Accelerating All-Edge Common Neighbor Counting on Three Processors

Yulin Che
HKUST
Hong Kong, China
yche@cse.ust.hk

Zhuohang Lai
HKUST
Hong Kong, China
zlai@cse.ust.hk

Shixuan Sun
HKUST
Hong Kong, China
ssunah@cse.ust.hk

Qiong Luo
HKUST
Hong Kong, China
luo@cse.ust.hk

Yue Wang
HKUST
Hong Kong, China
ywangby@connect.ust.hk

ABSTRACT

We propose to accelerate an important but time-consuming operation in online graph analytics, which is the counting of common neighbors for each pair of adjacent vertices (u,v), or edge (u,v), on three modern processors of different architectures. We study two representative algorithms for this problem: (1) a merge-based pivot-skip algorithm (MPS) that intersects the two sets of neighbor vertices of each edge (u,v) to obtain the count; and (2) a bitmap-based algorithm (BMP), which dynamically constructs a bitmap index on the neighbor set of each vertex u, and for each neighbor v of u, looks up v's neighbors in u's bitmap. We parallelize and optimize both algorithms on a multicore CPU, an Intel Xeon Phi Knights Landing processor (KNL), and an NVIDIA GPU. Our experiments show that (1) Both the CPU and the GPU favor BMP whereas MPS wins on the KNL; (2) Across all datasets, the best performer is either MPS on the KNL or BMP on the GPU; and (3) Our optimized algorithms can complete the operation within tens of seconds on billion-edge Twitter graphs, enabling online analytics.

CCS CONCEPTS

• **Computing methodologies** → **Shared memory algorithms**.

KEYWORDS

All-Edge Common Neighbor Counting, Set Intersections, Hybrid Merge, Bitmap Index, Multi-core CPU, KNL, GPU

ACM Reference Format:
Yulin Che, Zhuohang Lai, Shixuan Sun, Qiong Luo, and Yue Wang. 2019. Accelerating All-Edge Common Neighbor Counting on Three Processors. In *48th International Conference on Parallel Processing (ICPP 2019), August 5–8, 2019, Kyoto, Japan.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3337821.3337917

ICPP 2019, August 5–8, 2019, Kyoto, Japan

ACM ISBN 978-1-4503-6295-5/19/08…$15.00
https://doi.org/10.1145/3337821.3337917

1 INTRODUCTION

Graphs model real-world relationships such as social networks and the world wide web. Given an undirected graph $G = (V, E)$, graph analytical systems often utilize the common neighbor count $|N(u) \cap N(v)|$ between adjacent vertices, where $(u, v) \in E$ and $N(u)$ is the neighbor set of u, for graph structural clustering and similarity queries [8, 9, 18, 19, 21, 25–27]. The results can be applied to advertising and epidemiology. For example, online platforms maintain graphs of user co-purchasing relations and analyze the data on the fly to recommend products of potential interest to the user while the user is shopping. Such applications require fast performance on big graphs; however, counting the common neighbors for all adjacent vertices, or all edges, is a time-consuming operation. Therefore, we study how to accelerate this operation on modern processors.

Given two vertices $u, v \in V$, we can compute the common neighbor count $cnt[(u, v)]$ by intersecting the neighbor sets of u and v. We consider two set intersection algorithms: (1) MPS: a merge-based algorithm on two sorted arrays [14, 15] with optimizations of finding pivots and skipping elements in the presence of skew; and (2) BMP: an index-based algorithm which builds a bitmap on one array, and loops over the other array to look up matches through the index.

Modern processors provide new opportunities for improving the efficiency of our algorithms. The AVX2 instruction sets on modern CPUs enable simultaneous execution of eight 32-bit integer operations with a single instruction; an Intel Knights Landing Processor (KNL) has 64 cores, two vector processing units (VPUs) per core, and an on-package 16GB high-bandwidth multi-channel DRAM (MCDRAM); the Nvidia GPU has tens of Streamming Multiprocessors (SMs), on which hundreds of thread blocks can be executed simultaneously. Therefore, there exists great potential for performance improvement, if we parallelize both MPS and BMP and optimize them to fully utilize the computation and memory resources on these three processors.

We parallelize both algorithms using a general OpenMP skeleton on the KNL and the CPU. The main idea is to group a fixed number of neighbor set intersections as a task and dynamically schedule these tasks. On the GPU, due to its fine-grained parallelism and hardware-managed scheduling, we treat the neighbor set intersections of a single vertex as a task and map it to a thread block using

CUDA (Compute Unified Device Architecture). Specifically, we parallelize MPS by launching two merge kernels respectively for the set intersections with and without the data skew, and parallelize BMP by launching a single kernel, given the differences in skew handling of these two algorithms.

We propose optimization techniques for the two algorithms. For MPS, we adopt the vectorization technique to utilize the vector execution units on the CPU and KNL; For BMP, we propose a range filtering technique to utilize the sparsity of matches in the set intersection and fit the auxiliary bitmap filter into the CPU cache or the GPU shared memory. Additionally, we perform hardware-specific optimizations on the KNL and GPU. On the KNL, we study the effects of utilizing MCDRAM in the cache and flat modes. On the GPU, we utilize the unified memory feature, and propose (1) CPU-GPU co-processing to overlap independent computations on the CPU and GPU; (2) multi-pass processing on the GPU to preserve the data locality and minimize page swaps caused by the on-demand load of the unified memory; and (3) tuning the number of warps per thread block.

We conduct extensive experiments on real-world big graphs to evaluate the effectiveness of individual techniques for MPS and BMP. Subsequently, we compare the two optimized parallel algorithms on each processor. We further analyze the performance differences among the best algorithms on three processors, and report our findings. We find that: (1) MPS works best on the KNL, because of 128 VPUs and the high-bandwidth memory MCDRAM; (2) CPU favors BMP, because its L3 cache reduces the memory access latency; (3) GPU also favors BMP, due to the warp-level parallelism; and (4) the best performance is achieved by either MPS on the KNL or BMP on the GPU, and the worst by MPS on the GPU, due to the match between algorithms, hardware and datasets. We show that our optimized algorithms complete the all-edge common neighbor counting in 21.5 seconds on the 680 million edge twitter graph (BMP on the GPU) , and 34 seconds on the 1.8 billion edge friendster graph (MPS on the KNL) .

2 BACKGROUND AND RELATED WORK

In this section, we give the problem statement of all-edge common neighbor counting, discuss related work on set intersection and exact triangle counting algorithms.

2.1 Preliminaries

In this paper, we focus on an undirected graph $G = (V, E)$, where V is a set of vertices and E is a set of edges. Given a vertex $u \in V$, $N(u)$ is the set of neighbors of u, and d_u denotes the degree of u (i.e., $d_u = |N(u)|$). $ID(u)$ is the ID of vertex u. All vertex IDs in G are unique 32-bit unsigned integers in $[0, |V|)$. $N^+(u)$ is the neighbors of u whose IDs are greater than $ID(u)$.

Common Neighbor Count. Given an undirected graph $G = (V, E)$ and vertices $u, v \in V$, the common neighbors of u and v are the set of vertices $\{w | w \in N(u) \wedge w \in N(v)\}$. The common neighbor count of u and v, denoted as $cnt[(u, v)]$, is the number of their common neighbors. The common neighbor count of a pair of adjacent vertices is an important local coefficient for graph analytics, for example, the graph structural clustering and the similarity queries of vertex pairs [8, 9, 18, 19, 21, 25–27]. The problem addressed in this paper is as follows.

Problem Statement. Given a graph $G = (V, E)$, the all-edge common neighbor counting is to compute the common neighbor count $cnt[(u, v)]$ for each edge $(u, v) \in E$.

Degree-Descending Graph Ordering. In BMP, to lower the time complexity of each set intersection than $O(d_v)$, we add a degree-descending reordering of vertices, to remap the vertex IDs in an edge list. The reordering takes the time complexity $O(|V| \log |V| + |E|)$ - $O(|V| \log |V|)$ for sorting and $O(|E|)$ for vertex re-mapping for all edges. It took less than 3 seconds on the billion-edge twitter graphs in our experimental setting. The degree-descending reordering ensures $\forall_{u,v \in V}(u < v \rightarrow d_u > d_v)$, which enables us to construct bitmaps for the larger degree vertex and loop over the neighbor set of the smaller degree vertex in each intersection.

Storage Format. Given a graph $G = (V, E)$, whose original storage format is a list of edges, we preprocess G to convert it to the *compressed sparse row* (CSR) format, which is widely used in graph algorithms [8, 9, 22, 25]. A CSR consists of an offset array and a neighbor array, denoted as off and dst respectively. The neighbor array stores v, the destination of each edge $(u, v) \in E$. For brevity, we denote an edge offset in the CSR as $e(u, v)$, where $dst[e(u, v)] = v$. The range $[off[u], off[u + 1])$ is the range of offsets of u's neighbors, which we call an offset range, i.e., $e(u, v) \in [off[u], off[u + 1])$. The neighbor set of vertex u is denoted by $dst[off[u] : off[u + 1])$, which is sorted in the ascending order. Without causing confusion, we use u for both vertex and vertex id, $(u, v) \in E$ for an edge, $e(u, v)$ for the edge offset of (u, v), $cnt[e(u, v)]$ for the common neighbor count of two adjacent vertices u and v.

2.2 Related Work

Given an edge $(u, v) \in E$, we can compute the common neighbor count $cnt[e(u, v)]$ with a set intersection between $N(u)$ and $N(v)$, i.e., $cnt[e(u, v)] = |N(u) \cap N(v)|$. Also, counting the common neighbors for a single edge is essentially counting the number of triangles based on the edge.

2.2.1 Set Intersection. The set intersection algorithms can be categorized into three types: (1) merge-based ones on sorted arrays, (2) bitmap-based ones, and (3) index-based nested loop. The *merge-based* algorithms require both sets to be represented in sorted arrays. They scan the two arrays and compare the elements to find matches. The data parallelism can be exploited for the algorithms via the vectorization technique [14, 15, 24]. When the sizes of the two sets are significantly different, search algorithms [2–4, 10, 11, 15, 24] are introduced to improve the efficiency of merge. Specifically, the merge is done by iteratively fixing a pivot in one array and skipping in the other array quickly to find a possible match via an efficient search algorithm. Recently, a *sparse bitmap* is proposed to represent a neighbor set, which consists of offset and bit-state arrays [1, 13, 16]. An intersection of two sparse bitmaps is done by merging and filtering on the offset arrays and intersecting the underlying bit-states associated with the offsets when an offset match occurs. However, to make bit-states compact, graph reordering is required, which is time-consuming and is performed offline. In comparison, *index-based* methods invest both time and memory resources to build auxiliary data structures such as hash tables and skip lists

Algorithm 1: MPS

```
   Input: a graph G = (V, E) and a tunable threshold t
   Output: the all-edge comon neighbor counts cnt
1  foreach (u, v) ∈ E and u < v do
2      if d_u / d_v ≤ t and d_v / d_u ≤ t then
3          cnt[e(u, v)] ← IntersectM(N(u), N(v))
4      else cnt[e(u, v)] ← IntersectPS(N(u), N(v))
5      cnt[e(v, u)] ← cnt[e(u, v)]
6  Procedure IntersectM(A_1, A_2)
7      c ← 0, off_1 ← 0, off_2 ← 0, end_1 ← |A_1|, end_2 ← |A_2|
8      while off_1 < end_1 and off_2 < end_2 do
9          if A_1[off_1] < A_2[off_2] then off_1 ← off_1 + 1
10         else if A_1[off_1] > A_2[off_2] then off_2 ← off_2 + 1
11         else off_1 ← off_1 + 1, off_2 ← off_2 + 1, c ← c + 1
12     return c
13 Procedure IntersectPS(A_1, A_2)
14     c ← 0, off_1 ← 0, off_2 ← 0, end_1 ← |A_1|, end_2 ← |A_2|
15     while true do
16         off_1 ← LowerBound(A_1 + off_1, A_1 + end_1, A_2[off_2])
17         if off_1 ≥ end_1 then return c
18         off_2 ← LowerBound(A_2 + off_2, A_2 + end_2, A_1[off_1])
19         if off_2 ≥ end_2 then return c
20         if A_1[off_1] == A_2[off_2] then
21             off_1 ← off_1 + 1, off_2 ← off_2 + 1, c ← c + 1
22             if off_1 ≥ end_1 or off_2 ≥ end_2 then return c
```

[5, 12, 20]. Once the indices are built, an indexed nested loop join is performed to iterate over one set and look up the index to find matches in the other set.

We extract representative elements from existing set intersection algorithms, especially those on graph analytics, and design the two algorithms MPS and BMP. Our focus is to study the parallelization and optimization of these algorithms on modern processors.

2.2.2 Exact Triangle Counting. If we sum up the all-edge common neighbor counts in a graph and divide the sum by six, we get the total number of triangles in the graph. However, the triangle counting differs from our problem in two aspects. Firstly, with an order constraint $u < v < w$ and the symmetric breaking technique, for an adjacent vertex pair (u, v), only an intersection of $N^+(u)$ and $N^+(v)$ is required in the triangle counting [23]; whereas our problem requires to compute the intersection of full neighbor sets $N(u)$ and $N(v)$. Secondly, the triangle counting does not require to store edge associative values; whereas our problem requires to store the $|E|$ counts. Merge-based and hash-index-based algorithms are designed for the triangle counting on multi-core CPUs [23].

Our work differs from the state-of-the-art parallel triangle counting work on multi-core CPUs [23] on three aspects: (1) *handling degree-skew cases* for real-world graphs in our merge-based algorithm; (2) *dynamically building an index* to save the memory cost of auxiliary structures in our index-based algorithm; and (3) parallelizing, optimizing and evaluating both algorithms on *three heterogeneous modern processors*.

3 METHODOLOGY

In this section, we describe the design of MPS and BMP, both of which utilize a symmetric assignment technique. Since common neighbor counts $|N(u) \cap N(v)|$ and $|N(v) \cap N(u)|$ are identical, we can reduce the workload by computing the intersections only for the edges (u, v) satisfying $u < v$ and symmetrically assigning $cnt[e(u, v)]$ to $cnt[e(v, u)]$. The edge offset $e(v, u)$ can be found via a binary search of the value u on the sorted array of $N(v)$.

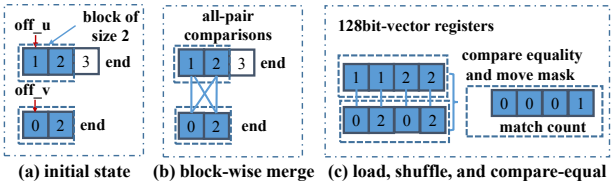


Figure 1: An example of VB

3.1 MPS

MPS (Algorithm 1) iterates through all the edges (u, v) with a condition of $u < v$, and utilizes the symmetric assignment technique to avoid unnecessary intersections (Line 5). Merging two sorted arrays and counting the matches are basic components of MPS. We describe a vectorized block-wise merge (VB), suitable for no degree-skew cases; and a pivot-skip-based merge (PS), suitable for degree-skewed cases.

Vectorized Block-wise Merge (VB). To exploit the data parallelism of merge, a vectorized block-wise merge [14] (Figure 1) was proposed, which conducts all-pair comparisons for the counting and advances a block of elements each time. There are three steps of the block-wise all-pair comparisons. Firstly, sorted integer blocks are loaded and shuffled within the vector register. Secondly, all elements are compared pair-wise simultaneously, the results of which are stored in masks. Thirdly, the counts of matches from the masks are added to a vector register that stores the intermediate match counts. After the block-wise all-pair comparisons, the last element in each block is compared, and the offset of the block with the smaller last element is advanced at a block-size step. The final match count result can be computed from the sum of integers in the vector register of match counts.

Pivot-Skip-based Merge (PS). On twitter graphs, VB does not work well for degree-skewed cases. In order to skip quickly for a possible match, we design a PS. The key idea of PS is to fix a pivot in one of the two sorted arrays, and then in the other array skip the offset directly to the lower bound of the sorted elements not less than the pivot value, to find a possible match. Each merge (Lines 13-22) consists of iterative computations in three steps: (1) given the pivot $A_2[off_2]$, find the lower bound in the A_1 not less than the pivot to advance off_1 for a possible match; (2) similarly advance off_2 on the other array; and (3) increment the match count and advance both offsets by one, if a match is found.

To implement an efficient lower bound algorithm, we apply two techniques: vectorized linear search [9] and galloping-search [2–4, 11, 15]. First, we utilize the vectorized instruction set to conduct a linear search of the target pivot value. If the linear search fails to find the lower bound, we continue with a galloping-search to skip the offset at exponentially increasing sizes $2^4, 2^5, ..., 2^i, 2^{i+1}$, until we find an offset pointing to a value greater than the pivot. After these exponential skips, a binary-search for the pivot value in the relative offset range $[2^i, 2^{i+1})$ finally locates the lower bound.

Time Complexity of PS. Let d_s and d_l denote the degrees of vertices v and u ($d_s \leq d_l$), $s[i]$ denotes the skip size in the u's neighbor set. In each skip step, the galloping skips and the binary search both take the $O(log(s[i]))$ time complexity, and there are at most $2 \cdot d_s$ iterations to advance the offset of v's neighbor set to the end. We sum up the cost of each skip step up, and get the time complexity $O(\Sigma_{i \in [0, 2d_s)} \log(s[i]) + d_s)$. In practice, the average

Algorithm 2: BMP

```
   Input: a graph G = (V, E)
   Output: the all-edge comon neighbor counts cnt
1  foreach u ∈ [0, |V|) do
2      B ← a bitmap of cardinality |V|
3      foreach v ∈ N(u) do
4          Set v's bit in bitmap B
5      foreach v ∈ N(u) and u < v do
6          cnt[e(u, v)] ← IntersectBMP(B, N(v))
7          cnt[e(v, u)] ← cnt[e(u, v)]
8      foreach v ∈ N(u) do
9          Flip v's bit in bitmap B
10 Procedure IntersectBMP(B, A)
11     c ← 0
12     foreach w ∈ A do
13         if w's bit is a 1-bit in bitmap B then c ← c + 1
14     return c
```

logarithms of skip size is close to a constant. Thus, we can rewrite the time complexity as $O(c \cdot d_s)$.

Combining VB and PS. When the degrees of two vertices are similar, PS may advance offsets only one element at a time to find the lower bound of the pivot, whereas VB can advance at a block size. To take advantage of both merge algorithms, we combine these two merge algorithms. Specifically, we adopt PS for two sets of high cardinality skew (i.e., $d_u >> d_v$ or $d_u << d_v$) and VB otherwise. We set a tunable threshold for the degree-skew ratio (Lines 2-4).

3.2 BMP

BMP (Algorithm 2) dynamically constructs a bitmap index of $N(u)$ for a vertex u, and reuses the index to do indexed nested loop joins for every $N(u) \cap N(v)$, $\forall v \in N(u)$. In the vertex computation of u (Lines 1-9), a bitmap index of $N(u)$ is constructed and reused for bitmap-array intersections (Lines 10-14).

Different from building auxiliary structures offline [23], we utilize the all-edge computation feature to dynamically construct and reuse our bitmap index, and amortize the construction and clearing costs. The motivation of adopting a bitmap instead of other structures such as skipping-based, tree-based, hash-based index structures [5, 12, 20] is to support put and lookup operations at the actual constant time cost via simple bit operations.

Specifically, we introduce a bitmap of cardinality $|V|$, which is used for the existence checking of a vertex w in the u's neighbor set $N(u)$ via a peek at the corresponding bit (Value 1 indicates the existence, and 0 non-existence). A bitmap is dynamically constructed and cleared for each vertex computation. The count $|N(u) \cap N(v)|$ can be computed in a bitmap-array intersection with the time complexity $O(d_v)$ by looping over all v's neighbors $w \in N(v)$ and counting when w is also in the bitmap index of $N(u)$.

In each vertex computation of u, there are three steps: (1) getting the bitmap B, and setting the bits representing its neighbors $N(u)$; (2) iterating through each of u's neighbor v satisfying the constraint $u < v$, computing $cnt[e(u, v)]$ and symmetrically assigning its value to the $cnt[e(v, u)]$; and (3) clearing the bitmap B by flipping the 1-bits set by u's neighbors.

Index Cost. The dynamic bitmap construction and clearing cost takes an amortized constant time for each set intersection. This is because each of u's neighbor v can account for its own bit's set and flip operations in the bitmap of $N(u)$, which are used for the computation of $cnt[e(u, v)]$ and $cnt[e(v, u)]$. The memory cost of a bitmap is $|V|/8$ bytes. In parallel execution, the number of bitmaps will be the same as the context, which is a few gigabytes in our experiments.

Algorithm 3: Parallel MPS and BMP using OpenMP

```
   Input: a graph G = (V, E) in a CSR = (off and dst arrays) and a tunable task size |T|
   Output: the all-edge common neighbor counts cnt
1  #pragma omp parallel for schedule(dynamic, the task size |T|)
2  foreach e(u, v) ∈ [0, |E|) do
3      u ← FindSrc(off, e(u, v))
4      if u < v then
5          cnt[e(u, v)] ← ComputeCntMPS(u, v)
6          cnt[e(v, u)] ← cnt[e(u, v)]
7  Procedure FindSrc(off, e(u, v))
8      u_tls ← a static thread-local integer, initially u_tls = 0
9      if e(u, v) ≥ off[u_tls + 1] then
10         u_tls ← LowerBound(off[u_tls + 1], off[|V|], e(u, v))
11         if off[u_tls] > e(u, v) then
12             while d_{u_tls − 1} == 0 do u_tls ← u_tls − 1
13             u_tls ← u_tls − 1
14         else while d_{u_tls} == 0 do u_tls ← u_tls + 1
15     return u_tls
16 Procedure ComputeCntMPS(u, v)
17     return IntersectMPS(N(u), N(v))
18 Procedure ComputeCntBMP(u, v)
19     pu_tls ← a static thread-local integer, initially pu_tls = −1
20     B_tls ← a static thread-local bitmap, initially B_tls = all-zero bits
21     if u != pu_tls then
22         if pu_tls != −1 then
23             B_tls ← ClearBitmap(B_tls, N(pu_tls))
24         B_tls ← ConstructBitmap(B_tls, N(u)), pu_tls ← u
25     return IntersectBMP(B_tls, N(v))
```

Time Complexity. Given a degree-descending reordered graph and the computation constraint $u < v$ (Line 5) during the iterations of u's neighbors, the time complexity of each bitmap-array intersection is $O(min(d_u, d_v))$. This is because we have $d_u > d_v$ given $u < v$, and each bitmap-array intersection on edge (u, v) is in the time complexity $O(d_v)$ for the loop of $N(v)$.

4 PARALLELIZATION

To exploit the hardware features of the three modern processors, we propose parallel algorithms for both MPS and BMP on each platform. In our problem, there are $|E|$ set intersections; and the sizes of most neighbor sets are often small. Thus, the parallelism in each set intersection is limited, so we group multiple set intersections into a single task. We define two types of tasks: (1) a fine-grained task, which takes a single-edge neighbor set intersection as the unit; and (2) a coarse-grained task, which takes d_u neighbor set intersections of a vertex u as the unit.

We use T to denote a set of units in a task. On the one hand, constructing large tasks (large $|T|$) makes the task queue maintenance cost negligible. On the other hand, smaller tasks achieve a better load balance than that of large tasks. Therefore, a trade-off of the overhead and load balance should be made. For a fine-grained task, it is feasible to find a fixed $|T|$, and group $|T|$ edge set intersections into a task and dynamically schedule $|E|/|T|$ tasks for the load balance. However, for a coarse-grained task, since each unit is a vertex computation of d_u intersections, and d_u may differ significantly in real world graphs; we choose to fix a small number of units to achieve the load balance (e.g. $|T| = 1$).

4.1 Parallelization on the CPU and KNL

On the CPU and KNL, in the consideration of both load balance and negligible dynamic scheduling overhead, we adopt the fine-grained tasks with a fixed number of units per task. We parallelize both

MPS and BMP with a general skeleton (Algorithm 3). We show a parallel MPS and discuss how to extend it for a parallel BMP.

Specifically, a task is represented in a pair of edge offsets that defines the processing range of edges. For the all-edge common neighbor counting, we separate the edge offset range $[0, |E|)$ into $|E|/|T|$ sub-ranges with an OpenMP parallel for directive. Then, we dynamically schedule these tasks. In each task, we iterate through all the destination vertices v in the edge offset range of the current task. For each vertex v, we proceed in the following three steps (Lines 3-6): (1) compute the source vertex u for the edge pointed by the offset $e(u, v)$; (2) compute the common neighbor count $|N(u) \cap N(v)|$ for the edge (u, v) if the condition $u < v$ holds; and (3) symmetrically assign $cnt[e(u, v)]$ to $cnt[e(v, u)]$.

The main challenge in Algorithm 3 is how to efficiently find the source vertex u in the first step without materializing the source vertex array. A naive way to find u is that for each edge offset $e(u, v)$, we find the lower bound of edge offsets not less than the target offset $e(u, v)$. In other words, we find the first vertex u satisfying $e(u, v) \in [off[u], off[u + 1])$. If that vertex u has at least one neighbor, then we find the correct u, otherwise we move on to find the first occurrence of vertex u with a non-zero degree. To improve the efficiency, we further amortize the cost of the source vertex finding. We record the previously processed source vertex u in a task with a static thread-local variable, and perform the time-consuming lower bound operations only when the current processed edge offset $e(u, v)$ is out of stashed vertex u's neighbor set offset range ($e(u, v) \geq off[u + 1]$). We show the details in the procedure $FindSrc(off, e(u, v))$ (Lines 7-15).

To support a parallel BMP, we only need to replace the common neighbor counting procedure $ComputeCntMPS(u, v)$ (Line 5) with the $ComputeCntBMP(u, v)$. The essential differences of these two procedures are the static thread-local bitmap clearing and construction operations. Specifically, we update a thread-local bitmap for the $N(u)$ indexing when the source vertex of processed edges changes. We use a static thread-local integer pu_{tls} to record the last processed vertex u, and update the bitmap when pu_{tls} differs from the current u. We show the details in a procedure (Lines 18-25). For a vertex u with a large degree, all the threads assigned to a task on u will construct a bitmap index for $N(u)$. Nevertheless, each thread only constructs an index for a vertex once. As such, the amortized cost of bitmap clearing and construction is not large, compared to that of bitmap-array intersections.

4.2 Parallelization on the GPU

On the GPU, benefiting from the low overhead of the hardware thread block scheduler, we adopt the coarse-grained tasks and map the common neighbor counting for a vertex u into a thread block, which simplifies the GPU programming.

The capacity of global memory on the GPU is limited (12 GB) compared to that of DRAM on the CPU and KNL (100+ GB). For billion-edge graphs, it is infeasible to directly allocate all the data structures on the global memory. To deal with the challenge, we utilize the unified memory feature available on Pascal GPUs, which enables us to have a single declared memory area addressable by both CPU and GPU. Memory pages will be swapped by the CUDA runtime automatically on the page faults.

Algorithm 4: Parallel MPS and BMP using CUDA

```
Input: a graph G = (V, E) in a CSR = (off and dst arrays)
Output: the all-edge common neighbor counts cnt
1  LaunchCUDAKernels(), AssignOffsetsOnCPU()
2  SynchronizeDevice()
3  foreach (u, v) ∈ E in parallel do
4  |  if u > v then cnt[e(u, v)] ← cnt[cnt[e(u, v)]]
5  Procedure AssignOffsetsOnCPU()
6  |  foreach (u, v) ∈ E in parallel do
7  |  |  if u < v then cnt[e(v, u)] ← e(u, v)
```

Algorithm 5: Parallel MPS kernels

```
Input: a graph G = (V, E) in a CSR = (off and dst arrays) and a tunable threshold t
Output: the common neighbor counts cnt[e(u, v)] of all the adjacent vertex pairs
        (u, v) satisfying u < v
/* |V| thread blocks, 2D threads per block (blockDim.x: the warp size 32,
   blockDim.y: the number of warps per block)                              */
1  Launch MKernel(off, dst, cnt, t)
/* |V| threads blocks, 1D threads per block                                */
2  Launch PSKernel(off, dst, cnt, t)
3  Procedure MKernel(off, dst, cnt, t)
4  |  u ← blockIdx.x, off_warp ← off[u] + threadIdx.y
5  |  for i ← off_warp; i < off[u + 1]; i ← i + blockDim.y do
6  |  |  c ← 0, v ← dst[i]
7  |  |  if u > v or d_u/d_v > t or d_v/d_u > t then continue
8  |  |  c ← WarpWiseBlockMerge(N(u), N(v))
   |  |  /* A warp-wise reduction for the sum of counts                    */
9  |  |  foreach k ∈ {16, 8, 4, 2, 1} do
10 |  |  |  c ← c + __shfl_down(c, k)
11 |  |  if threadIdx.x == 0 then cnt[i] ← c
12 Procedure PSKernel(off, dst, cnt, t)
13 |  u ← blockIdx.x, off_thread ← off[u] + threadIdx.x
14 |  for i ← off_thread; i < off[u + 1]; i ← i + blockDim.x do
15 |  |  c ← 0, v ← dst[i]
16 |  |  if u > v or d_u/d_v ≤ t or d_v/d_u ≤ t then continue
17 |  |  cnt[i] ← InterSectPS(N(u), N(v))
```

Memory Allocation. We allocate the CSR arrays and the common neighbor count array on the unified memory for both MPS and BMP. For BMP, we allocate a pool of bitmaps on the global memory directly without using the unified memory feature, since the bitmaps are frequently accessed by thread blocks, so we avoid page swaps on the bitmaps.

4.2.1 CUDA Implementations. To avoid as much irregular memory access as possible on the GPU, for both MPS and BMP, we symmetrically assign the count ($cnt[e(v, u)] \leftarrow cnt[e(u, v)]$) on the CPU in parallel as a post-processing phase.

Co-Processing on the Same Count Array. Given an edge offset $e(u, v)$, the reverse edge offset finding of $e(v, u)$ is time-consuming due to the intensive binary searches. To hide the latency, we propose assigning $cnt[e(u, v)] \leftarrow e(v, u)$ for each edge (u, v) satisfying $u > v$ on the CPU, which overlaps the common neighbor counting on the GPU for each edge (u, v) satisfying $u < v$. The reverse edge offsets are utilized for the fast final symmetric assignment, after all the counting is completed on the GPU. The final assignment on the CPU is done by $cnt[e(u, v)] \leftarrow cnt[cnt[e(u, v)]]$ $(u > v)$. The co-processing is enabled by the concurrent access of the same array on both CPU and GPU, a feature supported on recent GPUs. We summarize the main program with the co-processing technique for both MPS and BMP in Algorithm 4, and show the CUDA kernels of MPS and BMP respectively in Algorithms 5 and 6.

MPS. We launch two kernels for the non-degree-skewed and degree-skewed cases respectively (Algorithm 5). The block-wise kernel (Lines 3-11) exploits the warp-level parallelism, using each thread warp (32 threads) to handle a set intersection $N(u) \cap N(v)$

Algorithm 6: Parallel BMP kernels

```
   Input: a graph G = (V, E) in a CSR = (off and dst arrays)
   Output: the common neighbor counts cnt[e(u, v)] of all the adjacent vertex pairs
           (u, v) satisfying u < v
1  B_A ← an array of bitmaps, BS_A ← a bitmap occupation status array
2  n_C ← the maximum number of concurrent thread blocks per SM
   /* |V| thread blocks, 2D threads per block (blockDim.x: the warp size 32,
      blockDim.y: the number of warps per block)                          */
3  Launch BMPKernel(off, dst, cnt, B_A, BS_A, n_C)
4  Procedure BMPKernel(off, dst, cnt, B_A, BS_A, n_C)
5      u ← blockIdx.x, tid ← threadIdx.x+ blockDim.x·threadIdx.y
6      if tid == 0 then B ← AcquireBitmap(B_A, BS_A, n_C)
7      __syncthreads()
8      AtomicConstrucBitmap(B, N(u))
9      __syncthreads()
10     off_warp ← off[u] + threadIdx.y
11     for i ← off_warp; i < off[u + 1]; i ← i + blockDim.y do
12         c ← 0, v ← dst[i]
13         if u > v then continue
14         foreach w ∈ N(v) in warp-wise parallel do
15             if the w's bit is a 1-bit in the bitmap B then
16                 c ← c + 1
17         foreach k ∈ {16, 8, 4, 2, 1} do
18             c ← c + __shfl_down(c, k)
19         if threadIdx.x == 0 then cnt[i] ← c
20     __syncthreads()
21     ClearBitmap(B, N(u)), ReleaseBitmap(BS_A)
22 Procedure AcquireBitmap(B_A, BS_A, n_C)
23     i ← 0, sm_id ← the id of the current SM
24     while atomicCAS(&BS_A[sm_id · n_C + i], 0, 1)! = 0) do
25         i ← i + 1
26     return B_A[sm_id · n_C + i]
```

and conduct a block-wise merge, the logic of which is similar to the vectorized one on the CPU [14]. The multiplication of block sizes for the $N(u)$ and $N(v)$ is 32 (warp size). For each warp, we load 32 elements of a neighbor set on demand into its own shared memory region for the merge. After the merge, intermediate match counts are stored in the register of each thread within a warp. We adopt a warp-shuffling intrinsic function to perform a reduction and get the common neighbor count, after which a single thread in each warp writes the count back to the global memory. Different warps in a thread block utilize their own shared memories without any contention. Thus, no synchronization for a thread block is required.

However, the pivot-skip-based merge involves irregular lower bound operations (including exponential and binary searches), which consists of irregular gatherings. The warp-level parallelism cannot be exploited in these cases. Thus, in our pivot-skip-based merge kernel (Lines 12-17), we directly map each single-edge neighbor set intersection to a thread without using the shared memory.

BMP. We launch a single CUDA kernel for BMP (Algorithm 6). We allocate a bitmap of cardinality $|V|$ for each parallel execution context (i.e., a thread block). To manage a pool of bitmaps (B_A), we introduce a bitmap occupation status array (BS_A). The total number of bitmaps is the number of SMs multiplying the maximum number of concurrent thread blocks on an SM (n_C). The logic of the kernel is as follows. A single thread in a block acquires the bitmap by atomically checking and updating the corresponding area in the BS_A for the current SM (Lines 22-26). Before index construction, we synchronize threads in each thread block (Line 7). After the bitmap is acquired for the thread block, all the threads in the block construct the index by atomic-or operations on the corresponding word. The index construction is completed with the synchronization within each thread block. With the index, each warp in a thread block process edges for the counting using the bitmap-array intersections.

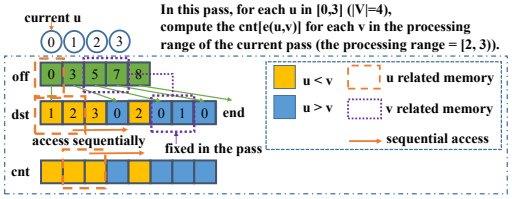


Figure 2: Illustrating the multi-pass processing on the GPU

Each thread within the same warp checks whether its corresponding bit for a vertex is set, and updates its local counter after finding a match. A warp-wise reduction of intermediate results computes the final result of $|N(u) \cap N(v)|$. After all the threads in a block complete the work, the corresponding bitmap is cleared and released in a similar logic to its allocation and construction.

4.2.2 Multi-Pass Processing Technique. When the graph does not fit in the global memory, to preserve data access locality and minimize page swaps, we propose a multi-pass processing technique. The key idea is to process a subset of adjacent vertices in a pass, which increases the memory access locality. We give an example of multi-pass task processing in Figure 2. We split a range $[0, |V|)$ into multiple vertex ranges and assign each range to a pass. In a pass, we iterate through all the vertices $u \in V$, and compute counts only for u's neighbor v in the vertex range of the current pass (in our example $v \in [2, 3)$). In this way, we access a subset of the neighbors only during the iteration. To minimize page swaps, we estimate the *maximum processing range capacity* of each pass by deducting the total global memory capacity Mem_{global} by (1) the bitmap memory consumption Mem_{B_A}, and (2) the tunable reserved memory for the sequential access of the CSR and count array $Mem_{reserved}$. The number of passes is computed via dividing the CSR memory consumption Mem_{CSR} by the maximum processing range capacity, i.e., $\lceil Mem_{CSR}/(Mem_{global} - Mem_{reserved} - Mem_{B_A}) \rceil$.

4.3 Optimizations

To fully utilize the hardware features of modern processors, we design four optimization techniques: (1) vectorization techniques for MPS (discussed in Section 3.1); (2) the bitmap range filtering for BMP to reduce access to the bitmap of cardinality $|V|$, and utilize the cache on the CPU and KNL or the shared memory on the GPU; (3) the memory allocation of data structures on the MCDRAM on the KNL; and (4) the co-processing which overlaps the reversed edge offset assignment on the CPU and the counting on the GPU for both MPS and BMP (discussed in Section 4.2.1).

Bitmap Range Filtering. In real-world graphs, only a small portion of a neighbor set contribute to the common neighbor counting. In other words, the matches of two neighbor sets are sparse. To make use of the sparsity, we introduce a small bitmap, a bit of which is to indicate the existence of 1-bit in a certain range of the underlying bitmap of cardinality $|V|$. If a quick lookup on the small bitmap tells us that in the current range there is no 1-bit, we can avoid accessing the underlying bitmap. The range size is tunable via setting the bitmap range scale to fit the small bitmap in the cache on the CPU and KNL or the shared memory on the GPU.

High Bandwidth Memory Utilization. A neighbor set is represented in a sorted array, and both arrays and bitmaps are accessed sequentially in MPS and BMP for the intersections. Such access

patterns can fully utilize the high bandwidth memory on the KNL to improve the scalability to number of threads for both algorithms. There are two ways to use the MCDRAM: (1) directly configure the cache mode without changing the implementations; and (2) switch to the flat mode, and allocate bitmaps and CSR arrays on the MCDRAM using a memkind library.

5 EVALUATION

In this section, we evaluate the effectiveness of individual techniques for both MPS and BMP on three processors, compare the optimized algorithms for each processor on five real-world billion-edge graphs, and summarize our experimental findings.

5.1 Experimental Setup

Since the architectures of the KNL and CPU are more similar than the GPU, we evaluate the effectiveness of four common techniques for the two processors: (1) the degree-skew handling (**DSH**) for MPS and BMP, (2) the vectorization or instruction-level parallelization (**V**) for MPS, (3) the task-level parallelization (**P**) for MPS and BMP, and (4) the bitmap range filtering (**RF**) for BMP. We further evaluate the high bandwidth memory MCDRAM usage (**HBW**) on the KNL.

We start from the sequential implementations of the algorithms; on the KNL, we initially configure the MCDRAM at the flat-mode. Subsequently, we enable the techniques one by one to evaluate their effectiveness in the order of DSH, V, P, RF and HBW. We first evaluate the techniques DSH and V with sequential implementations to study the workload reduction effects from algorithmic and vectorization optimizations. The RF and HBW are evaluated after parallelization, since these two orthogonal techniques influence the scalability to number of threads by reducing the latency and improving the bandwidth of the memory access respectively.

On the GPU, we evaluate four techniques: (1) the co-processing (**CP**) to reduce the CPU's post-processing time for MPS and BMP, (2) the multi-pass processing (**MPP**) for MPS and BMP, (3) the bitmap range filtering (**RF**) with the shared memory for BMP, and (4) the block size tuning (**BST**) for MPS and BMP.

We evaluate the four orthogonal techniques on the GPU by enabling them one by one in the order of CP, MPP, RF and BST. By default, we use 4 warps per thread block, resulting in at most 16 (2048/128) concurrent thread blocks per SM. According to the Nvidia document, 16 is the maximum number of thread blocks simultaneously scheduled on a SM of the TITAN Xp GPU. Our default setting targets a maximum of 100% occupancy of the GPU.

Finally, we compare the optimized implementations of MPS and BMP for each processor on five real-world graphs.

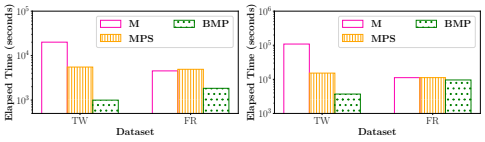
Environment. We implement the algorithms on the CPU and KNL in C++, and compile them with icpc 17.0.0. We implement the GPU algorithms in CUDA 8.0, and compile them with nvcc 8.0.61. We conduct experiments on two Linux servers: (1) a CPU server with CPUs and Nvidia TITAN Xp GPUs, and (2) a KNL server with a KNL processor. The CPU server has two 14-core 2.4GHz Intel Xeon E5-2680 CPUs. The L1, L2, L3 cache and DRAM of the CPU server are 64KB, 256KB, 35MB and 512GB respectively. The Nvidia GPU on the CPU server has 30 SMs, each of which supports at most 2048 threads. The KNL server has a 64-core 1.3GHz Intel Xeon Phi 7210 Processor, configured in the quadrant mode. The L1, L2 cache, MCDRAM and RAM of the KNL server are 64KB, 1024KB, 16GB and 96GB respectively.

Table 1: Real-world Graph Statistics

Dataset	$\|V\|$	$\|E\|$	$\overline{d}$	max d
livejournal (LJ)	4, 036, 538	34, 681, 189	17.2	14, 815
orkut (OR)	3, 072, 627	117, 185, 083	76.3	33, 312
web-it (WI)	41, 291, 083	583, 044, 292	28.2	1, 243, 927
twitter (TW)	41, 652, 230	684, 500, 375	32.9	1, 405, 985
friendster (FR)	124, 836, 180	1, 806, 067, 135	28.9	5, 214

Table 2: Percentage of the highly skewed set intersections.

Dataset	LJ	OR	WI	TW	FR
Percentage	1.2%	1.8%	27.9%	31.0%	1.6%



(a) on the CPU (b) on the KNL

Figure 3: Effect of degree skew handling (single threaded)

Datasets. We select five real-world graphs (shown in Table 1) downloaded from SNAP [17] and WebGraph [6, 7], which are widely used in the evaluation of graph algorithms [1, 8, 9, 13, 23]. For the all-edge common neighbor counting, WI and TW incur more degree-skewed set intersections (i.e., $d_u >> d_v$ for $N(u) \cap N(v)$), than the other datasets LJ, OR and FR. We show the percentage of highly skewed intersections in the counting ($d_u/d_v > 50$ supposing $d_u > d_v$)[1] in Table 2.

Metric. We run each experiment multiple times and report the average in-memory processing time. Specifically, we measure the elapsed time from the end of graph loading into the memory to the computation of the all-edge counting.

5.2 Evaluation of Individual Techniques

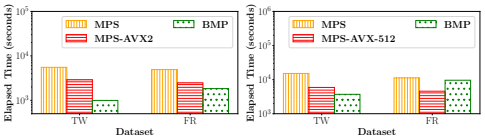
For brevity, in this section, we present the evaluation of individual techniques with the two representative datasets TW and FR.

5.2.1 Techniques on the CPU and KNL. We evaluate the four common techniques on both processors and the high bandwidth memory MCDRAM utilization on the KNL.

Effect of Degree Skew Handling. We implement a basic merge-based algorithm (M) without the pivot-skip technique as our baseline. We compare both MPS and BMP with M, and show the results in Figure 3. We firstly evaluate on the TW, where the percentage of highly skewed set intersections is 31% (Table 2). On the TW, MPS is 3.6x and 7.1x faster than M, on the CPU and KNL respectively. On the TW, BMP is 20.1x and 29.3x faster than M, on the CPU and KNL respectively. We then evaluate on the FR. MPS achieves a similar performance to M. BMP runs 2.5x and 1.1x faster than M, on the CPU and KNL respectively. The improvement on the KNL is much less than that on the CPU, because the KNL benefits more from the regular memory access in M than the CPU, and the CPU favors the saving from pivot-skip and the bitmap-index than the KNL.

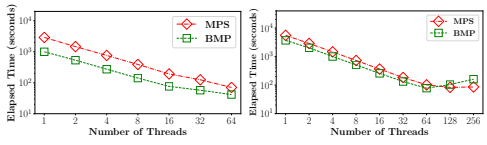
Effect of Vectorization. We utilize the AVX2 and AVX-512 instruction sets on the CPU and KNL respectively to improve MPS. We compare the original MPS, vectorized MPS, and BMP in Figure 4. On the CPU, MPS-AVX2 runs 1.9x and 2.0x faster than MPS on the

[1] We choose an empirical number 50 as the threshold to control the merge algorithm selection in MPS (Algorithm 1).

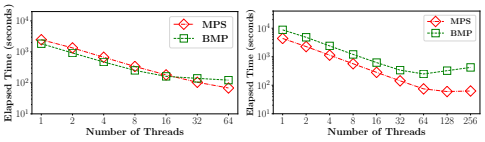


(a) on the CPU (b) on the KNL

Figure 4: Effect of vectorization (single threaded)



(a) TW, on the CPU (b) TW, on the KNL



(c) FR, on the CPU (d) FR, on the KNL

Figure 5: Scalability to number of threads (with V enabled)

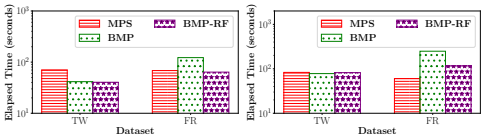
TW and FR, and MPS-AVX-512 runs 2.6x and 2.5x faster than MPS on the TW and FR. The vectorized MPS shows better improvements on the KNL than on the CPU, due to a 2x larger vector register width on the KNL than on the CPU. Comparing vectorized MPS with BMP, we observe CPU favors BMP; on the TW, MPS-AVX-512 is slower than BMP (taking 1.6x time), whereas on the FR, MPS-AVX-512 is 2.1x faster than BMP. This result suggests that BMP works better than MPS except on the FR on the KNL.

Effect of Parallelization. We evaluate the scalability to number of threads for both parallel MPS and BMP in Figure 5. Since the CPU and the KNL support two and four hyper threads respectively; we vary the number of threads from 1 to 64 on the CPU and from 1 to 256 on the KNL. MPS scales well on the CPU and achieves 41.1x and 36.1x speedups over the vectorized sequential one with 64 threads on the TW and FR respectively, which are greater than the number of cores 28, due to the regular and predictable memory access patterns and the hyper-threading techniques. MPS also scales well on the KNL before the number of threads becomes greater than 64, when the memory bandwidth is saturated. Thus, the speedups of MPS on the KNL are up to 67.0x and 72.0x on the TW and FR respectively, even though there are 128 VPUs on the KNL. We next study the scalability of BMP. In each bitmap index based intersection $N(u) \cap N(v)$, both the bitmap of $N(u)$ and the sorted array of $N(v)$ are accessed. The accesses of bitmap may cross a wide range, since $N(v)$ can scatter in any position of $[0, |V|)$. On the CPU, BMP scales well with the number of threads, but only achieves 24.0x and 15.0x speedups respectively on the TW and FR, due to the poor memory access pattern than MPS. The speedup with 64 threads on the FR is worse than that on the TW, due to a 3x wider range of bitmap indices and more sparsity of $N(v)$. On the KNL, with 128 and 256 threads, BMP slows down due to the poorer memory access locality from the increasing number of thread-local bitmaps.

Effect of Bitmap Range Filtering. Recall that we introduce a small bitmap with each bit indicating the existence of non-all-zero range for the underlying large bitmap. We set the size ratio of the two bitmaps at 4096, to make the small bitmap fit into L1 cache. We show the memory consumption of each thread-local bitmaps in Table 3. We compare BMP enabled the technique RF (BMP-RF) with both BMP and MPS in Figure 6. The results on the two processors show similar trends. On the TW dataset, BMP and BMP-RF have a similar performance. In comparison, on the FR, BMP-RF runs 1.9x and 2.1x faster than BMP for the CPU and the KNL respectively. This is because (1) bitmaps on the FR has a larger cardinality and consumes more memory than on the TW; and (2) FR is a more uniform dataset than TW on vertex degrees and the range filtering helps greatly in avoiding access of the big bitmaps for non-degree-skewed intersections.

Table 3: Memory consumption of each thread-local bitmap

Dataset	TW		FR	
Bitmap Type	small	large	small	large
Memory Consumption	1.3KB	5.0MB	3.7KB	14.9MB



(a) on the CPU (b) on the KNL

Figure 6: Effect of bitmap range filtering (with V and P enabled)

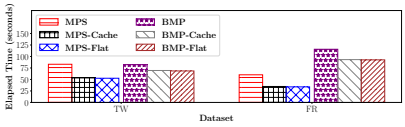


Figure 7: Effectiveness of MCDRAM utilization

Table 4: Comparison with the baseline M (seconds).

Dataset	TW		FR	
Processor	*CPU*	*KNL*	*CPU*	*KNL*
T_M	20065.3	108418.6	4528.8	11199.9
T_{MPS}	5527.2	15244.4	4919.1	11224.1
T_{MPS+V}	2891.6	5904.0	2470.7	4569.4
$T_{MPS+V+P}$	70.3	83.1	68.3	60.1
$T_{MPS+V+P+HBW}$	N/A	52.7	N/A	33.9
T_{BMP}	996.2	3704.3	1837.2	9591.3
T_{BMP+P}	41.5	78.1	122.5	248.7
$T_{BMP+P+RF}$	40.4	82.1	63.8	115.7
$T_{BMP+P+RF+HBW}$	N/A	68.5	N/A	92.6
Best MPS Speedup over M	286x	2,057x	66x	330x
Best BMP Speedup over M	497x	1,583x	71x	121x

Effect of MCDRAM Utilization. With all the four techniques enabled, we further study the effect of utilizing the high bandwidth memory MCDRAM on the KNL. We compare the MCDRAM in the cache and flat modes, and show the results in Figure 7. MPS-Flat runs 1.6x and 1.8x faster than MPS on TW and FR respectively, because MCDRAM improves the parallel efficiency of MPS. For example, MPS-Flat achieves 112.0x speedup over the sequential MPS-Flat, more significant than the 71.0x speedup of MPS over the sequential MPS. The result indicates that MPS is memory-bandwidth bounded. BMP-Flat runs 1.2x and 1.3x faster than BMP on the TW and FR respectively, which are less significant to the improvements of MPS-Flat over MPS. This is because the bitmap access is more sensitive to the memory access latency than the bandwidth. Algorithms under the cache mode show competitive performance with those under the flat mode, because the capacity of MCDRAM is large (16GB) and the accesses have good locality. On both datasets, both MPS-Cache and BMP-Cache are slightly slower than MPS-Flat and BMP-Flat respectively, due to the data movement overhead in the cache mode.

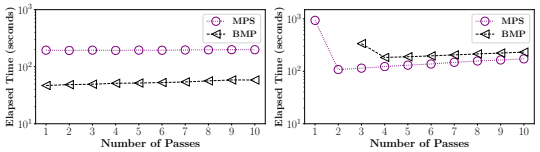
Summary. We summarize the performance improvement from the five techniques in Table 4. The degree-skew handling techniques in MPS and BMP achieve 7x and 30x speedups over the baseline M

Table 5: Post-processing time on the CPU (seconds)

Dataset	TW		FR	
Enabling Co-Processing	No	Yes	No	Yes
Elapsed Time	5.6	0.9	19.0	3.8

Table 6: Memory consumption of data structures and estimated number of passes

Dataset	TW		FR	
Algorithm	MPS	BMP	MPS	BMP
Mem_{cnt}	5.2GB		13.5GB	
Mem_{CSR}	5.3GB		13.9GB	
Mem_{B_A} (480 bitmaps)	0	2.3GB	0	7.0GB
Estimated number of passes	1	1	2	4



(a) TW (b) FR

Figure 8: Effect of number of passes

respectively. The two orthogonal techniques V and P for MPS bring 79x-84x and 183x-186x speedups over the sequential MPS on the CPU and KNL respectively. The technique HBW further improves the memory bandwidth in MPS, achieving 1.6x-1.8x speedups over the parallel vectorized MPS. The orthogonal techniques P and RF for BMP achieve 25x-28x and 47x-83x speedups on the CPU and KNL respectively. The technique HBW brings only 10-20% improvements for the parallel BMP-RF, which is random access dominant. CPU favors BMP whereas KNL favors MPS, because BMP works better with CPU's large caches than KNL and KNL's many cores perform the computation of MPS faster than CPU.

5.2.2 Techniques on the GPU. We evaluate four techniques for the two algorithms on the GPU, with 4 warps (128 threads) per thread block by default.

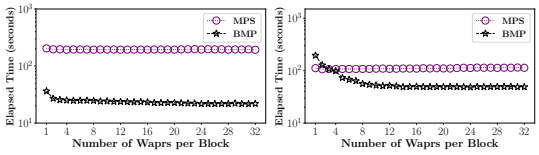
Effect of Co-Processing. We show the symmetric assignment post-processing time on the CPU with and without the co-processing in Table 5. The results show that co-processing reduces the elapsed time from 5.6 to 0.9 and from 19.0 to 3.8 seconds on the TW and FR respectively, due to the overlap of concurrent computations on the CPU and GPU. In the following evaluation on the GPU, we enable this technique; and our reported elapsed time includes the post-processing time on the CPU.

Effect of Multi-Pass Processing. We show the memory consumption of data structures and our estimated number of passes for both MPS and BMP in Table 6. Recall that the number of passes is estimated based on $\lceil Mem_{CSR}/(Mem_{global} - Mem_{reserved} - Mem_{B_A})\rceil$ where $Mem_{CSR}, Mem_{reserved}$ and Mem_{B_A} are the memory consumption of the CSR, the reserved memory and the bitmap respectively, and Mem_{global} is the capacity of the global memory. In our setting, the global memory is 12GB and the reserved memory size is 500MB. We use 128 threads per thread block for both MPS and BMP. Thus, there are 16 (2048/128) concurrent blocks per SM. Given 30 SMs in total, we need to allocate 480 bitmaps for BMP.

We evaluate the technique MPP by varying the number of passes for both MPS and BMP. To avoid excessively long execution time due to thrashing page swaps, we set the time limit to an hour, and stop the non-completed executions. We show the results in Figure 8. On the TW, given more passes, the elapsed time of both algorithms

Table 7: Elapsed time of BMP on the GPU (seconds)

Dataset	TW		FR	
Enabling Range Filtering	No	Yes	No	Yes
Elapsed Time	46.8	24.9	184.1	97.5



(a) TW (b) FR

Figure 9: Effect of block size

increases slightly. As the memory consumption of both algorithms is within the global memory capacity, the ascending trend of both curves is resulted from increased memory loads from the multi-pass processing. In contrast, on the FR, BMP fails when the number of passes is less than three, due to thrashing page swaps. The results show that our estimated number of passes is effective in minimizing page swaps and preserving the memory access locality.

Effect of Bitmap Range Filtering. The technique RF for BMP on the GPU is similar to that on the KNL and CPU. The only difference is that we write the small bitmap for range filtering into the on-chip shared memory on the GPU (48KB per SM). We show the results in Table 7. The range filtering speeds up BMP by 1.9x on the TW and FR, due to the reduction of global memory loads.

Effect of Block Size Tuning. We tune the number of warps per thread block from 1 to the maximum 32, and show the results in Figure 9. In our setting, there are 2048 threads per SM, and at most 16 thread blocks concurrently scheduled in a SM. Then, one and more than three warps per thread block corresponds to 25% and 100% theoretical occupancy respectively. The curves of MPS are flat on both TW and FR, which indicates MPS is not sensitive to the changes of computation resources and is bounded by memory access. BMP's performance improves when the number of warps increases from 1 to 4. This performance improvement in BMP is because a larger block size leads to fewer thread blocks and fewer bitmaps, which results in fewer number of passes on the FR. When the thread block size is sufficiently large, the memory access latency is fully hidden in the parallel computation, and BMP's performance flattens. In particular, on the FR, BMP with 32 warps per block runs 2x faster than that with the default block size.

Summary. The technique CP reduces more than 80% of post-processing time on the GPU on both TW and FR datasets. MPP reduces the elapsed time by orders of magnitude for both algorithms on the FR. RF reduces the time by half on both data sets, whereas BST achieves 2.2x and 3.8x speedups over the non-optimized BMP on the TW and the FR respectively. Overall, the GPU favors BMP, because BMP has less workload than MPS, exploits warp-level parallelism and utilizes GPU resources more effectively than MPS.

5.3 Comparison of Optimized Algorithms

We compare the performance of the optimized MPS and BMP on three processors with five datasets in Figure 10. Firstly, we compare MPS and BMP for each processor. CPU favors BMP, because (1) BMP has less workload for each set intersection than MPS; and (2) the out-of-order instruction execution and the cache-memory hierarchy of the CPU hides the memory access latency of BMP. KNL favors MPS, because (1) MPS exploits the vectorization technique and scales

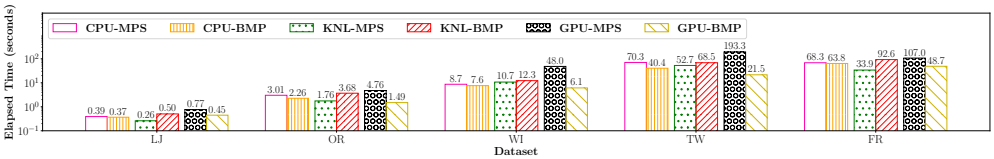


Figure 10: Elapsed time of optimized algorithms for each processor on five real-world graphs

better than BMP, and has a better memory access pattern; and (2) KNL is equipped with 128 VPUs and AVX-512 instruction sets, and high bandwidth memory MCDRAM. GPU favors BMP, because (1) BMP incurs fewer global memory loads than MPS; (2) BMP exploits warp-level parallelism which results in better occupation of GPU resources; and (3) the pivot-skip merge kernel for MPS on the GPU is inefficient due to irregular memory gathering.

Secondly, we compare the performance of CPU-BMP, KNL-MPS and GPU-BMP on each graph. All the computations on each processor can complete within tens of seconds, and the performance of the best algorithms on three processors differs by a factor of 2.5x at maximum. On the relatively smaller datasets LJ and OR, all the algorithms complete within 2.3 seconds. On the WI and TW with more degree-skewed set intersections, GPU-BMP works best due to the warp-level parallelism and full utilization of 30 SMs. On the FR, KNL-MPS works better because it fully exploits its vectorization, parallelization techniques and utilizes the VPUs and MCDRAM. CPU-BMP is less competitive but has acceptable performance.

5.4 Summary

- MPS exploits the vectorization technique, and scales better to number of threads than BMP; while BMP involves fewer workloads for each neighbor set intersection than that of MPS.
- The performance of the two algorithms is close on both CPU and KNL, but may differ up to an order of magnitude on the GPU. MPS works best on the KNL, because of the 128 VPUs and the high bandwidth memory MCDRAM. CPU favors BMP, because its L3 cache reduces the memory access latency. GPU also favors BMP, since the warp-level parallelism helps utilize the resources.
- BMP on the GPU and MPS on the KNL work best respectively for degree-skewed and non-degree-skewed large graphs. The performance of both algorithms on the CPU is moderate, at most 1.9x slower than the best algorithms on both the KNL and the GPU. MPS on the GPU is always the slowest, followed by BMP on the KNL.
- Our optimized parallel algorithms complete the computation within tens of seconds on billion-edge graphs, enabling online graph analytics.

6 CONCLUSION

To accelerate the operation of all-edge common neighbor counting, we study MPS and BMP two representative merge-based and index-based algorithms respectively. To exploit the hardware features of modern processors, we parallelize and optimize both algorithms. Furthermore, we evaluate individual techniques for the two algorithms on each platform, and compare the optimized algorithms. We show that for each processor, our optimized algorithms can complete the operation within tens of seconds on billion-edge graphs, enabling online graph analytics.

7 ACKNOWLEDGMENTS

This work was partly supported by grants 16206414 from the Hong Kong Research Grants Council and MRA11EG01 from Microsoft.

REFERENCES

[1] Christopher R Aberger, Andrew Lamb, Susan Tu, Andres Nötzli, Kunle Olukotun, and Christopher Ré. 2017. Emptyheaded: A relational engine for graph processing. *TODS* 42, 4 (2017), 20.
[2] Ricardo Baeza-Yates. 2004. A fast set intersection algorithm for sorted sequences. In *CPM*. Springer, 400–408.
[3] Ricardo Baeza-Yates and Alejandro Salinger. 2005. Experimental analysis of a fast intersection algorithm for sorted sequences. In *SPIRE*. Springer, 13–24.
[4] Jérémy Barbay, Alejandro López-Ortiz, Tyler Lu, and Alejandro Salinger. 2009. An experimental investigation of set intersection algorithms for text searching. *JEA* 14 (2009), 7.
[5] Philip Bille, Anna Pagh, and Rasmus Pagh. 2007. Fast evaluation of union-intersection expressions. In *ISAAC*. Springer, 739–750.
[6] Paolo Boldi, Marco Rosa, Massimo Santini, and Sebastiano Vigna. 2011. Layered label propagation: A multiresolution coordinate-free ordering for compressing social networks. In *WWW*. ACM, 587–596.
[7] Paolo Boldi and Sebastiano Vigna. 2004. The webgraph framework I: compression techniques. In *WWW*. ACM, 595–602.
[8] Lijun Chang, Wei Li, Xuemin Lin, Lu Qin, and Wenjie Zhang. 2016. pSCAN: Fast and exact structural graph clustering. In *ICDE*. IEEE, 253–264.
[9] Yulin Che, Shixuan Sun, and Qiong Luo. 2018. Parallelizing Pruning-based Graph Structural Clustering. In *ICPP*. ACM, 77.
[10] Yangjun Chen and Weixin Shen. 2016. An efficient method to evaluate intersections on big data sets. *TCS* 647 (2016), 1–21.
[11] Erik D Demaine, Alejandro López-Ortiz, and J Ian Munro. 2000. Adaptive set intersections, unions, and differences. In *SODA*. Citeseer, 743–752.
[12] Bolin Ding and Arnd Christian König. 2011. Fast set intersection in memory. *PVLDB* 4, 4 (2011), 255–266.
[13] Shuo Han, Lei Zou, and Jeffrey Xu Yu. 2018. Speeding Up Set Intersections in Graph Algorithms using SIMD Instructions. In *SIGMOD*. ACM, 1587–1602.
[14] Hiroshi Inoue, Moriyoshi Ohara, and Kenjiro Taura. 2014. Faster set intersection with SIMD instructions by reducing branch mispredictions. *PVLDB* 8, 3 (2014), 293–304.
[15] Daniel Lemire, Leonid Boytsov, and Nathan Kurz. 2016. SIMD compression and the intersection of sorted integers. *SPE* 46, 6 (2016), 723–749.
[16] Daniel Lemire, Gregory Ssi-Yan-Kai, and Owen Kaser. 2016. Consistently faster and smaller compressed bitmaps with roaring. *SPE* 46, 11 (2016), 1547–1569.
[17] Jure Leskovec and Andrej Krevl. 2014. SNAP Datasets: Stanford Large Network Dataset Collection. http://snap.stanford.edu/data.
[18] Sungsu Lim, Seungwoo Ryu, Sejeong Kwon, Kyomin Jung, and Jae-Gil Lee. 2014. LinkSCAN*: Overlapping community detection using the link-space transformation. In *ICDE*. IEEE, 292–303.
[19] Son T Mai, Martin Storgaard Dieu, Ira Assent, Jon Jacobsen, Jesper Kristensen, and Mathias Birk. 2017. Scalable and interactive graph clustering algorithm on multicore CPUs. In *ICDE*. IEEE, 349–360.
[20] William Pugh. 1998. *A skip list cookbook*. Technical Report.
[21] Hiroaki Shiokawa, Yasuhiro Fujiwara, and Makoto Onizuka. 2015. SCAN++: efficient algorithm for finding clusters, hubs and outliers on large-scale graphs. *PVLDB* 8, 11 (2015), 1178–1189.
[22] Hiroaki Shiokawa, Tomokatsu Takahashi, and Hiroyuki Kitagawa. 2018. ScaleSCAN: Scalable Density-Based Graph Clustering. In *DEXA*. Springer, 18–34.
[23] Julian Shun and Kanat Tangwongsan. 2015. Multicore triangle computations without tuning. In *ICDE*. IEEE, 149–160.
[24] Shixuan Sun, Yulin Che, Lipeng Wang, and Qiong Luo. 2019. Efficient Parallel Subgraph Enumeration on a Single Machine. In *ICDE*. IEEE.
[25] Tomokatsu Takahashi, Hiroaki Shiokawa, and Hiroyuki Kitagawa. 2017. SCAN-XP: Parallel Structural Graph Clustering Algorithm on Intel Xeon Phi Coprocessors. In *SIGMOD Workshop on NDA*. ACM, 6.
[26] Dong Wen, Lu Qin, Ying Zhang, Lijun Chang, and Xuemin Lin. 2017. Efficient structural graph clustering: an index-based approach. *PVLDB* 11, 3 (2017), 243–255.
[27] Xiaowei Xu, Nurcan Yuruk, Zhidan Feng, and Thomas AJ Schweiger. 2007. Scan: a structural clustering algorithm for networks. In *SIGKDD*. ACM, 824–833.