Compiling Generalized Histograms for GPU

Troels Henriksen
University of Copenhagen
Copenhagen, Denmark

Sune Hellfritzsch
University of Copenhagen
Copenhagen, Denmark

Ponnuswamy Sadayappan
University of Utah
Salt Lake City, USA

Cosmin Oancea
University of Copenhagen
Copenhagen, Denmark

***Abstract*—We present and evaluate an implementation technique for histogram-like computations on GPUs that ensures both work-efficient asymptotic cost, support for arbitrary associative and commutative operators, and efficient use of hardware-supported atomic operations when applicable. Based on a systematic empirical examination of the design space, we develop a technique that balances conflict rates and memory footprint.**

We demonstrate our technique both as a library implementation in CUDA, as well as by extending the parallel array language Futhark with a new construct for expressing generalized histograms, and by supporting this construct with several compiler optimizations. We show that our histogram implementation taken in isolation outperforms similar primitives from CUB, and that it is competitive or outperforms the hand-written code of several application benchmarks, even when the latter is specialized for a class of datasets.

***Index Terms*—GPU, parallelism, functional programming**

I. Introduction

Parallel programming can be made accessible to the non-expert programmer by providing high-level building blocks, such as `map`, `scan` (prefix sum), or `reduce`, that can be implemented efficiently just once, and then reused as library routines. A key question then is: which building blocks should be provided, and how should they be implemented? In this paper we make the case for a new building block, the *generalized histogram*, which can be used in a wide range of problems, as detailed in section VI. We show how to implement it efficiently on GPUs as well as how to represent it within the IR of an optimising compiler.

Consider k-means clustering, which partitions n d-dimensional points into k clusters, such that each point belongs to the cluster with the nearest centroid. The typical algorithm has three steps: (1) for each point, find the cluster with the nearest centroid; (2) recompute cluster centroids as the average of the points having that cluster as the nearest; (3) repeat until convergence. Step (2) can be further divided into two substeps: (2a) compute the number of points assigned to each cluster; (2b) sum the points belonging to each cluster, and then divide by the number of points. While these two steps can be expressed using standard `map`/`scan`/`reduce` building blocks, this is in practice not efficient.[1] Instead, we note that these steps actually correspond to a generalization of histograms.

A *histogram* computes for each element of some input array an integral *bin number*, then counts how many elements belong to each bin. A simple case is illustrated in C-like pseudocode

```
int bins[H] = {0, ..., 0}
for(int i = 0; i < N; i++) {
  j = input[i];
  bins[j]++;
}
```

(a) Normal histogram computation.

```
int bins[H] = {0⊕, ..., 0⊕}
for(int i = 0; i < N; i++) {
  (j, v) = f(input[i])
  bins[j] = bins[j] ⊕ v;
}
```

(b) Generalized histogram computation with operator $\oplus$ and neutral element $0_\oplus$.

Fig. 1. Normal & generalized histograms with k buckets and n input elements.

in Figure 1a. A *generalized histogram* allows an arbitrary operator $\oplus$ with a neutral element $0_\oplus$, instead of just integer addition. Further, the bin number and the value passed to $\oplus$ is computed from the input elements with an arbitrary function f. This is illustrated in Figure 1b. The standard histogram can be obtained by setting $\oplus$ to integer addition and making $f(x)$ return the pair $(x, 1)$. If $\oplus$ is associative, then the histogram can be computed in parallel. For better efficiency, we also require that $\oplus$ is commutative.

There are two main challenges for efficiently computing generalized histograms on GPUs. The first is reducing the amount of bin conflicts, where multiple threads simultaneously try to update the same bin. Such conflicts are resolved by sequentialising the conflicting updates, which may significantly hinder the efficiency of parallel execution. A way to alleviate this problem is to compute multiple *subhistograms*, and then combine them at the end. At one extreme, each thread may be given its own subhistogram, which completely eliminates bin conflicts, but potentially incurs significant memory and operational overhead. At the other extreme, all threads share the same histogram. This has no memory or operational overhead, but suffers from conflicts, as noted above.

The second challenge is keeping the resident memory set small enough to fit some spot in the memory hierarchy, for example in shared memory[2] or in the L2 global memory cache. This is addressed by performing a multi-pass traversal of the input, in which each pass updates a range of bins small enough to fit the desired memory level.

Both solutions implement a classic time-space tradeoff, but they have conflicting effects, as subhistogramming increases the amount of memory used, which may no longer fit in the desired memory level. Prior work studies *specific instances* of these trade-offs, typically in the simpler setting of pure histograms. For example, subhistograms have been maintained

[1]This is because such histograms requires sorting in the general case. We measure the impact in Section V-B.

[2]We use CUDA terminology in which "shared memory" refers to the scratchpad/fast memory.

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

per-thread or per-warp in shared memory [1], [2], [3], [4], with the multi-pass technique used when the subhistograms exceed the amount of shared memory [3]. However, no prior work systematically explores these trade-offs or the development of adaptive schemes to balance this tension for histograms of arbitrary size, sparsity, operator and element type.

This paper develops an adaptive technique for computing generalized histograms of arbitrary operators and sizes on contemporary GPUs, by utilising available hardware-supported atomic operations in either global or shared memory. Further, a generalized histogram abstraction is added to the Futhark programming language, along with type-driven compiler optimization for the abstraction, and is also implemented as a standalone library, which can be invoked from CUDA code.

We first observe that any histogram operator can be implemented by means of one of three atomic primitives: it is either directly supported by GPU hardware (e.g., `atomicAdd`), or can be implemented by means of `atomicCAS`—if the element type is up to a 64-bit word—or less efficiently, by implementing a spinlock with `atomicExch`. We then develop an analytic model that predicts good values for the multi-histogram and multi-pass degrees for a given hardware, operator, histogram size and sparsity, where the latter can be estimated by means of a simple inspector that relies on input sampling. We validate our model by implementing and measuring the performance of a CUDA library for computing generalized histograms.

We use these findings to add a generalized histogram construct to Futhark. The language construct is aimed at generality, and can for example be nested inside sequential and/or parallel code. At compiler level, we support efficient compilation of the construct by code transformations that:

- vertically fuse the parallel loops (`maps`) producing the indices and values arrays with the histogram computation, such that those arrays are not manifested in memory;
- horizontally fuse independent histogram computations whose inputs originate in the same array;
- perform loop distribution and interchange to separate batch histogram computations from the rest of parallel code, thus enabling code generation;
- avoids whenever possible the use of spinlocks in favour of the other two primitives. For example a k-length vectorized addition would by default be performed inside a critical region, but instead, it can be implemented by k independent `atomicAdd` operations.

Finally, we present an empirical evaluation demonstrating that (1) our analytical model produces good estimations of the multi-histogram and multi-pass degrees for both shared and global memory, (2) that the performance of the compiler implementation outperforms an implementation using CUB [5] on three operators that exercise all three atomic primitives, and (3) that the compiler implementation matches or outperforms the hand-written code of several reference implementations (e.g., loop `inl1100` in *435.gromacs*, k-means, *histo*, *tpacf*).

II. Motivation for Generalized Histogram as a Programming Primitive

Histograms are an important statistical tool for data analysis and are used by many computational algorithms from various fields, such as data mining and image processing. In the following we focus on motivating the applicability of generalized histograms, albeit the techniques we present apply equally well to "pure" histograms (which increment by one).

An important research direction in the area of Symbolic Data Analysis is to summarize massive amounts of data by various statistical representations, and to adapt analyses such that they are performed directly on the summarized data. A widely-used representation is that of histogram-valued data, a.k.a., categorical histogram: in simple words, the difference is that, rather than each individual incrementing one bin by one, each individual now updates a constant number of neighboring bins (categories) with contributions that sum up to one [6].

Histogram-valued data has been used in a number of analyses from the fields of machine learning and image processing, for example principle component analysis [7], learning decision trees [8], image similarity [9] and registration [10].

Another example of the use of generalized histograms is in the visualization of large scale datasets, where the number of data points is reduced through data binning and aggregation to generate a new set of data points with a much smaller size [11], [12].

Other notable examples of applications that use generalized-histogram computations are (i) the structural-mechanics benchmark 454.calculix[3] from SPEC2006, (ii) the high-performance implementation of gradient boosting [14] in the XGBoost library, and (iii) the z-buffering algorithm [15] from the field of computer graphics, which uses `argmin` as the combining operator of the histogram. Similarly, Parboil's *histo* benchmark uses saturated-integer addition as operator.

Having the notion of generalized histogram in the language helps the user to analyze parallelism by deconstructing it into building-blocks constructs. For example, Section V-D demonstrates that our histogram-based, but otherwise naive implementation of k-means is competitive with the `kmcuda` library [16], which is used in production and which applies algorithmic improvements. Similarly, we show in Section V-C that gromacs is essentially a histogram computation.

Our solution can also be used to directly extend frameworks that lack specialized support for histogram computations, such such as PyTorch [17]. The easy integration is enabled by Futhark supporting specialized code generators to C+OpenCL/CUDA and Python+OpenCL [18]. We demonstrate such a scenario at the end of Section V-D, in the context of an image-registration technique [10] whose PyTorch implementation uses a sort-scan approach to implement the histogram computation step. Using the proposed histogram construct results in two-order of magnitude speedups. Finally, having generalized histograms as a language construct:

[3]Loop `mafillsm_do7` covers 74% of the sequential runtime and primarily consists of generalized-reductions computations [13].

α	input array element type
β	value type of the generalized histogram
$\oplus$	$: \beta \to \beta \to \beta$, associative and commutative operator
$0_{\oplus}$	$: \beta$ the neutral element of $\oplus$
f	$: \alpha \to (\texttt{int}, \beta)$, function producing an index/value pair for updating the histogram
N	length of the input array
H	number of histogram bins (typically $H \leq N$)
S	count of the sequential multi-pass loop
H_{chk}	$\left\lceil \frac{H}{S} \right\rceil$ number of bins processed in a single pass
L	shared memory size (bytes)
$L2$	L2 cache size (bytes)
$L2_{sz}^{ln}$	the L2 cache line size (bytes)
T_{hw}	the maximal number of concurrent hardware threads
T	$min(T_{hw}, N)$, i.e., concurrent threads utilized
B	maximal CUDA block size
C	number of threads cooperatively working on the same subhistogram
M	histogram's multiplication degree
RF	denotes the average fraction of distinct bins that are accessed by groups of H consecutive elements.
K	thread blocks per histogram (in batched case)
N_{out}	number of batched histograms we are computing

Fig. 2. Summary of notation.

- promotes performance portability: rather than porting each application to each hardware, specialized backends can be written once for each platform and used for diverse applications;
- enables efficient implementation of other language extensions such as automatic differentiation [19] (AD), e.g., in reverse-mode AD, a histogram construct may enable more efficient translation rules than the one obtained by differentiating its low-level implementation.

III. Prototyping a Solution at a High Level

A. Design Space Exploration

This section discusses the pros and cons of several possible generalized histogram implementations, which are shown in Figure 3, where `input` and `histo` denotes the input (of length N) and histogram (of H bins) arrays, respectively. Throughout the paper we will define and make use of many symbolic quantities. For future reference, the important ones are summarized in Figure 2.

Assuming that f takes $O(1)$ time, the sequential algorithm, shown in Figure 1b, has $O(N)$ work and span, because there are N sequential loop iterations.

Listing 1 shows an extreme point of the design space: all T threads apply f in parallel, and then concurrently update the corresponding elements of the same `histo` array. This approach has optimal $O(N)$ work, but may have suboptimal span, due to bin conflicts. For example, if the number of active threads A is much larger than the number of bins H, then at least $A \div H$ atomic updates will occur at the same time on the same element and will be sequentialized.

```
$\beta$ histo[$H$] = {$0_{\oplus}$, ..., $0_{\oplus}$}
forall (int i = 0; i < N; i++) {
  (j, v) = f(input[i]);
  atomic { histo[j] $\oplus$= v; } }
```

Listing 1. Fully atomic: $O(N)$ work, but potential for frequent conflicts

```
$\beta$ histos[T][H];
forall (int t=0; t<T; t++) {
  forseq (int j=0; j<H; j++)
    histos[t][j] = $0_{\oplus}$;
  forseq (int i=t; i<N; i+=T) {
    (j, v) = f(input[i]);
    histos[t][j] $\oplus$=  v;
} }
histo = map (reduce ($\oplus$) $0_{\oplus}$) (transpose histos)
```

Listing 2. Data parallel, efficiently sequentialized: $O(N + H \cdot T)$ work

```
$\beta$ histos[T/C][H];     //assumes C | T
forseq(int off = 0; off < H; off += Hchk) {
  forall(int tt = 0; tt < T; tt += C) {
    forall(int j=off; j<min(off+Hchk,H); j++)
      histos[tt/C][j] = $0_{\oplus}$;
    forall(int t=tt; t<tt+C; t++) {
      forseq(int i=t; i<N; i+=T) {
        (j, v) = f(input[i]);
        if ( off $\leq$ j < min(off+Hchk, H) )
          atomic{histos[tt/C][j] $\oplus$= v;}
} } } }
histo = map (reduce ($\oplus$) $0_{\oplus}$) (transpose histos)
```

Listing 3. Data parallel and atomics: $O(N + \frac{H \cdot T}{C})$ work, reduced conflicts

```
jvs = sort_by_fst (map f input);
(js,vs) = reduce_same_index ($\oplus$) jvs;
histo = scatter {$0_{\oplus}$,...,$0_{\oplus}$} js vs;
```

Listing 4. Sort and scan: $O(N)$ work, $O(\log N)$ span

Fig. 3. Possible Generalized Histogram Implementations.

Listing 2 shows a data-parallel implementation, in which the parallelism in excess is efficiently sequentialized: each of the T threads maintains its own subhistogram by initializing it with $0_{\oplus}$ and then updating it with the elements corresponding to its chunks of the input.[4] The partial subhistograms, stored in `histos`, are finally summed elementwise, by first transposing `histos`, and then by summing up each of the resulting rows, i.e., `map (reduce (`$\oplus$`)` $0_{\oplus}$`) (transpose histos)`. This approach does not use atomics, yet is fully parallel, but may require suboptimal work $O(N + H \cdot T)$—which is inefficient when, for example, H is close to N.

The design point taken in this paper, sketched in Listing 3, is to combine the idea of a sequential multi-pass traversal of the input with an adaptive middle-ground between the approaches sketched in Listings 1 and 2. The multi-pass traversal is aimed at enabling the computation to fit in some level of memory—e.g., shared memory or L2 cache—and is implemented by the outer sequential loop of index `off`, in which each iteration/pass processes only histogram indices in `[off, off+`H_{chk}`]`. This is achieved at the expense of redundantly computing f for each input element in every pass.

[4]The sequential traversal of the input chunk with stride T results in coalesced access to global memory on GPU.

Further, instead of each thread maintaining its own subhistogram, C threads cooperatively build one partial subhistogram by means of atomic updates. Assuming for simplicity that C divides T, it follows that there are $M = \frac{T}{C}$ groups of C threads, with each group operating on a different subhistogram, hence the computation across groups is independent.

The cooperation level C implements a trade-off between work overhead and bin conflict rates. Assuming that the count of the multi-pass loop $S = \lceil H \div H_{chk} \rceil$ is a constant, we observe that choosing $C = min(T, H_{chk} \div k)$ for some constant k preserves the optimal work complexity

$$O(N{+}H{\cdot}M) = O(N{+}H{\cdot}\frac{T}{C}) = O(N{+}max(H, k{\cdot}S{\cdot}T)) = O(N)$$

under the assumption that T and H are smaller than N, which is the common case (otherwise $C = T$ is best). Finally, increasing k decreases C and thus the number of bin conflicts.

An important question that we will answer in Sections III-C and III-D is how to choose values for S and M that not only make sense from an asymptotic standpoint, but also result in good performance for arbitrary histogram size, sparsity, element type and hardware characteristics.

For completeness, Listing 4 shows an implementation based on sort and prefix sum (**`scan`**) operators: (1) f is applied to each input-array element, i.e., **`map`** `f input`, (2) the result is sorted in increasing order of its keys/indices, (3) all consecutive values corresponding to the same key are summed by $\oplus$, and (4) optionally, the sums are published at their indices in `histo`. Assuming radix sort is used, this approach has optimal work $O(N)$ and span $O(\log N)$ complexities, and offers stable performance in practice because it is not affected by how the keys are spatially distributed.

The last two approaches complement each other. We expect the atomic-based approach to be fastest on most datasets, because sorting requires significant data movement. Conversely, adversarial datasets for the atomic-based approach can be easily built, for example when $H \gg N$ and/or when the conflict rate is high (only a few distinct keys).

The rest of the paper will discuss the approach of listing 3, as it cannot be implemented by the user in a deterministic (data-) parallel language, as these typically do *not* provide atomic update primitives. We assume that the user will explicitly select between the sort-and-scan approach—which can be provided by a library—and the one implemented with atomics.

B. Type-Driven Selection of the Atomic Primitive

This section discusses the implementation of the atomic-update primitive implied by the **`atomic`** keyword in the pseudocode of Figure 3. Our aim is to support generalized histograms as a second-order language construct, operating over arbitrary element types β and combining operators $\oplus : \beta \times \beta \rightarrow \beta$. This is possible since modern hardware supports a hierarchy of synchronization primitives that trade off performance for generality of use, and a sufficient subset of these are available through CUDA and OpenCL. Similarly, we aim to use the most efficient primitive supported by the hardware, by statically discriminating based on the type β and by pattern matching whenever possible the $\oplus$ operator:

```
template<class T> struct indval{uint32_t index; T value;};

__global__ void locMemHist(int N, int H, int Hchk, int T,
                          int M, int off, α* input, β* histos) {
  extern __shared__ volatile int loc_hists[];
  int tid= threadIdx.x, lhid = (tid % M)*Hchk;
  int gid = blockIdx.x*blockDim.x + tid;
  // initialize subhistograms and locks
  for(int i=tid; i < M*Hchk; i+=blockDim.x) {
    loc_hists[i] = 0⊕;
  } __syncthreads();
  // compute shared-memory histograms
  for(int i = gid; i < N; i += T) {
    struct indval<β> iv = f<β>(input[i], H);
    if( iv.ind >= off && iv.ind < min(H, off+Hchk) )
      atomicUpdate( lhid+iv.ind-off, iv.val
                  , ⊕, loc_hists, NULL);
  } __syncthreads();
  // reduce block histos and copy to global memory
  for(int i=tid; (i<Hchunk) && (off+i< H); i+=blockDim.x){
    β acc = loc_hists[i];
    for(int j = 1; j < M; j++)
        acc = acc ⊕ loc_hists[i+j*Hchk];
    histos[blockIdx.x*H + off + i] = acc;
} }
```

Fig. 4. Sketch of CUDA kernel for generalized histograms. For the lock-based implementation, one also needs to allocate, initialize and use a shared-memory array of locks (instead of NULL).

HDW: This class of operators correspond to those directly supported by the hardware. For example, integer addition (`atomicAdd`), min/max, and bitwise and/or operations are typically supported, and CUDA also supports float addition.

CAS: If β is some 32 or 64-bit type, then we use the compare and exchange primitive (`atomicCAS`), where `(j,v)` denotes the index-value pair to be updated:

```
β assumed, old = histo[j];
do { β new = old ⊕ v;
     assumed = old;
     old = atomicCAS(&histo[j], old, new);
} while (assumed != old);
```

XCG: Any other type β and operator $\oplus$ will use a busy-waiting strategy implemented by means of the `atomicExch` primitive and an array of locks:

```
while(!done) {
    int done = 0;
    if(atomicExch(&locks[j], 1) == 0) {
        histo[j] = histo[j] ⊕ v;
        mem_fence();
        locks[j] = 0;   done = 1;
}   }
```

C. Computing Histograms in Shared Memory

This section presents the GPU kernel code together with our strategy of choosing the cooperation level C—or the per-block multi-histogram degree M—for the case when the histogram is computed in CUDA shared memory.

Figure 4 shows the CUDA kernel used for experimentation, which computes a chunk of size H_{chk}—starting at position `off`—of a histogram of length H from an N-element input array. `loc_hists` is the shared-memory array of subhistogram chunks to be computed by the current CUDA block, hence its length is $M \cdot H_{chk}$. Otherwise, the code is a direct translation of the pseudocode of Figure 3: the M subhistograms are first initialized, then they are computed by means of atomic updates—where `atomicUpdate` is selected as explained in

Section III-B—and finally the M histograms computed by the current block are combined. Thus, the result is a global memory array that stores one histogram per block; these are further reduced with a second kernel.

The access pattern into `loc_hists` is chosen such that groups of M consecutive threads access M different histograms, as demonstrated by the computation of `lhid = (tid % M) *` H_{chk}. The rationale is that such strided access significantly reduces the number of conflicts appearing across several consecutive groups/warps of threads executing in lockstep.[5] In comparison with the "naive" approach of consecutive threads cooperating on the same sub-histogram, this optimization produces speedups as high as $2-3\times$ for the optimal cooperation C level, and as high as one order of magnitude for larger values of C.

1) Automatically Selecting M and S for Shared Memory: The strategy comes down to choosing the highest multi-histogram degree M—or lowest C—that still fits in shared memory (of size L). If the histogram is too big to fit, then we use a multi-pass strategy by splitting the histogram into chunks of maximal size H_{chk} that fit in L.[6] Section V-A demonstrates that, depending on the operator type, computing in shared memory with values of S between 2 and 6 can be faster than computing in global memory.

We use the maximal block size $B = 1024$, and for simplicity of exposition we assume that this matches the maximal number of concurrent threads per SM, such that one block utilizes the whole amount of shared memory L on an SM.[7]

We denote by el_{size} the size of the histogram element (`sizeof(`β`)`) to which we add 4 bytes when explicit locking is used (the XCG case). The strategy of choosing M and S is formalized by the following equations:

$$blocks = \lceil T/B \rceil \quad m' = min\Big(\frac{L}{el_{size}},\ \Big\lceil \frac{N}{blocks} \Big\rceil\Big)/H \qquad (1)$$

$$M = max(1,\ min(\lfloor m' \rfloor,\ B)) \qquad C = \Big\lceil \frac{B}{M} \Big\rceil \qquad (2)$$

$$len = \frac{L}{el_{size} \cdot M} \quad S = \Big\lceil \frac{H}{len} \Big\rceil \quad H_{chk} = \Big\lceil \frac{H}{S} \Big\rceil \qquad (3)$$

D. Computing Histograms in Global Memory

Computing histograms in global memory follows a similar strategy, except that we aim at fitting in the L2 cache rather than in shared memory. This complicates things, because when the histogram is sparse, some of the histogram's memory blocks will never be brought into cache, making the resident set smaller. It follows that the same L2 cache can accommodate a higher M for sparse histograms in comparison to the case when indices are uniformly distributed.

We start by computing a race factor RF, which estimates the histogram's sparsity. RF denotes the average fraction of active bins—i.e., for a given RF we expect that on average, $H \div RF$ bins are accessed by H consecutive input elements. RF can be cheaply approximated by an inspector that samples groups of H consecutive input elements: for each group, a parallel scatter operation writes `1` at the resulted indices in an H-length array of zeros, followed by summing up the array.[8] RF is obtained by averaging the results of the sampled groups.

We use RF to compute a race-expansion factor of the L2 cache size as:

$$RACE^{exp} = max\Big(1.0,\ \frac{k_{RF} \cdot RF}{min(1.0,\ L2^{ln}_{sz} \div el^{avg}_{size})}\Big) \qquad (4)$$

where $k_{RF} = 0.75$ was chosen experimentally, and el^{avg}_{size} is the average size of the histogram element. The intuition behind the formula is that, if all the data associated with a bin fills exactly one cache line, and only $\frac{H}{RF}$ bins are active, then expanding the multi-histogram degree by a RF factor would still result in the same cache behavior as the uniformly-distributed histogram. However, if the bin data is less than the cache-line size—and assuming the worst case in which no two active bins share a cache line—then $L2^{ln}_{sz} - el^{avg}_{size}$ space is wasted on each cache line. It follows that we need to divide RF with $L2^{ln}_{sz} \div el^{avg}_{size}$ to match the cache behaviour of uniformly-distributed indices.

For the HDW and CAS cases, el^{avg}_{size} is the size of β. For the XCG case, it depends on the memory layout:

1) if the histogram element and the lock are contiguously laid out in memory, then $el^{avg}_{size} = \texttt{sizeof}(\beta) + 4$.
2) if the histogram element has contiguous memory layout—i.e., array of tuples layout—but the locks are stored in a separate array then $el^{avg}_{size} = \frac{\texttt{sizeof}(\beta)+4}{2}$.
3) if $\beta = (\alpha_1, \ldots, \alpha_u)$ such that each $\alpha_{i\in\{1,\ldots,q\}}$ has contiguous layout, but each of them and the locks are stored in different arrays—i.e., tuple of arrays layout—then $el^{avg}_{size} = \frac{\texttt{sizeof}(\alpha_1)+\ldots\texttt{sizeof}(\alpha_q)+4}{q+1}$.

With this refinement, we derive H_{chk} as the maximal histogram chunk that still fits in L2 and allows for a cooperation level of at most H/k^{ct}_{min}, for some small k^{ct}_{min} (chosen as 2):

$$C_{max} = min\Big(T, \frac{H}{k^{ct}_{min}}\Big) \quad M_{min} = max\Big(1, \Big\lfloor \frac{T}{C_{max}} \Big\rfloor\Big)$$
$$S = \Big\lceil \frac{M_{min} \cdot H \cdot el^{avg}_{size}}{f_{L2} \cdot L2 \cdot RACE^{exp}} \Big\rceil \quad H_{chk} = \Big\lceil \frac{H}{S} \Big\rceil \qquad (5)$$

In the formula for S—the number of sequential passes—the nominator is the memory space required by the multi-histogram degree M_{min} corresponding to C_{max}, and the

[5]Within a thread block, only several warps truly execute in parallel; the rest execute concurrently to hide latency. Thus, spreading the accesses of consecutive threads across histograms minimizes the number of conflicts.

[6]The intuition here is that the overhead of redundant computation required by the multi-pass technique dominates the overhead introduced by having even high bin-conflict rates in shared memory; it follows that it does not pay off to reduce the conflict rates at the expense of increasing the multi-pass count S (which will allow a higher M).

[7]In practice, the maximal number of concurrent threads per SM may be a multiple k of the maximal block size B. A more rigorous treatment is to preferentially use $L \div k$ shared memory per block, and only in the case when $M = 1$ still does not fit, then spawn only one block per SM, which utilizes the whole amount of shared memory.

[8]Furthermore, RF can be multiplied by an average factor estimating how many bins share a cache line; this can also be computed relatively cheaply by a segmented-scan operation.

denomiator represents the expanded L2 size. F_{L2} denotes the fraction of the L2 cache used for histograms (chosen as 0.4).

After computing H_{chk}, we may still be in the case where the multi-histogram degree that fits in the L2 cache is larger than the one imposed by k^{ct}_{min}. To this purpose, we recompute the minimal and maximal C and M that still fits in L2.

In the formula below, we chose $u = 2.0$ for atomic add, because its hardware implementation is more robust and it accommodates a smaller M; the computation of k_{max} can be intuitively interpreted as the minimal number of (histogram) elements per active thread that fit in the L2 cache.

$$C = min\Big(T, \frac{u \cdot H_{chk}}{k_{max}}\Big) \qquad M = max\Big(1, \Big\lfloor \frac{T}{C} \Big\rfloor\Big);$$
$$\text{where} \quad u = \begin{cases} 1.0 & \text{for CAS, XCG} \\ 2.0 & \text{for HDW} \end{cases} \tag{6}$$
$$k_{max} = \frac{min\Big(\frac{F_{L2} \cdot L2 \cdot RACE^{exp}}{el_{size}},\ N\Big)}{T}$$

IV. Implementation in the Futhark Compiler

Futhark is a purely functional data-parallel array language that supports regular nested parallelism, with a compiler that generates CUDA or OpenCL code for GPUs. Except where noted, we use the CUDA backend for our measurements in this paper, but the exact same implementation technique is also used to generate OpenCL. Syntactically, Futhark resembles a mixture of OCaml and Haskell, and is based on the programmer making parallelism explicit via the use of parallel constructs, rather than through complex compiler analysis. In particular, parallelism is expressed via *second-order array combinators* (SOACs), such as **map** and **reduce**, appearing to the programmer as ordinary higher-order functions, but are treated specially by the compiler. Futhark is intended to be significantly more productive than low-level CUDA programming, but with comparable performance.

Section IV-A presents the compiler IR for generalized histograms, Section IV-B presents the rewrite-rules for horizontal and vertical fusion, Section IV-C briefly discusses how (batched) histogram computations are extracted from arbitrary code, and Section IV-D addresses vectorized operators.

A. Compiler Intermediate Representation

In the following, we write $\overline{z}^{(n)}$ to range over sequences of n objects. Depending on context, the elements of the sequence may have separators, or be merely juxtaposed. For example, we may use the same notation to shorten a function application $f\ \overline{v}^{(n)} \equiv f\ v_1\ \cdots\ v_n$ or a tuple $(\overline{v}^{(n)}) \equiv (v_1, \ldots, v_n)$.

Context determines which separator, if any, is intended. For more complicated sequences, where not all terms under the bar are variant, the variant term is subscripted with i. For example, $(\overline{[d]v_i}^{(n)}) = ([d]v_1, .., [d]v_n)$ and $(\overline{[d_i]v_i}^{(n)}) = ([d_1]v_1, .., [d_n]v_n)$

The type of integer arrays with n elements is written as $[n]$`int`. The most important parallel construct is **map**, with the usual semantics, which we extend to operate on an arbitrary number of input arrays that have the same size (here m):

```
for (int j = 0; j < $d$; j++) {
  (($i_1$, $v_1$),...,($i_k$, $v_k$)) = $f$($vs_1$[j], ..., $vs_n$[j]);

  if $i_1$ >= 0 && $i_1$ < length($dest_1$)
  { $dest_1$[$i_1$] = $dest_1$[$i_1$] $\oplus_1$ $v_1$ }
  ⋮
  if $i_k$ >= 0 && $i_k$ < length($dest_k$)
  { $dest_k$[$i_k$] = $dest_k$[$i_k$] $\oplus_k$ $v_k$ }
}
```

Fig. 5. Pseudocode, assuming that all arrays vs have length d, corresponding to **genhist** $(\overline{vs}^{(n)})\ f\ (\overline{(dest_i, \oplus_i, 0_{\oplus_i})}^{(k)})$.

F1 (vertical fusion: map into genhist):

$$\textbf{genhist}\ (\overline{e}^{(n)}, \textbf{map}\ g\ xs, \overline{e}^{(m)})\ f\ ops$$
$$\Downarrow$$
$$\textbf{genhist}\ (\overline{e}^{(n)}, xs, \overline{e}^{(m)})$$
$$(\lambda \overline{y}^{(n)}\ x\ \overline{z}^{(m)} \rightarrow\ f\ \overline{y}^{(n)}\ (g\ x)\ \overline{z}^{(m)})\ ops$$

(Applies only if none of the arrays mentioned in *ops* alias **xs** or occur as free variables in **g**.)

F2 (horizontal fusion of genhist):

$$(\textbf{genhist}\ \overline{e}^{(n)}\ f\ \overline{ops}^{(a)},\ \textbf{genhist}\ \overline{e}^{(m)}\ g\ \overline{ops}^{(b)})$$
$$\Downarrow$$
$$\textbf{genhist}\ (\overline{e}^{(n)}, \overline{e}^{(m)})$$
$$(\lambda \overline{x}^{(n)} \overline{y}^{(m)} \rightarrow (f\ \overline{x}^{(n)}, g\ \overline{y}^{(m)}))\ (\overline{ops}^{(a)}, \overline{ops}^{(b)})$$

(Applies only if the intersection of $\overline{e}^{(n)}$ and $\overline{e}^{(m)}$ is nonempty.)

Fig. 6. Fusion rules for **genhist**.

$$\textbf{map}\ f\ \overline{x}^{(n)} = [f\ \overline{x_i[0]}^{(n)}, \ldots, f\ \overline{x_i[m-1]}^{(n)}]$$

Generalized histograms are made available as the function **genhist**. To permit fusibility, the type is fairly complicated, and can be skipped without significantly impacting understanding of the paper:

$$\begin{array}{rll} \textbf{genhist} & : & \forall d \overline{x}^{(k)} \overline{\alpha}^{(k)} \overline{\beta}^{(n)}. \\ & & (\overline{[d]\beta_i}^{(n)}) \rightarrow (\overline{\beta}^{(n)} \rightarrow (\overline{\texttt{int}, \alpha_i}^{(k)})) \\ & \rightarrow & (\overline{([x_i]\alpha_i, \alpha_i \rightarrow \alpha_i \rightarrow \alpha_i, \alpha_i)}^{(k)}) \\ & \rightarrow & (\overline{[x_i]\alpha_i}^{(k)}) \end{array}$$

genhist computes k simultaneous histograms (each with its own operator and neutral element), based on n input arrays of the same length, from which elements are passed to a function that constructs k pairs of bin numbers and values. The semantics are illustrated in imperative pseudocode in Figure 5.

B. Horizontal and Vertical Fusion

The Futhark compiler performs two kinds of fusion: *vertical* (producer-consumer) fusion avoids intermediate results in memory, and *horizontal* fusion merges two otherwise independent operations over same input arrays. The **genhist** fusion rules are shown on Figure 6. The rules are applied greedily via dataflow graph reduction [20], which is not necessarily optimal, but performs well in practice.

The vertical fusion rule F1 handles the case where an input array to **genhist** comes from a **map**, in which case the **map** can be inlined directly in **genhist**.

The horizontal fusion rule F2 combines two adjacent **genhist**s that have the same arrays as input into a single **genhist** that computes multiple histograms. While this reduces the number of accesses to global memory, the resulting histograms may no longer fit in shared memory. There is therefore a risk that aggressive application of rule F2 can be harmful to application performance. We have not yet encountered this issue in real programs.

C. Extracting Histograms from Nested Parallelism

Futhark permits regular nested data parallelism, which means that **genhist** constructs can occur within an arbitrary amount of nested **map**s and other looping constructs. The compiler applies a flattening technique, involving loop distribution and interchange, to isolate parallel constructs such as **genhist**, optionally with perfect **map** nests on top [21]. These expressions that are passed on to the code generator in the form of **genhist** constructs enclosed in **map**s, representing batched or *segmented* histograms. For example,

$$\textbf{map}\ (\lambda \texttt{xs}\ \texttt{ys} \rightarrow \textbf{genhist}\ \texttt{xs}\ f\ \ (\texttt{ys}, +, 0))\ \texttt{xss}\ \texttt{yss}$$

computes a histogram for each array xs in two-dimensional array xss. The computation of N_{out} H-bin histograms can be done by treating it as a single $N_{out} \times H$-bin histogram on a flattened input array (xss in the above example). However, this is not necessarily optimal, as it discards locality information. In our implementation, when H fits in shared memory, we instead assign K thread blocks to each histogram, with the usual subhistogramming for small H. We set

$$K = \left\lceil \frac{T \div B}{N_{out}} \right\rceil$$

which guarantees that the amount of exploited parallelism is at least T. Our shared memory implementation does not support a single thread block computing multiple batched histograms simultaneously. While this example shows only a single **map** on top of **genhist**, our implementation permits any number.

D. Optimizing Vectorised Operators

It is sometime useful to compute histograms where the operator is of the form **map** $\oplus$ for some $\oplus$, corresponding to a generalization of vector addition. The resulting histogram then contains an array in each bin. By section III-B, such an operator would necessarily require the XCG strategy, since no available atomic can update an entire vector at a time. However, our compiler detects the case where the operator has this vectorised form, and applies the atomic update on each *element* of the vector, rather than the vector as a whole. This causes simultaneous updates of the same bin to be interleaved, but this is safe because we are still properly synchronising the updates to the elements of the vector in each bin.

M,S shared	**H**=31	127	505	2K	6K	12K	24K	48K
HDW/CAS	396,1	96,1	24,1	6,1	2,1	1,1	1,2	1,4
XCG	132,1	32,1	8,1	2,1	1,2	1,3	1,6	1,12

M,S global	**H**=12K	24K	48K	192K	384K	768K	1.5M
HDW RF=1	23,1	11,1	5,1	1,1	1,1	1,2	1,3
HDW RF=63	69,1	34,1	17,1	4,1	2,1	1,1	1,1
CAS RF=1	46,1	23,1	11,1	2,1	1,1	1,2	1,3
CAS RF=63	138,1	69,1	34,1	8,1	4,1	2,1	1,1
XCG RF=1	15,1	7,1	3,1	1,1	1,2	1,3	1,5
XCG RF=63	69,1	34,1	17,1	4,1	2,1	1,1	1,1

TABLE I
THE VALUES OF (M, S) COMPUTED BY OUR ANALYTICAL MODEL FOR SHARED (TOP) AND GLOBAL MEMORY (BOTTOM), RESPECTIVELY.

V. EXPERIMENTAL EVALUATION

This section is organized as follows: Section V-A validates our design point for shared and global memory by comparing model-driven selection of the multi-histogram degree with a range of fixed choices on a representative set of histogram sizes and operators. Section V-B compares our compiler implementation with CUB for the same set of histogram sizes and operators, section V-C shows the applicability of generalized histograms to the 435.gromacs benchmark from SPEC CPU2006, and section V-D compares our compiler implementation with several applications from standard benchmark suites, and details on the compiler optimizations that provide performance gains. All experiments are run on an NVIDIA RTX2080 Ti (Turing architecture) with CUDA 10.1 and L = 48KiB, $L2$ = 5.5MiB, T_{hw} = 69632. Reported runtimes do not include the time for transferring the program input and result arrays between device and host. All runtimes are the average of at least a hundred consecutive runs. Our experiments have also been run on other GPUs (e.g., RTX2070, GTX1050 Ti) with similar results. All code is publicly available[9].

A. Model Validation

Figure 7 shows the runtime for computing histograms for various H and under various multi-histogram degrees in shared and global memory; see caption for details. We used an input of 50 million elements, where the input array holds uniformly distributed 32-bit integers (α), and the index result of f for an input elm is obtained by the formula `iv.ind = (elm % max(1, H/RF))*RF`. Thus, $RF = 1$ corresponds to a uniform distribution of indices, while increasing RF directly increases the bin-conflict rate since only $H \div RF$ bins are actively used (with a stride equal to RF).

The operator is chosen as follows to demonstrate the three atomic-update implementations of section III-B:

HDW:	32-bit integer addition (β = int);
CAS:	24-bit saturated addition (β = int);
XCG:	argmax, receiving (index,value) pairs and picking pair with the largest value, using the index as tie-breaker (β = (int, int)).

[9]https://github.com/diku-dk/futhark-sc20

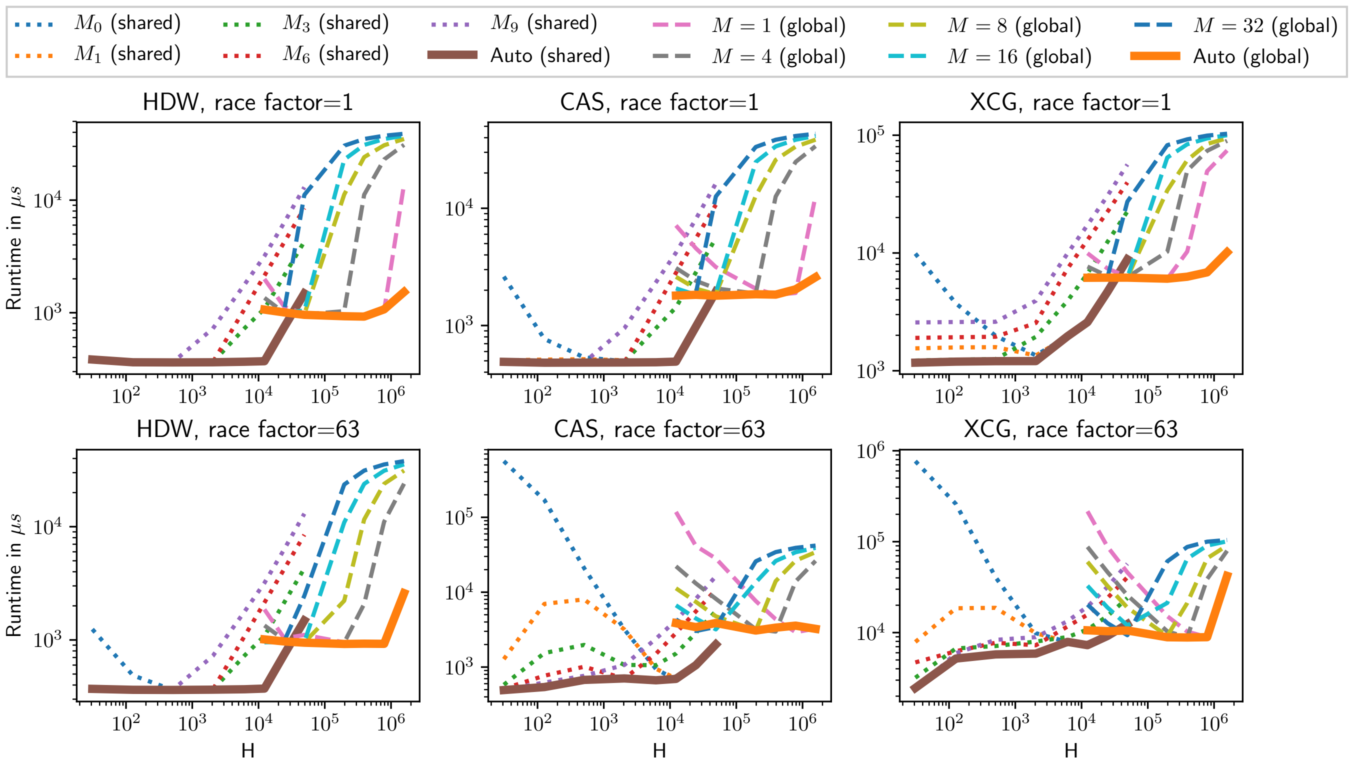

Fig. 7. Runtimes for shared- and global memory histograms computed with our CUDA implementation, on NVIDIA RTX2080 Ti. The experiment uses block sizes 1024 and 256 for shared and global memory, $\mathtt{T}_{hw} = 69632$ and an input of 50 million integers. The number of bins of the tested histograms are: $H = 31, 127, 505, 2K, 6K, 12K, 24K, 48K, 192K, 384K, 768K, 1.5M$. Shared memory uses $M_0 = 1$ and $M_{k \in \{1,3,6,9\}} = k \cdot \left\lfloor \frac{B}{min(H,B)} \right\rfloor$. Whenever $M_k \cdot H$ does not fit in shared memory, the multi-pass strategy is used. For global memory, $M \in \{1, 4, 8, 16, 32\}$ do not use multi-passing. "Auto" denotes automatic selection of M and S based on our analytical model.

We compare automatic selection of (M, S) in Table I and denoted by "Auto"—with a set of statically-chosen values: For shared memory we use $M_{k \in \{1,3,6,9\}} = \left\lfloor \frac{k \cdot B}{min(H,B)} \right\rfloor$, and $M_0 = 1$, and we use the multi-pass technique accordingly whenever $M_k \cdot H$ is too large to fit in shared memory. For global memory we compared with multi-histogram degrees $\{1, 4, 8, 16, 32\}$, which do not use the multi-pass technique.

The top graphs correspond to the input being uniformly distributed to bins ($RF = 1$), while the bottom graphs correspond to sparse histograms, in which H consecutive elements hit only $H \div 63$ distinct bins ($RF = 63$).

- Our strategy produces near-optimal (M, S) values. The maximal slowdown with respect to the best static choice of (M, S) is 5%.
- In the case of HDW, shared memory, the static choice of M_1 is near optimal as well.
- For the remaining cases, no fixed/static choice of (M, S) is close to producing near-optimal results, for all histogram sizes and RF values. For example:
  - In the HDW case in global memory with $RF = 1$, $M = 1$ is about $9\times$ slower than Auto for $H = 1.5$M because it does not use the multi-pass strategy. The other choices of M are significantly worse.
  - The differences become much more pronounced for the cases of CAS and XCG. For example, in the case of shared-memory CAS with $RF = 1$, M_1 is near optimal, but for $RF = 63$, the same M_1 results in $2.6 - 12.8\times$ slowdowns for all histograms up to 2K.
- Table I demonstrates that our analytical model takes into account histogram sparsity: $RF = 63$ has higher multi-histogram degree M or smaller multi-pass degree S than $RF = 1$, and still produces near-optimal performance in comparison with the static choices of M, S.
- The results in Figure 7 support a simple compiler heuristic for choosing the switching point between shared and global memory: if the multi-pass degree S computed for the shared-memory is less or equal to 3, 4 and 6 for HDW, CAS, XCG, respectively, then the histograms are computed in shared memory, otherwise in global memory. This is not perfect: for example for CAS with $RF = 1$, $H = 48$K requires 4 passes and is mapped to shared memory, in spite of being $1.08\times$ slower than the global memory version (but for $RF = 63$ it pays off since execution in shared memory is about $2\times$ faster in this case).

B. Comparison with CUB

We compare our library with CUB 1.8.0 [5], on the same operators, input and histogram lengths as in figure 7. We have used CUB's `HistogramEven` for the HDW case, and

Speedup vs CUB	**H**=31	127	505	2K	6K	12K	24K	48K	192K	384K	768K	1.5M
CUB-HDW (miliseconds)	1.40	1.34	1.44	1.55	1.44	9.39	22.13	29.66	36.87	39.20	42.00	46.59
FUT-HDW RF=1 (speedup)	3.4×	3.3×	3.5×	3.6×	3.1×	19.6×	24.7×	22.5×	30.5×	32.6×	32.7×	27.5×
FUT-HDW RF=63(speedup)	3.4×	3.2×	3.5×	3.7×	3.1×	20.3×	25.2×	22.5×	29.9×	31.8×	33.9×	14.4×
CUB-CAS (miliseconds)	2.40	3.52	3.84	4.96	5.17	5.37	5.57	6.45	6.80	7.00	7.22	8.09
FUT-CAS RF=1 (speedup)	3.9×	5.7×	6.2×	7.9×	8.1×	8.1×	4.4×	2.7×	2.7×	2.8×	2.8×	2.6×
FUT-CAS RF=63 (speedup)	3.9×	4.9×	4.1×	5.0×	5.4×	5.4×	4.0×	2.6×	1.9×	2.0×	2.0×	2.2×
CUB-XCG (miliseconds)	6.75	9.29	11.11	13.57	14.83	15.94	17.06	18.11	20.11	21.28	22.44	23.28
FUT-XCG RF=1 (speedup)	4.0×	5.5×	6.5×	7.8×	5.9×	4.9×	2.8×	2.3×	2.7×	3.1×	3.0×	2.0×
FUT-XCG RF=63 (speedup)	2.6×	1.6×	1.6×	1.9×	1.7×	2.2×	2.0×	1.8×	2.0×	2.4×	2.7×	2.7×

TABLE II
SPEEDUP OF FUTHARK-GENERATED (FUT) CODE OVER CUB; CUB RUNTIME IS SHOWN IN MILISECONDS.

`SortKeys` followed by `ReduceByKey` for the CAS and XCG case. Our measurements favor CUB in several ways: (1) we do not account for the runtime of CUB's inspectors and neither for the creation of the key-value pairs, and (2) we run the CUB implementation on uniformly-distributed indices.[10]

Table II presents the speedup of Futhark generated code for $RF = 1$ and $RF = 63$ in comparison with CUB: For HDW, Futhark is only $3.1 - 3.7\times$ faster for histogram lengths of up to $12K$, after which the speedup increases dramatically.

For CAS and XCG, Futhark gets the upper hand even when $RF = 63$. The shared memory histograms shows the higher speedups—for example greater than $7\times$ on $H = 2K$—while the global memory speedups are between $1.8 - 3.1\times$.

This demonstrates the overhead of using the sorting-based approach to computing generalized histograms that we alluded to in the introduction.

C. *Case Study: 435.gromacs from SPEC CPU2006*

Molecular dynamics simulations do not immediately come to mind for histograms, yet an essential part of 435.gromacs can be expressed as generalized histograms. Specifically, we implement a simplified form of loop `inl1100` in Futhark. This loop is an important code pattern and a representative computational kernel, because similarly-structured loops—such as `inl1130`, `inl1000`—account for about 90% of the sequential execution runtime of 435.gromacs [13]. In one run, the loop updates the forces of a number `nri` of 3D particles and their associated neighbors, which are looked up by means of indirect arrays (`iinr` and `jjnr`). Its structure is as follows:

```
for(int n=0; n<nri; n++) {
  int i3=3*iinr[n]; float fix1=0,fiy1=0,fiz1=0;
  for(int k=jindex[n]; k<jindex[n+1]; k++) {
    int j3 = 3*jjnr[k];
    float tx11 = ..., ty11 = ..., tz11 = ...;
    fix1 += tx11; fiy1 += ty11; fiz1 += tz11;
    faction[j3] -= tx11; faction[j3+1] -= ty11;
    faction[j3+2] -= tz11;
  }
  faction[i3] += fix1; faction[i3+1] += fiy1;
  faction[i3+2] += fiz1;
}
```

Our experiment uses the reference dataset, but it expands the input to loop `inl1100` by merging together the largest 22 inputs on which the loop is called during CPU execution. The resulting dataset has `nri=30657`, the total number of neighbors is `nrj=jindex[nri]=1380160`, and the total number of particles is 23178, which leads to histogram of size 69534, because the forces are computed on each of the three dimensions.[11] Sampling the dataset revealed that the histogram is sparse, with a race factor $RF = 79$.

We use for comparison a straightforward CUDA implementation that flattens the parallelism of the outer and inner loops, and in which each thread performs six atomic updates (to `faction`) to simulate the interaction between a particle and a neighbor. The runtimes and relative speedups are shown in Table III. We report performance in two settings:

HDW: Using the hardware support for atomic-float addition (HDW) results in comparable performance, with Futhark having a small disadvantage ($0.91\times$).

CAS: We also test the case when the atomic updates are implemented by means of CAS, for example because AMD GPUs are typically programmed via OpenCL, which does not support a primitive for atomic-float addition; for consistency we use the OpenCL backend of Futhark. To reduce the number of (dynamic) races we restructured the Futhark program to perform one update per thread, e.g., `faction[j3]`, `faction[j3+1]`, and `faction[j3+2]` are updated by consecutive threads. The restructuring was relatively easy in Futhark, requiring about 20 new lines of code, but it would be tedious in a low-level CUDA/OpenCL implementation. The Futhark implementation with $RF = 79$ results in $2.65\times$ speedup in comparison to the CUDA version. Furthermore, accounting for the race factor has a significant impact: the version using $RF = 79$ results in multi-histogram degree $M = 22$ and is responsible for $1.6\times$ application-level speedup in comparison with the version using $RF = 1$, which results in $M = 6$.

[10]Histogram sparsity does not significantly changes the performance of the key-sorting version of CUB, because such a data-parallel implementation performs the same work, irrespective of sparsity.

[11]The reason for expanding the dataset is that the input of one loop run is too small to utilize well the GPU parallelism. Furthermore, the small input seems to be an artifact of a chunking technique aimed at optimizing the locality of CPU execution: in file `fnbf.c`, `inl1100` is actually called on different chunks from a parallel loop, which justifies expanding the dataset.

D. Other Application Benchmarks

This section compares Futhark to prior implementations of common problems or published benchmarks that make use of generalized histograms. Results are summarised in Table III.

1) k-means clustering: This application partitions a set of n points in d-dimensional space into k clusters, such that the distance from each point to the cluster centre is minimised. This involves two histograms: one that counts the number of points belonging to each cluster, and one that counts the sum of the individual points for each cluster. Assuming arrays $\texttt{membership} : [n]\texttt{int}, \texttt{points} : [n][d]\texttt{f32}$, this can be expressed as follows:

$$\begin{array}{l}(\textbf{genhist}\ \texttt{membership}\ (\lambda c \to (c,1))\ (\texttt{replicate}\ k\ 0, (+), 0),\\ \ \textbf{genhist}\ (\texttt{membership}, \texttt{points})\ (\lambda(c,p) \to (c,p))\\ \quad (\texttt{replicate}\ k\ (\texttt{replicate}\ d\ 0),\\ \quad\ \textbf{map}\ (+), \texttt{replicate}\ d\ 0))\end{array}$$

Note that the second histogram uses vector addition (**map** $(+)$) as its operator. The two histograms share an input array (membership), and so are fused horizontally, but because the points array is a factor d larger, the impact is neglible. We compare the Futhark implementation against kmcuda: a CUDA implementation of Yinyang k-means [16] (https://github.com/src-d/kmcuda). We benchmark with $n = 100000, d = 256, k = 1024$, where the histograms do not fit in shared memory, and slightly outperform kmcuda. We note that the kmcuda implementation is optimised for large k, and does not scale well for small k, with Futhark being about 30× faster for $k = 5$.[12]

2) Parboil: From Parboil [22] we take two benchmarks. *histo* computes a histogram with 996 × 1024 bins, where the operator is saturated integer addition. We use Parboils opencl_nvidia implementation, as this one performed the best on our hardware. *tpacf* computes two sets of 100 batched histograms of 22 bins each. Parboil implements this using 200 thread blocks, each of which compute a 22-bin histogram in shared memory. The Futhark compiler instead fuses the two histogram computations horizontally and launches $K = 300$ thread blocks (see Section IV-C), each of which compute two partial 22-bin histograms, which are then combined by a segmented reduction. This is advantageous because in Parboil's case, the two thread blocks computing a histogram pair will have significant overlap in which memory they access. In Futhark, the same memory accesses will contribute to *two* histograms, thus saving bandwidth.

3) CUDA samples: The CUDA SDK contains two histogram implementations specialised for different bin counts (64 and 256). For 64 bins, each thread is given its own private subhistogram. For 256 bins, each subhistogram is shared between the threads in a warp. In both cases, the subhistograms are stored in shared memory, and a final reduction pass computes the final histogram. This is similar to our approach. The CUDA implementations outperform Futhark

[12]We use Futhark's OpenCL backend for this benchmark, as NVIDIAs CUDA kernel compiler seems to unroll an inner loop too aggressively, reducing the speedup for $k = 5$ to 10×.

435.gromacs loop inl1100 ($N = 4140480$, $H = 69534$)

Workload	CUDA	Futhark	Speedup
HDW	333μs	362μs	0.91 ×
CAS	2432μs	919μs	2.65 ×

k-means clustering

Workload	kmcuda	Futhark	Speedup
$k = 1024$	4.2s	3.7s	1.14 ×
$k = 5$	2.98s	0.097s	30.72 ×

Parboil benchmarks

Benchmark	Parboil	Futhark	Speedup
histo	1.74ms	0.37ms	4.70 ×
tpacf	916ms	810ms	1.13 ×

CUDA samples

Workload	CUDA	Futhark	Speedup
$H = 64$	250μs	261μs	0.95 ×
$H = 256$	230μs	277μs	0.83 ×

Image Registration Implementation [10]

Workload	PyTorch (cuda)	Futhark (pyopencl)	Speedup
$N = 2M, H = 2809$	258ms	1.95ms	132×
$N = 8M, H = 41209$	3331ms	7.10ms	469×

TABLE III
SPEEDUP ON APPLICATION BENCHMARKS.

because the histogram input is conceptually a *byte* sequence. The CUDA implementation has each thread read a full 32-bit word, which is then used to perform four index/value updates. With **genhist**, each input element produces only a single index/value pair, and so Futhark's memory accesses are less efficient, as threads read only a single byte at a time.

4) Image Registration [10]: Finally, we compare with a PyTorch implementation of the image registration technique from Darkner and Sporring [10], which builds on the technique of Locally Orderless Images [23] and requires computating local intensity histograms. Since PyTorch [17] does not support a generalized histogram construct, the implementation uses a sort-scan approach, in which the values corresponding to the same (unique) key are grouped together—by using PyTorch's unique and split constructs—and then they are summed together to produce the histogram result.

We present the histogram computation in a C-style, loop-based pseudocode that is more accessible than the original implementation and is straightforwardly translated to Futhark:

```
void interp(float x, float* vals) {
  float t = x - floor(x);
  vals[0] = (-t3+3.0*t2-3.0*t+1.0)/6.0;
  vals[1] = (3.0*t3-6.0*t2+4.0)/6.0;
  vals[2] = (-3.0*t3+3.0*t2+3.0*t+1.0)/6.0;
  vals[3] = t3/6.0;
}
// min_x, min_y and max_x, max_y are the minimal and
//    maximal elements of dimensions x and y of array inp.
int h1 = max_x - min_x + 4, h2 = max_y - min_y + 4;
float* histo = (float*)calloc(h1*h2, sizeof(float));

for(int k=0; k<N; k++) {
  float x = inp[k].x, y = inp[k].y;
  float vals1[4], vals2[4];
  interp(x, vals1); interp(y, vals2);
  int ind1 = x - min_x, ind2 = y - min_y;

  for(int i=0; i<4; i++) {
    for(int j=0; j<4; j++) {
      histo[(ind2+i)*h1 + (ind1+j)] += vals1[i] * vals2[j];
} } }
```

The input is an N-element array of two-dimensional points. For each point `inp[k]`, and for each dimension `x` and `y`, the function `interp` computes four contributions which always sum up to one and which are stored in arrays `vals1` and `vals2`. The outer product of the contributions (`vals1[i]*vals2[j]`, $\forall 0 \leq i, j < 4$) is used to update a 4×4 neighborhood in the histogram.

Table III shows the running times for the histogram computation step (i) of the original PyTorch implementation running with CUDA and (ii) of the Futhark implementation, which is easily integrated with the Python program by means of Futhark's `pyopencl` (Python+OpenCL) code generator [18]. We report $132\times$ and $469\times$ speedups in favor of Futhark on two random datasets, in which the input has length $N = 2 \cdot 10^6$ and $8 \cdot 10^6$ and with histograms of sizes $h1 \cdot h2 = 53 \cdot 53$, which is computed in shared memory, and $h1 \cdot h2 = 203 \cdot 203$, which is computed in global memory, respectively. The high speedups demonstrate the usefulness of supporting histograms as a language construct, rather than relying on the users to assemble their own implementations out of lower-level constructs.

VI. Related Work

Pure Histograms. A body of work has investigated CUDA accelerated (pure) histogram implementations. Podlozhnyuk proposes two methods [4] for histograms of size 64 and 256 bins, which maintain subhistograms in shared memory, at thread and warp level, respectively. Since atomics were not yet supported in hardware, the access to warp-level subhistograms was implemented with a software-tagging scheme, which exploits that the threads in a warp execute in lockstep.

Independently, Shams and Kennedy propose two methods for computing histograms on GPUs [3]. The first maintains subhistograms at warp level in shared memory, and is similar to the one of Podlozhnyuk, except that the multi-pass strategy is applied in order to support histograms of arbitrary bin count. The second method is aimed at sparse histograms, and it relies on a collision-free structure, implemented as a $H \times T$ matrix in global memory, in which each of the T threads owns a subhistogram. Furthermore, the high latency of global memory is amortized by packing a number of bins in a double word maintained in shared memory, and by updating the global memory subhistogram only when that double word is about to overflow. (This technique assumes increment as operator.)

Nugteren et al. examine a fixed scenario, where the input is a 2048×2048 8-bit pixel image, from which it computes a histogram with 256 bins [24]. The work examines several techniques for reducing the race conflicts, for example, (1) the input is shuffled by block transposition, because images tend to have correlated pixels, or (2) each thread processes a chunk of consecutive pixels, even if this introduces uncoalesced accesses. These techniques assume images as input.

Brown and Snoeyink present a collision-free technique for computing histograms with 256 bins [1]. The idea is to maintain per-thread partial subhistograms in register memory, by using 8-bit bin counters (instead of 32-bit integers). Before overflow can happen (every 63 elements) the partial results are accumulated in a shared-memory histogram, maintained at CUDA block level, by performing a linear serial scan of the bit counts, collectively, with all the threads in the block. This method also assumes pure histogram computations.

Gómez-Luna et al. propose an implementation [25] applicable to histograms up to 4096 bins where a bin size is 32 bits. Similar to us, they compute histograms in shared memory by choosing the maximal subhistogram degree that fits shared memory, but do not consider global memory or sparsity.

In comparison with this body of work, our implementation and related optimizations are applicable to arbitrary operators rather than just integer increment. Moreover, our analytical model takes into account the sparsity of the histogram and determines near-optimal values for the combination of multi-histogram and multi-pass degrees, while allowing the computation to be carried out in either shared and global memory.

Sort-Scan Implementations. Another body of work is aimed at designing the implementation such as its performance is oblivious to the race factor. An example was already presented in Listing 4, where the input is sorted according to the bin number it falls into, followed by key reduction. Such solutions are complementary to our approach, and the comparison with the CUB library hints that the sweet point in performance is a relatively-high bin-conflict factor.

In the same spirit, Dhanasekaran and Rubin present a generalized histogram algorithm on GPUs [26], in which subhistograms are maintained at CUDA block level. The idea is that the input to a CUDA block is chunked and each chunk is partially sorted in shared memory, by means of an indirect (permutation) array, which is computed by using `atomicAdd` operations. Then an segmented reduction is performed through the indirect array, such as work items corresponding to the same bin are reduced.

Language and Compiler Implementations. Ravi et al. propose compiler and runtime support for computing generalized reductions on heterogeneous hardware [2]. Much of the work is aimed (i) at the compiler analysis required for identifying such computations (supported on known commutative operators such as addition), (ii) at generating the host and device code and (iii) at the work distribution schemes required for cooperatively performing the computation on both CPU and GPU. The implementation has each thread compute its own subhistogram, which are merged at the end, and are possibly maintained in shared memory if they fit.

Similarly, Reddy et al. extend the Pencil framework [27] with support for generalized reductions [28] for known commutative operators. They report the extensions necessary for integrating reductions in polyhedral analysis and code generation for GPU hardware, which also supports fusion. The implementation seem to maintain one subhistogram per CUDA block whenever the histogram fits in shared memory, or otherwise, all threads work directly on one histogram, which is stored in global memory.

Finally, Accelerate [29] offers the `permute` function which has similar semantics with generalized reductions. Similar to our work, the operator is statically translated to the most efficient hardware-supported atomic primitive for the given element type, and defaults to a spinlock in general. In Accelerate, vectorized histogram operators are implemented with a spinlock and the code-generation and optimization support is simpler because Accelerate does not allow nested parallelism.

We observe that the imperative language and compiler solutions do not allow for arbitrary histogram operators, and no solution utilizes the multi-pass strategy or takes into consideration histogram sparsity.

VII. Conclusions

We have defined the notion of a generalized histogram construct and presented a GPU implementation strategy. As an advancement over prior work, our analytical model predicts the best combination of the multi-histogram and multi-pass degrees for a given histogram size, operator element type, and sparsity. We have experimentally validated our analytical model by comparing performance of its selected parameter against fixed parameter choices.

We have fully integrated the generalized histogram construct—nestable inside arbitrary parallel code—in Futhark, a high-level parallel programming language and compiler, isolating the programmer from having to know low-level details of GPU programming. Vectorized operators, which would naively require locking, are instead implemented by a sequence of efficient atomic operators, such as `atomicAdd` and `atomicCAS`. Our experimental validation shows that benchmark problems implemented in Futhark with generalized histograms are at least comparable to hand-written code and existing library solutions, and often significantly faster, ranging from $\times 0.83$ slowdown to $\times 30$ speedup.

References

[1] S. Brown and J. Snoeyink, "Modestly faster histogram computations on GPUs," in *Proceedings of Innovative Parallel Computing Foundations & Applications of GPU, Manycore, and Heterogeneous Systems (INPAR)*, ser. INPAR'12, 2012, pp. 1–7.

[2] V. T. Ravi, W. Ma, D. Chiu, and G. Agrawal, "Compiler and runtime support for enabling generalized reduction computations on heterogeneous parallel configurations," in *Proceedings of the 24th ACM International Conference on Supercomputing*, ser. ICS'10. ACM, 2010, pp. 137–146. [Online]. Available: https://doi.org/10.1145/1810085.1810106

[3] R. Shams and R. A. Kennedy, "Efficient histogram algorithms for NVIDIA CUDA compatible devices," in *Proceedings of International Conference on Signal Processing and Communication Systems (ICSPCS)*, 2007, pp. 418–422.

[4] V. Podlozhnyuk, "Histogram calculations in cuda," 2007. [Online]. Available: https://developer.download.nvidia.com/compute/cuda/1.1-Beta/x86_website/projects/histogram64/doc/histogram.pdf

[5] D. Merrill, "CUDA Unbound (CUB) Library," 2015. [Online]. Available: https://nvlabs.github.io/cub/

[6] A. Irpino and R. Verde, "Basic statistics for distributional symbolic variables: a new metric-based approach," *Advances in Data Analysis and Classification*, vol. 9, no. 2, pp. 143–175, 2015.

[7] E. Diday, "Principal component analysis for categorical histogram data: Some open directions of research," in *Classification and Multivariate Analysis for Complex Data Structures*, B. Fichet, D. Piccolo, R. Verde, and M. Vichi, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 3–15.

[8] R. B. Gurung, T. Lindgren, and H. Boström, "Learning decision trees from histogram data using multiple subsets of bins," in *Proceedings of the Twenty-Ninth International Florida Artificial Intelligence Research Society Conference*. AAAI Press, 2016, p. 430–435, [ed] Zdravko Markov, Ingrid Russell. [Online]. Available: http://urn.kb.se/resolve?urn=urn:nbn:se:kth:diva-221498

[9] S. Satoh, "Generalized histogram: Empirical optimization of low dimensional features for image matching," in *Computer Vision - ECCV 2004*, T. Pajdla and J. Matas, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 210–223.

[10] S. Darkner and J. Sporring, "Locally orderless registration," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 35, no. 6, pp. 1437–1450, 2013.

[11] K. Lu and H. Shen, "Multivariate volumetric data analysis and visualization through bottom-up subspace exploration," in *2017 IEEE Pacific Visualization Symposium, PacificVis 2017, Seoul, South Korea, April 18-21, 2017*, 2017, pp. 141–150.

[12] K. Wang, K. Lu, T. Wei, N. Shareef, and H. Shen, "Statistical visualization and analysis of large data using a value-based spatial distribution," in *2017 IEEE Pacific Visualization Symposium, PacificVis 2017, Seoul, South Korea, April 18-21, 2017*, 2017, pp. 161–170.

[13] C. E. Oancea and L. Rauchwerger, "Logical inference techniques for loop parallelization," in *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '12. New York, NY, USA: ACM, 2012, pp. 509–520. [Online]. Available: http://doi.acm.org/10.1145/2254064.2254124

[14] R. Mitchell, "Gradient boosting, decision trees and xgboost with cuda," 2017. [Online]. Available: https://devblogs.nvidia.com/gradient-boosting-decision-trees-xgboost-cuda/

[15] E. E. Catmull, "A subdivision algorithm for computer display of curved surfaces." Ph.D. dissertation, 1974, aAI7504786.

[16] Y. Ding, Y. Zhao, X. Shen, M. Musuvathi, and T. Mytkowicz, "Yinyang k-means: A drop-in replacement of the classic k-means with consistent speedup," in *Proceedings of the 32Nd International Conference on International Conference on Machine Learning - Volume 37*, ser. ICML'15. JMLR.org, 2015, pp. 579–587. [Online]. Available: http://dl.acm.org/citation.cfm?id=3045118.3045181

[17] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer, "Automatic differentiation in pytorch," in *NIPS 2017 Workshop on Autodiff*, 2017. [Online]. Available: https://openreview.net/forum?id=BJJsrmfCZ

[18] T. Henriksen, M. Dybdal, H. Urms, A. S. Kiehn, D. Gavin, H. Abelskov, M. Elsman, and C. Oancea, "APL on GPUs: A TAIL from the Past, Scribbled in Futhark," in *Procs. of the 5th Int. Workshop on Functional High-Performance Computing*, ser. FHPC'16. New York, NY, USA: ACM, 2016, pp. 38–43.

[19] A. G. Baydin, B. A. Pearlmutter, A. A. Radul, and J. M. Siskind, "Automatic differentiation in machine learning: A survey," *J. Mach. Learn. Res.*, vol. 18, no. 1, p. 5595–5637, Jan. 2017.

[20] T. Henriksen and C. E. Oancea, "A T2 graph-reduction approach to fusion," in *Proceedings of the 2Nd ACM SIGPLAN Workshop on Functional High-performance Computing*, ser. FHPC '13. New York, NY, USA: ACM, 2013, pp. 47–58. [Online]. Available: http://doi.acm.org/10.1145/2502323.2502328

[21] T. Henriksen, F. Thorøe, M. Elsman, and C. Oancea, "Incremental flattening for nested data parallelism," in *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '19. New York, NY, USA: ACM, 2019, pp. 53–67. [Online]. Available: http://doi.acm.org/10.1145/3293883.3295707

[22] J. A. Stratton, C. Rodrigues, I.-J. Sung, N. Obeid, L.-W. Chang, N. Anssari, G. D. Liu, and W.-m. W. Hwu, "Parboil: A revised benchmark suite for scientific and commercial throughput computing," *Center for Reliable and High-Performance Computing*, vol. 127, 2012.

[23] J. Koenderink and A. V. Doorn, "The structure of locally orderless images," *International Journal of Computer Vision*, vol. 31, no. 2, pp. 159–168, 1999.

[24] C. Nugteren, G.-J. van den Braak, H. Corporaal, and B. Mesman, "High performance predictable histogramming on GPUs: Exploring and evaluating algorithm trade-offs," in *Proceedings of the Fourth Workshop on General Purpose Processing on Graphics Processing Units*, ser. GPGPU-4. ACM, 2011, pp. 1:1–1:8.

[25] J. Gómez-Luna, J. M. González-Linares, J. I. Benavides, and N. Guil, "An optimized approach to histogram computation on GPU," *Machine Vision and Applications*, vol. 24, no. 5, pp. 899–908, Jul 2013.

[26] B. Dhanasekaran and N. Rubin, “A new method for gpu based irregular reductions and its application to k-means clustering,” in *Proceedings of the Fourth Workshop on General Purpose Processing on Graphics Processing Units*, ser. GPGPU-4. New York, NY, USA: ACM, 2011. [Online]. Available: https://doi.org/10.1145/1964179.1964182

[27] R. Baghdadi, U. Beaugnon, A. Cohen, T. Grosser, M. Kruse, C. Reddy, S. Verdoolaege, A. Betts, A. F. Donaldson, J. Ketema *et al.*, “Pencil: a platform-neutral compute intermediate language for accelerator programming,” in *2015 International Conference on Parallel Architecture and Compilation (PACT)*. IEEE, 2015, pp. 138–149.

[28] C. Reddy, M. Kruse, and A. Cohen, “Reduction drawing: Language constructs and polyhedral compilation for reductions on gpu,” in *Proceedings of the 2016 International Conference on Parallel Architectures and Compilation*, ser. PACT ’16. New York, NY, USA: ACM, 2016, pp. 87–97. [Online]. Available: http://doi.acm.org/10.1145/2967938.2967950

[29] T. L. McDonell, M. M. Chakravarty, G. Keller, and B. Lippmeier, “Optimising Purely Functional GPU Programs,” in *Procs. of Int. Conf. Funct. Prog. (ICFP)*, 2013.