X-BQSR: Holistic Acceleration of Base Quality Recalibration for Scalable Genomic Analysis

Eugenio Marinelli*, Ivan Donchev Kabadzhov*,† and Raja Appuswamy

EURECOM

Abstract. With the continued decline in sequencing costs, genomics-based precision medicine is gaining increasing importance. The Genome Analysis Toolkit (GATK) is the standard framework for converting raw genomic data into actionable variants, but one of its most expensive steps is base quality score recalibration (BQSR), which corrects systematic sequencing errors. Despite significant work on accelerating other parts of the variant calling pipeline, BQSR has not been systematically explored on GPUs.

In this work, we analyze BQSR to identify computational and I/O bottlenecks and show that simply offloading key stages to an accelerator is insufficient to achieve overall gains. We then present X-BQSR, a holistic redesign of the entire BQSR pipeline, including I/O, compression, and kernel execution, to better leverage GPU parallelism. We implement X-BQSR using portable SYCL kernels for NVIDIA, AMD, and Intel GPUs, along with native CUDA and HIP backends for comparison, and we evaluate its performance against previous FPGA work. Our evaluation shows up to 141.7× acceleration compared to single-threaded GATK and 8.7× acceleration compared to a proprietary GPU-accelerated solution.

Keywords: base quality recalibration, genomics, GPU acceleration, high-performance computing

1 Introduction

Over the last two decades, the cost of DNA sequencing has dropped faster than Moore's law, from over $1,000,000 in 2001 to well under $1,000 today [10]. As we move into the era of $100 sequencing, genomics-driven precision medicine efforts have become widespread in both research and clinical practice. Germline genomic analysis, or the task of studying mutations that are inherited and present in every cell, is of vital importance as it has several applications, from clinical diagnosis of rare diseases in humans, to population-genomics studies that map genetic diversity in other organisms. The gold standard computational pipeline used for germline genomic analysis is the Genome Analysis Toolkit (GATK) [9,4,1] germline variant calling pipeline shown in Figure 1.

It is well known that the GATK pipeline is computationally intensive. For instance, an evaluation by Intel [12] showed that the pipeline took 1.5 days to run the pipeline for a whole-genome sequencing dataset at 30× coverage on a 36-core CPU. This naturally

* These authors contributed equally to this work.

† Corresponding author: ivan.kabadzhov@eurecom.fr

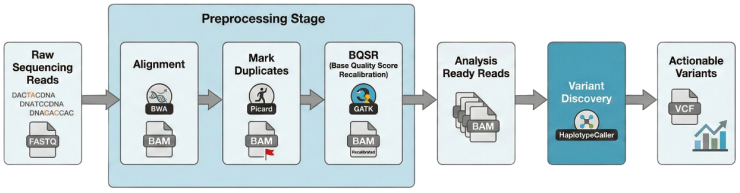


Fig. 1: GATK germline variant calling pipeline.

makes the pipeline a key bottleneck in any genomic analysis that processes multiple such sequences, and hence, a key optimization target for hardware-accelerated solutions from both industry and academia. As shown in Figure 1, Base Quality Score Recalibration (BQSR) is the final preprocessing step that recalibrates systematic errors to ensure variant discovery receives high-fidelity, analysis-ready reads. While prior research has accelerated alignment and discovery using optimized CPU threading (e.g., elPrep [3]), GPUs (e.g., CUSHAW [7]), and FPGAs [2], BQSR remains a bottleneck. To our knowledge, the BQSR stage has been the focus of only one academic study, which proposed an algorithm-hardware co-optimization strategy using FPGA by Lo et al [8]. While that work demonstrated the potential of hardware acceleration, it only targeted the BaseRecalibrator phase of BQSR, neglected the ApplyBQSR phase, and did not perform end-to-end integration within the full GATK pipeline. On the other hand, NVIDIA Parabricks library [11] provides a closed-source solution for BQSR acceleration targeting NVIDIA GPUs. But their CUDA implementation is closed source, and not portable to multi-vendor GPUs and accelerators, like Intel Flex or AMD MI250.

In this work, we bridge this gap by making several contributions. First, we present an in-depth analysis of BQSR to identify various computational and I/O bottlenecks in the CPU-based implementation and to identify GPU-offload targets for an accelerated redesign. Second, we present **X-BQSR**, a redesign of the core BQSR algorithms to unlock the massive SIMT parallelism of GPUs. In doing so, we not only develop optimized GPU kernels for performing the core computation but also perform a holistic redesign of all other aspects of BQSR including I/O, compression, memory management, and data transfers, to ensure that the improvement unlocked by GPU acceleration is not overshadowed by other hidden overheads. Third, we implement three variants of **X-BQSR** using CUDA, HIP, and SYCL. Our SYCL **X-BQSR** implementation is the first, portable, vendor-neutral, open-source implementation of BQSR that can run on multicore CPUs, integrated GPUs, and multi-vendor discrete GPUs using a single code base. Finally, we present a rigorous evaluation comparing **X-BQSR** with the original single-threaded GATK BQSR, Spark-enabled multi-threaded GATK BQSR, FPGA BQSR from Lo et al., and NVIDIA Parabricks BQSR. In doing so, we show that: (i) **X-BQSR** can outperform NVIDIA Parabricks by 3.6× to 8.7× across varying dataset scales; (ii) SYCL **X-BQSR** matches or exceeds the performance of its CUDA counterpart while remaining portable across NVIDIA, AMD, and Intel GPUs; and (iii) unlike FPGA implementations that require external data pre-formatting, our implementation serves as a drop-in replacement in the GATK germline pipeline, providing up to a 141.7× speedup at the BQSR level and a 2.7× speedup for the GATK preprocessing

Table 1: Pipeline and BQSR breakdown for HG004 (NovaSeq)

Stage	Time (min)	Share (%)
Alignment (bwa-mem2)	156.0	18.9
GATK Sort / Markdup	219.0	26.5
BQSR model build (`BaseRecalibrator`)	223.8	27.1
BQSR apply (`ApplyBQSR`)	226.8	27.5

(a) Pipeline-level breakdown

Component	Time (min)	Share (%)
Input Processing	**68.46**	**32.3**
Parsing	62.71	29.6
Decompression	5.56	2.6
Computation	**58.14**	**27.4**
`BaseRecalibrator`	38.63	18.2
`ApplyBQSR`	19.51	9.2
Output Processing	**85.50**	**40.3**
Formatting	27.50	13.0
Compression	56.98	26.8

(b) BQSR workflow breakdown

stage as a whole compared to optimized multithread CPU workflows. We plan to make our code fully open source together with the camera-ready version of our manuscript.

2 Design

Modern high-throughput sequencing (HTS) instruments read DNA one base at a time and assign a quality score to each base call that is intended to approximate the probability that the predicted base is incorrect. While instrument calibration provides a reasonable initial estimate, systematic biases commonly remain that lead to errors in estimates. These errors may depend on sequencing context (e.g., local k-mer patterns), cycle (position within the read), and lane-specific effects. Since downstream variant callers weight evidence using base qualities, miscalibration can translate into misweighted observations at scale, affecting the accuracy of variant calls [5,4]. BQSR calibrates base-quality scores by estimating empirical error rates conditioned on covariates while excluding known-variant loci. The workflow consists of two stages: (1) model construction (`BaseRecalibrator`), which takes as input a binary alignment map (BAM) file containing aligned reads and builds recalibration statistics, and (2) score application (`ApplyBQSR`), which writes a new BAM file with corrected quality scores based on the learned model.

To motivate the need for holistic acceleration of BQSR, Table 1a reports an end-to-end breakdown of the GATK preprocessing pipeline on a whole-genome sequencing (WGS) dataset (D4 from Section 3). As can be seen, the two BQSR phases together account for 54.6% of the total preprocessing time, making BQSR a high-leverage target for reducing overall workflow latency. Following Amdahl's Law, this establishes a maximum theoretical speedup of approximately $2.2\times$ for the entire pipeline if BQSR execution time were reduced to zero.

Table 1b further decomposes the total runtime of the algorithm into (i) input processing (disk I/O, BGZF decompression, and parsing), (ii) core BQSR computation, and (iii) output processing (formatting, BGZF compression, and writing). The latter is valid

only for `ApplyBQSR`, as `BaseRecalibrator` does not write out a new BAM file but only a text file containing the recalibration tables, whose I/O cost is negligible, as the tables span only a few KB. The results show that end-to-end performance is constrained not only by compute but also by input/output processing (parsing and (de)compression); `BaseRecalibrator` is dominated by input handling, whereas `ApplyBQSR` is dominated by output formatting and compression.

These measurements motivate the holistic redesign adopted by **X-BQSR**, where we designed optimized GPU kernels to handle the complex computation of various stages of BQSR (as described in Section 2.1), and wrapped these kernels in a scalable pipeline that efficiently parallelizes serialization/compression and overlaps computation with I/O to eliminate bottlenecks in the data path (as described in Section 2.2).

2.1 GPU Kernel Design

X-BQSR implements the BQSR pipeline as a three-stage SIMT-optimized architecture: (1) **Sequence Clipping** via metadata transformations, (2) **Feature Derivation** using base-parallel flattening for balanced workload distribution, and (3) **Table Aggregation** via a two-level atomic strategy. This section details how these kernels resolve the sequential bottlenecks and memory contention inherent in the original CPU algorithms.

`BaseRecalibrator` Sequence Clipping Unlike traditional CPU-based implementations that process genomic reads as individual objects, X-BQSR utilizes a Structure-of-Arrays (SoA) layout. Each field (CIGAR, sequence, quality) is stored in a large flat vector, with strings from different reads placed contiguously. We parallelize at the read level, assigning one thread to scan the CIGAR field and identify adapter regions. By using the SoA format, we implement clipping as a simple metadata update to a *span vector* that stores the starting position and length, avoiding expensive string manipulation or memory copies. This design ensures coalesced memory access and explains why the Clip stage accounts for a negligible fraction of the total runtime in our evaluation (Figure 4c).

`BaseRecalibrator` Feature Detection This stage computes two per-base binary vectors. The first (`isSNP`) marks sequencing error events by comparing each aligned read to the reference genome. This involves scanning the CIGAR string, advancing read and reference pointers according to edit operators (M, I, D) [6] and marking each aligned mismatch as an observed error event. The second (`skip mask`) indicates bases that must be excluded from training. This includes positions that (i) overlap known variant sites, (ii) contain ambiguous nucleotides, or (iii) lie outside the post-clipping span. Skip mask is generated by intersecting the read's reference alignment interval with known-site intervals and mapping the resulting locations back to read coordinates.

There are two potential parallelization strategies that can be adopted for computing these two binary vectors on the GPU, namely, a one-thread-per-read mapping or a one-thread-per-base mapping.

With one-thread-per-read mapping, each thread processes a single read and performs three steps. First, it determines the interval of the reference genome covered by

the clipped alignment. Second, it intersects this interval with the list of known variant sites to identify which reference coordinates must be ignored during recalibration. Third, the thread walks the CIGAR operations to translate those reference coordinates into the corresponding read indices. Whenever a reference position associated with a known site maps to a base in the read, the kernel marks that position in the per-read skip mask. SNP events detection is computed similarly, with each thread traversing both read and reference based on the CIGAR and comparing the bases: if they differ, a mismatch is marked in a `SNP vector`.

The base-parallel design, in contrast, is based on the key idea that a CIGAR string encodes the alignment as a sequence of pairs $< length, operation >$, where the operation can be I (insertion), D (deletion), M (match/mismatch), H (hard clip), and so on. For example, a CIGAR such as $(10M2D5M)$ describes ten aligned bases, followed by two deletions and five additional aligned bases. Thus, one possible transformation is to expand this representation so that each operation is repeated per base. In the previous example this would produce a vector such as $MMMMMMMMMMDDMMMMM$. By computing a prefix sum over this expanded vector (reset at the beginning of each read), threads can directly derive the corresponding read and reference coordinates without sequentially walking the CIGAR. This removes the intra-read dependency and produces a more uniform workload across GPU threads.

In **X-BQSR**, we implemented both these approaches. In our early evaluation, we found read-level parallelism to outperform base-level one as the transformation requires materializing additional vectors and performing a prefix scan, increasing global memory traffic. Thus, we retained the read-level implementation for this stage.

`BaseRecalibrator` Table Aggregation Once eligible bases are identified in the previous step, `BaseRecalibrator` computes the covariates associated with each position. Covariates are discrete features that capture systematic biases in sequencing output. In the `BaseRecalibrator` formulation, the covariate tuple for base i includes:

- Reported quality score Q_i: directly extracted from the Phred-encoded quality string; its cardinality is bounded (typically < 100).
- Cycle covariate $\texttt{cycle}_i$: the base index within the read, computed in a strand-aware manner such that comparable sequencing cycles map to similar values; cardinality is proportional to the maximum read length.
- Context covariate $\texttt{context}_i$: a strand-corrected k-mer centered at the base (commonly $k = 2$ or 3), encoded in a compact 2-bit representation per base; cardinality is at most $4^k + 1$ (including an invalid sentinel).

Concretely, after clipping, masking, and mismatch detection, `BaseRecalibrator` uses various vectors produced in these stages to compute the covariate tuples, for example, $(RG, Q, cycle, context)$. These tuples represent multidimensional indexes that are used to update tables recording the number of times each covariate appears and the number of mismatches with respect to the reference genome for that base quality.

In the CPU-based GATK `BaseRecalibrator` implementation, covariate construction is naturally implemented as a single pass where (i) the covariate keys are derived from the eligible bases and (ii) the tables are updated directly. As default GATK

`BaseRecalibrator` is single-threaded, there is no contention when updating the tables. However, when parallelized on the GPU, the tables will be updated simultaneously for thousands of bases. Consequently, updates must be protected with atomics and might experience significant contention. To avoid performance issues, we implemented this stage of `BaseRecalibrator` using two distinct kernels rather than one.

Covariate emission (base-parallel). The first kernel in **X-BQSR** for this stage just computes covariate indices.

Because this step has no intra-read dependencies, we parallelize it at the base level rather than at the read level. To do so, we flatten all reads into a single list of bases. Rather than assigning one thread per read, we partition this flattened list into nearly equal contiguous segments that are processed by GPU threads (or groups of threads). Each thread therefore receives approximately the same number of bases to process, independent of the original read boundaries. To determine which read a given base belongs to, we precompute a prefix-sum array over the read lengths. This array stores, for each read, the starting offset of that read in the flattened base list. Given a global base index, a thread performs a search over this prefix array to identify the corresponding read and the offset within that read. Once this mapping is known, the thread can access the appropriate entries in the previously computed vectors (skip mask, mismatch flags, context keys, and quality scores) and compute the covariate indices for that base. This strategy effectively converts the irregular workload caused by variable-length reads into a balanced distribution of work across GPU threads. As a result, threads within a warp tend to execute a similar amount of work, improving overall GPU utilization.

Scalable table aggregation (contention-aware). Once the covariate indices and the corresponding per-base SNP information have been computed, the second kernel in **X-BQSR** updates the covariate tables. Each update consists of indexing the tables with the covariate index and incrementing two values stored at that entry: (i) a counter representing how many times that covariate appears and (ii) a counter tracking the number of mismatches observed for that base. Since the tables are relatively small and the number of updates is extremely large (billions of bases), concurrent updates on the GPU would result in a very large number of atomic operations that must be serialized due to contention, drastically hurting performance. To mitigate this issue, we adopt a two-level histogram update strategy. As all tables fit in the GPU shared memory, each thread block first updates a local copy of the tables. Atomic operations within a block are significantly faster than atomics in global memory. Then, a single thread per block merges the local histogram into the global table, greatly reducing global memory contention.

ApplyBQSR Having discussed the design of **X-BQSR**'s `BaseRecalibrator` phase, we now focus on the `ApplyBQSR` phase which rewrites the quality score of each base by evaluating a hierarchical Bayesian model built using the covariate tables computed in the previous phase. Similar to the covariate-emission in the Feature Detection stage, we use base-level parallelism for redesigning `ApplyBQSR`. In particular, for each base, a thread reconstructs the covariate tuple $(\text{readGroup}, \text{qual}, \text{cycle}, \text{context})$, similar to what is done in the `BaseRecalibrator` algorithm. This tuple is used to retrieve the empirical

statistics needed to evaluate the Bayesian model and obtain the recalibrated quality score. The resulting quality score is then quantized into the final integer Phred value.

A key observation in this kernel is that many bases may share the same combination of covariates, for example when they have the same quality score, cycle, and context. As a result, the recalibrated quality score may be recomputed repeatedly for identical covariate tuples. To avoid this redundancy, we introduce a device-wide lookup table that stores the result of the computation for each unique covariate combination. The table is implemented as a flattened multidimensional array over the covariate domains (read group, quality score, context, and cycle). The table size is bounded: up to 94 quality values, roughly 1,000 context values, and up to 1,000 cycle values (500 positions across two strands). Since each entry stores a 1-byte quantized score, the table occupies about 1 MB. Because BQSR models covariates independently for each read group, the table must be replicated per read group. In practice, however, samples usually contain only 1–10 read groups, so the overall memory footprint remains small. During execution, each thread computes the covariate indexes and checks whether the recalibrated value is already present in the table. If so, it reuses the cached value directly. Otherwise, it computes the value and atomically updates the table.

2.2 Pipeline Optimizations

Having presented the design of GPU kernels in **X-BQSR**, we now describe various pipeline optimizations that ensure that the acceleration achieved by kernel offload to GPUs is not overshadowed by overheads in the input–output I/O path.

Optimizing BAM input As shown in Table 1b, BAM parsing in input processing was a major overhead for `BaseRecalibrator`. The primary obstacle to scaling BAM reading is record boundary discovery; after BGZF decompression, the BAM byte stream is a concatenation of variable-length records, and identifying record starts requires a sequential scan. We address this issue by leveraging the `.bai` index that typically accompanies coordinate-sorted BAM files in best-practice workflows. The BAI linear index provides BGZF virtual offsets that correspond to valid BAM record starts. We extract and deduplicate these virtual offsets and use them as candidate slice boundaries. Because a virtual offset encodes both a compressed block position and an uncompressed offset within the block, it represents a safe restart point for BGZF inflation, enabling deterministic search and decoding.

To balance slices, we build a lightweight BGZF block index recording block offsets and `ISIZE` values, which lets us estimate decoded slice size and choose the furthest boundary within a target budget. A final tail slice covers all remaining records.

Finally, once the slices are defined, ingestion proceeds via an asynchronous, double-buffered pipeline. Batches correspond to work units transferred to the GPU, which are fixed-size slices that fit within the device's memory. While a batch is parsed and converted to our SoA-based in-memory representation, a separate thread simultaneously reads and decompresses the next batch. This overlap hides the disk I/O and BGZF decompression latency behind the CPU-side parsing and packing work, allowing for sustained throughput to be fed to GPUs.

Optimizing BAM output While the `BaseRecalibrator` algorithm only generates and writes to disk a small file containing the recalibration tables computed, `ApplyBQSR` in addition to input and compute stages, must also write a BAM file containing the updated quality score values for each read. The output of a large BAM file can be significant, and it can dominate the end-to-end time of the entire pipeline. This is because it involves several steps: (1) serialization of the BAM record as a BAM record in a flat byte stream; (2) compression of the byte stream in blocks of at most 64 KB; (3) writing the compressed byte blocks (BGZF blocks) to a file.

In order to optimize the BAM output, we exploit multithread CPU parallelism. Each CPU thread serializes a batch coming from the GPU after quality score recalibration and enqueues the serialized byte buffer (containing the BAM records) for a writer thread to process. It is important to note that the batch can come out-of-order due to different GPUs' processing speeds; therefore, the writer is in charge of processing those batches in order. Once the correct serialized batch arrives, the writer splits it into chunks of 64 KB and assigns each chunk to different threads to be compressed. After compression, each chunk is enqueued to be written on disk.

Pipelining computation and I/O Since the pipeline end-to-end throughput is heavily impacted by input and output paths, in order to minimize the latency of the different stages, we built a pipeline where the full input path, the GPU processing stages, and the output path overlap: readers prepare batches asynchronously, GPU worker threads submit them for execution, and completed batches are returned to the host for output processing. While one batch is executing on the GPU, the next batch is prepared concurrently to keep the device busy. When the algorithm requires writing a BAM file (as in the case of `ApplyBQSR`), the batch goes through the output stage: the batches are serialized, compressed in parallel, and written on disk.

3 Evaluation

In this section, we present the results from our evaluation of **X-BQSR**.

Software Setup. We developed X-BQSR in SYCL to provide a high-performance, vendor-neutral solution for the entire BQSR pipeline. To evaluate the portability of our design, we also implemented CUDA and HIP versions of the `BaseRecalibrator` stage. We compare **X-BQSR** against single-threaded GATK and multi-threaded GATK-Spark v4.6.2.0, NVIDIA Parabricks v4.6.0-1, and an FPGA-accelerated `BaseRecalibrator` from prior work. All `X-BQSR` kernels are compiled using the Intel oneAPI DPC++ Compiler v2025.1.0. The CUDA and HIP implementations utilize the CUDA Toolkit (nvcc) v12.6 for NVIDIA hardware and HIP v6.3.4 (ROCm v6.3.3) for AMD hardware.

Hardware Setup. Benchmarks were performed on a dual-socket Intel Xeon Platinum 8454H server (128 logical cores) with 1 TB of RAM. For accelerated workloads, we utilize an NVIDIA L40S (48 GB), an AMD Instinct MI210 (64 GB), and an Intel Flex 170 (16 GB), all via PCIe 4.0. FPGA comparisons utilize a Xilinx VCU118 platform clocked at 100 MHz, connected via PCIe 3.0.

Datasets. We use six dataset samples from GIAB individuals (Table 2): the NA12878 (HG001) and three members of the Ashkenazi trio (HG002, HG003, and HG004). We

Table 2: Datasets used for validation

Dataset	Name	#Reads	Read Len
D0	HG001-HiSeq2500	168.544.418	100
D1	HG001-HiSeq4000	88.774.172	150
D2	HG002-HiSeqX	777.447.412	150
D3	HG003-HiSeqX	1.001.580.652	150
D4	HG004-NovaSeq	1.166.740.818	150
D5	HG002-HiSeq2500	898.865.538	250

use the GRCh38 reference for all experiments. After alignment, we process the reads following the GATK Best Practices workflow [1]. Because GATK BQSR is designed for error profiles of short-read data, true long-read datasets are outside the scope of this work. The output of this duplicate-marking step serves as the input for our evaluation. Across all datasets, **X-BQSR** and NVIDIA Parabricks maintain bit-perfect parity to the gold-standard single-core GATK.

In all experiments, input and output data reside on local NVMe storage to ensure consistent I/O performance. All experiments were conducted 10 times to ensure statistical rigor; the reported values represent the average of these runs, and because the results exhibited a negligible deviation of less than 5%, error bars are omitted from the figures.

3.1 Comparison with State-of-the-Art

We start the evaluation by benchmarking `X-BQSR` against the standard single-threaded GATK implementation. The results for the `BaseRecalibrator` and `ApplyBQSR` stages (Figure 2) clearly show the performance gap between traditional CPU methods and accelerated implementations. The single-threaded GATK baseline serves as the upper bound for execution time, requiring over 5 hours (18,134 seconds) for the D5 dataset. Across the production-scale datasets (D2–D5), **X-BQSR** `BaseRecalibrator` achieves speedups over the GATK gold standard of 125.0× (D2), 129.0× (D3), 141.7× (D4), and 136.3× (D5). For `ApplyBQSR`, **X-BQSR** achieves improvement ranging from 75.2× to 84.3× over single-threaded CPU execution.

We compare **X-BQSR** with the proprietary NVIDIA Parabricks-BQSR tool. In `BaseRecalibrator`, **X-BQSR** consistently outperforms Parabricks with speedups ranging from 6.1× to 8.7× for the larger datasets (D2–D5). For instance, on the D4 dataset, **X-BQSR** completes the stage in 95 seconds compared to 830 seconds for Parabricks. For `ApplyBQSR`, **X-BQSR** exhibits a consistent 3.6× to 3.9× speedup over Parabricks across the D2–D5 datasets. Specifically, **X-BQSR** processes the 1.16 billion reads of D4 in 162 seconds (vs. 616 seconds for Parabricks) and the 898 million reads of D5 in 205 seconds (vs. 743 seconds for Parabricks). As Parabricks is provided as a precompiled binary within a Docker container, its optimization strategies and kernel configurations are opaque. Thus, it is not possible for us to identify the specific causes of the performance gap with Parabricks. We ensure a fair comparison by using the latest stable version (v4.6.0-1) and following the vendor's recommended settings. In any case, these results demonstrate that our open, standards-based solution outperforms state-of-the-art proprietary ones.

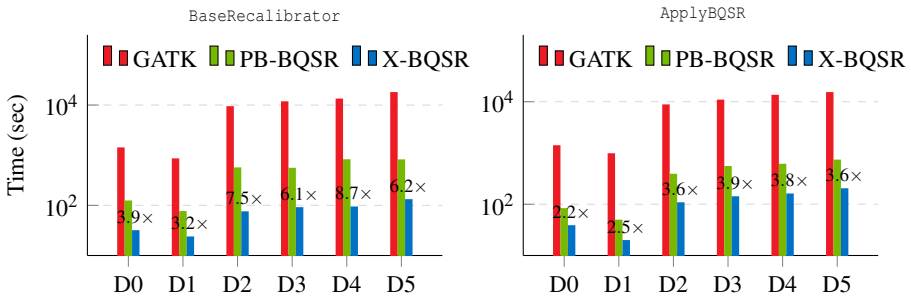


Fig. 2: Execution time comparison of **X-BQSR** against GATK and NVIDIA Parabricks (D0–D5). Speedup values (×) are relative to NVIDIA Parabricks.

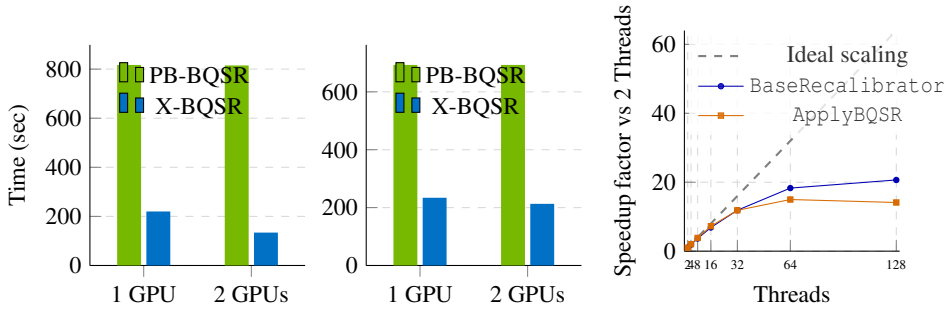
3.2 Multi-GPU and CPU Scalability

Beyond single-device performance, we evaluate the scalability of our implementation by distributing the BQSR workload across multiple accelerators using dataset D5. Each GPU operates as an independent worker in an asynchronous, double-buffered pipeline. As shown in Figures 3a and 3b, **X-BQSR** demonstrates superior scaling compared to Parabricks. In `BaseRecalibrator`, **X-BQSR** reduces execution time from 220 seconds to 134 seconds when moving from one to two GPUs, achieving 1.64× speedup. In contrast, Parabricks shows almost no improvement (815s vs. 813s). In the `ApplyBQSR` stage, achieving linear scaling is more difficult due to the requirement of a single-threaded writer process that must serialize data to disk in a specific order to maintain GATK compatibility. As the number of GPUs increases, the computational portion of the workload is reduced, but this ordering requirement leads to output I/O becoming a dominating time sink, effectively capping the maximum achievable speedup.

While our SYCL implementation is technically portable to CPU architectures, **X-BQSR** is architecturally optimized for the massive parallelism and memory hierarchy of GPUs. GATK-Spark provides a multi-threaded CPU baseline via Apache Spark. We excluded it from our primary end-to-end comparison in Figure 2 because GATK-Spark is currently designated as a BETA tool by Broad Institute and is not recommended for production benchmarking. However, we show the scaling results here for completeness. Figure 3c illustrates the scalability of GATK-Spark from 2 to 128 threads on the D5 dataset. While GATK-Spark shows consistent speedup as thread counts increase, reaching a 20.66× speedup for `BaseRecalibrator` and 14.12× for `ApplyBQSR` at 128 threads relative to the 2-thread baseline, it still falls significantly short of "ideal" linear scaling. More importantly, even when utilizing all 128 threads of the dual-socket server, the CPU-based Spark implementation remains significantly slower than a single GPU running X-BQSR. Specifically, the GATK-Spark `BaseRecalibrator` stage is 3.1× slower (411s vs. 133s) and the `ApplyBQSR` stage is 6.0× slower (1,226s vs. 205s) than a single NVIDIA L40S running **X-BQSR**.

3.3 Kernel Performance and Cross-Architecture Portability

In this section, we present a study of performance portability of **X-BQSR** kernels on multi-vendor GPUs. By focusing on kernel-only execution time (Figure 4a and 4b), we



(a) GPU `BaseRecalibrator` (b) GPU `ApplyBQSR` (c) GATK-Spark Scalability

Fig. 3: Scaling D5 dataset across GPU (**X-BQSR**, Parabricks) and CPU (GATK-Spark)

isolate the algorithmic performance from host-side I/O and driver overhead. As can be seen, the SYCL-based **X-BQSR** kernels are highly competitive with native toolchains. On the D5 dataset, our `BaseRecalibrator` kernels on the NVIDIA backend outperform optimized native CUDA kernels by approximately 2.8% (76.78 s vs. 78.99 s). For AMD hardware, while our initial SYCL port demonstrated portability, the native HIP implementation provides a measurable speedup, reducing the total `BaseRecalibrator` kernel time for D5 from 195.98 s to 191.35 s. This native implementation leverages AMD-specific compiler optimizations and avoids the intermediate abstraction layers of the SYCL runtime. In the `ApplyBQSR` phase, the performance hierarchy aligns with raw hardware specifications, with the NVIDIA L40S leading, followed by the Intel Flex 170 and the AMD MI210.

Figure 4c presents the breakdown of execution time across various stages of **X-BQSR** to identify specific architectural bottlenecks. In `BaseRecalibrator`, the **Feature Detection** stage, which encompasses skip-masking, SNP detection, and context covariate emission, is the primary execution driver on AMD hardware. While we ensure high GPU occupancy, the complex instruction mix and address generation required for context emission become compute-limited on the CDNA2 SIMD units.

Conversely, the **Table Aggregation** stage highlights the efficacy of our two-level aggregation strategy. By utilizing shared-memory histograms to minimize global atomic contention, architectures with high L2-cache atomic resolution, like the AMD MI210, remain highly competitive. Intel, however, shows significant overhead in this stage, as the Xe-HPG architecture resolves atomics with higher latency, confirming that while our design successfully mitigates hardware contention, performance remains bounded by the underlying atomic throughput of the GPU.

To further understand the architectural drivers behind the performance trends observed across multi-vendor GPUs, we use two SYCL microbenchmarks to isolate the hardware limits of BQSR: (i) the contended atomics benchmark (mimicking **Table Aggregation**) and (ii) the STREAM triad benchmark (mimicking the `ApplyBQSR` phase streaming pattern). The contended-atomics benchmark performs R `fetch_add` operations into a global table of 94 bins using `sycl::atomic_ref`. The 94-bin size is chosen to match the **Cycle** covariate, which is the primary source of atomic contention

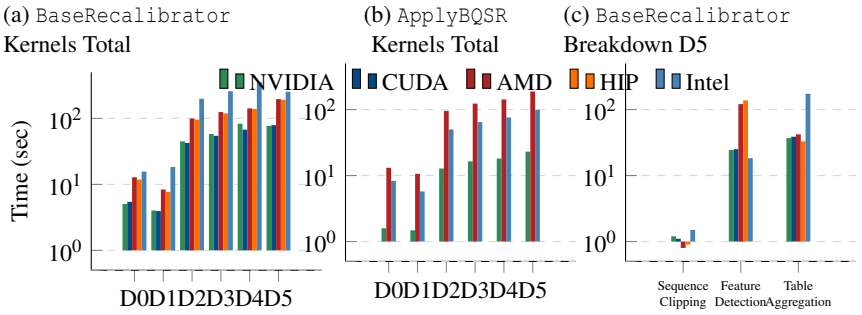


Fig. 4: Cross-architecture kernel performance comparison

Table 3: Atomic update throughput (Gop/s) and sustained device bandwidth (GB/s)

Device	Contended Atomics (Gop/s)	STREAM (GB/s)
NVIDIA L40S	3.09	689
AMD MI210	11.37	1302
Intel Flex 170	0.90	344

in `BaseRecalibrator`. The STREAM triad benchmark measures steady-state device bandwidth, mirroring the memory-access pattern of the `ApplyBQSR` phase.

Table 3 presents the results from these benchmarks. The AMD MI210 reaches 11.37 Gop/s on atomics, 4× higher than the NVIDIA L40S. Hardware counters showed that >99% of these operations are resolved within the MI210's L2 cache. This explains why AMD remains highly competitive in the **Table Aggregation** stage despite having lower raw compute clock speeds than the L40S. For memory bandwidth, the MI210 sustains 1302 GB/s, while the L40S sustains 689 GB/s. Despite this 1.9× bandwidth advantage, the `ApplyBQSR` phase is substantially slower on AMD. Profiling shows high VALU utilization on the MI210, suggesting that the SYCL instruction mix for address generation and hierarchical Bayesian calculation becomes compute-limited on the CDNA2 SIMD units, preventing the kernels from saturating the available HBM2e bandwidth. The Intel Flex 170 sustains 344 GB/s but is limited by its lower atomic throughput (0.90 Gop/s), which is consistent with its longer **Table Aggregation** times. A quantitative analysis confirms the predictive value of these metrics; while atomic throughput successfully predicts aggregation rankings, the `ApplyBQSR` performance on AMD highlights the critical importance of compiler-level instruction scheduling in cross-architecture SYCL deployments.

3.4 FPGA Comparison

To further contextualize our performance, we compare **X-BQSR** against a state-of-the-art FPGA redesign of BQSR by Lo et al. [8], which provides a solution for accelerating only the `BaseRecalibrator` stage of BQSR. They do not focus on the second phase of BQSR (`ApplyBQSR`). We present a comparison of our GPU-accelerated `BaseRecalibrator` in **X-BQSR** with Lo et al's FPGA solution for completeness. With the FPGA accelerator architecture, the host partitions the dataset into read chunks of

Table 4: BQSR execution time and speedup relative to the GATK single-core baseline

Data	GATK (s)	Extract (s)	Construct (s)	FPGA-BQSR (s / Speedup)	X-BQSR FPGA (s / Speedup)	X-BQSR GPU (s / Speedup)
1M	16.20	50.77	22.68	0.70 / 23.2×	12.80 / 1.27×	0.05 / 341.3×
2M	30.00	82.57	43.54	1.33 / 23.5×	13.12 / 2.29×	0.10 / 300.0×
4M	60.00	163.09	85.95	2.60 / 23.1×	12.91 / 4.65×	0.21 / 292.5×
8M	126.00	336.77	494.99	5.11 / 24.6×	13.77 / 9.15×	0.42 / 303.6×
16M	246.00	721.07	1,133.82	10.15 / 24.2×	15.12 / 16.3×	0.83 / 295.3×

size 100,000 and sends them to the FPGA sequentially for writing into SDRAM. Due to limited FPGA resources, we cannot process all 100,000 reads in a single iteration. Only 16 processing elements are implemented on the FPGA, so all 100,000 reads are processed across 6,250 iterations. Upon completing a chunk of reads, the BQSR accelerator asserts an interrupt to notify the host to send the next chunk. To maximize throughput, the host transfers the next chunk of reads to DRAM concurrently while the FPGA processes the previous chunk.

Unlike our GPU implementation, the FPGA architecture requires data to be preformatted into a specific binary structure. Consequently, an end-to-end FPGA workflow necessitates two additional pre-processing phases: (i) the extraction of bases, quality scores, and error metadata from the input BAM file, and (ii) the construction of the FPGA-specific data structures. Table 4 summarizes the execution times for the three phases of the FPGA workflow alongside the GATK baseline. While the standalone FPGA-BQSR compute kernel achieves a speedup of up to 24.6× over GATK, this is lower than the 40× speedup reported in the original study. Our profiling indicates that the discrepancy stems from memory latency: the Load stage (fetching data from FPGA DRAM) requires 865 clock cycles in our environment, compared to the 534 cycles reported by Lo et al. using a 4-channel DRAM configuration. At 100 MHz, this results in a BQSR execution time of approximately 0.054 s per 100,000 reads ($865 \times 6,250 \times 10 \times 10^{-9} \approx 0.054$ s).

The primary limitation of the FPGA approach is revealed when considering end-to-end integration. The required pre-processing phases (Extract and Construct) take 4× to 7× longer than the entire single-threaded GATK execution, effectively negating the gains of the hardware accelerator. In contrast, our **X-BQSR** implementation provides a holistic redesign that accelerates both I/O and computation. By eliminating hidden pre-computation overheads, **X-BQSR** serves as a seamless drop-in replacement for GATK, providing true end-to-end acceleration for germline variant calling pipelines.

4 Conclusion

In this work, we focused on BQSR, one of the most time-consuming steps of the widely used GATK germline variant calling pipeline. We motivated the need for hardware acceleration of BQSR and showed how focusing only on GPU acceleration of the computational part, as done by prior work, will only yield marginal improvement in overall time due to I/O bottlenecks. We presented **X-BQSR**, our holistic redesign with several

I/O and compute optimizations, and showed how **X-BQSR** can provide up to 141.7× improvement over GATK and up to 8.7× improvement over closed-source solutions while being vendor-agnostic, open source[1], and performance portable. In future work, we first plan to integrate **X-BQSR** with other preprocessing stages (alignment, sorting and mark duplicates) and then with downstream variant callers to develop a fully open, standards-based, GPU-accelerated solution for scaling genomic data analysis.

Acknowledgments. This work was supported by the European Union Horizon 2020 programme [101136962]; UK Research and Innovation (UKRI) under the UK Government's Horizon Europe funding guarantee [10098097, 10104323] and the Swiss State Secretariat for Education, Research and Innovation (SERI). Additional support was provided by the Horizon Europe Project "SYCLOPS", funded by the European Union HE Research and Innovation programme under grant agreement No. 10109287.

Disclosure of Interests. The authors have no competing interests to declare that are relevant to the content of this article.

References

1. Broad Institute: Genome analysis toolkit (gatk) 4 (2019)
2. Chen, Y., Schmidt, B., Maskell, D.L.: A hybrid short read mapping accelerator. BMC Bioinformatics (2013)
3. Decap, D., Fostier, J., Reumers, J., Costanza, P., Herzeel, C.: elprep: high-performance preparation of sequence alignment/map files for variant calling. PLOS ONE (2015)
4. DePristo, M.A., Banks, E., Poplin, R., Garimella, K.V., Maguire, J.R., Hartl, C., Philippakis, A.A., del Angel, G., Rivas, M.A., Hanna, M., McKenna, A., Fennell, T.J., Kernytsky, A.M., Sivachenko, A.Y., Cibulskis, K., Gabriel, S.B., Altshuler, D., Daly, M.J.: A framework for variation discovery and genotyping using next-generation dna sequencing data. Nature Genetics (2011)
5. Ewing, B., Hillier, L., Wendl, M.C., Green, P.: Base-calling of automated sequencer traces using phred. i. accuracy assessment. Genome Research (1998)
6. Li, H., Handsaker, B., Wysoker, A., Fennell, T., Ruan, J., Homer, N., Marth, G., Abecasis, G., Durbin, R.: The sequence alignment/map format and SAMtools. Bioinformatics (2009)
7. Liu, Y., Schmidt, B., Maskell, D.L.: CUSHAW: a CUDA compatible short read aligner to large genomes based on the burrows–wheeler transform. Bioinformatics (2012)
8. Lo, M., Fang, Z., Wang, J., Zhou, P., Chang, M.C.F., Cong, J.: Algorithm-hardware co-design for bqsr acceleration in genome analysis toolkit. In: FCCM (2020)
9. McKenna, A., Hanna, M., Banks, E., Sivachenko, A., Cibulskis, K., Kernytsky, A., Garimella, K., Altshuler, D., Gabriel, S., Daly, M., DePristo, M.A.: The genome analysis toolkit: a mapreduce framework for analyzing next-generation dna sequencing data. Genome Research (2010)
10. NHGRI: The cost of sequencing a human genome (2019)
11. NVIDIA Corporation: Nvidia parabricks (2020)
12. Prabhakaran, A., Gururaj, K., Naik, M., Gopalan, S., Shargorodskiy, A., Brau, E.: Accelerate genomics research with the broad-intel genomics. In: Intel/Broad Genomics (2017)

[1] `https://gitlab.eurecom.fr/dislab/bqsr`