International Journal of Networking and Computing – www.ijnc.org
ISSN 2185-2839 (print) ISSN 2185-2847 (online)
Volume 7, Number 2, pages 208–226, July 2017

# Efficient Algorithms for Stream Compaction on GPUs

Darius Bakunas-Milanowski
IBM Corporation, 1111 Superior Avenue,
Cleveland, OH, USA

Vernon Rego
Dept. of Computer Science
Purdue University, West Lafayette, IN, USA

Janche Sang
Dept. of Electrical Engineering and Computer Science
Cleveland State University, Cleveland, OH, USA

and

Chansu Yu
Dept. of Electrical Engineering and Computer Science
Cleveland State University, Cleveland, OH, USA

Received: February 15, 2017
Revised: April 27, 2017
Accepted: June 2, 2017
Communicated by Yasuaki Ito

**Abstract**

Stream compaction, also known as stream filtering or selection, produces a smaller output array which contains the indices of the only wanted elements from the input array for further processing. With the tremendous amount of data elements to be filtered, the performance of selection is of great concern. Recently, modern Graphics Processing Units (GPUs) have been increasingly used to accelerate the execution of massively large, data parallel applications. In this paper, we designed and implemented two new algorithms for stream compaction on GPU. The first algorithm, which can preserve the relative order of the input elements, uses a multi-level prefix-sum approach. The second algorithm, which is non-order-preserving, is based the hybrid use of the prefix-sum and the atomics approaches. We compared their performance with other parallel selection algorithms on the current generation of NVIDIA GPUs. The experimental results show that both algorithms run faster than Thrust, an open-source parallel algorithms library. Furthermore, the hybrid method performs the best among all existing selection algorithms on GPU and can be two orders of magnitude faster than the sequential selection on CPU, especially when the data size is large.

*Keywords:* Stream Compaction, Parallel Selection, CUDA Thrust Library, GPU, SIMT

## 1 Introduction

Stream compaction is a common programming function that produces a smaller output array, containing the indices of the only wanted elements from the input array. So the further processing can benefit from the reduced data size and hence the performance can be improved. This primitive building block has been used in a wide range of applications, such as database, statistics, artificial intelligence, image processing, simulations, etc.[3][4][10]. With the tremendous amount of data elements to be processed, the performance of selection becomes an important factor in running these applications efficiently. Therefore, exploiting the availability and the power of multiprocessors to speed up the filtering process is of considerable interest.

Recently, modern Graphics Processing Units (GPUs) have been increasingly used to accelerate the execution of massively large, data parallel applications[9]. It is now much more convenient to create application software that will run on current GPUs for processing large amounts of data, without the need to write low-level assembly language code. NVIDIA provides a parallel computing platform and programming model called CUDA (Compute Unified Device architecture). Furthermore, there are a few accelerated, high performance libraries which enable an easy way of adding GPU-acceleration to the broad class of applications. For example, programmers can get even more flexibility and speed by writing their own GPU-accelerated programs using the CUDA Thrust Library[6], which provides a comprehensive development environment for C and C++ developers.

The NVIDIA Kepler GPU(Compute Capability 3.0) consists of a scalable number of streaming multiprocessors (SMXs), each containing 192 streaming processors (SPs) or cores to execute the light-weighted threads. The kernel function, which is executed on the device, is composed of a grid of threads. Note that a grid is divided into a set of blocks and each block contains multiple warps of threads. Blocks are distributed evenly to the different SMXs to run. A warp, which has 32 consecutive threads bundled together, is executed using the Single Instruction, Multiple Threads (SIMT) style (term coined by NVIDIA manufacturer). In addition to the main memory on the CPU motherboard, the GPU device has its own off-chip device memory (i.e. global memory). Furthermore, registers and shared memory in a SMX are on-chip memory and can be accessed very fast. They are per-block resources and are not released until all the threads in the block finish execution. Fancy warp shuffle functions are firstly supported in Kepler[11]. They permit exchanging of variables (i.e. registers) between threads within a warp without use of shared memory. Note that the NVIDIA's newer model Maxwell(Compute Capability 5.0) has similar architecture. The number of cores in each Maxwell streaming multiprocessor is reduced to 128 but with improved energy efficiency. It also provides dedicated shared memory and faster shared memory atomic operations [13].

In this paper, we focused on the design and implementation of two efficient stream compaction algorithms on GPU. The first algorithm uses a multi-level prefix-sum approach and hence the relative order of the input elements can be preserved. The second algorithm, which is non-order-preserving, is based the hybrid use of the prefix-sum and the atomics approaches. Both algorithms also adopt the new shuffle instructions to let threads within a warp have direct register-to-register data exchanges. We compared the performance of our algorithms with other parallel selection methods on CUDA-enabled GPUs. All tests were performed using CUDA Toolkit on a PC with a consumer grade NVIDIA Quadro K620 GPU and also on the Ohio Supercomputer Center's cluster Ruby, outfitted with professional NVIDIA Tesla K40 GPUs. The former belongs to the NVIDIA Maxwell, while the latter belongs to the NVIDIA Kepler architecture. The experimental results show that the multi-level prefix-sum based algorithm and the hybrid algorithm can be 3.7 times and 5.6 times, respectively, faster than the NVIDIA Thrust library. Furthermore, the hybrid method performs the best among all existing stream compaction algorithms on GPU and can be more than 120 times faster than the sequential selection on CPU.

The organization of this paper is as follows. Section 2 describes the background and the related work. Section 3 goes into details of our algorithms and in Section 4, the experiments and the results for performance evaluation are presented. We give a short conclusion in Section 5.

## 2 Background and Related Work

To support heterogeneous parallel computing, the CUDA programming model extends the C/C++ language by adding a set of keywords, predefined variables, and functions. For example, The qualifier keyword __host__ , which is used in front of a function declaration, declares a CPU function that is callable from the CPU only. Similarly, the qualifier __device__ declares a GPU function that is callable from the GPU only, and __global__ declares a kernel function which can only be launched from CPU and will be executed by all threads on the GPU device. As mentioned earlier, CUDA threads are organized in a two-level hierarchy, grid and block. The dimension of the grid and the dimension of each block are stored in the predefined variables gridDim and blockDim, respectively. The coordinates of a thread are stored in the variables blockIdx and threadIdx, where blockIdx is the block index in a grid and threadIdx is the thread index in a block. Hence, each thread can access these built-in variables to identify themselves and then determine the data area it has to work on.

CUDA toolkit also provides a variety of functions. Described below in detail are the functions which are relevant to our work.

- Intra-warp Voting: The __ballot(int p) intrinsic function returns a 32-bit integer in which bit $k$ is set if and only if the predicate p provided by the thread with lane ID $k$ is non-zero. Note that the lane ID is the thread's index within a warp, ranging from 0 to 31. In other words, the __ballot() function collects the predicates 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. Namely, it performs the population count operation. By combining the __ballot() and __popc() functions, all threads can quickly get the voting result, i.e., the number of true values within the warp.

- Atomic Operations: In CUDA, an atomic function enables a thread to perform a read-modify-write atomic operation without interference from other threads. CUDA supports a collection of functions which perform atomic operations, such as addition, subtraction, minimum, maximum, etc., in global or shared memory. For example, the function atomicAdd(&total, v) reads the value of the variable total, add the value with v, and store the result back to total. All these three operations are performed in an atomic way. The function returns the old value of total.

- Warp Shuffles: The traditional way threads exchange data with each other is through the shared memory. The new shuffle functions, which are available on the Kepler and later GPUs (Compute Capability 3.0 and above), allow threads within the same warp to read each ether's registers. Using the function __shfl(int v, int srcLane) as an example, the caller thread will get the value of the variable v held by the thread with lane ID srcLane. It behaves the same as broadcasting if every thread in the warp copies from the same source lane. For another example, the function __shfl_up(int v, int d) will let the lane $k$ thread read the variable v held by the lane $(k - d)$ thread.

The implementation of sequential stream compaction is straightforward. As shown in the code below, we can use a variable $k$ to keep track of the number of selected elements and store their indices into the corresponding positions indicated by $k$ in the target array.

```
int k = 0;
for(int i=0; i < DataSize; i++) {
  if( Source[i] meets the criteria )
      Target[k++] = i;
}
```

However, to implement the parallel stream compaction, the challenging is how to determine the positions (i.e. indices) of the selected elements in the destination array. In general, the approaches to performing the stream compaction on multiprocessors can be classified into two categories: one is based on the prefix-sum algorithm, while the other is based on the atomic operation. The former can let the output keep the relative order of the input elements, while the latter cannot guarantee the order will be preserved. Recent GPU-based parallel stream compaction approaches are described below in detail.

## 2.1 Prefix-Sum based approaches

As shown in Figure 1(a), this approach firstly initializes an array with 1 for selected elements (i.e. $s[i] <= 5$) and 0 for unwanted elements. Then, it calculates the prefix sums of the array in parallel[15]. One such implementation is adopted by the Thrust library, specifically a method called copy_if()[8], which is a fairly good implementation, simple to use and may prove to be the best choice for most GPU programmers needing this operation. By digging into its implementation details, we found that it uses 2-level sums. First, it calculates the number of selected elements within the block (using index counter inside the shared memory) and stores the result in an intermediate array of size N / block_size (N is the total number of input elements) in the global memory. Then it performs the parallel prefix sum on this array and uses the outcome in the final phase to determine the indices of output elements. As a result, the relative order of the input elements is also preserved. Furthermore, as shown in the later section, Thrust algorithm execution time does not depend on the number of passing elements.

Unlike the Thrust copy_if() which needs several kernel calls, the InK-Compact approach presented in [7] uses only one kernel to maintain the input-output data ordering. In this method, the data is divided into many sections where each section has 32 blocks. It computes the offset for a selected element by using: section_offset+ block_offset + warp_offset + intra_warp_offset. The intra_warp_offset can be determined by using the intra-warp voting functions __ballot() and __popc()[2][5], while the warp_offset can be computed by using the binary-bit scan operations[5]. It uses the function _syncthreads_count() to get the total number of wanted elements within the block and then writes the number to an intermediate block-count array. The block offset can be calculated by applying the prefix-sum operation on the block-count array. Each section has a controller block which uses the wait/signal mechanism to let the sections be processed in order and hence the section_offset can be obtained. To let the shared offsets be visible to all threads in the block, the functions __threadfence() and __syncthreads() are required.

Our new order-preserving algorithm adopts the multi-level prefix-sum approach. It used the intra-warp scan (i.e. using __ballot() and __popc() ) to find the offset of a selected element in a warp quickly. Each warp is responsible for handling a 1024-element sized group which contains 32 warp-sized subgroups. That is, each warp iterates 32 times and then uses prefix-sum to get the offset of each subgroup. The numbers of selected elements from each group are written into an intermediate array and then apply the prefix-sum again on these numbers to calculate the offset of each group.

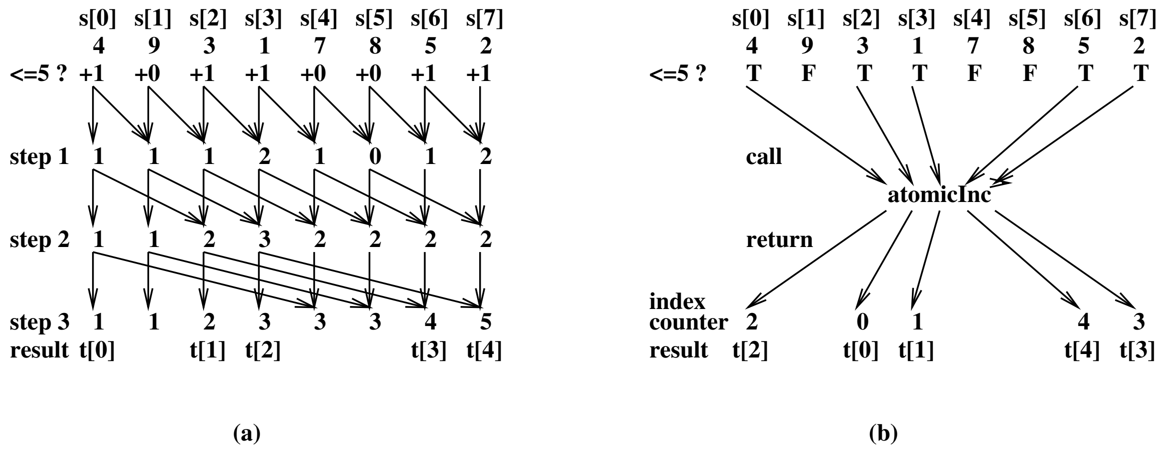

Figure 1: Parallel Selection using (a) Parallel Prefix-Sum (b) Atomic Operation

## 2.2 Atomic operation based approaches

As shown in the sequential code earlier, the index counter $k$ will be incremented by one for each newly selected element. For parallel implementation on GPUs, it's possible that there are many threads which need to update the counter $k$ simultaneously. To avoid the race condition, the increment to the variable $k$ has to be an mutually exclusive operation. This can be simply done by using CUDA atomicInc() function, as shown in Figure 1(b). Note that a CUDA atomic function performs a read-modify-write atomic operation on one 32-bit or 64-bit word residing in global or shared memory.

The main problem with this naive approach is that these atomic operations become a major bottleneck when the input contains a large amount of elements that pass the selection criteria. This is due to the very large number of threads competing to increment the single counter inside the global memory.

One possible improvement is to use shared memory atomics. This will essentially decrease the number of atomic collisions to a block size. Unfortunately, its performance still suffers from the thread synchronization. Same as the naive approach, the execution time is still directly proportional to the number of passing items[1].

The InK-Collate approach presented in [7] is a variation of the InK-Compact method without using sections. It uses the same way as in the InK-Compact to get the warp_offset and the intra_warp_offset. However, to obtain the block_offset, it simply invokes the atomicAdd() function. Therefore, the number of calling atomicAdd() can be greatly reduced as compared with the naive approach.

Another filtering approach is based on Warp-Aggregated Atomics[1]. It only computes the warp offset and the intra-warp offset. The advantage is that it does not need shared memory and hence avoids using memory fence and synchronization within a block. The voting functions __ballot() and __popc() are used to perform the intra-warp scan and to find out the number of wanted elements within a warp. This number will be added to the global counter via the atomicAdd() function to get the warp offset. The shuffle intrinsic[11] is used to broadcast the warp offset to all of the threads in a warp. Since each warp will issue at most one atomicAdd() request, its execution time is not be proportional to the number of passing elements and hence can be reduced greatly.

Our second algorithm is focused on the hybrid use of the ideas in our multi-level prefix-sum based algorithm and in [1]. A 1024-element sized group which contains 32 warp-sized subgroups will be processed by a warp. In addition to using the voting functions to get the intra-subgroup offset, the group offset is obtained via atomicAdd(), while the subgroup offset is calculated by the parallel prefix sum. It does not need shared memory nor synchronization within a block.

## 3 New Algorithms

This section presents our new algorithms for stream compaction on GPUs. One is order-preserving by using multi-level prefix-sum through three kernel calls, the other is non-order-preserving with the hybrid use of prefix-sum and atomics inside one kernel function. Note that stream compaction usually refers to stable filtering, i.e., the relative order of the input elements will be preserved in the final output. However, this ordering constraint is not necessary for some applications. Using the large-scale discrete-event simulations presented in [14] as an example, events are simply stored in an unsorted array. The execution of simulation needs to select parallel events based on the timestamps and then process these events. The order of the parallel events selected is not important because they are independent of each other. This procedure will be repeated until certain criterion is reached. Hence, the performance of the selection is a concern. This motivates us to design efficient stream compaction algorithms, as described in detail below.

### 3.1 A New Algorithm based on Multi-Level Prefix-Sum

The order-preserving algorithm implementation consists of three major phases. Each phase is required a kernel call. The phase 1 kernel starts by evaluating a predicate for each subgroup of 32 elements and saving the result into a predicate array (see Figure 2). The number of passing elements in the subgroup is also determined (using __popc() instruction) and saved inside the register variable cnt for each thread in the warp. This operation is performed for 32 iterations. As a result, each warp processes the total number of 1024 elements. When this loop completes, a parallel reduction operation is performed on the cnt register values, resulting in the number of selected elements for each 1024-element group (see Figure 3).

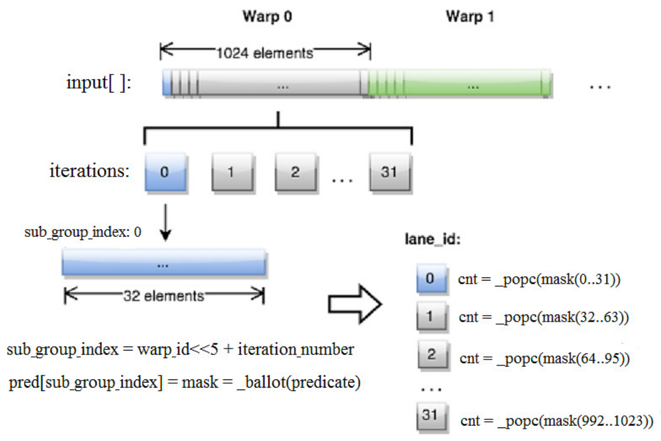
Warp 0
Warp 1
1024 elements
input[ ]:
...
iterations:
0
1
2
31
sub_group_index: 0
32 elements
lane_id:
cnt = _popc(mask(0..31))
cnt = _popc(mask(32..63))
cnt = _popc(mask(64..95))
cnt = _popc(mask(992..1023))
sub_group_index = warp_id<<5 + iteration_number
pred[sub_group_index] = mask = _ballot(predicate)

Figure 2: Phase 1a: fill predicate and counter arrays

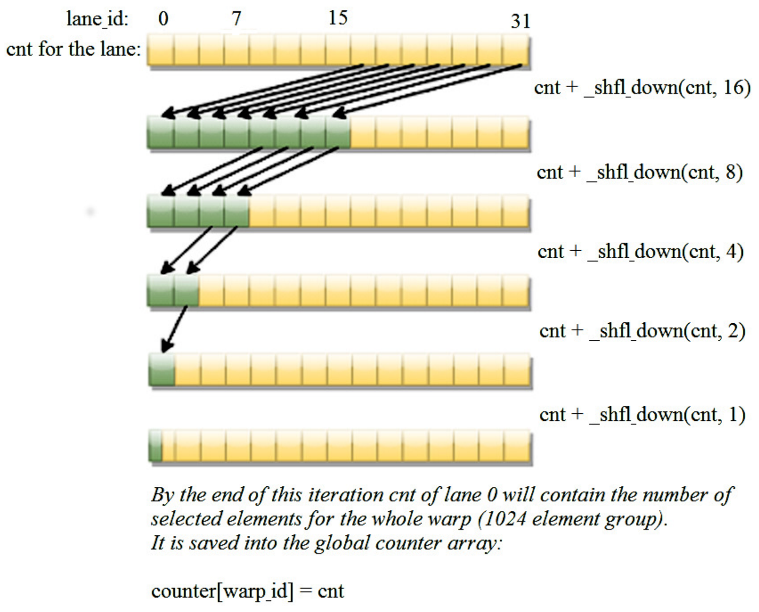
lane_id: 0 7 15 31
cnt for the lane:
cnt + _shfl_down(cnt, 16)
cnt + _shfl_down(cnt, 8)
cnt + _shfl_down(cnt, 4)
cnt + _shfl_down(cnt, 2)
cnt + _shfl_down(cnt, 1)
By the end of this iteration cnt of lane 0 will contain the number of selected elements for the whole warp (1024 element group).
It is saved into the global counter array:
counter[warp_id] = cnt

Figure 3: Phase 1b: perform reduction on cnt values

Note that __shfl_down() instruction was used, that essentially allows passing down register values from the lane with higher ID relative to the caller lane. The result is saved in the counter array. The detail of the code for the Phase 1 Kernel can be found in Figure 4.

```
#include <cuda.h>
#include "algorithms.h"
#include "err.h"

#define WARP_SZ 32

__device__ inline int lane_id(void) { return threadIdx.x % WARP_SZ; }

__global__ void parsel_kernel_phase1(float *input,
  unsigned int *counter,
  unsigned int *pred,
  float percent,
  const unsigned int num_items)
{
   int tid = blockIdx.x * blockDim.x + threadIdx.x;

   if (tid >= (num_items >> 5) )  // divide by 32
      return;

   int lnid = lane_id();
   int warp_id = tid >> 5; // global warp number

   unsigned int mask;
   int cnt;

   for(int i = 0; i < 32 ; i++) {
       mask = __ballot(input[(warp_id<<10)+(i<<5)+lnid] <= percent);

       if (lnid == 0)
           pred[(warp_id<<5)+i] = mask;

       if (lnid == i)
           cnt = __popc(mask);
   }
   // para reduction to a sum of 1024 elements
#pragma unroll
   for (int offset = 16 ; offset > 0; offset >>= 1)
       cnt += __shfl_down(cnt, offset);

   if (lnid == 0)
       counter[warp_id] = cnt; // store the sum of the group
}

```

Figure 4: Phase 1 kernel

In phase 2, a prefix sum operation is applied to the counter array (Figure 5). As a result, there are counter[k-1] valid elements before the group k. Note that for this operation we simply used Thrust implementation thrust::inclusive_scan(x), which was fast enough and sufficient for our purposes.

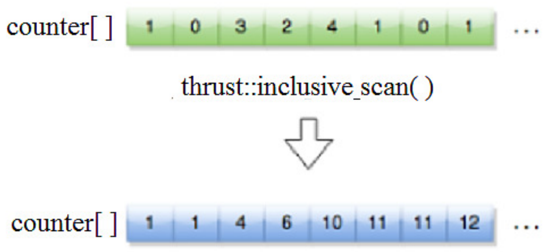

Figure 5: Phase 2: perform inclusive scan operation on the counter array

Phase 3 produces the final result. The process begins by reading the predicate array that was produced in the phase 1. The number of set bits is determined (using __popc() instruction) for each predicate value and saved into the cnt register for each lane inside the warp (Figure 6).

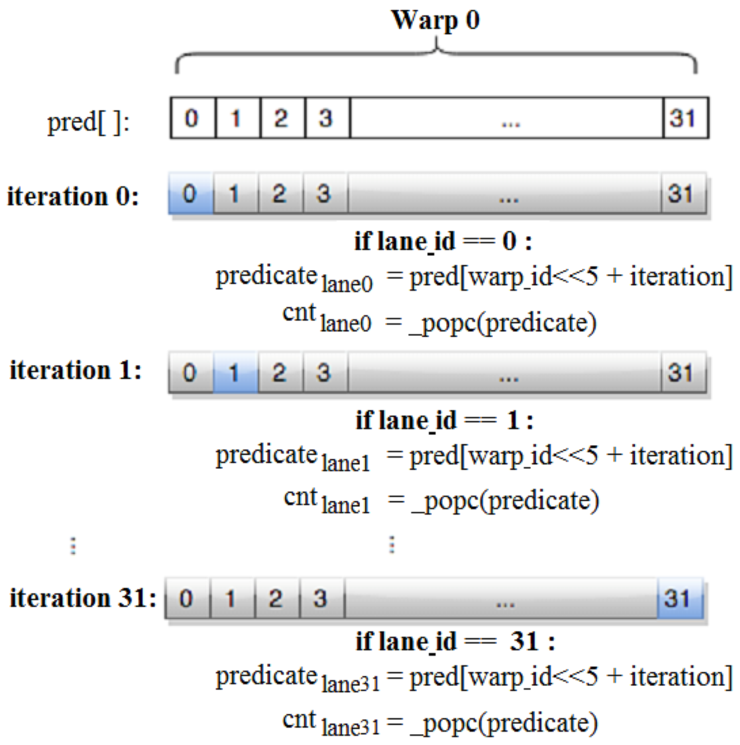
Warp 0
pred[ ]:
0 1 2 3 ... 31
iteration 0:
if lane_id == 0 :
predicate lane0 = pred[warp_id<<5 + iteration]
cnt lane0 = _popc(predicate)
iteration 1:
if lane_id == 1 :
predicate lane1 = pred[warp_id<<5 + iteration]
cnt lane1 = _popc(predicate)
iteration 31:
if lane_id == 31 :
predicate lane31 = pred[warp_id<<5 + iteration]
cnt lane31 = _popc(predicate)

Figure 6: Phase 3a: get bit counts for the predicate values

To calculate the subgroup index, the prefix sum operation is performed on these cnt values (Figure 7). After this operation, each cnt $i$ (for thread $i$) holds the number of valid elements before the $i$th subgroup. Note that for this operation, we used __shfl_up() instruction, which allows passing register values form the lane with lower ID relative to the caller lane.

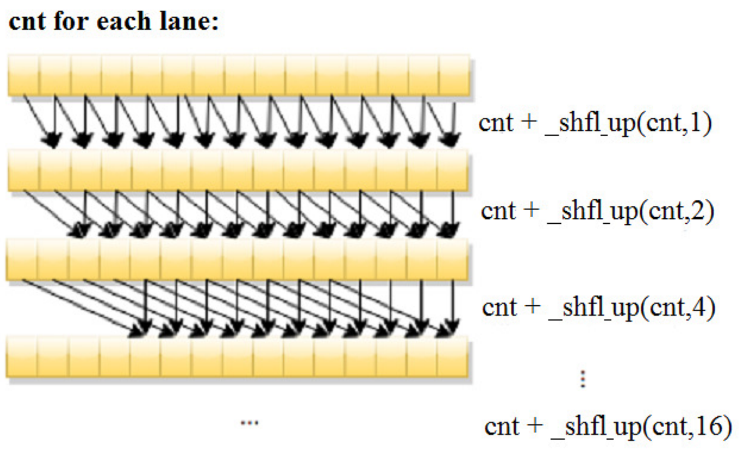

Figure 7: Phase 3b: perform prefix sum on cnt values

As a result, three indexes were produced – one for the 1024-element group (i.e. global_index), an index for each 32-element subgroup and since we have saved all predicate values – individual index within this 32- element subgroup can be calculated. This information essentially allows us to determine the indices of the valid elements in the destination array (Figure 8). Note that both kernels use grid configuration as follows: "dim3 grid((N / 32 + block.x - 1) / block.". The detail of the code for the Phase 3 Kernel can be found in Figure 9.

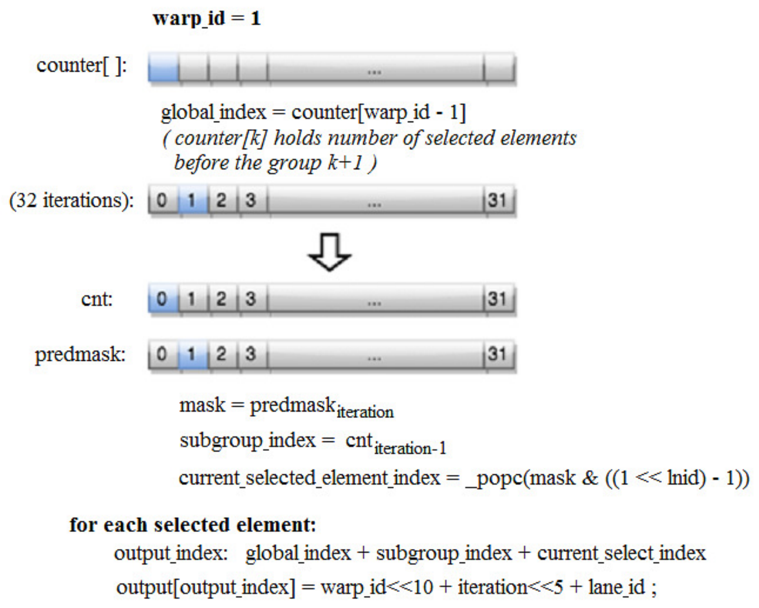
warp_id = 1
counter[ ]:
global_index = counter[warp_id - 1]
( counter[k] holds number of selected elements before the group k+1 )
(32 iterations):
0 1 2 3 ... 31
cnt:
predmask:
mask = predmask_iteration
subgroup_index = cnt_iteration-1
current_selected_element_index = _popc(mask & ((1 << lnid) - 1))
for each selected element:
output_index: global_index + subgroup_index + current_select_index
output[output_index] = warp_id<<10 + iteration<<5 + lane_id ;

Figure 8: Phase 3c: calculate the final indexes in the output array

```
/* PHASE3: produce final result array */
__global__ void parsel_kernel_phase3(int *output,
  unsigned int *counter,
  unsigned int *pred,
  const unsigned int num_items)
{
   int tid = blockIdx.x * blockDim.x + threadIdx.x;
   if (tid >= (num_items >> 5) )  // divide by 32
       return;

   int lnid = lane_id();
   int warp_id = tid >> 5; // global warp number

   unsigned int predmask;
   int cnt;

   for(int i = 0; i < 32 ; i++) {
       if (lnid == i) {
           // each thr take turns to load its local var (i.e regs)
           predmask = pred[(warp_id<<5)+i];
           cnt = __popc(predmask);
       }
   }
// parallel prefix sum

#pragma unroll
   for (int offset=1; offset<32; offset<<=1) {
       int n = __shfl_up(cnt, offset) ;
       if (lnid >= offset) cnt += n;
   }

   int global_index =0 ;
   if (warp_id > 0)
       global_index = counter[warp_id -1];

   for(int i = 0; i < 32 ; i++) {
       int mask = __shfl(predmask, i); // broadcast from thr i
       int subgroup_index = 0;
       if (i > 0)
           subgroup_index = __shfl(cnt, i-1); // broadcast from thr i-1 if i>0

       if (mask & (1 << lnid ) ) // each thr extracts its pred bit
           output[global_index + subgroup_index +
             __popc(mask & ((1 << lnid) - 1))] = (warp_id<<10)+ (i<<5) + lnid  ;
   }
}

```

Figure 9: Phase 3 Kernel

## 3.2 A New Hybrid Implementation

Our idea originated from comparing the performance of the InK-Collate and the Warp-Aggregated Atomics approaches. We found that the Warp-Aggregated approach performed faster than the InK-Collate method. Note that the InK-Collate method has fewer calls to the atomicAdd() functions, but it needs to synchronize threads within a block in order to access the shared memory. So we decided to pattern after the Warp-Aggregated Atomics approach to let the warp be the only computation unit and hence no need to call __syncthreads() or __threadfence(). We also found that if we simply apply the grid-stride loops optimizing technique to the Warp-Aggregated Atomics approach, the performance can be improved. This motivated us to let each warp handle a larger number of input elements.

As mentioned earlier, our algorithm divides the input elements into many 1024-element sized groups. Each group will be further divided into 32 subgroups and each subgroup has 32 elements. Furthermore, each group will be processed by a warp. As illustrated in Figure 10, there are four processing stages, all done within one kernel function. In the first stage, a warp will iterate 32 times to deal with the 32 subgroups one by one. Within each iteration, each thread in a warp reads an element from the input array and checks the element meets the selection criteria or not. The evaluated predicate (true or false) will be cast into the __ballot() function to get the whole subgroup voting result back. Same as other approaches using the intra-warp voting functions, the number of wanted elements can be obtained by calling the __popc() with the voting result as the parameter. The programming technique which we use here is that we save each subgroup's ballot result and the population count into two variables (i.e. registers) `votes` and `cnt`, respectively. These two variables are stored in a specific thread which depends on the iteration number. That is, for each iteration $i$, $0 <= i < 32$, the thread with the lane ID $i$ (i.e. thread ID % 32) will be the host of these two variables. The population count for each subgroup is saved because we will use them to calculate the subgroup offset in the next stage. The reason we also save the ballot result is because in the last stage we do not need to evaluate the predicates again.

At the end of the iterations, we will have 32 population count values which are stored in the variable `cnt` in each thread. Therefore, in the second stage we can apply the parallel prefix-sum operation on the `cnt` variables to obtain the offset for each subgroup. This can be done efficiently by using the CUDA Samples code [12] which uses the __shfl_up(val,i) instruction to pass a variable's value to the thread with $i$ higher lane ID than the caller thread. After the parallel prefix-sum scan, each `cnt` in the lane $i$ holds the number of valid elements among the first $i$ subgroups. Therefore, the `cnt` variable in the last lane has the total number of selected elements for the whole group.

In the third stage, the last lane thread will call the function atomicAdd() to add the `cnt` value to the global counter. The return value from atomicAdd() is the value of global counter before addition, which is the offset of the group. Then, this group offset can be broadcasted to all of the threads in a warp by using the __shfl() function.

We will write the indices of the selected elements to the output array in the last stage. Same as the first stage, we need to loop 32 times to process one subgroup at a time. During each iteration $i$, the ballot result, which we saved in the `votes` variable hosted in the lane $i$, will be broadcasted to every thread in the warp. So the intra-subgroup offset can be obtained by using bit-masking and __popc(). The subgroup offset calculated in the stage 2 can be also broadcasted from the thread with lane ID $i - 1$. Adding the group offset, the subgroup offset, and the intra-subgroup offset together, we can get the indices of the valid elements in the destination array.

## 4 Experimental Results

We compared our two new algorithms with the Thrust copy_if() method, the InK-Compact[7], the InK-Collate[7], and the method using the warp-aggregated atomics[1]. The experiments were conducted on one of the nodes in the Ruby cluster provided by the Ohio Supercomputer Center. The GPU used in this particular computing platform was the NVIDIA Kepler-based Tesla K40m, which contains 15 streaming multiprocessors(2880 CUDA cores in total), 12GB GDDR5 memory and 745 MHz GPU clock rate. For comparison purpose, we also ran the experiments on our instruction lab PC with the NVIDIA Maxwell-based Quadro K620, which contains 3 streaming multiprocessors(384 CUDA cores in total), 2GB DDR3 memory, and 1124 MHz GPU clock rate. Both of the GPU device programs use the CUDA driver/runtime version 7.5. The execution time we measured for each method refers to kernel execution time, excluding the data transfer time between host and device. In all experiments, we adopted uniform random distribution between [0,1] for our data source. We

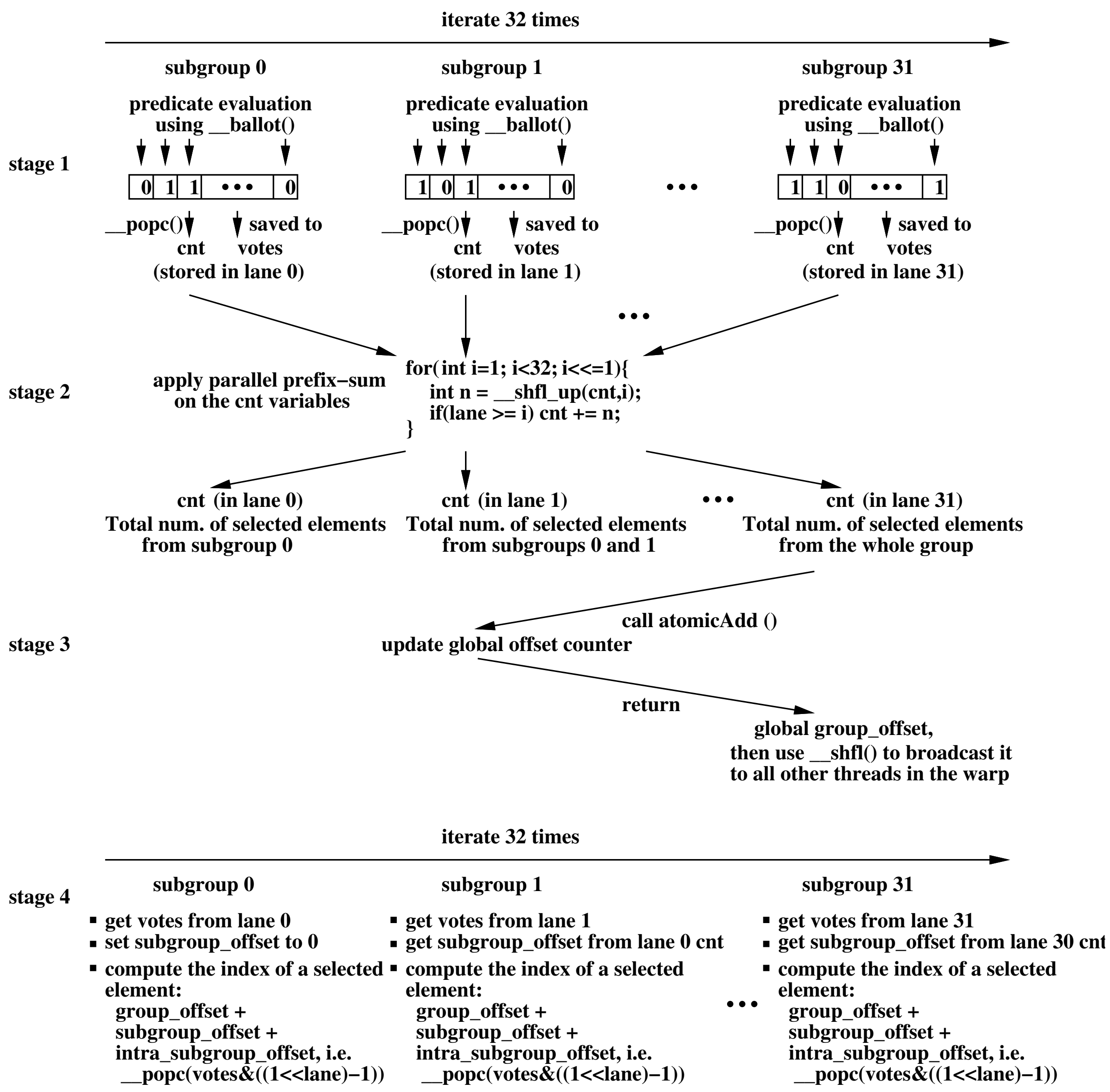
iterate 32 times
subgroup 0
subgroup 1
subgroup 31
predicate evaluation using __ballot()
stage 1
0 1 1 ••• 0
1 0 1 ••• 0
1 1 0 ••• 1
__popc() cnt (stored in lane 0)
saved to votes
__popc() cnt (stored in lane 1)
__popc() cnt (stored in lane 31)
stage 2
apply parallel prefix−sum on the cnt variables
for( int i=1; i<32; i<<=1){
int n = __shfl_up(cnt,i);
if(lane >= i) cnt += n;
}
cnt (in lane 0) Total num. of selected elements from subgroup 0
cnt (in lane 1) Total num. of selected elements from subgroups 0 and 1
cnt (in lane 31) Total num. of selected elements from the whole group
stage 3
call atomicAdd ()
update global offset counter
return
global group_offset, then use __shfl() to broadcast it to all other threads in the warp
iterate 32 times
stage 4
subgroup 0
▪ get votes from lane 0
▪ set subgroup_offset to 0
▪ compute the index of a selected element: group_offset + subgroup_offset + intra_subgroup_offset, i.e. __popc(votes&((1<<lane)−1))
subgroup 1
▪ get votes from lane 1
▪ get subgroup_offset from lane 0 cnt
▪ compute the index of a selected element: group_offset + subgroup_offset + intra_subgroup_offset, i.e. __popc(votes&((1<<lane)−1))
subgroup 31
▪ get votes from lane 31
▪ get subgroup_offset from lane 30 cnt
▪ compute the index of a selected element: group_offset + subgroup_offset + intra_subgroup_offset, i.e. __popc(votes&((1<<lane)−1))

Figure 10: The Hybrid Approach

used another input parameter $p$ to control the number of selected elements. That is, given the input data size $N$, there will be $(N * p)$ wanted elements in the final output. For example, if $p = 0.5$, half of the input elements will be selected.

In the first experiment, we measured the execution times for all of the algorithms by varying the number of threads per block (see Figure 11 and Figure 12), where input data size $N$ is set to 128M (M: one million). This experiment allowed us to select the optimal kernel configuration of the block size for our future experiments. We found that 128 threads per block work best on both devices for all of the algorithms except the InK-Compact method which runs the best with 512 threads per block. The execution times for the Thrust copy_if() remain the same because the Thrust library hides the details of kernel launch configurations. Also note that the using smaller block sizes, such as 32 or 64, causes the resources not to be fully utilized and also prevents creating sufficient parallelism to hide memory access latency. In particular, the InK-Compact and the InK-Collate methods perform much worse when using 32 as the block size. This is because the memory fence and the cross-warp synchronization used in these two methods become totally unnecessary for a one-warp sized block. Furthermore, the InK-Compact method adopts the busy waiting mechanism to synchronize the sections. A smaller block size will cause the InK-Compact method to have too many sections to handle. Another interesting thing is that the running times in Figure 12 are probably 5-6 times longer than the times in Figure 11. This is because the consumer grade Quadro K620 has much less number of cores than the Tesla K40. Usually, the overall performance of a GPU can be roughly estimated by using the number of cores times the GPU clock rate. So we can get the performance ratio of the two GPUs:

$TeslaK40/QuadroK620 = (2880 * 745)/(384 * 1124) \approx 5$ which is close to the experimental results.

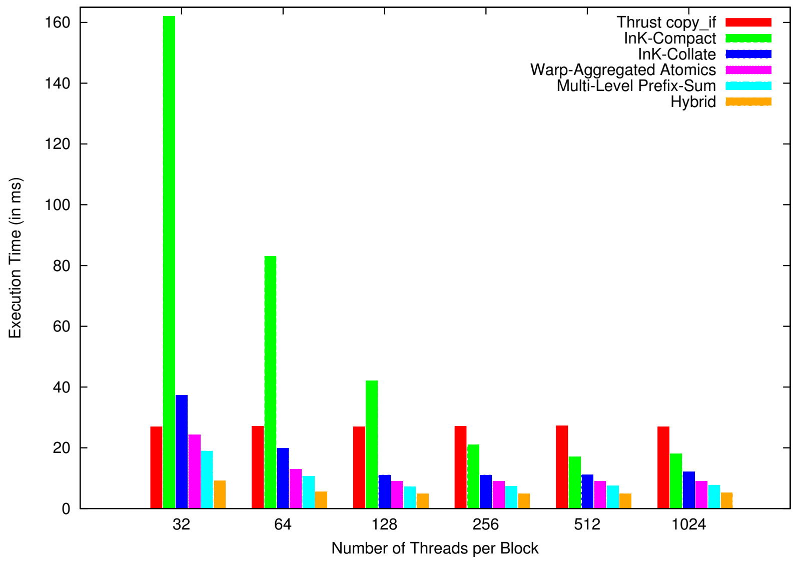

Figure 11: Performance metrics for various kernel block dimensions tested ($N$ = 128M, p=0.5, Tesla K40)

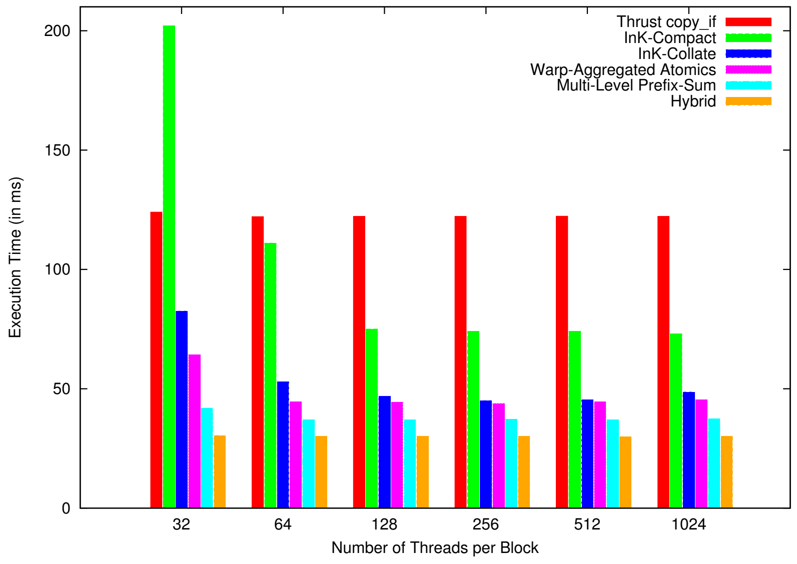

Figure 12: Performance metrics for various kernel block dimensions tested ($N$ = 128M, p=0.5, Quadro K620)

In the second experiment we measured the execution performance of these algorithms by varying the number of selected items, which is controlled by an input parameter $p$ (see Figure 13 and Figure 14). The block size is set to 512 threads for the InK-Compact while 128 threads for the rest methods. It can be seen that the running times of all methods do not depend on the selecting ratio $p$ (i.e. the number of valid elements) significantly. They are slightly increased when the ratio becomes higher due to the increased number of writes to the output array. The Warp-Aggregated Atomics approach can perform better when the selecting ratio is less than 5% because if there is no element selected within a warp, this warp does not need to call the function atomicAdd(). Among the six methods compared, the Thrust copy_if() perform the worst because it does not use the voting functions within a warp and has lots of intermediate data to write. The InK-Compact approach is the second slowest due to its use of the busy waiting operation and its use of shared memory among warps. Hence, memory fence and synchronization are needed within the block. These generate more overhead. The InK-Collate method also uses memory fence and synchronization and has the same problem. Our multi-level prefix-sum based algorithm and our hybrid algorithm can be 3.7 times and 5.6 times, respectively, faster than Thrust copy_if() on the high-end GPU Tesla K40. Hence, our methods suggest a feasible improved implementation for the future version of the Thrust copy_if(). Note that all of the InK-Collate, the Warp-Aggregated Atomics, and our hybrid approach use the atomicAdd() function, but our hybrid approach has fewer number of invocation of atomicAdd() as compared with the Warp-Aggregated Atomics method, and does not need block-level synchronization as compared with the InK-Collate approach.

To investigate more about the performance of our multi-level prefix-sum based algorithm, we measured the execution breakdown times for its three phases (i.e. kernel calls). As shown in Table 1 and in Table 2, each row presents a breakdown of execution time as well as the corresponding percentage for a particular input data size and a selecting ratio $p$. It can be seen that Phase 1 and Phase 3 are the biggest contributors to overall performance of the algorithm, which is also why we chose not to implement the inclusive scan operation for Phase 2 by ourselves. Furthermore, for a fixed input data size, when the selecting ratio $p$ increases, the Phase 3 execution time also increases, while Phase 1 and Phase 2 execution times remain the same. The reason is that there will be $(N * p)$-write operations occurred in Phase 3 and hence for a larger ratio $p$, more write operations to the output array are needed. This also justifies the result of the second experiment. In addition, the increased write operations also cause that for a fixed ratio $p$, the larger the input data size, the higher breakdown percentage for the Phase 3, especially on Tesla K40.

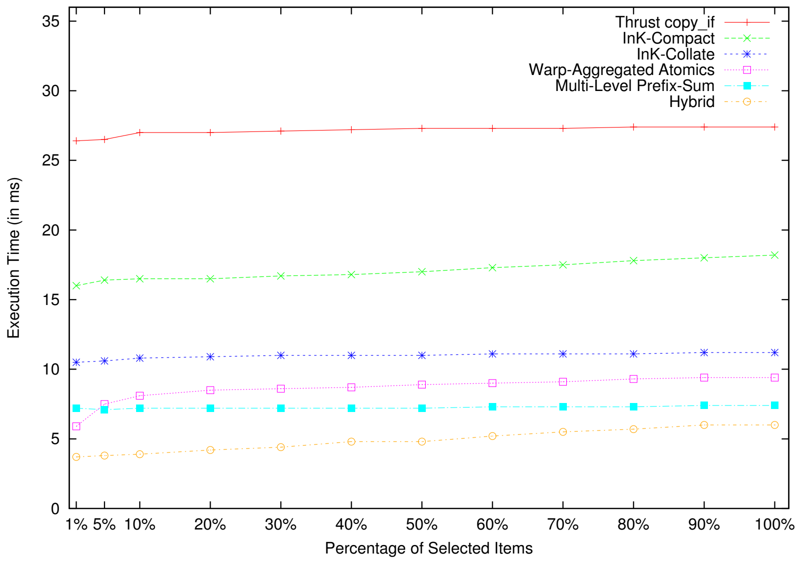

Figure 13: Performance comparison with different number of selected items ($N$ = 128M, Tesla K40)

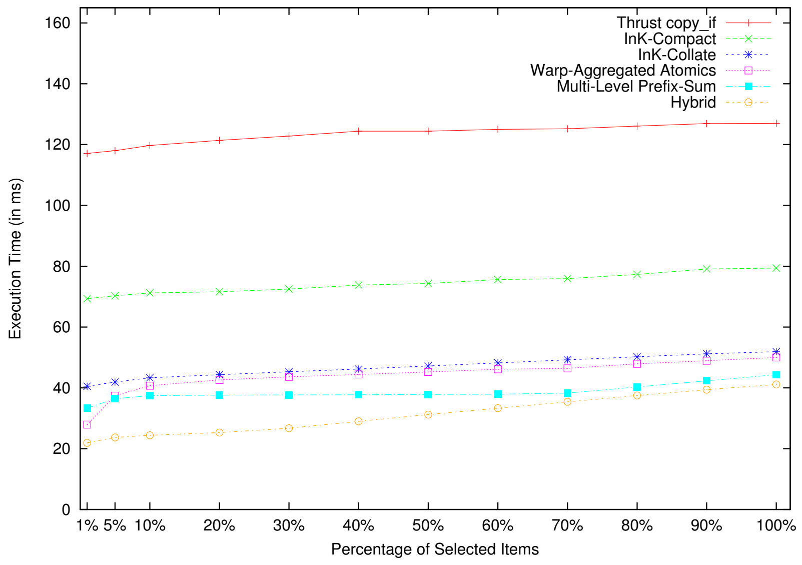

Figure 14: Performance comparison with different number of selected items ($N$ = 128M, Quadro K620)

Table 1: The Breakdown Exec Time (in ms) of the Multi-Level Prefix-Sum based Algorithm (Tesla K40)

| Data Size | p | Phase 1 time | Phase 2 time | Phase 3 time |
|---|---|---|---|---|
| 32M | 0.2 | 1.18 (51.8%) | 0.29 (12.8%) | 0.81 (35.4%) |
| | 0.5 | 1.18 (51.3%) | 0.29 (12.8%) | 0.83 (35.9%) |
| | 0.8 | 1.17 (50.7%) | 0.28 (12.3%) | 0.86 (37.0%) |
| 64M | 0.2 | 2.04 (52.0%) | 0.29 (7.5%) | 1.59 (40.5%) |
| | 0.5 | 2.03 (51.4%) | 0.29 (7.3%) | 1.63 (41.3%) |
| | 0.8 | 2.03 (50.7%) | 0.29 (7.2%) | 1.68 (42.1%) |
| 128M | 0.2 | 3.77 (52.2%) | 0.30 (4.1%) | 3.15 (43.7%) |
| | 0.5 | 3.76 (51.6%) | 0.30 (4.1%) | 3.23 (44.3%) |
| | 0.8 | 3.76 (50.9%) | 0.29 (3.9%) | 3.34 (45.2%) |

Table 2: The Breakdown Exec Time (in ms) of the Multi-Level Prefix-Sum based Algorithm (Quadro K620)

| Data Size | p | Phase 1 time | Phase 2 time | Phase 3 time |
|---|---|---|---|---|
| 32M | 0.2 | 5.43 (57.5%) | 0.21 (2.2%) | 3.81 (40.3%) |
| | 0.5 | 5.45 (57.5%) | 0.21 (2.2%) | 3.82 (40.3%) |
| | 0.8 | 5.45 (54.7%) | 0.21 (2.1%) | 4.32 (43.3%) |
| 64M | 0.2 | 10.71 (57.7%) | 0.22 (1.2%) | 7.62 (41.1%) |
| | 0.5 | 10.72 (57.1%) | 0.22 (1.2%) | 7.82 (41.7%) |
| | 0.8 | 10.83 (55.5%) | 0.22 (1.1%) | 8.49 (43.4%) |
| 128M | 0.2 | 21.29 (57.9%) | 0.24 (0.7%) | 15.24 (41.4%) |
| | 0.5 | 21.32 (57.5%) | 0.24 (0.7%) | 15.53 (41.8%) |
| | 0.8 | 21.26 (54.7%) | 0.24 (0.6%) | 17.34 (44.7%) |

In another experiment, we measured algorithm performance by varying the input size, while using the fixed block size and fixed selecting ratio. As shown in Figure 15 and Figure 16, the execution times of all tested algorithms were linearly proportional to the number of input elements. That is, when the input size $N$ is doubled, the running time is doubled. As mentioned before, the InK-Collate approach will issue one invocation of atomicAdd() for each block, the Warp-Aggregated Atomics method will call at most one atomicAdd() for each warp, while our approach will invoke atomicAdd() once for each group. More precisely, the number of invocations of the atomicAdd() function for each of the InK-Collate approach, Warp-Aggregated Atomics method, and our hybrid approach will be $(N/block\_size)$, $(N/32)$, and $(N/1024)$, respectively. Their execution times depend majorly on the input number of elements, not the number of selected elements.

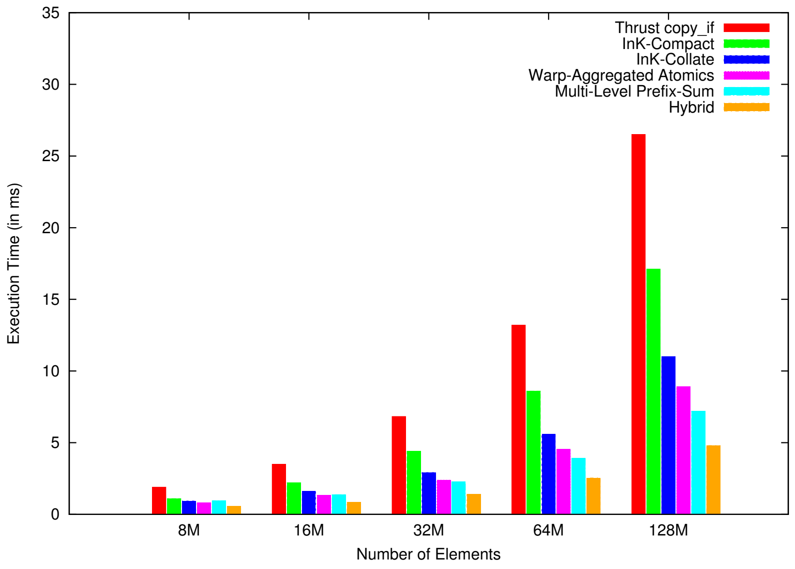

Figure 15: Performance comparison with different number of elements in the source array (p = 0.5, Tesla K40)

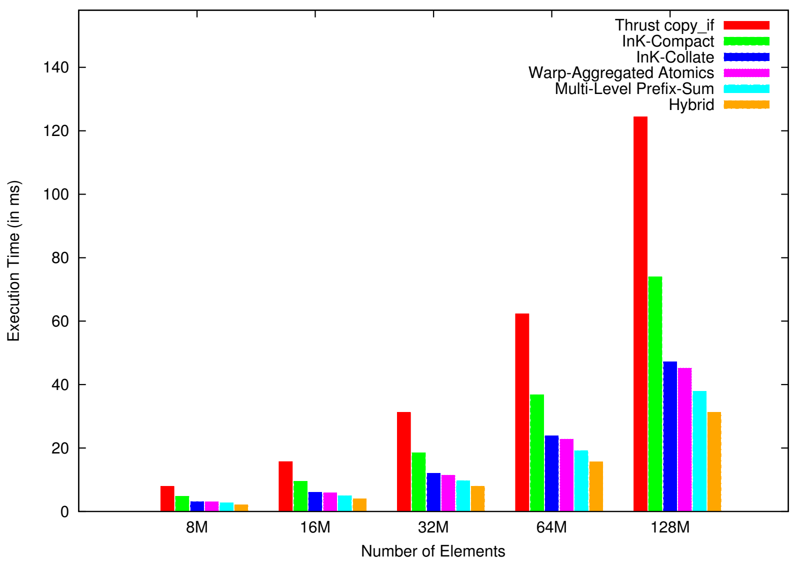

Figure 16: Performance comparison with different number of elements in the source array (p = 0.5, Quadro K620)

We also ran the sequential stream compaction on CPU. The host of the Tesla K40 has an Intel Xeon CPU E5-2670 with the clock rate 2.50GHz, while the host of the Quadro K620 uses the Intel Xeon CPU E3-1231 with the clock rate 3.40GHz. The speed-up ratios, which are the ratios of the sequential time to the parallel time, are shown in Figure 17 and Figure 18. It can be seen that for large data size ($N = 128M$ in our case), our hybrid approach can be more than 120 times faster than running sequentially on the CPU using the machine in the Ohio Supercomputer center. This implies that even if the 10 hardware cores in Intel Xeon CPU E5-2670 are used to speed up the CPU execution, which can be at best 10 times faster, GPU still can be a better choice for some large-scale, data-parallel applications which require stream compaction.

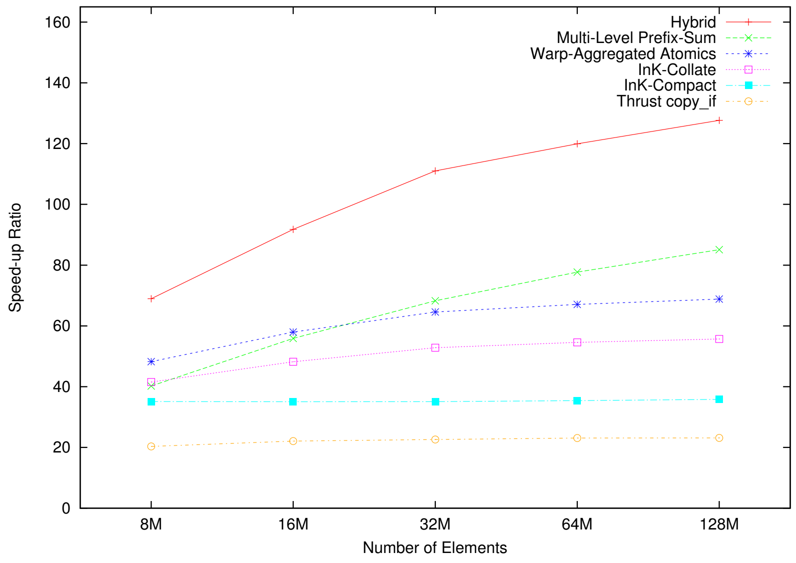

Figure 17: Speed Up (p = 0.5, Tesla K40)

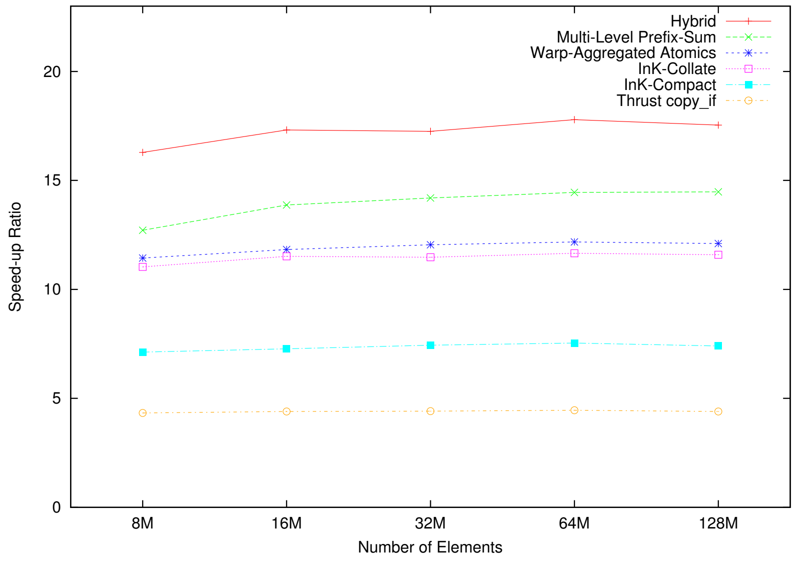

Figure 18: Speed Up (p = 0.5, Quadro K620)

## 5 Conclusion and Future Work

We presented two fast implementations of stream compaction on GPUs. The order-preserving one is based on the multi-level prefix-sum approach, while the non-order-preserving one is based on the hybrid use of the prefix-sum and the atomics operations. Both algorithms exploit the new warp shuffle functions on GPUs to let each warp process more input elements in a group and exchange data among threads efficiently. This results in increased computation-to-communication ratio and removal of the memory fence and synchronization overheads within a block. In addition, because of the larger group size, the multi-level prefix-sum algorithm has fewer number of reads/writes to the intermediate array, and the hybrid algorithm can also have less number of atomicAdd() invocation. The experimental results are encouraging, as both algorithms are able to achieve 3.7 and 5.6 times faster performance than the Thrust implementation. Furthermore, the hybrid algorithm outperforms the other stream compaction methods on GPUs and can be two orders of magnitude faster than the sequential selection on CPU, especially when the data size is large. In the future, we will integrate our fast stream compaction algorithms into the GPU-accelerated discrete-event simulation tool developed in [14]. We also plan to extend the current compaction method by performing one more dimensional parallel prefix-sum for a 32 times larger group and investigate any further performance improvement or not.

## Acknowledgments

This research was supported by Cleveland State University Undergraduate Summer Research Award Program and by allocation of computing time from the Ohio Supercomputer Center.

## References

[1] A. V. Adinetz. CUDA Pro Tip: Optimized Filtering With Warp-Aggregated Atomics, PARALLEL FORALL. `http://devblogs.nvidia.com/parallelforall/cuda-pro-tip-optimized-filtering-warp-aggregated-atomics/`, 2015.

[2] M. Billeter, O. Olsson, and U. Assarsson. Efficient Stream Compaction on Wide SIMD Many-Core Architectures. In *Proceedings of the Conference on High Performance Graphics*, 2009.

[3] G. Diamos, H. Wu, A. Lele, J. Wang, and S. Yalamanchili. Efficient relational algebra algorithms and data structures for GPU. Technical report, Technical Report GIT-CERCS-12-01, CERCS, Georgia Institute of Technology, 2012.

[4] K. Garanzha, S. Premoze, A. Bely, and V. Galaktionov. Grid-based SAH BVH construction on a GPU. *The Visual Computer*, 27(6-8):697–706, 2011.

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

[6] Jared Hoberock and Nathan Bell. Thrust: A parallel algorithms library which resembles the C++ Standard Template Library (STL). `http://thrust.github.io`, 2015.

[7] D. M. Hughes, I. S. Lim, M. Jones, A. Knoll, and B. Spencer. InK-Compact: In-Kernel Stream Compaction and Its Application to Multi-Kernel Data Visualization on General-Purpose GPUs. *Computer Graphics Forum*, 32(6):178–188, 2013.

[8] J. Hoberock and N. Bell. Stream Compaction. `https://thrust.github.io/doc/group__stream__compaction.html`, 2015.

[9] David B. Kirk and Wen-mei W. Hwu. *Programming Massively Parallel Processors: A Hands-on Approach.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2010.

[10] S. Lo, C. Lee, I. Chung, and Y. Chung. Optimizing Pairwise Box Intersection Checking on GPUs for Large-Scale Simulations. *ACM Trans. on Modeling and Computer Simulation*, 23(3):19:1–19:22, July 2013.

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

[12] Nvidia.com. CUDA Parallel Prefix Sum with Shuffle Intrinsics (SHFL_Scan). `http://docs.nvidia.com/cuda/cuda-samples/index.html`.

[13] Nvidia.com. NVIDIA Maxwell Architecture. `https://developer.nvidia.com/maxwell-compute-architecture`.

[14] J. Sang, C. Lee, V. Rego, and C. King. A Fast Implementation of Parallel Discrete-Event Simulation on GPGPU. In *Proceedings of International Conference on Parallel and Distributed Processing Techniques and Applications*, 2013.

[15] James C. Wyllie. The Complexity of Parallel Computations. Technical report, PhD thesis, Cornell University, Ithaca, NY, USA, 1979.