An OpenACC Optimizer for Accelerating Histogram Computation on a GPU

Kei Ikeda, Fumihiko Ino and Kenichi Hagihara
Graduate School of Information Science and Technology
Osaka University
1-5 Yamadaoka, Suita, Osaka 565-0871, Japan
Email: {i-kei, ino}@ist.osaka-u.ac.jp

***Abstract*—This paper presents a source-to-source OpenACC optimizer that automatically optimizes a histogram computation code for a graphics processing unit (GPU). Parallel histogram computation codes typically deploy multiple copies of histograms and update them with atomic operations. This duplication method can be implemented as an OpenACC code. However, the structure of sequential code blocks must be manually rewritten owing to the limitation on OpenACC directives. Such a rewritten code does not always achieve the highest performance on arbitrary platforms, and thus, the duplication method degrades the performance portability of the code. To tackle this issue, we propose an optimizer that identifies histogram-related blocks in a naive OpenACC code and automatically rewrites the detected blocks such that multiple copies of histograms can be exploited for acceleration. In experiments, we apply our optimizer to three practical applications and investigate their performance on three platforms: an NVIDIA GPU, an AMD GPU and an Intel CPU. Experimental results show that our automated approach is useful for OpenACC codes to maximize the performance of histogram computation, and thereby enhancing the performance portability of the code.**

***Keywords*-Histogram computation; automated tuning; OpenACC; GPU;**

I. Introduction

Histogram computation is one of the important computational patterns that frequently appear in applications of various fields such as medical image processing [1], code assessment [2] and image recognition [3]. A *histogram* is an estimate of the probability distribution of a variable, and consists of a sequence of *bins*, which store the frequency of observations over categories (i.e., intervals of a variable). In most cases, bins are updated randomly due to the irregularity of observed values. Consequently, the irregularity in memory access patterns is the key issue that must be resolved for efficient parallelization of histogram computation. To deal with this challenging issue, parallel histogram computation is typically implemented with atomic operations to allow multiple threads to update the same bin correctly with serialization. Because the memory performance determines the throughput of bin updates, histogram computation is usually implemented on accelerators, which provide a magnitude of order higher memory bandwidth than conventional CPUs.

Accelerators such as the graphics processing unit (GPU) [4] and Xeon Phi [5] are emerging as green-aware many-core hardware in high-performance computing systems. Various memory- or compute-intensive applications [6]–[9] have been accelerated using accelerators successfully. These accelerators usually require application code rewriting to implement performance bottleneck part of the target application with a unique programming framework. As such a framework, the compute unified device architecture (CUDA) [10] is widely used for NVIDIA GPUs.

One issue on this unique programming style is the lack of *performance portability*. The performance portability of a code is defined as a code characteristic that represents how fast the code can run on different machines with less manual modification. Because the CUDA requires rewriting of the CPU code, the performance portability of the CUDA code is not so high. To reduce programming efforts, the GPU community standardized OpenACC [11], which allows application codes to be executed on different accelerators with minimum code modification. With OpenACC, programmers can easily implement an accelerated code by adding compiler directives in their sequential code. The OpenACC compiler then automatically generates a parallel code with data decomposition and parallel schemes specified by the inserted directives. Because an accelerator-specific code is automatically generated according to the directives, the original code can be tuned without modifying key statements that represent the essence of the computation. We think that this advantage, i.e., separation of platform-specific description from the general code, is useful to maximize the performance portability of the code.

However, as compared with CUDA, high-level OpenACC has several limitations on the programmability at lower levels, so that usually results in a low performance. Therefore, in most cases, manual rewriting of the sequential code is needed to tune the OpenACC code on the deployed accelerator. Such rewriting efforts degrade the performance portability of the code. For example, an acceleration method for histogram computation, called the duplication method, cannot be naively implemented by adding OpenACC directives to the sequential code; statements in code blocks must be rewritten to manage multiple histograms. These rewritten statements are redundant if the code is compiled as a sequential code, and thus, degrading the execution efficiency on a sequential machine. In addition, the rewritten

code does not always produce the highest performance on arbitrary accelerators.

In this paper, we present a source-to-source OpenACC optimizer capable of automatically tuning the performance of a histogram computation code for the GPU architecture. Given a naive OpenACC code, our optimizer detects code blocks where histograms are computed. The detected blocks are then tuned with entirely rewritten code blocks and appropriate directives such that the number of atomic conflicts are reduced with multiple local histograms on the GPU. Because the presented optimizer is fully automated, programmers are freed from managing several versions of the application code, each optimized for a specific platform. Our automated approach allows programmers to concentrate on developing the original, single OpenACC code.

The remainder of this paper is structured as follows. Section II introduces previous studies on GPU-accelerated histogram computation and directive-based programming frameworks. Section III describes the duplication method, which is the basis of our optimizer. Section IV summarizes technical issues on directive-based histogram computation. After that, Section V describes the design and implementation of our optimizer. Section VI shows experimental results obtained with several practical applications. Finally, Section VII concludes this paper with future directions.

II. Related Work

There are many studies that accelerated histogram computation on a CUDA-compatible GPU. Podlozhnyuk [12] accelerated 256-bin histogram computation for image processing applications. This CUDA-based implementation exploited on-chip shared memory [10] to realize efficient histogram computation. Similar approaches [13]–[15] exploited the shared memory to accelerate histogram computation. OpenACC provides the `cache` directive which allows programmers to specify the variables to be put into the shared memory. However, such low-level implementations cannot be fully realized with OpenACC, which hides detailed memory hierarchy from programmers [16]. Similarly, intrinsics such as the vote function [17] cannot be called from the OpenACC code. In contrast, the OpenACC code can run on a CPU if the code is compiled without OpenACC directives. The CUDA code does not possess such flexibility.

Some researchers extended the OpenACC specification [11] to improve the performance portability of the code. An extension of OpenACC presented in [18] allows a sequential code to process large data on the GPU. Because the capacity of the video memory is a magnitude lower than that of the main memory, OpenACC programmers are usually enforced to rewrite their code to swap out large data from the video memory. This extension hides such additional, accelerator-specific description to minimize programming efforts. Hoshino *et al.* [19] proposed an OpenACC extension for data layout transformation. This extension frees programmers from changing their data structures needed for performance tuning on accelerators. Similar directives were presented by Beyer *et al.* [20]. These extensions improve the programmability of OpenACC, but the target code must be rewritten to adapt itself to the extensions. In contrast, our optimizer automates rewriting of the OpenACC code, so that its performance can be tuned without modifying the original code.

Similar to OpenACC, OpenMPC [21] and XcalableACC [22] directives were proposed for GPU-accelerated computation. OpenMPC is an extension of OpenMP, which is designed for multithreading on a multi-core CPU. XcalableACC is an extension of XcalableMP [23], or a partitioned global address space (PGAS) language for distributed-memory multiprocessors such as cluster systems. Although our optimizer currently accepts only OpenACC codes, its key idea can be applied to other directive-based parallel frameworks.

```
1  /* Histogram initialization */
2  for (int i = 0; i < bins; i++) {
3    hist[i] = 0
4  }
5
6  /* Histogram computation */
7  for (int i = 0; i < n; i++) {
8    hist[data[i]]++;  /* 0 <= data[i] < bins */
9  }
```

Figure 1. Sequential code of histogram computation.

III. Histogram Computation on GPU

Figure 1 shows an example of a sequential code that implements histogram computation on a CPU. In this example, array `data` of size `n` stores the population to be examined, and array elements are sampled to enumerate observed values. Array `hist` of size `bins` represents a histogram, and its elements correspond to histogram bins. In Fig. 1, the histogram is initialized and computed at lines 2–4 and 7–9, respectively.

Typically, histogram computation can be parallelized by exploiting the data parallelism inherent in the computation. That is, the population to be examined is decomposed into small segments, which are then assigned to threads. Each thread then examines the assigned segment to update a histogram. Figure 2 shows a naive method that shares a single histogram among all threads. In this figure, array `hist` is updated by massively-parallel threads that are responsible for a segment of array `data`. Atomic operations are required to update the shared histogram correctly, because multiple threads can increment the same bin of the histogram. Such simultaneous votes to the same bin result in an atomic conflict, which can significantly drop the memory throughput due to serialization. Therefore, some conflict-reducing mechanisms are needed to achieve efficient acceleration for histogram computation.

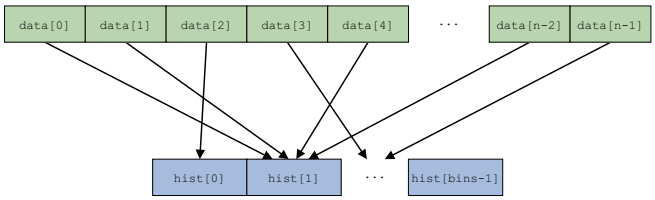


Figure 2. Parallel histogram computation with the naive method. A single histogram is shared among threads, each responsible for a segment of the input array data. Simultaneous votes to the same bin result in serialization of atomic writes.

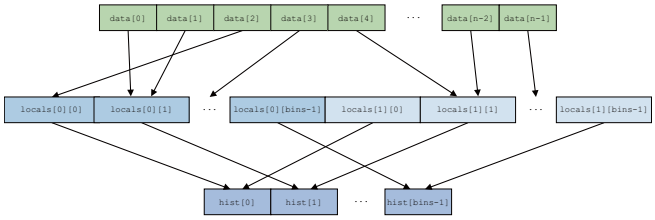


Figure 3. Parallel histogram computation with the duplication method. Two local histograms are shared among threads in this example. Threads associated with different local histograms never cause atomic conflicts between them.

To reduce the number of atomic conflicts, the duplication method shown in Fig. 3 allocates local histograms, or multiple copies of the original histogram. In this figure, two local histograms are allocated as a two-dimensional array `locals`. Similar to the naive method, these local histograms are accessed by multiple threads simultaneously, but threads associated with different local histograms do not cause any conflict between them; by contrast, threads with the same local histogram causes atomic conflicts. After this local histogram computation, the final histogram is generated by applying parallel reduction to local histograms.

There is a trade-off relation between the number of atomic conflicts and the locality of memory access. That is, the duplication method can reduce the number of atomic conflicts by increasing the number of local histograms. However, excessive local histograms result in a low performance due to random memory accesses from threads that update bins located in a wide range of memory regions. Furthermore, the overhead of parallel reduction increases with the number of local histograms. Therefore, the duplication method must be executed with an appropriate number of local histograms to maximize the performance. To the best of our knowledge, there is no scheme that estimates the appropriate number of local histograms that maximizes the performance. Therefore, we have to experimentally determine the number of local histograms.

IV. Histogram Computation with OpenACC

Both the naive method and the duplication method can be implemented with OpenACC. In this section, we summarize these implementations with technical issues.

```
1  #pragma acc data copyin(data[0:n]) copyout(hist[0:bins])
2  {
3  /* Histogram initialization */
4  #pragma acc parallel loop
5  for (int i = 0; i < bins; i++) {
6    hist[i] = 0
7  }
8
9  /* Histogram computation */
10 #pragma acc parallel loop
11 for (int i = 0; i < n; i++) {
12   #pragma acc atomic update
13   hist[data[i]]++;  /* 0 <= data[i] < bins */
14 }
15 }
```

Figure 4. OpenACC code for the naive method.

Figure 4 shows an example of an OpenACC code that implements the naive method. As compared with the sequential code (see Fig. 1), this OpenACC code has only four additional directives as follows.

- `#pragma acc data` at line 1, which specifies data transfer between the host and device, i.e., the CPU and GPU, respectively. This `data` directive indicates that the input data `data` and the output data `hist` must be transferred before and after histogram computation, respectively.
- Two `#pragma acc parallel loop` at lines 4 and 10, which specify the parallelization scheme for the loops to be executed on the device. The histogram is initialized and computed at lines 4–7 and 10–14, respectively.
- `#pragma acc atomic update` at line 12, which indicates that the next sentence causes memory references that must be processed atomically.

In other words, the blocks in this code is kept as is in the sequential code. The only differences over the sequential code are the abovementioned directives, which can be easily ignored when compiling the code as a sequential code. Consequently, this code has high performance portability.

In contrast to the naive method, more efforts are required to implement the duplication method with OpenACC. Figure 5 presents an example of an OpenACC code that implements the duplication method. In this example, eight local histograms are allocated, and each is assigned to a group of gangs. A *gang* [11] is a group of threads that can exploit coarse-grained parallelism on the device; a gang in OpenACC corresponds to a thread block [10] in CUDA. Similarly, a vector is a group of threads that can run in a single-instruction, multiple-data (SIMD) manner; the vector size corresponds to the number of threads in a thread block. This code has many differences over the sequential code in Fig. 1. In particular, this OpenACC code has not only additional directives but also modified code blocks to manage multiple local histograms. Owing to the modified blocks, which require more memory references than the original blocks, the modified code cannot run efficiently as

```
1  const int vector_size = 256;
2  const int gangs = (int)ceil((double)n/vector_size);
3
4  const int num_locals = 8;
5  int *locals = (int*)malloc(sizeof(int)*bins*num_locals);
6
7  #pragma acc data copyin(data[0:n]) copyout(hist[0:bins])
       create(locals[0:bins*num_locals])
8  {
9  /* Histogram initialization */
10 #pragma acc parallel loop
11 for (int i = 0; i < bins; i++) {
12   hist[i] = 0
13 }
14
15 /* Local histogram initialization */
16 #pragma acc parallel loop tile(256,1)
17 for (int j = 0; j < num_locals; j++) {
18   for (int i = 0; i < bins; i++) {
19     locals[bins*j + i] = 0;
20   }
21 }
22
23 /* Local histogram computation */
24 #pragma acc parallel loop num_gangs(gangs) vector_length
       (vector_size)
25 for (int i = 0; i < n; i++) {
26   int num_iters = (int)ceil((double)n/gangs);
27   int gang_id = i/num_iters;
28   int charge = gang_id%num_locals;
29   #pragma acc atomic update
30   locals[bins*charge + data[i]]++;  /* 0 <= data[i] <
         bins */
31 }
32
33 /* Local histogram reduction */
34 #pragma acc parallel loop tile(256,1)
35 for (int j = 0; j < num_locals; j++) {
36   for (int i = 0; i < bins; i++) {
37     #pragma acc atomic update
38     hist[i] += locals[bins*j + i];
39   }
40 }
41 }
42
43 free(locals);
```

Figure 5. OpenACC code for the duplication method. A gang of size `vector_size` corresponds to a thread block of vector threads.

a sequential program. Therefore, the modified code has low performance portability though it maximizes the effective performance on the GPU.

In Fig. 5, array `locals` of size `num_locals`×`bins` holds local histograms, where `num_locals` and `bins` specify the number of local histograms and that of bins in each histogram, respectively. The `malloc` function at line 5 allocates a main memory region for `num_locals` local histograms. Similarly, a memory region must be allocated on the device memory with the `create` clause at line 7. After that, local histograms are initialized at lines 16–21, and then computed at lines 24–31 in parallel with an `atomic` clause. Finally, local histograms are reduced into the final global histogram at lines 34–40. This parallel reduction is implemented with an `atomic` clause. The `tile` clause at line 16 expresses a data locality in the nested loop.

Notice that gang indexes are required to associate local histograms with specific gangs. Such an association can be easily established with CUDA. However, this association cannot be explicitly specified with OpenACC, which does not provide an API function that returns a gang index; OpenACC hides not only the hierarchy of the memory architecture but also that of the processor architecture. Our solution for this issue is to estimate gang indexes from the value of the loop variable. That is, we assume that gang indexes can be estimated from loop information, because the `parallel` construct is designed for data parallelism, which mainly exists in loops. We confirmed that this assumption works fine with the PGI compiler [24], which parallelizes the loop body with a block assignment scheme; given a loop structure with n iterations, the number g of gangs is given by $\lceil n/v \rceil$, where $v = 256$ represents the vector size; each of g gangs is then responsible for $c = \lceil n/g \rceil$ contiguous iterations. In Fig. 5, the value of c is stored in variable `num_iters` and a gang index is stored in variable `gang_id`. Variable `charge` holds the histogram index associated with a gang of index `gang_id`.

V. OpenACC Optimizer

Given an OpenACC C code as an input, our source-to-source optimizer outputs a tuned OpenACC C code. To realize this, it firstly detects code blocks where histograms are computed. The detected blocks are then rewritten such that multiple local histograms are initialized, computed and reduced in parallel, as presented in Fig. 5.

Figure 6 shows a typical compilation flow using our optimizer, which must be applied before compiling the OpenACC code. Our optimizer is separated from the deployed OpenACC compiler. This separated design contributes to establish high flexibility, allowing our optimization framework to be used with any compiler.

A. Code Block Detection

A code block where histograms are computed can be detected as a `for` loop that satisfies the following two conditions.

C1) The device executes the `for` loop. This condition becomes true if the `for` loop is associated with a `#pragma acc loop` directive. Instead of this, other variations such as `#pragma acc kernels loop` and `#pragma acc parallel loop` are acceptable.

C2) An array element is incremented with atomic operations in the body of the `for` loop. This condition becomes true if a `#pragma acc atomic update` directive is specified to the sentence that increments the array. The `update` clause can be omitted.

After this detection, we assume that histograms are stored in the arrays found according to condition C2).

As mentioned above, our detection strategy assumes that the target code blocks are parallelized with OpenACC.

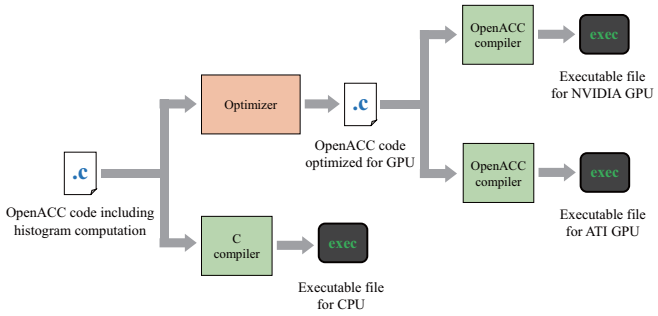


Figure 6. A typical flow with the proposed OpenACC optimizer.

Therefore, this strategy fails to detect code blocks where histograms are computed sequentially. Furthermore, histogram computation is usually a stage of a bigger algorithm, as presented later in Section VI. Thus, in real life applications it would be important to fuse several loops in the same kernel whenever possible. Such fused loops may require a different detection strategy for optimization.

B. Code Generation

In order to generate a tuned OpenACC code, our optimizer applies the following code rewrite rules to the detected `for` loops.

a) The rule on memory management for local histograms. To allocate and free a main memory region for local histograms, `malloc` and `free` functions are placed before and after the detected `for` loop, respectively. Furthermore, a `create` clause is added to the existing `#pragma acc data` directive to allocate a device memory region for local histograms. This `create` clause must have an argument to specify the arrays that store local histograms. If a `create` clause already exists in the original code, the abovementioned argument is simply added to the existing `create` clause.

b) The rule on local histogram initialization and reduction. Initialization and reduction operations are added before and after the detected `for` loop, respectively. These operations can be implemented with a double-loop in which the outer loop traverses local histograms while the inner loop traverses bins of a local histogram. Both operations are executed on the device by adding a `#pragma acc parallel loop tile` directive to the double loop. The parallel reduction is implemented with atomic operations.

c) The rule on local histogram computation. As mentioned in Section IV, the duplication method assigns each local histogram to a group of gangs. This assignment requires the number of gangs that participate in histogram computation. If the original code does not specify this number, our optimizer specifies the number explicitly by adding either a `gang` clause or a `num_gangs` clause; the former and the latter are used for loops with a `kernel` construct and those with a `parallel` construct, respectively. The number of gangs depends on the vector size, which is experimentally determined as 256. Furthermore, a statement is added within the body of the detected `for` loop to compute the index of the local histogram assigned to each gang. Finally, references to the original histogram are replaced with those to a local histogram.

The number of local histograms must be experimentally determined to maximize the performance of irregular memory access. This number is restricted by the capacity of the device memory, because the amount of video memory usage increases with the number of local histograms. Therefore, our optimizer produces a flexible code such that the code can vary the number of local histograms.

Table I
SPECIFICATION OF COMPUTED HISTOGRAMS AND JOINT HISTOGRAMS. EACH BIN STORES 32-BIT DATA.

Application	Histogram size (bin)	(KB)	Number of bin updates
Color histogram	768	3	$1920 \times 1080 \times 3$
Mutual information	256	1	$256 \times 256 \times 89$
	256	1	$256 \times 256 \times 89$
	256×256	256	$256 \times 256 \times 89$
Line detection	6003×182	4268	(Number of edge pixels) $\times 180$

C. Implementation

Our optimizer is implemented using the ROSE compiler infrastructure [25], which provides C and C++ frontend to generate an abstract syntax tree (AST) of the given code. The generated AST is then traversed to detect the vertices that satisfy conditions C1) and C2). The detected vertices are marked explicitly for modification, so that our rewrite rules are applied to them in the next traversal. Finally, the modified AST is given to the ROSE code generator to obtain a tuned OpenACC code.

VI. EXPERIMENTS

We applied our optimizer to three practical applications and measured the performance of the generated OpenACC code. Table I summarizes the specification of computed histograms and joint histograms.

1) Color histogram computation. A color histogram, namely the distribution of colors in an image, is generated. Because a pixel consists of red, green and blue colors, an observation updates three bins in the histogram.
2) Mutual information computation [1]. Many image registration algorithms adopt mutual information as a similarity measure of images to be aligned. This computation generates two histograms and a joint histogram, each with a different number of bins. An observation updates a single bin of all (joint) histograms.

Table II
SPECIFICATION OF OUR EXPERIMENTAL MACHINE.

Item	Machine #1	Machine #2
OS	Ubuntu 14.04.2 LTS	
CPU	Intel Xeon E5-2680 v2	Intel Xeon E5-2660 v3
Main memory	512 GB	64 GB
GPU	NVIDIA Tesla K40	AMD Radeon HD 7970
Video memory	12 GB	3 GB
Compiler	PGI C compiler 15.5 [24]	
Compiler option	`-acc -O3 -ta=tesla,cc35`	`-acc -O3 -ta=radeon,tahiti`

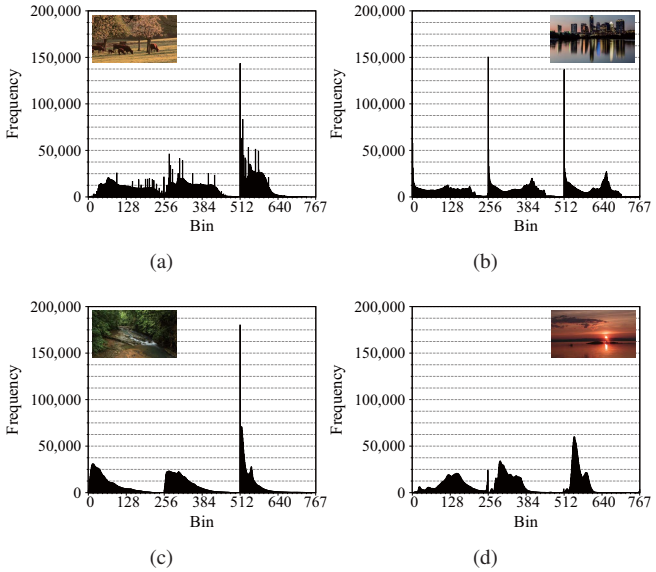


Figure 7. Computed color histograms. Results for (a) animals, (b) austin, (c) nature and (d) sunset datasets. Bins in the range $[0, 255]$, $[256, 511]$ and $[512, 767]$ correspond to red, green and blue channels.

3) Line detection using Hough transform [26]. The Hough transform is useful for image recognition applications. A single joint histogram is computed during this transform, and an observation updates 180 bins in the joint histogram. The original code was developed by modifying an example code in the OpenCV library [27].

Table II shows the specification of our experimental machines. The error check and correct (ECC) capability of the Tesla card was turned off during measurement. The PGI C compiler 15.5 [24] was used to compile the experimental applications. CPU-based implementations ran on a single core of the Xeon E5-2660 CPU.

A. Color Histogram Computation

Color histograms are computed for four 24-bit color images obtained from a media repository called Wikimedia Commons (https://commons.wikimedia.org). These images consist of 1920×1080 pixels, each having 24-bit RGB channels. Because each color has 8-bit depth, a histogram consists of 768 $(= 256 \times 3)$ bins, as shown in Fig. 7.

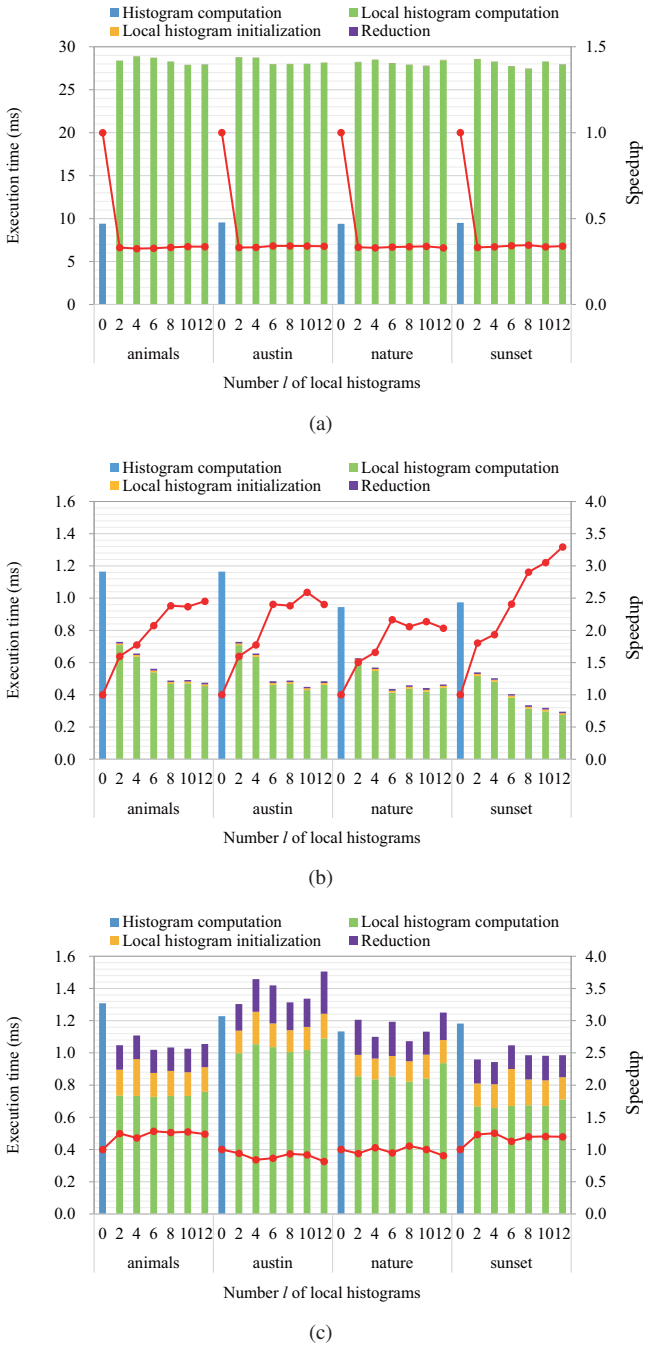


Figure 8. Execution times for color histogram computation. Results on (a) Xeon E5-2660, (b) Tesla and (c) Radeon. The naive method corresponds to $l = 0$.

Using these datasets, we measured the execution times for color histogram computation. The execution times in Fig. 8 do not include the data transfer time between the CPU and GPU, because color histogram computation typically is a pipeline stage of an image processing application; the input images exist on the device memory and the output images are rendered on the display without transferring back to the

host memory.

In Fig. 8, the best speedups over the naive method were $\times$2.5, $\times$2.6, $\times$2.2 and $\times$3.3 for animals, austin, nature and sunset dataset, which are obtained on the Tesla card with $l = 12$, 10, 6 and 12, respectively. The maximum memory throughput reached 112 GB/s when using $l = 12$ for the sunset dataset. This effective throughput was 39% of the peak memory bandwidth (288 GB/s). For all datasets except the sunset dataset, the performance was saturated with more than $l = 8$ local histograms. These datasets frequently incremented a few bins, as shown in Fig. 7. Such bottleneck bins were frequently accessed in a small image region, and thereby increasing the number l of local histograms was not so effective when $l \geq 8$. By contrast, the sunset dataset had a relatively uniform distribution compared to the remaining datasets. Such a uniform distribution was efficiently processed with the duplication method.

The duplication method also slightly improved the performance on the Radeon card, but it failed to outperform the naive method for the austin and nature datasets. Similarly, the performance on the Xeon CPU was decreased by 33%, due to the overhead for managing multiple local histograms. Thus, the duplication method is not a perfect solution for arbitrary datasets and platforms. Our optimizer automatically rewrites the naive code for the duplication method, so that it allows application programmers to concentrate on managing the naive code, which excludes platform-specific description.

B. Mutual Information Computation

As clinical datasets to be aligned, we used a set of computed tomography (CT), T1- and T2-weighted magnetic resonance (MR) images obtained from the Vanderbilt Database [28]. These datasets consist of $256 \times 256 \times 89$ voxels, each having 8-bit intensity value. Therefore, their histogram and joint histogram have 256 bins and 256×256 bins, respectively.

Figure 9 shows histograms and joint histograms generated for mutual information computation. Most of the observations occurs at only several bins. The maximum frequency was below 4M and 1M for the CT and MR images, respectively.

Figure 10 shows the execution times of mutual information computation for each pair of images. The best speedups were observed on the Tesla card: $\times$3.2, $\times$3.6 and $\times$2.9 for CT–MR-T1, CT–MR-T2 and MR-T1–MR-T2 pairs, respectively. Because only a few tens of bins collected votes, local histograms successfully reduced the execution time. Consequently, the duplication method outperformed the naive method on the Radeon card.

Although this application generated three (joint) histograms, the overhead of the duplication method was negligible. For example, parallel reduction for $l = 12$ local histograms took 0.07 ms on the Tesla card, which was

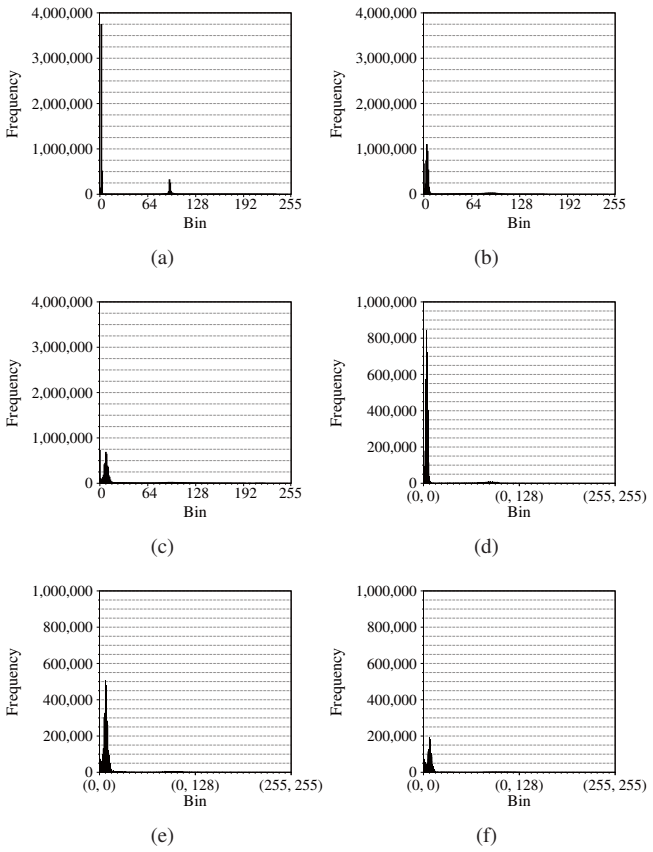


Figure 9. Histograms and joint histograms generated for mutual information computation. Histograms for (a) CT, (b) MR-T1 and (c) MR-T2 images. Joint histograms for (d) CT–MR-T1, (e) CT–MR-T2 and (f) MR-T1–MR-T2 pairs.

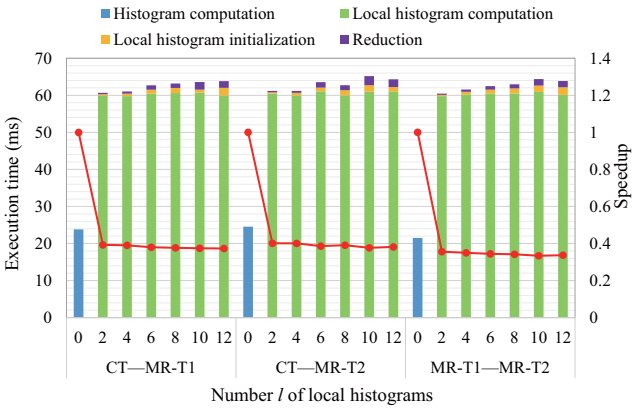
4% of the total execution time. Similarly, local histogram initialization completed within 0.05 ms when $l = 12$.

In summary, our generated code achieves a high speedup over the naive method if sampled data intensively updates a small number of bins. In particular, a significant speedup can be expected if a small number of bins collect at least millions of votes.

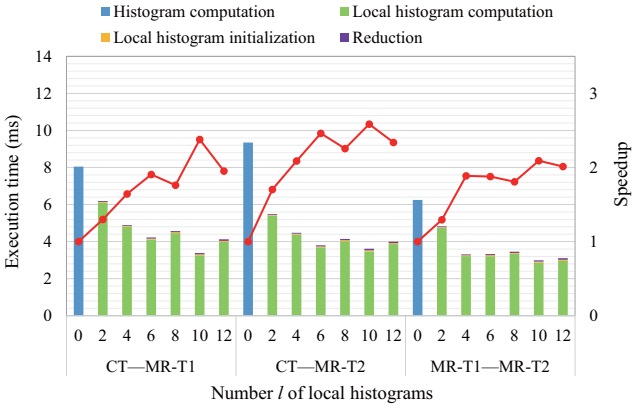
C. Line Detection using Hough Transform

We used four images obtained from Wikimedia Commons (https://commons.wikipedia.org). These images consist of 1920×1080 pixels, each having 4-byte data. The ratios of edge pixels were 5%, 11%, 16% and 21% for court, keely, nature and animals datasets, respectively. The generated joint histograms consist of 6003×182 bins, each with 32-bit data. As shown in Table I, this application generated a relatively large joint histogram than the remaining two applications.

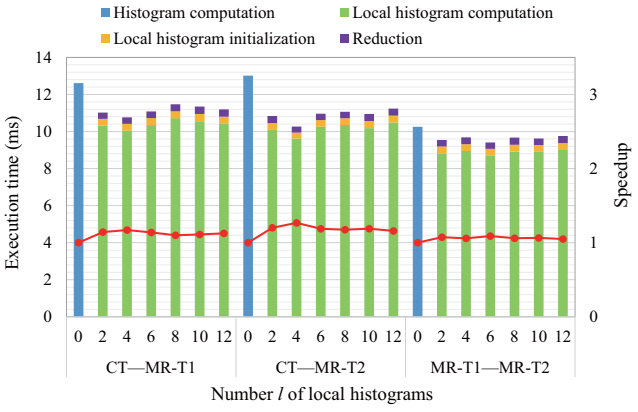
Figure 12 shows the execution times for line detection. Surprisingly, the duplication method failed to outperform the naive method on all platforms for all images; the speeups ranged from $\times$0.67 to $\times$0.78 on the Tesla card. This behavior can be explained by the examined populations,



(a)



(b)



(c)

Figure 10. Execution times for mutual information computation. Results on (a) Xeon E5-2660, (b) Tesla and (c) Radeon. The naive method corresponds to $l = 0$.

which had relatively small standard deviations, as shown in Fig. 11; almost all bins have at least 100 votes and at most 1200 votes. Consequently, simultaneous accesses to the same bin were not so frequent as compared with the first two applications. Furthermore, the data size of the joint histogram was 4268 KB, which cannot be cached

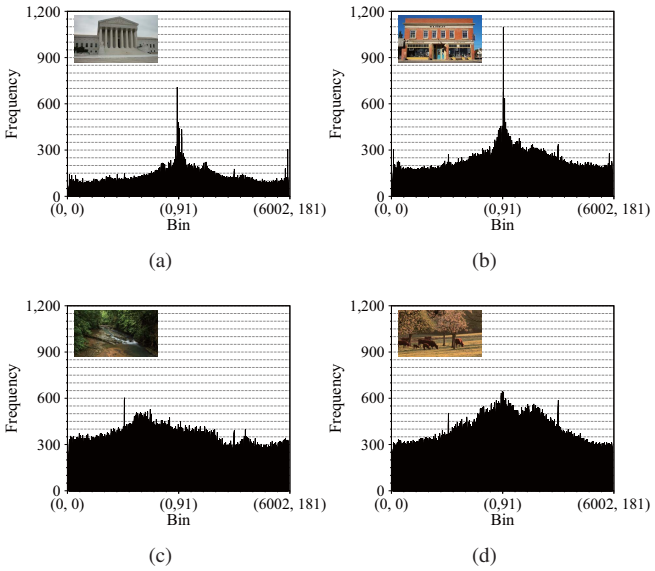


Figure 11. Joint histograms computed for line detection. Results for (a) court, (b) keely, (c) nature and (d) animals datasets.

entirely. Because increasing the number l of local histograms degrades the locality of memory access, such large local histograms caused many cache misses.

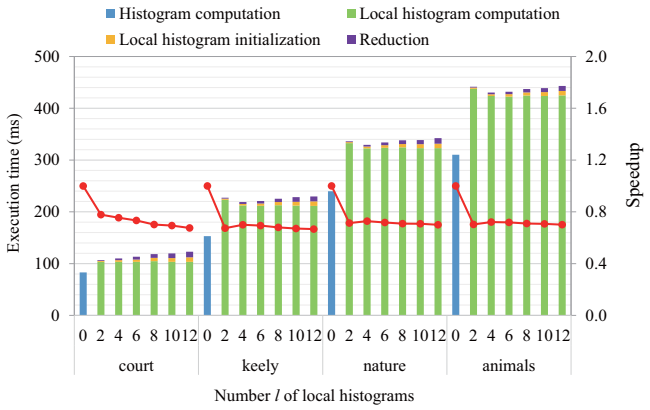
Thus, the duplication method was ineffective for this application. Our automated approach is useful for such applications, because our optimizer allows prograrmmers to keep the naive code.

VII. Conclusion

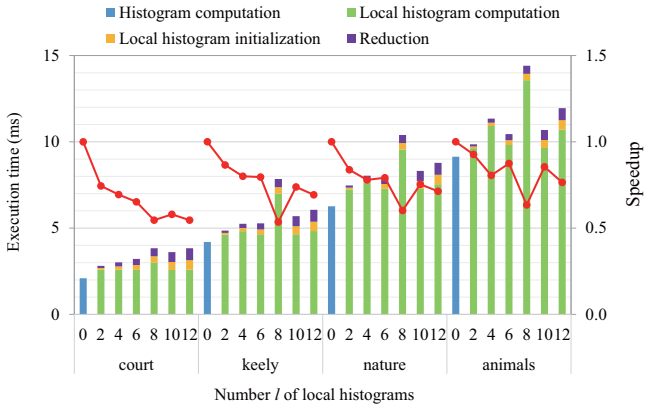
We have presented an optimizer capable of tuning an OpenACC code that implements histogram computation on a GPU. To realize this, our optimizer detects code blocks where histograms are computed. The detected code blocks are then tuned with the duplication method, which distributes atomic operations over multiple local histograms. Because our optimizer automates rewriting of statements in the code, the effective performance on the GPU can be maximized with high performance portability.

In experiments, we applied our optimizer to three practical applications. The achieved performance shows that our optimizer is useful to accelerate GPU-enabled histogram computation without modifying the OpenACC code. The achieved speedups over the naive method ranged from $\times 0.7$ to $\times 3.6$, according to the distribution of the population to be sampled for histogram computation. The duplication method is useful to reduce the number of atomic conflicts, particularly for irregular applications that intensively update a few tens of bins in the histogram.

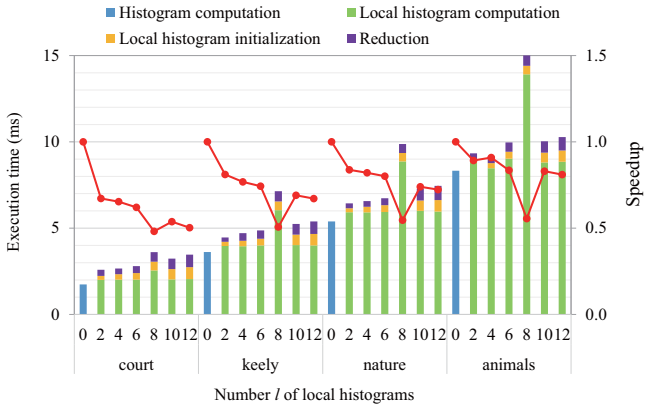
One future work is to develop a tuning mechanism that automates estimation of an appropriate number of local histograms.



(a)



(b)



(c)

Figure 12. Execution times for line detection. Results on (a) Xeon E5-2660, (b) Tesla and (c) Radeon. The naive method corresponds to $l = 0$.

ACKNOWLEDGMENTS

This study was supported in part by the Japan Society for the Promotion of Science KAKENHI Grant Numbers 15K12008 and 15H01687, and the Japan Science and Technology Agency CREST program, "An Evolutionary Approach to Construction of a Software Development Environment for Massively-Parallel Computing Systems." The authors would like to thank the anonymous reviewers for helpful comments to improve their paper.

REFERENCES

[1] K. Ikeda, F. Ino, and K. Hagihara, "Efficient acceleration of mutual information computation for nonrigid registration using CUDA," *IEEE J. Biomedical and Health Informatics*, vol. 18, no. 3, pp. 956–968, May 2014.

[2] S. Ando, F. Ino, T. Fujiwara, and K. Hagihara, "Enumerating joint weight of a binary linear code using parallel architectures: multi-core CPUs and GPUs," *Int'l J. Networking and Computing*, vol. 5, no. 2, pp. 290–303, Jul. 2015.

[3] J. Xiao, J. Hays, K. A. Ehinger, A. Oliva, and A. Torralba, "SUN database: Large-scale scene recognition from abbey to zoo," in *Proc. 23rd IEEE Conf. Computer Vision and Pattern Recognition (CVPR'10)*, Jun. 2004, pp. 3485–3492.

[4] NVIDIA Corporation, "NVIDIA's Next Generation CUDA Compute Architecture: Kepler GK110," May 2012. [Online]. Available: http://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf

[5] Intel Corporation, "Intel Xeon Phi product family." [Online]. Available: http://www.intel.com/content/www/us/en/processors/xeon/xeon-phi-detail.html

[6] J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone, and J. C. Phillips, "GPU computing," *Proceedings of the IEEE*, vol. 96, no. 5, pp. 879–899, May 2008.

[7] D. Okada, F. Ino, and K. Hagihara, "Accelerating the Smith-Waterman algorithm with an interpair pruning method for all-pairs comparison of base sequences," *BMC Bioinformatics*, vol. 16, no. 321, Oct. 2015, 15 pages.

[8] F. Ino, Y. Munekawa, and K. Hagihara, "Sequence homology search using fine grained cycle sharing of idle GPUs," *IEEE Trans. Parallel and Distributed Systems*, vol. 23, no. 4, pp. 751–759, Apr. 2012.

[9] Y. Okitsu, F. Ino, and K. Hagihara, "High-performance cone beam reconstruction using CUDA compatible GPUs," *Parallel Computing*, vol. 36, no. 2/3, pp. 129–141, Feb. 2010.

[10] NVIDIA Corporation, "CUDA C Programming Guide Version 7.5," Sep. 2015. [Online]. Available: http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf

[11] OpenACC-Standard.org, "The OpenACC application programming interface, version 2.0," Aug. 2013.

[12] V. Podlozhnyuk, "Histogram calculation in CUDA," Jul. 2012. [Online]. Available: http://docs.nvidia.com/cuda/samples/3_Imaging/histogram/doc/histogram.pdf

[13] R. Shams and R. A. Kennedy, "Efficient histogram algorithms for NVIDIA CUDA compatible devices," in *Proc. Int'l Conf. Signal Processing and Communications Systems (ICSPCS'07)*, Dec. 2007, pp. 418–422.

[14] 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 *Proc. 4th Workshop General Purpose Processing on Graphics Processing Units (GPGPU'11)*, Mar. 2011.

[15] U. Milic, I. Gelado, N. Puzovic, A. Ramirez, and M. Tomasevic, "Parallelizing general histogram application for CUDA architectures," in *Proc. Proc. 13rd Int'l Conf. Embedded Computer Systems: Architectures, Modeling and Simulation (SAMOS'13)*, Jul. 2013.

[16] A. Lashgar and A. Baniasadi, "Employing software-managed caches in OpenACC: Opportunities and benefits," *ACM Trans. Modeling and Performance Evaluation of Computing Systems*, vol. 1, no. 1, article. 1, Dec. 2015.

[17] I. Egielski, J. Huang, and E. Z. Zhang, "Massive atomics for massive parallelism on GPUs," in *Proc. 4th Int'l Symp. Memory Management (ISMM'14)*, Jun. 2014, pp. 93–103.

[18] T. Kato, F. Ino, and K. Hagihara, "PACC: An extension of OpenACC for pipelined processing of large data on a GPU," in *Poster 27th Int'l Conf. High Performance Computing, Networking, Storage and Analysis (SC'14)*, Nov. 2014.

[19] T. Hoshino, N. Maruyama, and S. Matsuoka, "An OpenACC extension for data layout transformation," in *Proc. 1st Workshop Accelerator Programming using Directives (WACCPD'12)*, Nov. 2014, pp. 12–18.

[20] J. Beyer, D. Oehmke, and J. Sandoval, "Transferring user-defined types in OpenACC," in *Proc. Cray User Group (CUG'14)*, May 2014.

[21] S. Lee and R. Eigenmann, "OpenMPC: extended OpenMP for efficient programming and tuning on GPUs," *Int'l J. Computational Science and Engineering*, vol. 8, no. 1, pp. 4–20, 2013.

[22] M. Nakao, H. Murai, T. Shimosaka, A. Tabuchi, T. Hanawa, Y. Kodama, T. Boku, and M. Sato, "XcalableACC: Extension of XcalableMP PGAS language using OpenACC for accelerator clusters," in *Proc. 2nd Workshop Accelerator Programming using Directives (WACCPD'13)*, Nov. 2014, pp. 27–36.

[23] M. Nakao, J. Lee, T. Boku, and M. Sato, "Productivity and performance of global-view programming with XcalableMP PGAS language," in *Proc. 12th IEEE/ACM Int'l Symp. Cluster, Cloud and Grid Computing (CCGRID'12)*, May 2012, pp. 402–409.

[24] NVIDIA Corporation, "PGI compiler," 2015. [Online]. Available: http://www.pgroup.com/

[25] C. Liao, D. J. Quinlan, R. Vuduc, and T. Panas, "Effective source-to-source outlining to support whole program empirical optimization," in *Proc. 23rd Int'l Workshop Languages and Compilers for Parallel Computing (LCPC'10)*, Oct. 2010, pp. 308–322.

[26] P. V. C. Hough, "Machine analysis of bubble chamber pictures," in *Proc. 2nd Int'l Conf. High-Energy Accelerators and Instrumentation (HEACC '59)*, Sep. 1959, pp. 554–558.

[27] "The OpenCV library," 2015. [Online]. Available: http://opencv.org/

[28] J. M. Fitzpatrick, "The retrospective image registration evaluation project," 2008. [Online]. Available: http://www.insight-journal.org/rire/