# Performance Enhancement of a Computational Persistent Homology Package

Alan Hylton
*Space Communications and Navigation*
*NASA Glenn Research Center*
Cleveland, OH, USA
alan.g.hylton@nasa.gov

Greg Henselman
*Dept. of Electrical and Systems Engineering*
*University of Pennsylvania*
Philadelphia, PA, USA
grh@seas.upenn.edu

Janche Sang
*Dept. of Elect. Eng. and Computer Science*
*Cleveland State University*
Cleveland, OH, USA
sang@eecs.csuohio.edu

Robert Short
*Dept. of Mathematics*
*Lehigh University*
Bethlehem, PA, USA
rss212@lehigh.edu

***Abstract*—In recent years, persistent homology has become an attractive method for data analysis. It captures topological features, such as connected components, holes, voids, etc., from a point cloud by finding out when these features appear and disappear in the filtration sequence. In this project, we focus on improving the performance of Eirene, a fancy computational persistent homology package. Eirene is a 5000-line open-source software implemented by using the dynamic programming language Julia. We use the Julia profiling tools to identify the performance bottlenecks and develop different methods to manage the bottlenecks, including the parallelization of some time-consuming functions on the multicore/manycore hardware. The empirical results show that the performance can be greatly improved.**

***Keywords*-Performance Optimization, Profiling, Persistent Homology, Multicore/Manycore Computing**

## I. Introduction

Persistent Homology, formalized by Edelsbrunner, Letscher and Zomorodian [1] at the beginning of the last decade, is a method from algebraic topology adapted in an algorithmic context to topological data analysis. It extracts the homology classes, such as connected components, cycles, voids, etc., from a point cloud data set and keeps track of when these classes appear and disappear in the filtration sequence. Persistent homology finds numerous applications in research areas involving very large data sets, including biology [2], image analysis [3], sensor networks [4], Cosmology [5], etc. A detailed roadmap for the computation of persistent homology can be found in [6].

Eirene is a state-of-the-art open-source platform for computational persistent homology [7]. It is implemented in Julia [8], a high-performance dynamic scripting language for numerical and scientific computation. Eirene adopts the novel relationship between the Schur complement (in linear algebra), discrete Morse Theory (in computational homology), and minimal bases (in combinatorial optimization) unearthed in [9] to build the algorithms. Hence, its performance can be much faster than most of other computational persistent homology packages. Furthermore, it provides user-friendly utilities for graphical visualization of homological classes. Figure 1 is a 3D visualization WorldMap graph displayed by Eirene which shows a persistent 1-cycle of cities embedded in the Eurasian continent.

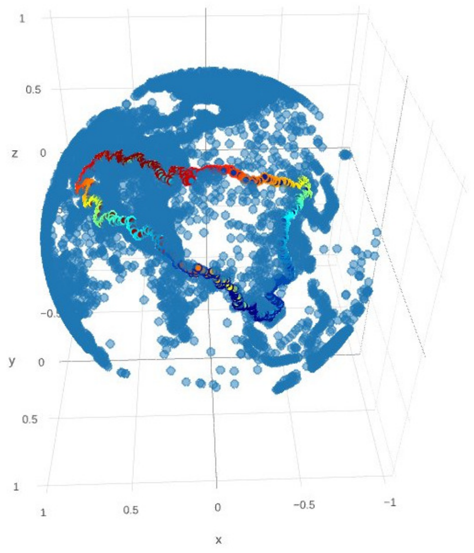


Fig. 1. A Cycle in the WorldMap displayed by the Eirene tool

However, we noticed that Eirene runs slower when we used the recent release Julia v0.6. Therefore, the objective of this project was to optimize the performance of Eirene. We used the profiling technique to identify the potential performance bottlenecks. Note that the software profiling tool, which can display the call graph and the amount of time spent in each function, has been used to tune program performance for several decades [10] [11]. After locating each of the

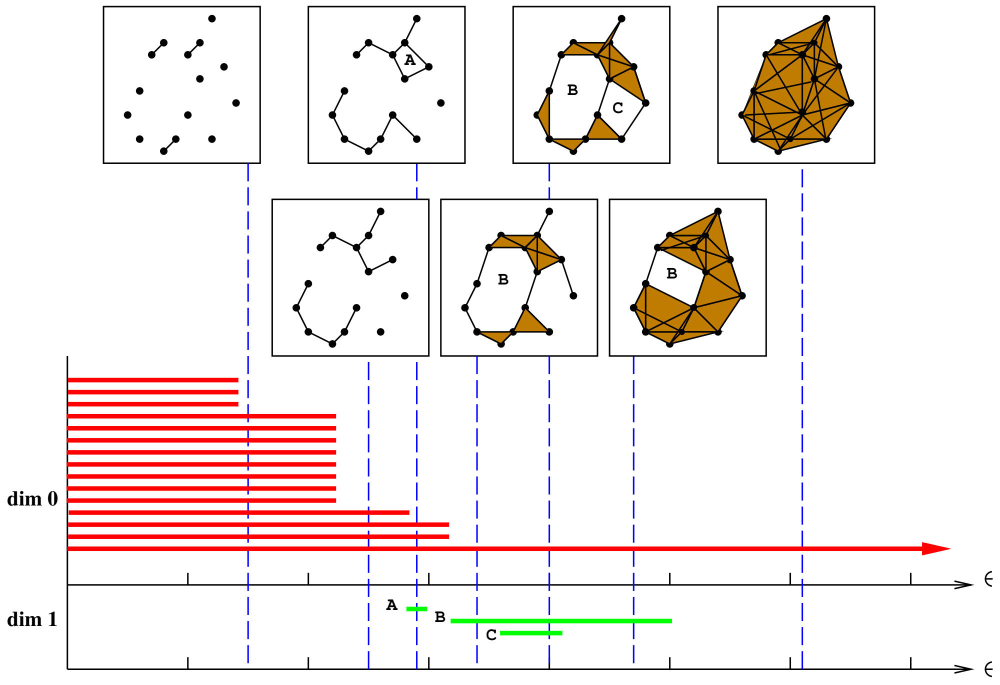


Fig. 2. An example of zero- and one-dimensional barcodes for a sequence of Vietoris-Rips complexes

bottlenecks, we found the cause of it and developed a method to solve the problem. For some time-consuming functions, we re-implemented the code and ported them on the multicore and manycore architectures, by using pthreads [12] and CUDA threads [13], respectively. The experimental results showed that the performance can be improved significantly.

The rest of the paper is organized as follows. In Section 2, we briefly review the necessary background on persistent homology. In Section 3, we identify the bottlenecks and propose different performance-improving methods. We evaluate our methods by conducting benchmark experiments in Section 4. A short conclusion is given in Section 5.

## II. Background

Homology is a tool in algebraic topology for analyzing the connectivity of simplicial complexes, such as points, edges, solid triangles, solid tetrahedra, and other higher dimensional shapes. By using homology, we can measure several features of the data in metric spaces – including the numbers of of connected components (0-cycle), holes(1-cycle), voids(2-cycle), etc. In this project we limit the study in the homology of Vietoris-Rips complexes only. To construct a Vietoris-Rips complex, a distance threshold $\epsilon$ is chosen first. Then, any two points which are less than the distance $\epsilon$ from each other are connected by an edge. A solid triangle is created if all its three edges have been generated. A solid tetrahedra is constructed if all its face triangle have been created. Similar constructions are used to build higher-dimensional simplexes

Note that using different distance thresholds to generate the Vietoris-Rips complexes from the same point cloud data set may result in very different homologies, i.e different numbers and types of cycles. Because cycles are the topologically significant features to be captured from the data set, we do not want to miss any of them. For solving this problem, a method, which is called persistent homology [1], is to generate Vietoris-Rips complexes from a set of points at every distance threshold and then derive when the cycles appear and disappear in the complexes as the distance increases. Therefore, the distance threshold is often referred to as time.

The topological data produced by using persistent homology can be visualized through a barcode [14]. A barcode is a collection of intervals, where each interval represents a cycle that exists in at least one Vietoris-Rips complex generated from a point cloud. The left endpoint and the right endpoint of an interval represent the birth and the death times of a cycle, respectively. The horizontal axis corresponds to the distance threshold and the vertical axis represents an arbitrary ordering of captured cycles.

Figure 2 illustrates an example of zero- and one-dimensional barcodes for a sequence of Vietoris-Rips complexes. The red bars represent the lifetime of the connected components. Note that at beginning the number of components is the same as the number of points. When the distance threshold $\epsilon$ increases, the number of components is decreased because more and more components are connected together. The green bars represents the lifetime of the cycles in dimension 1. The cycle disappear when it is completely filled in by solid triangles. It can be observed that a cycle, denoted as B in the diagram, is significantly longer than the others, while the cycle A is short-lived and can be considered as noise. A barcode also encodes all the information regarding the Betti numbers on different scales. Betti numbers count topological features, like connected components (0th Betti number), holes (1st Betti number), voids (2nd Betti number), etc., for an individual simplicial complex. For one of the Vietoris-Rips complexes in the barcode, these numbers can be obtained by counting the numbers of the intersections between a vertical line and the bars in the diagram.

For the computation of persistent homology, there are several open-source software packages available, such as javaPlex [15], Dionysus [16], Perseus [17], etc. Basically, for computing persistence intervals of a finite filtration, a program firstly constructs filtered simplicial complexes from a point cloud and store the information into a matrix. Next, it applies matrix reduction techniques and obtains the intervals of the barcode by pairing the simplices in the reduced matrix. As mentioned before, the key feature of Eirene is that it adopts discrete Morse Theory to reduce the matrix size and uses the Schur complement to perform the matrix reduction efficiently.

## III. Performance-Improving Methods

Code analysis tools are important for programmers to understand program behavior. Software profiling measures the time and memory used during the execution of a program to gain this understanding and thus helps in optimizing the code. To develop efficient software, it is essential to identify the major bottlenecks and focus the optimization on the bottlenecks. Therefore, our strategy is to use Julia's built-in Profile module, a statistical profiler, to find the key bottlenecks in Eirene and then develop different performance-improving methods to solve them.

To profile an execution of Eirene, we simply need to put the macro `@profile` before calling the main function of Eirene, e.g.

```
@profile C = eirene(data-file-path,...)
```

Note that the Julia profiling tool works by periodically taking a backtrace during the execution of a program. Each backtrace takes a snpshot of the current state of execution, i.e. the current running function and line number along with the complete chain of function calls which led to this line. Therefore, a busy line of code, such as the code inside nested loops, has a higher opportunity to be sampled and hence appeared frequently in the set of all backtraces. However, profiling a very long-running task may cause the backtrace buffer to fill full. Programmers can use the configuration function `Profile.init(n, delay)` to either increase the total number of backtrace instruction pointers `n` (default: 10ˆ6) or the sampling interval `delay` (default: 0.001) or both.

We used three benchmarks: HIV, WorldMap, and Dragon2, to locate the bottlenecks in Eirene. The HIV benchmark contains the Hamming distances between 1088 different genomic sequence of the HIV virus. The WorldMap benchmark includes the data of 7322 cities in the world. The Dragon2 data set contains 2000 points sampled from the Stanford Dragon graphic. Both of the HIV and the Dragon2 benchmarks are available from [6] , while the WorldMap data can be obtained from [7].

After the major bottlenecks were identified, we figured out how each of the bottlenecks was formed and developed solutions to fix them, as described in detail below.

### *A. Dynamic-type Any*

Note that the Julia's sampling profiler only displays the results in textual format. To have a faster comprehension of the results, we used another package called ProfileView which can give users a graphical view of the data collected by the profiler. The function `ProfilevView.view()` plots a flame graph [18] which is a visual representation of the call graph of the code just being profiled. In such a graph, the vertical axis (from bottom to top) represents the stack of function calls, while the horizontal axis represents the number of backtraces being sampled at each line. To identify a potential bottleneck, users can move the mouse to a long bar usually located on the top two levels in the graph and the corresponding backtrace of the function name and the line number will be shown on the screen.

Figure 3 shows our first identification of a performance bottleneck located in the function `getstartweights_subr2()` using the HIV benchmark. It spans 90% of the horizontal axis and similar results can be found in the other two benchmarks. Examining the code in this function, we found that the use of the dynamic-type `Any` in the following three lines

```
val = Array(Any,m)
supp = Array(Any,m)
suppDown = Array(Any,m)
```

causes the Julia interpreter to generate many dynamic invoker objects when these three arrays appeared in an expression and hence the performance is greatly suffered. Fortunately, we also figured out the use of the dynamic-type `Any` is unnecessary in this function because the array `supp[]` stores the indices (i.e. of integer type) of non-zero elements returned from the Julia function `find()`. After replacing the dynamic-type `Any` with the static-type `Int64`, this bottleneck has been solved.

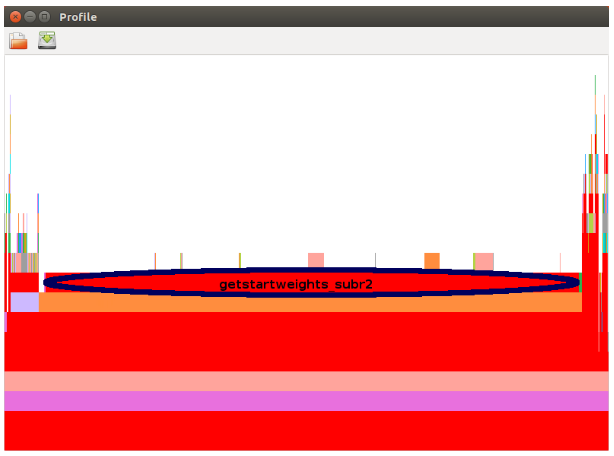


Fig. 3. Identification of the bottleneck caused by the Dynamic-type Any

## B. Redundant Calculations

We ran the code and performed the profiling procedure again after fixing the first large bottleneck. We noticed that there is a certain amount of time spent in the function `integersinsameorderbycolumn()` when running the Dragon2 benchmark, as shown in Figure 4 displayed by the ProfileView tool. After looking the original code of this function(see the top box in Figure 5), we found that there are some unnecessary calculations. The array `y` is used to calculate the prefix sums of the array `x` and only some of them will be copied to the array `z` to be returned to its caller. When the `maxvalue`, a parameter passed to this function, is large, the inner loop `for i = 1:maxvalue` will let this function be executed much longer. As shown in the bottom box in Figure 5, we modified the code to find out the range first and then calculate only the prefix sums within this range. Furthermore, we can just use the array `x` to accumulate the prefix sums of itself instead of using another local array `y`. The initialization of the whole array `x` inside the beginning of the loop can also be replaced by cleaning up the dirty elements before the end of the loop.

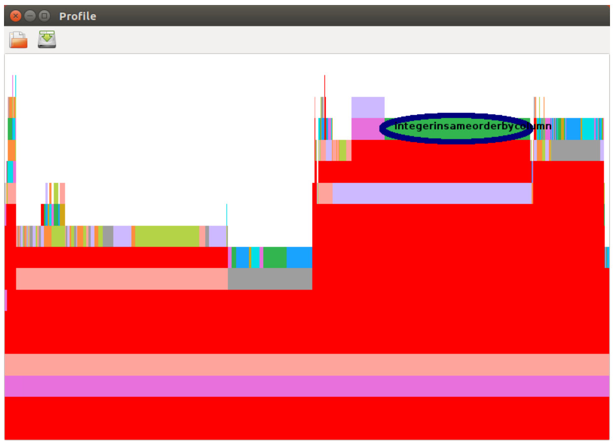


Fig. 4. Identification of the bottleneck in the function integersinsameorderbycolumn

```
        ...
        for j = 1:numcols
                x[:] = 0
                for i = colptr[j]:(colptr[j+1]-1)
                        x[v[i]]+=1
                end
                y[1] = colptr[j]
                for i = 1:maxvalue
                        y[i+1]=y[i]+x[i]
                end
                for i = colptr[j]:(colptr[j+1]-1)
                        u = v[i]
                        z[i] = y[u]
                        y[u]+=1
                end
         end
         return z
```

```
         ...
         x[:] = 0
         for j = 1:numcols
                 for i = colptr[j]:(colptr[j+1]-1)
                         x[v[i]]+=1
                 end

                 maxv = v[colptr[j]];    minv = maxv
                 for i = (colptr[j]+1):(colptr[j+1]-1)
                     if v[i] > maxv
                        maxv = v[i]
                     elseif v[i] < minv
                        minv = v[i]
                     end
                 end

                 prevsum = colptr[j]
                 for i = minv:maxv
                         sum = prevsum + x[i]
                         x[i] = prevsum
                         prevsum = sum
                 end
                 for i = colptr[j]:(colptr[j+1]-1)
                         u = v[i]
                         z[i] = x[u]
                         x[u]+=1
                 end

                 for i = minv:maxv
                         x[i] = 0
                 end
         end
         return z
```

Fig. 5. The original(top) and the modified(bottom) function integersinsameorderbycolumn() in Eirene

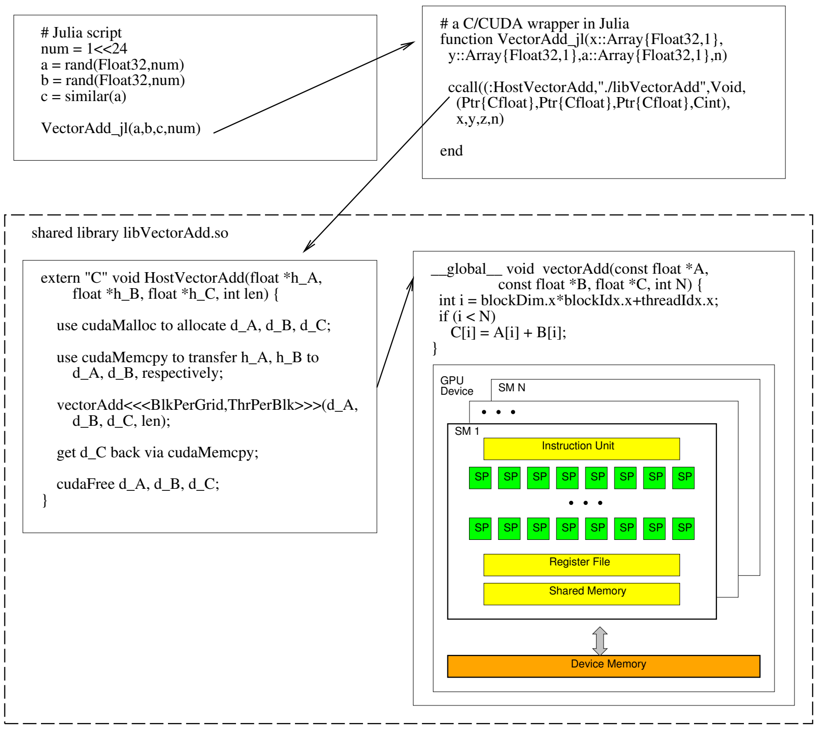
## C. Deeply Nested Loops

Even we removed the overhead caused by the dynamic-type `Any` in the function `getstartweights_subr2()`, the execution of the Dragon2 benchmark still spends a reasonable amount of time in this function. As shown in Figure 6, this function calculates the weight for each column in the matrix `s`. It firstly finds the indices of the non-zero elements in each column and then uses nested loops to increment the counter of the column if certain conditions are met. When there are many non-zero elements in the matrix `s`, the deeply nested loops will consume much CPU computation time.

To deal with the deeply nested loops, one feasible approach is to use the GPU to accelerate the execution. NVIDIA provides a parallel computing platform and programming model called CUDA (Compute Unified Device architecture). Therefore, now it is much more convenient to write application

```
function getstartweights_subr2(s::Array{Int64,2},
            w::Array{Int64,1},m::Int64)
   ...
   for i = 1:m
       supp[i] = find(s[:,i])
       l[i] = length(supp[i])
       ...
   end
   for i = 1:m
       Si = supp[i]
       ...
       for jp = 1:l[i]
           ...
           for ...
              if conditions met
                  w[i]+=1
              end
           end
       end
   end
end
```

Fig. 6. The original function getstartweights_subr2() in Eirene



```
# Julia script
num = 1<<24
a = rand(Float32,num)
b = rand(Float32,num)
c = similar(a)

VectorAdd_jl(a,b,c,num)
```

```
# a C/CUDA wrapper in Julia
function VectorAdd_jl(x::Array{Float32,1},
  y::Array{Float32,1},a::Array{Float32,1},n)

  ccall((:HostVectorAdd,"./libVectorAdd",Void,
    (Ptr{Cfloat},Ptr{Cfloat},Ptr{Cfloat},Cint),
    x,y,z,n)

end
```

shared library libVectorAdd.so

```
extern "C" void HostVectorAdd(float *h_A,
       float *h_B, float *h_C, int len) {

    use cudaMalloc to allocate d_A, d_B, d_C;

    use cudaMemcpy to transfer h_A, h_B to
       d_A, d_B, respectively;

    vectorAdd<<<BlkPerGrid,ThrPerBlk>>>(d_A,
       d_B, d_C, len);

    get d_C back via cudaMemcpy;

    cudaFree d_A, d_B, d_C;
}
```

```
__global__ void  vectorAdd(const float *A,
            const float *B, float *C, int N) {
int i = blockDim.x*blockIdx.x+threadIdx.x;
if (i < N)
   C[i] = A[i] + B[i];
}
```

Fig. 7. An example of calling C/CUDA functions from Julia script

programs on the GPUs for processing large amounts of data, without the need to use low-level assembly language code.

The NVIDIA GPU architecture consists of a scalable number of streaming multiprocessors (SMs), each containing many streaming processors (SPs) or cores to execute the lightweight threads. The kernel function, which is declared by using the `__global__` qualifier keyword in front of a function heading, is executed on the GPU device. It consists of a grid of threads and these threads are divided into a set of blocks and each block contains multiple warps of threads. Blocks are distributed evenly to the different SMs to run. A warp, which has 32 consecutive threads bundled together, is executed using the Single Instruction, Multiple Threads (SIMT) style. Note that the GPU device has its own off-chip device memory (i.e. global memory). and on-chip faster memory such as registers and shared memory. Fancy warp shuffle functions are also supported in modern GPUs [19]. They permit exchanging of variables (i.e. registers) between threads within a warp without using shared memory.

Though there are some Julia packages which enable programmers to launch GPU kernel calls, we decided to implement our own wrappers because of more flexibility and more efficiency. Figure 7 shows an example of calling a CUDA function named `vectorAdd()` by way of the host function `HostVectorAdd()` in C. A wrapper function in Julia is needed which utilizes the `ccall()` function to invoke the host function. Note that the first argument of `ccall()` is a tuple pair `(:function,"library-path")`. The rest of the arguments include the function return type, a tuple of input parameter types, and then the actual parameters. The C/CUDA functions should be compiled and linked as a shared objects. It is worth mentioning that, when using the `nvcc` NVIDIA CUDA Compiler to compile the C/CUDA code, programmers need to use the options `-Xcompiler -fPIC` to pass the position-independent code (PIC) option from `nvcc` to `g++`.

Figure 8 shows our implementation of the function `getstartweights_subr2()`. Our idea is to use `m` warps to handle the outermost `for i=1:m` loop in the original code. Since the second `for i=1:m` loop needs the result from the first loop, we need to use two kernel functions: `init_supp()` and `calcstartweights()`, and launch them one after the other in the host function `Host_getstarweights()`. To find the indices of the non-zero elements for each column, each thread within a warp check the corresponding element and cast its one-bit vote via the `__ballot()` intrinsic function. The `__ballot()` collects the votes from all threads in a warp into a 32-bit integer and returns this integer to every thread. The `__popc(int v)` function returns the number of bits which are set to 1 in the 32-bit integer v. That is, it performs the population count operation. By combining the `__ballot()` and `__popc()` functions along with bit-masking , each thread in a warp can quickly find out how many non-zero elements in front of it and then stores its index into the corresponding location in the array `supp[]`. This procedure will be repeated stride by stride until all elements in a column have been processed. Note that similar strategy has been used in [20] and [21] to perform efficient stream compactions on GPU.

After getting the indices of the non-zero elements for each column, each thread in the second kernel function `calcstartweights()` uses a local counter (i.e. a register) and increments this counter by one if certain conditions are met. When all non-zero elements have been checked, a parallel reduction sum operation via the efficient shuffle function `__shfl_down()` is performed. All of the local counter values in each warp will be added together and stored into the counter at the first thread (i.e lane ID 0). This thread then writes the weight to the output array `w`.

```
_global__ void init_supp(const long long *s,
            int *supp, int *l, ..., long long m) {
   int tid = blockIdx.x * blockDim.x + threadIdx.x;
   int lnid = threadIdx.x % WARP_SIZE ; // lane id
   int warp_id = tid >> 5; // global warp number
   if(warp_id >= m) return;

   int supplen = 0;
   int j = lnid;

   while (j < m ) {
      int b = s[warp_id*m + j] != 0 ;
      int votes = __ballot( b ); // cast b if non-zero
      int lidx = __popc( votes & ((1 << lnid) - 1)) ;

      if (b)
        supp[warp_id*m + supplen+lidx] = j;

      supplen += __popc(votes);
      j += WARP_SIZE; // next stride
   }
   ...
   if(lnid == 0) {
      l[warp_id] = supplen;
       ...
   }
}

_global__ void calcstartweights(const long long *s,
    int *supp, int *l, long long *w, long long m) {
 // Same as in init_supp, calc. tid , lnid, and warp_id ;
   if(warp_id >= m) return;

   int supplen = l[warp_id];
   int jp = lnid;
   int wt = 0; // each thread has a counter (in register)
   while (jp < supplen) {
      int j = supp[warp_id*m  + jp];
       ...
          for ... {
              if ( conditions met )
                wt += 1;
          }
      jp += WARP_SIZE; // next stride
   }
   // parallel reduction sum through registers
   for(int offset = WARP_SIZE>>1; offset>0; offset >>= 1)
       wt += __shfl_down(wt, offset);

   if (lnid == 0) w[warp_id] = wt;
}

extern "C" void Host_getstarweights(long long * s,
                 long long *h_w , long long m) {
   use cudaMalloc to allocate d_s, d_supp, d_l, d_w, etc.

   use cudaMemcpy to transfer data to d_s on GPU

   init_supp<<<BlkPerGrid, ThrPerBlk>>>(d_s, d_supp,
                 d_l,..., m);
   calcstartweights<<<BlkPerGrid, ThrPerBlk>>>(d_s, d_supp,
                 d_l, d_w, m);
   use cudaMemcpy to get the weights h_w from device d_w
}
```

Fig. 8. Implementation of the getstartweights_subr2() on GPU

## *D. Sortperm a Large Array*

Another bottleneck occurs in the function `ordercanonicalform()` when it calls Julia's `sortperm(v)` to find the rank of each element in the distance matrix. The `sortperm(v)` computes a permutation of the array v's indices that puts the array into sorted order. For example, if the input array `v` is

```
    v = [ 7, 3, 8, 4, 2 ] ,
```

then the output from `sortperm(v)` will be

```
    [ 5, 2, 4, 1, 3 ] .
```

When the size of the matrix is large, e.g. the WorldMap benchmark, `sortperm(v)` performs poorly. We found out that if we change the default sorting algorithm from MergeSort to RadixSort, the performance can be greatly improved, especially for the WorldMap benchmark.

Furthermore, CUDA Thrust is a powerful library [22] which provides a rich collection of data parallel primitives such as sort, scan, reduction, etc. Hence, using CUDA Thrust to build GPU applications, programming efforts can be reduced greatly. Though CUDA Thrust does not support the `sortperm`-like function directly, we can simply use the `sequence()` and `sort_by_key()` to implement it on GPU quickly, as shown in Figure 9.

```
extern "C" void
sortperm_thrust(double *h_s, long long *h_idx, long long n)
{
   thrust::device_ptr<long long> d_idx =
                 thrust::device_malloc<long long>(n);
   // create an array with elements 1, 2, 3, ..., n
   thrust::sequence(d_idx, d_idx + n, 1);

   thrust::device_ptr<double> d_s =
                 thrust::device_malloc<double>(n);

   thrust::copy(h_s, h_s+n, d_s); // copy s to device

   thrust::sort_by_key(d_s, d_s + n, d_idx);

   // we are interested in idx
   thrust::copy(d_idx, d_idx + n, h_idx);

   thrust::device_free(d_s);
   thrust::device_free(d_idx);
}
```

Fig. 9. Implementation of sortperm() on GPU

## *E. Sparse Matrix Multiplication and Addition*

For efficiently computing the matrix reduction, the Eirene library uses the Schur complement to encapsulate the LU factorization [9]. Given a matrix M with four sub-matrices A, B, C, and D in it, i.e.

$$M = \begin{bmatrix} A & B \\ C & D \end{bmatrix}$$

the Schur complement S of the block A of the matrix M is

$$S = D + CA^{-1}B$$

using the modulo-2 operation. Inside the Schur complement function `schurit4!()` in Eirene, the function blockprodsum() is invoked to compute $D + CE$ after getting $E = A^{-1}B$. Even Eirene uses sparse matrices for computing, we found that the function blockprodsum() becomes a bottleneck when the matrices are large.

To alleviate this problem, we decided to take the advantage of the multicore architecture and adopted the master/workers parallel computation model to speed up the execution. We partitioned the matrices D and E columnwise (due to the use of Compressed Sparse Column(CSC) format in Eirene) into several workers (i.e. pthreads) and let each worker $i$ compute $S_i = D_i + CE_i$, as illustrated in Figure 10. Unlike the dense matrix multiplication in which the product matrix is fully data parallel after partitioned, the index pointers in CSC which point to the starting locations of every column have to be adjusted one after the other. We let the master do the adjusting work when it copies the result to its caller.

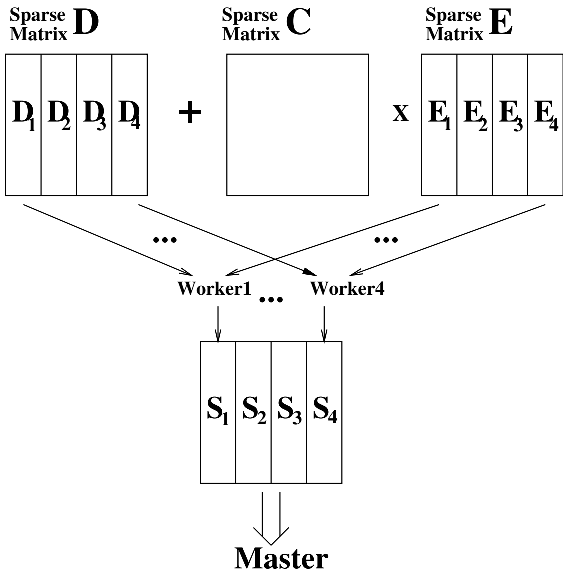


Fig. 10. Parallel Implementation of blockprodsum() using Master/Workers

## IV. Experimental Results

To evaluate the effects of the performance-improving methods discussed in the previous section, we ran the experiments with three different versions of Eirene: the original version which is Eirene v0.3.5 released in January 2017; the modified version which removes the dynamic-type Any, avoids unnecessary calculations, and uses RadixSort in sortperm(); the enhanced version which is a superset of the modified version and utilizes manycore/multicore to calculate the getstartweights_subr2(), sortperm(), and the blockprodsum() (using 4 workers).

We firstly adopted the workstation at the Ohio Supercomputing Center(OSC) to conduct the experiments. The machine has the Intel Xeon E5-2680 v4 CPU (2.4GHz), 28 cores per node, 128 GB of memory, as well as a cutting-edge NVIDIA Pascal P100 GPU(1.33GHz, 3584 CUDA cores, 16GB) running CUDA Driver Version 8.0. Tables I, II, III show the major bottlenecks' execution times and the total execution time for the HIV, WorldMap and Dragon2 benchmarks, respectively. In these three tables, we use A, B, C, D, and E to denote the major bottlenecks caused by dynamic-type "Any" , integersinsameorderbycolumn(), getstartweights_subr2(), sortperm(), and blockprodsum().

TABLE I
Execution Times (in seconds) of the Major Bottlenecks using the HIV Benchmark
(Intel Xeon E5-2680 v4 and NVIDIA Tesla P100 (Pascal))

| | Original | Modified | Enhanced |
|---|---|---|---|
| A | 95.7 | 0 | 0 |
| B | 0.078 | 0.002 | 0.002 |
| C | 3.2 | 3.2 | 0.068(Manycore) |
| D | 0.42 | 0.043(RadixSort) | 0.008(Manycore) |
| E | 0.135 | 0.135 | 0.134(Multicore) |
| Total | 110.8 | 13.1 | 9.9 |

TABLE II
Execution Times (in seconds) of the Major Bottlenecks using the WorldMap Benchmark
(Intel Xeon E5-2680 v4 and NVIDIA Tesla P100 (Pascal))

| | Original | Modified | Enhanced |
|---|---|---|---|
| A | 18.6 | 0 | 0 |
| B | 2.2 | 0.002 | 0.002 |
| C | 1.3 | 1.3 | 0.29(Manycore) |
| D | 38.8 | 4.1 (RadixSort) | 0.39(Manycore) |
| E | 1.18 | 1.18 | 0.90(Multicore) |
| Total | 80.6 | 23.4 | 17.1 |

TABLE III
Execution Times (in seconds) of the Major Bottlenecks using the Dragon2 Benchmark
(Intel Xeon E5-2680 v4 and NVIDIA Tesla P100 (Pascal))

| | Original | Modified | Enhanced |
|---|---|---|---|
| A | 416.6 | 0 | 0 |
| B | 36.2 | 0.05 | 0.05 |
| C | 15.4 | 15.4 | 0.72(Manycore) |
| D | 2.4 | 0.41(RadixSort) | 0.03(Manycore) |
| E | 15.5 | 15.5 | 9.6(Multicore) |
| Total | 565.2 | 109.9 | 89.0 |

It can be seen that the removal of unnecessary use of the dynamic-type Any can greatly improve the performance for all benchmarks. The other methods also have positive impacts on the performance for different benchmarks. Note that there is no significant improvement for the HIV and Worldmap benchmarks when using multicore to speed up the blockprodsum() due to the small amount of computation. Moreover, we used different number of threads to compute the blockprodsum() for the Dragon2 benchmark. As shown in Table IV, using a few more threads can still further improve the performance. For parallel dense matrix multiplication, usually linear or close to linear speedups can be obtained. The reason we cannot get linear speedups here is because the matrices are sparse and the workload is not fully balanced among the worker threads. Currently, a load balancing implementation of the `blockprodsum()` function is being developed.

TABLE IV
EXECUTION TIMES (IN SECONDS) OF THE BLOCKPRODSUM() IN DRAGON2 BENCHMARK WITH DIFFERENT NUMBER OF WORKERS)

| Num. of Workers | 1 | 2 | 4 | 6 | 8 |
|---|---|---|---|---|---|
| Time | 15.5 | 11.3 | 9.6 | 9.2 | 8.6 |

To show our methods which can also work well on different hardware platforms, we ran the benchmarks using the workstations in our laboratories. Table V shows the timing results on a workstation with Intel Core i7-4770K(3.50Hz, 32GB memory) and NVIDIA GeForce GTX 760(Kepler) GPU, while Table VI displays the results from a workstation with Intel Xeon CPU E3-1231(3.40GHz, 8GB memory) and NVIDIA Quadro K620 (Maxwell) GPU.i These machines have higher CPU clock rate but older GPU models than the OSC workstation.

TABLE V
EXECUTION TIMES (IN SECONDS) OF THE THREE BENCHMARKS USING INTEL CORE I7-4770K AND NVIDIA GEFORCE GTX 760 (KEPLER)

| | Original | Modified | Enhanced |
|---|---|---|---|
| HIV | 91.7 | 9.92 | 8.53 |
| WorldMap | 67.5 | 17.7 | 11.9 |
| Dragon2 | 473 | 84.3 | 72.9 |

TABLE VI
EXECUTION TIMES (IN SECONDS) OF THE THREE BENCHMARKS USING INTEL XEON CPU E3-1231 AND NVIDIA QUADRO K620 (MAXWELL)

| | Original | Modified | Enhanced |
|---|---|---|---|
| HIV | 108.4 | 13.0 | 11.1 |
| WorldMap | 74.9 | 23.1 | 17.9 |
| Dragon2 | 543 | 110.3 | 94.0 |

## V. CONCLUSION

We used the profiling tools in Julia to identify the bottlenecks in Eirene, a fancy open-source platform for computing persistent homology. Several performance-improving methods targeting the bottlenecks have been developed, such as removing unnecessary use of dynamic-type, eliminating redundant computation, use of manycore/multicore to accelerate execution, etc. Experimental results demonstrate that the performance can be greatly improved.

## ACKNOWLEDGMENT

This research was supported by the NASA GRC Summer Faculty Fellowship and by allocation of computing time from the Ohio Supercomputer Center.

## REFERENCES

[1] H. Edelsbrunner, D. Letscher, and A. Zomorodian, "Topological persistence and simplification," *Discrete & Computational Geometry*, vol. 28, no. 4, pp. 511–533, 2002.

[2] J. M. Chan, G. Carlsson, and R. Rabadan, "Topology of viral evolution," in *Proceedings of the National Academy of Sciences of the United States of America*, vol. 110, 2013.

[3] G. Carlsson, V. d. S. T. Ishkhanov, and A. Zomorodian, "On the local behavior of spaces of natural images," *International Journal of Computer Vision*, vol. 76, pp. 1–12, 2008.

[4] V. D. Silva and R. Ghrist, "Coverage in sensor networks via persistent homology," *Algebraic & Geometric Topology*, vol. 7, pp. 339–358, 2007.

[5] P. Pranav, H. Edelsbrunner, R. van de Weygaert, G. Vegter, M. Kerber, B. J. Jones, and M. Wintraecken, "The topology of the cosmic web in terms of persistent betti numbers," *Monthly Notices of the Royal Astronomical Society*, vol. 465, no. 4, pp. 4281–4310, 2016.

[6] N. Otter, M. Porter, U. Tillmann, P. Grindrod, and H. Harrington, "A roadmap for the computation of persistent homology," arXiv:1506.08903.

[7] G. Henselman, "Eirene: a platform for computational homological algebra," http://gregoryhenselman.org/eirene.html.

[8] "The Julia Language," http://https://julialang.org/.

[9] G. Henselman, "Matroids, filtrations, and applications," PhD thesis, University of Pennsylvania, 2016.

[10] S. L. Graham, P. B. Kessler, and M. K. McKusick, "An execution profiler for modular programs," *Software: Practice and Experience*, vol. 13, no. 8, pp. 671–685, August 1983.

[11] M. K. McKusick, "Using gprof to tune the 4.2bsd kernel," in *Proceedings of the European UNIX Users Group Meeting, Nijmegen, Netherlands.*, April 1984.

[12] B. Nichols, D. Buttlar, and J. P. Farrel, *Pthreads Programming*, 1st ed. Sebastopol, CA 95472: O'Reilly & Associates, Inc., 1996.

[13] D. B. Kirk and W.-m. W. Hwu, *Programming Massively Parallel Processors: A Hands-on Approach*, 3rd ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2016.

[14] R. Ghrist, "Barcodes: The persistent topology of data," *Bulletin of the American Mathematical Society*, vol. 45, pp. 61–75, 2007.

[15] A. Tausz, M. Vejdemo-Johansson, and H. Adams, "JavaPlex: A research software package for persistent (co)homology," in *Proceedings of ICMS 2014*, ser. Lecture Notes in Computer Science 8592, H. Hong and C. Yap, Eds., 2014, pp. 129–136.

[16] D. Morozov, "Dionysus," http://www.mrzv.org/software/dionysus/.

[17] V. Nanda, "erseus, the persistent homology software," http://www.sas.upenn.edu/∼vnanda/perseus.

[18] B. Gregg, "The flame graph," *Communications of the ACM*, vol. 59, no. 6, pp. 48–57, 2016.

[19] M. Harris, "CUDA Pro Tip: Do The Kepler Shuffle, PARALLEL FORALL," http://devblogs.nvidia.com/parallelforall/cuda-pro-tip-kepler-shuffle/, 2015.

[20] M. Harris and M. Garland, *Optimizing Parallel Prefix Operations for the Fermi Architecture*. San Francisco, CA, USA: Chapter 3 of the book "GPU Computing Gems - Jade Edition", Morgan Kaufmann Publishers Inc., 2011.

[21] V. Rego and J. Sang and C. Yu, "A Fast Hybrid Approach for Stream Compaction on GPUs," in *Proceedings of International Workshop on GPU Computing and Applications*, 2016.

[22] J. Hoberock and N. Bell, "Thrust: A parallel algorithms library which resembles the C++ Standard Template Library (STL)," http://thrust.github.io, 2015.