Language Parallelism. *The Journal of Chemical Physics* 161, 8 (08 2024), 082501. doi:10.1063/5.0215352

[2] Andrey Asadchev and Edward F. Valeev. 2023. High-Performance Evaluation of High Angular Momentum 4-Center Gaussian Integrals on Modern Accelerated Processors. *The Journal of Physical Chemistry A* 127, 51 (2023), 10889–10895. doi:10.1021/acs.jpca.3c04574

[3] Andrey Asadchev and Edward F. Valeev. 2024. 3-Center and 4-Center 2-Particle Gaussian AO Integrals on Modern Accelerated Processors. *The Journal of Chemical Physics* 160, 24 (2024), 244109. doi:10.1063/5.0217001

[4] Carsten Bauer, Tobias Kenter, Michael Lass, Lukas Mazur, Marius Meyer, Holger Nitsche, Heinrich Riebler, Robert Schade, Michael Schwarz, Nils Winnwa, Alex Wiens, Xin Wu, Christian Plessl, and Jens Simon. 2024. Noctua 2 Supercomputer. *Journal of Large-Scale Research Facilities* 9 (2024). doi:10.17815/jlsrf-8-187

[5] Christophe Bobda, Joel Mandebi Mbongue, Paul Chow, Mohammad Ewais, Naif Tarafdar, Juan Camilo Vega, Ken Eguro, Dirk Koch, Suranga Handagala, Miriam Leeser, Martin Herbordt, Hafsah Shahzad, Peter Hofste, Burkhard Ringlein, Jakub Szefer, Ahmed Sanaullah, and Russell Tessier. 2022. The Future of FPGA Acceleration in Datacenters and the Cloud. *ACM Trans. Reconfigurable Technol. Syst.* 15, 3, Article 34 (Feb. 2022), 42 pages. doi:10.1145/3506713

[6] Jason Cong, Muhuan Huang, Di Wu, and Cody Hao Yu. 2016. Invited - Heterogeneous Datacenters: Options and Opportunities. In *Proceedings of the 53rd Annual Design Automation Conference (DAC'16)*. Article 16, 6 pages. doi:10.1145/2897937.2905012

[7] Sambit Das, Bikash Kanungo, Vishal Subramanian, Gourab Panigrahi, Phani Motamarri, David Rogers, Paul M. Zimmerman, and Vikram Gavini. 2023. Large-Scale Materials Modeling at Quantum Accuracy: Ab Initio Simulations of Quasicrystals and Interacting Extended Defects in Metallic Alloys. In *SC'23: International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–12. doi:10.1145/3581784.3627037

[8] Michel Dupuis, John Rys, and Harry F. King. 1976. Evaluation of Molecular Integrals Over Gaussian Basis Functions. *The Journal of Chemical Physics* 65, 1 (07 1976), 111–116. doi:10.1063/1.432807

[9] N. Flocke and V. Lotrich. 2008. Efficient Electronic Integrals and Their Generalized Derivatives for Object Oriented Implementations of Electronic Structure Calculations. *Journal of Computational Chemistry* 29, 16 (2008), 2722–2736. doi:10.1002/jcc.21018

[10] Richard A. Friesner. 2005. Ab Initio Quantum Chemistry: Methodology and Applications. *Proceedings of the National Academy of Sciences* 102, 19 (2005), 6648–6653. doi:10.1073/pnas.0408036102

[11] Manuel Guidon, Florian Schiffmann, Jürg Hutter, and Joost VandeVondele. 2008. Ab initio Molecular Dynamics Using Hybrid Density Functionals. *The Journal of Chemical Physics* 128, 21 (06 2008), 214104. doi:10.1063/1.2931945

[12] W. J. Hehre, R. F. Stewart, and J. A. Pople. 1969. Self-Consistent Molecular-Orbital Methods. I. Use of Gaussian Expansions of Slater-Type Atomic Orbitals. *The Journal of Chemical Physics* 51, 6 (09 1969), 2657–2664. doi:10.1063/1.1672392

[13] T. Helgaker, P. Jorgensen, and J. Olsen. 2014. *Molecular Electronic-Structure Theory*. John Wiley & Sons Ltd.

[14] Hua Huang and Edmond Chow. 2018. Accelerating Quantum Chemistry with Vectorized and Batched Integrals. In *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*. 529–542. doi:10.1109/SC.2018.00044

[15] Volodymyr Kindratenko, Ivan Ufimtsev, and Todd Martínez. 2008. 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.

[16] Thomas D. Kühne, Marcella Iannuzzi, Mauro Del Ben, Vladimir V. Rybkin, Patrick Seewald, Frederick Stein, Teodoro Laino, Rustam Z. Khaliullin, Ole Schütt, Florian Schiffmann, Dorothea Golze, Jan Wilhelm, Sergey Chulkov, Mohammad Hossein Bani-Hashemian, Valéry Weber, Urban Borštnik, Mathieu Taillefumier, Alice Shoshana Jakobovits, Alfio Lazzaro, Hans Pabst, Tiziano Müller, Robert Schade, Manuel Guidon, Samuel Andermatt, Nico Holmberg, Gregory K. Schenter, Anna Hehn, Augustin Bussy, Fabian Belleflamme, Gloria Tabacchi, Andreas Glöß, Michael Lass, Iain Bethune, Christopher J. Mundy, Christian Plessl, Matt Watkins, Joost VandeVondele, Matthias Krack, and Jürg Hutter. 2020. CP2K: An Electronic Structure and Molecular Dynamics Software Package - Quickstep: Efficient and Accurate Electronic Structure Calculations. *The Journal of Chemical Physics* 152, 19 (05 2020), 194103. doi:10.1063/5.0007045

[17] Larry E McMurchie and Ernest R Davidson. 1978. One- and two-electron integrals over cartesian gaussian functions. *J. Comput. Phys.* 26, 2 (1978), 218–231. doi:10.1016/0021-9991(78)90092-X

[18] Johannes Menzel and Christian Plessl. 2025. Efficient and Distributed Computation of Electron Repulsion Integrals on AMD AI Engines. In *2025 IEEE 33rd Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. 95–104.

[19] Frank Neese. 2023. The SHARK Integral Generation and Digestion System. *Journal of Computational Chemistry* 44, 3 (2023), 381–396. doi:10.1002/jcc.26942

[20] Andrew Putnam, Adrian M. Caulfield, Eric S. Chung, Derek Chiou, Kypros Constantinides, John Demme, Hadi Esmaeilzadeh, Jeremy Fowers, Gopi Prashanth Gopal, Jan Gray, Michael Haselman, Scott Hauck, Stephen Heil, Amir Hormati, Joo-Young Kim, Sitaram Lanka, James Larus, Eric Peterson, Simon Pope, Aaron Smith, Jason Thong, Phillip Yi Xiao, and Doug Burger. 2014. A Reconfigurable Fabric for Accelerating Large-Scale Datacenter Services. In *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*. 13–24. doi:10.1109/ISCA.2014.6853195

[21] J. Rys, M. Dupuis, and H. F. King. 1983. Computation of Electron Repulsion Integrals Using the Rys Quadrature Method. *Journal of Computational Chemistry* 4, 2 (1983), 154–157. doi:10.1002/jcc.540040206

[22] Tosaporn Sattasathuchana, Peng Xu, Dossay Oryspayev, Colleen Bertoni, Luke B. Roskop, and Mark S. Gordon. 2024. Performance Portability of Electron Repulsion Integrals and Their Related Methods across Peta to Exascale Architectures. In *SC24-W: Workshops of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1943–1954. doi:10.1109/SCW63240.2024.00244

[23] Hafsah Shahzad, Ahmed Sanaullah, and Martin Herbordt. 2021. Survey and Future Trends for FPGA Cloud Architectures. In *2021 IEEE High Performance Extreme Computing Conference (HPEC)*. 1–10. doi:10.1109/HPEC49654.2021.9622807

[24] J. C. Slater. 1930. Atomic Shielding Constants. *Physical Review* 36 (Jul 1930), 57–64. Issue 1.

[25] Daniel G. A. Smith, Lori A. Burns, Andrew C. Simmonett, Robert M. Parrish, Matthew C. Schieber, Raimondas Galvelis, Peter Kraus, Holger Kruse, Roberto Di Remigio, Asem Alenaizan, Andrew M. James, Susi Lehtola, Jonathon P. Misiewicz, Maximilian Scheurer, Robert A. Shaw, Jeffrey B. Schriber, Yi Xie, Zachary L. Glick, Dominic A. Sirianni, Joseph Senan O'Brien, Jonathan M. Waldrop, Ashutosh Kumar, Edward G. Hohenstein, Benjamin P. Pritchard, Bernard R. Brooks, III Schaefer, Henry F., Alexander Yu. Sokolov, Konrad Patkowski, III DePrince, A. Eugene, Uğur Bozkaya, Rollin A. King, Francesco A. Evangelista, Justin M. Turney, T. Daniel Crawford, and C. David Sherrill. 2020. PSI4 1.4: Open-Source Software for High-Throughput Quantum Chemistry. *The Journal of Chemical Physics* 152, 18 (05 2020), 184108. doi:10.1063/5.0006002

[26] Philip Stachura, Guanyu Li, Xin Wu, Christian Plessl, and Zhenman Fang. 2024. SERI: High-Throughput Streaming Acceleration of Electron Repulsion Integral Computation in Quantum Chemistry using HBM-based FPGAs. In *2024 34th International Conference on Field-Programmable Logic and Applications (FPL)*. 60–68. doi:10.1109/FPL64840.2024.00018

[27] Ryan Stocks, Jorge L. Galvez Vallejo, Fiona C. Y. Yu, Calum Snowdon, Elise Palethorpe, Jakub Kurzak, Dmytro Bykov, and Giuseppe M. J. Barca. 2024. Breaking the Million-Electron and 1 EFLOP/s Barriers: Biomolecular-Scale Ab Initio Molecular Dynamics Using MP2 Potentials. In *SC'24: International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–12. doi:10.1109/SC41406.2024.00015

[28] Qiming Sun. 2024. The Updates in Libcint 6: More Integrals, API Refinements, and SIMD Optimization Techniques. *The Journal of Chemical Physics* 160, 17 (05 2024), 174116.

[29] The Linux Kernel Development Community. 2024. Power Capping Framework. https://www.kernel.org/doc/html/latest/power/powercap/powercap.html. [Accessed: March 27th, 2024].

[30] TOP500. 2025. TOP500 List - November 2024. https://top500.org/lists/top500/list/2024/11. Accessed: April 15, 2025.

[31] E. F. Valeev. 2024. Libint: A library for the evaluation of molecular integrals of many-body operators over Gaussian functions. http://libint.valeyev.net/.

[32] Jorge Luis Galvez Vallejo, Giuseppe M.J. Barca, and Mark S. Gordon and. 2023. High-performance GPU-accelerated evaluation of electron repulsion integrals. *Molecular Physics* 121, 9-10 (2023), e2112987. doi:10.1080/00268976.2022.2112987

[33] X. Wu, T. Kenter, R. Schade, T. D. Kühne, and C. Plessl. 2023. Computing and Compressing Electron Repulsion Integrals on FPGAs. In *2023 IEEE 31st Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. 162–173.