Conflict-Free Vectorization of Associative Irregular Applications with Recent SIMD Architectural Advances

Peng Jiang
Department of Computer Science and Engineering
The Ohio State University
Columbus, OH, USA
jiang.952@osu.edu

Gagan Agrawal
Department of Computer Science and Engineering
The Ohio State University
Columbus, OH, USA
agrawal@cse.ohio-state.edu

Abstract

Irregular applications that involve indirect memory accesses were traditionally considered unsuitable for SIMD processing. Though some progress has been made in recent years, the existing approaches require either expensive data reorganization or favorable input distribution to deliver good performance. In this work, we propose a novel vectorization approach called *in-vector reduction* that can efficiently accelerate a class of associative irregular applications. This approach exploits associativity in the irregular reductions to resolve the data conflicts within SIMD vectors. We implement in-vector reduction with the new conflict detecting instructions that are supported in Intel AVX-512 instruction set and provide a programming interface to facilitate the vectorization of such associative irregular applications. Compared with previous approaches, in-vector reduction eliminates a large part of the overhead of data reorganization and achieves high SIMD utilization even under adverse input distributions. The evaluation results show that our approach is efficient in vectorizing a diverse set of irregular applications, including graph algorithms, particle simulation codes, and hash-based aggregation. Our vectorization achieves 1.5x to 5.5x speedups over the original sequential codes on a single core of Intel Xeon Phi and outperforms a competing approach, conflict-masking based vectorization, by 1.4x to 11.8x.

CCS Concepts • **Computer systems organization → Single instruction, multiple data**; • **Software and its engineering → Massively parallel systems**;

Keywords SIMD parallelization, irregular applications

CGO'18, February 24–28, 2018, Vienna, Austria

ACM ISBN 978-1-4503-5617-6/18/02…$15.00
https://doi.org/10.1145/3168827

ACM Reference Format:
Peng Jiang and Gagan Agrawal. 2018. Conflict-Free Vectorization of Associative Irregular Applications with Recent SIMD Architectural Advances. In *Proceedings of 2018 IEEE/ACM International Symposium on Code Generation and Optimization (CGO'18).* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3168827

1 Introduction

SIMD or fine-grain parallelism has been a common feature in popular processors for many years. Because of the requirements of aligned and continuous memory accesses, early SIMD units were considered suitable only for accelerating *regular applications* such as dense matrix multiplication and pixel-wise image processing [4, 13]. As wider SIMD vectors and more flexible SIMD instructions (e.g., *gather/scatter* for indirect memory accesses) are being incorporated into processors in recent years, there has been a growing interest in exploiting SIMD features to accelerate *irregular applications* such as graph algorithms, particle simulation codes, and certain types of database operations [1, 11, 17, 18, 25].

In this work, we focus on exploiting the recent SIMD architectural advances to accelerate a class of irregular applications that involve associative irregular reductions. An example of such applications is shown in Figure 1, which is a code snippet of the inner loop of PageRank. Array $n1$ and $n2$ store the source and sink indices of all the edges in a graph. In each iteration, the *rank* of a vertex is divided and distributed to all of its neighbors. While the division of *rank* can be safely conducted in SIMD, the summation is an irregular reduction. Because multiple SIMD lanes may write to the same position in array *sum*, the conflicting updates have to be conducted one-by-one to ensure correctness, which will impede the vectorization performance. Similar to the example of PageRank, a broad class of irregular applications – including *wave-frontier* based graph algorithms [5, 11] and particle simulations code [6] – have irregular reductions at the core.

There are two existing approaches for vectorizing irregular reductions in existing works: *inspector/executor* and *conflict-masking*. Inspector/executor, as the name suggests, comprises two phases. The first or the *inspector* phase examines the memory access patterns of an irregular reduction and reorganizes the data layout to ensure that no conflicts will occur during SIMD processing. Then, the *executor* phase

```
// iterate over all edges
for(int j=0;j<nedges;j++) {
  // obtain vertex indices
  int nx = n1[j];
  int ny = n2[j];
  // add up rank values
  sum[ny] += rank[nx] / nneighbor[nx];}
```

Figure 1. A code snippet of the inner loop of PageRank

performs the computations safely in SIMD. Applied to the PageRank example, the edges are first reordered to ensure that no 16 consecutive entries of *n2* have identical values (assuming the SIMD vector has 16 lanes). Then, the loop is vectorized as DOALL with the assurance that no conflicting writes to *sum* will occur. An example and realization of inspector/executor is the *tiling-and-grouping* technique proposed by Chen *et al.* [1, 11]. They tile the edges in a graph to improve data locality and group the edges in each tile to remove data conflicts for SIMD processing. While their method is effective in resolving data conflicts, the overhead of data reorganization is simply assumed to be amortizable over the iterations.

Another approach to handling the data conflicts among SIMD lanes is *conflict-masking*. Instead of resolving the data conflicts by reordering the input data, conflict-masking identifies the non-conflicting lanes in SIMD vectors at runtime and only writes these non-conflicting lanes in each round of the execution. As an example, Polychroniou *et al.* [18] utilize SIMD features to accelerate building of a hash table. They first check if there are any conflicts among the multiple insertions by a *gather-after-scatter* technique, and then only write the key/value pairs on the non-conflicting lanes. The conflicting lanes are *masked* as invalid during a particular iteration and deferred to the next round of processing. To facilitate conflict-masking, a conflict detection instruction (*vpconflict*) has been even added in the new Intel AVX-512 instruction set [23]. However, the key to the performance of conflict-masking is not the conflict detection itself but the SIMD utilization, which is determined by the input distribution. In the worst case, when all (or most) of the lanes in a SIMD vector are writing to a single memory location, conflict-masking approach is (almost) the same as sequential execution because the lanes have to be processed one-by-one.

In this paper, we propose an *in-vector* reduction technique that can efficiently vectorize a class of associative irregular applications. The idea is that since the order of reduction does not affect the correctness (due to associativity), we can first perform a *partial reduction* within a SIMD vector, and then write the non-conflicting lanes that hold the partial reduction results safely to the main memory. We present an efficient implementation of in-vector reduction by exploiting the conflict detection instruction (*vpconflict*) in the Intel AVX-512 instruction set. Our approach greatly reduces

```
// iterate over all active edges
for(int j=0;j<active_vertices.size();j++) {
  // obtain vertex indices
  int nx = n1[j];
  int ny = n2[j];
  float dx = dis[nx];
  float w = weight[j];
  // relax distance of ny
  if(dis_new[ny] > dx + w) {
    dis_new[ny] = dx + w;
    active_vertices.add(ny);}}
```

Figure 2. The inner loop of Wave-frontier SSSP

the overhead of data reorganization compared with inspector/executor approach, and ensures high SIMD utilization even under adverse input distributions (unlike the conflict-masking approach).

We evaluate the efficiency of in-vector reduction in different irregular applications including PageRank, *wave-frontier* graph algorithms, Molecular Dynamics and Hash-based aggregation, on a single core of Intel Xeon Phi Knight Landing (KNL) processor. The results show that our approach eliminates the overhead of data reorganization while keeping/improving the the speedups by SIMD, and it outperforms conflict-masking on these irregular applications by 1.4x to 11.8x.

2 Background

This section provides background on recent SIMD architectural advances, associative irregular applications, and a commonly used technique for resolving data conflicts in SIMD processing – conflict-masking.

2.1 Recent SIMD Architectural Advances

The methods we present in this paper are based on the AVX-512 instruction set proposed by Intel in July 2013. The instruction set is currently supported in Intel Xeon Phi Knights Landing and Xeon Skylake processors and will be supported in future Skylake-X Core i7 and i9 models [19].

AVX-512 has a family of *gather/scatter* primitives for loading/storing data at unaligned and non-continuous memory addresses. It also supports a *mask* data type with a set of *mask* operations, which allow computations on only a specified subset of lanes within a SIMD vector. Particularly, there is a class of *mask_gather/mask_scatter* instructions that allow reading/writing data on specified lanes of a SIMD vector. These features have enabled a broad class of irregular applications to benefit from SIMD [1, 10, 20, 21].

More important to our discussion in this paper is a set of conflict detection instructions (*vpconflict*) in AVX-512 that can detect conflicting updates among different lanes of a SIMD vector. The instruction tests each element in the index vector for equality with all preceding elements, starting from the least significant bit. The output is a vector in which the value of each lane represents whether that lane has conflicts

```
idx = v_load(idx_arr);
m_todo = 0xFFFF;
do {
1.  old_val = v_gather(d_arr, idx);
2.  compute m_todo
3.  m_safe = get_conflict_free_subset(idx,
m_todo);
4.  compute new_val
5.  v_mask_scatter(m_safe, d_arr, idx,
new_val);
6.  update idx based on m_safe
} while (idx is valid);
```

Figure 3. Overview of conflict-masking approach

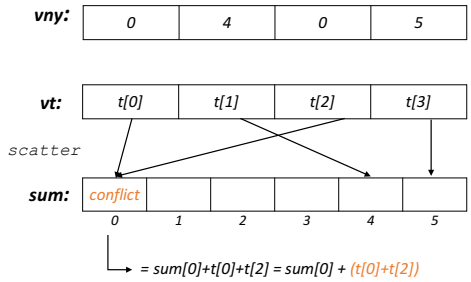


Figure 4. One snapshot of SIMD processing of PageRank in Figure 1: vt is a vector of intermediate results of $rank[nx]/nneighbor[nx]$

with other lanes. If the jth bit in the ith lane is set, it indicates that the ith lane has the same value with the jth lane in the index vector.

2.2 Associative Irregular Reductions

Chen *et al.* [1, 11] have recently shown that a broad class of irregular applications follow a *Sparse Matrix View*. The computation in these applications is (iteratively) conducted on the non-zeros of the sparse matrix, and the processing of each non-zero involves reading and updating values in the row and/or column nodes. An important property is that the updates in these nodes are associative in most cases. An example is the inner loop of PageRank shown in Figure 1. The loop iterates over all the edges in a graph that can be seen as the non-zeros of a sparse matrix. In each iteration, an edge is processed with memory accesses to array *sum*, *rank*, and *nneighbor* indexed by its row and column number *nx* and *ny*. The updates to array *sum* is a summation, which is an associative operation.

2.3 Conflict-masking

Another example is the wave-frontier graph algorithms. Figure 2 shows the inner loop of wave-frontier SSSP. Here, $n1$ and $n2$ are indirection arrays that store the vertex indices for the two end-points of each edge in a graph. The loop iterates over all the active edges in the graph. In each iteration, it first obtains the indices of x and y, the two end points, and then accesses the current distances of x and y and the weight of the edges. The program next checks if the distance of y is greater than the sum of distance of x and the weight of the edges. If so, the distance of y needs to be updated and y should be added to the *active_vertices* list for the next round of computation. The computation here also follows the Sparse Matrix View, despite that the non-zeros in the sparse matrix are the frontier vertices and constantly changing. The updates to array *dis_new* is a minimum operation which is also associative.

In addition to the above two graph algorithms, there are many other irregular applications that follow the Sparse Matrix View and involve associative irregular reductions. Such applications include many wave-frontier based graph algorithms such as Weakly Connected Component (WCC) and Single Source Widest Path (SSWP), particle simulation applications such as Molecular Dynamics and unstructured grid-based solver like Euler [6], and histogram applications such as group-by aggregation used in databases and data analytics.

Conflict-masking is a general solution to the data conflicts problem in SIMD parallelization, and thus used as a baseline to compare our approach against. Figure 3 shows the general workflow of conflict-masking approach on irregular applications involving conflicting memory access among SIMD lanes. The program iterates over all the input data. In each iteration, it first gathers data from the data array based on the index vector. Next, it determines which lanes need to be updated, and identifies the conflict-free lanes in the index vector among those lanes that need to be updated. The method then computes the new values on these lanes and writes their values to the conflict-free locations. Finally, it fills the computed lanes of the index vector and the data vector with new inputs for next iteration.

The vectorization performance of conflict-masking is dependent on the input distribution. If conflicts arise frequently in the input, conflict-masking will result in low SIMD utilization and thus poor vectorization performance. Thus, it can be beneficial to reorder the inputs to reduce data conflicts [1]. However, such data reorganization incurs overheads, and it is possible that the overhead can outweigh the SIMD speedups [10].

3 Conflict-free Vectorization of Associative Irregular Reductions

Instead of using conflict-masking, we propose an *in-vector reduction* approach that can resolve data conflicts in SIMD processing for associative irregular reductions.

3.1 Overview

Consider again the irregular reduction in PageRank (Figure 1). A straightforward vectorization of the loop involves processing multiple edges in the SIMD vector and then updating multiple values in the *sum* array simultaneously. Figure 4 shows one step of the SIMD processing and the memory accesses involved. For simplicity, we use vector width of 4 in

Algorithm 1 Invector_Reduction ($active$, $vindex$, $vdata$)

```
1:  mret ← v_get_conflict_free_subset(active, vindex)
2:  msafe ← mret
    ▷ Iterates over all conflicting lanes
3:  for each unset bit i in msafe do
        ▷ Get the lanes that have an identical index with vdata[i]
4:      mreduce ← v_compare_eq(vdata, vdata[i])
        ▷ Reduce these lanes
5:      res ← v_horizontal_reduce(mreduce, vdata)
        ▷ Find the first one of these lanes
6:      res_pos ← mreduce&(~mreduce + 1)
        ▷ Store the reduction result to that first lane
7:      vdata[res_pos] ← res
        ▷ Invalidate the reduced lanes
8:      msafe ← msafe | mreduce
9:  end for
10: return mret
```

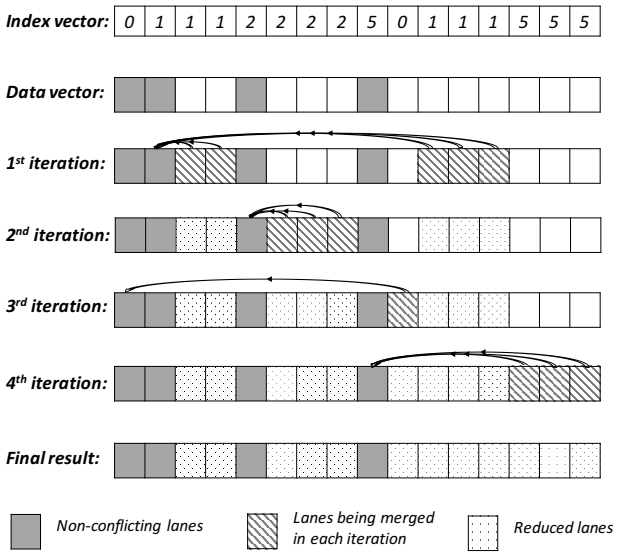


Figure 5. A running example of Algorithm 1

this illustration. Here, vny is one of the two index vectors that store the indices of the end-points of edges. The calculations involved produce the result vector vt, which needs to be updated to the array sum based on the indices in vny. As shown in the figure, a simple `scatter` of vt will cause conflict at $sum[0]$ since both $t[0]$ and $t[2]$ need to be added to $sum[0]$. However, using associative property, we can sum up $t[0]$ and $t[2]$ first and add the result to $sum[0]$ without impacting correctness. More generally, for vectorizing associative irregular reductions, we can reduce the values within the SIMD vector first and scatter the reduced vector into the main memory. We name this approach *in-vector reduction*.

Because the reduced vector contains only distinct indices and is conflict free for writing, in-vector reduction resolves the data conflicts among SIMD lanes without any data reorganization. This makes the SIMD processing of the *computation part* of these associative irregular applications (such as the computation of vt in the above example) have high utilization regardless of the input distribution. The only overhead incurred is for performing the in-vector reduction. We next give an efficient implementation of in-vector reduction based on the new *conflict detection* instruction from AVX-512 and discuss its overhead in Section 3.3.

3.2 Implementation of In-vector Reduction

We exploit the *conflict detection* and *horizontal reduction* instructions in Intel AVX-512 to implement in-vector reduction. The implementation is shown in Algorithm 1. The algorithm first obtains the *non-conflicting* subset of active lanes in the index vector, which are represented by set bits in a mask $mret$. These lanes will be used to store the partial reduction results from conflicting lanes, and then be updated to memory without conflict. This step ($v\_get\_conflict\_free\_subset$) is implemented directly with the *vpconflict* instruction followed by a comparison with a zero vector. The *vpconflict* instruction compares each lane in a vector with all of its preceding lanes from the least significant bit. If a match is found, the corresponding bit on that lane of the result vector will be set to 1 (all the bits are initially set to 0). The lanes that have the value zero in the result vector are the lanes that have no conflicting lanes ahead of them, and thereby form a non-conflicting subset. A safe mask that represents these non-conflicting lanes is then obtained by comparing the result vector with a zero vector. Next, the algorithm iterates over the unset bits of the safe mask, i.e., those representing conflicting lanes. For each unset bit, it compares the value on that particular lane with other lanes in the data vector and returns a *reduction mask*. This mask represents the matching lanes that have an identical index. Then, using the reduction mask, the algorithm uses the horizontal reduction instruction to get the merged result for these matching lanes. This is followed by calculating the position of the first lane among these matching lanes (which is the first set bit in $mreduce$ from the least significant bit) and moving the reduced result to that first lane in the data vector. Finally, the algorithm sets the bits in safe mask that are set in reduction mask, indicating that these reduced lanes are no longer useful.

As a demonstration, Figure 5 shows how Algorithm 1 is executed on a data vector with an index vector. Suppose all the SIMD lanes are active. Initially, the non-conflicting lanes are identified, as shown by the shadowed cells. In the first iteration, starting from the first conflicting lane, which is the third lane, the algorithm identifies all the lanes with the same index and merges them into the first one among them, which is the second lane. The algorithm then inactivate all the other merged lanes. In the second iteration, since the fourth lane has been masked out, the next conflicting lane is the sixth lane. The algorithm identifies all the lanes that have an identical index with the sixth lane and merges their values to the first one, which is the fifth lane. After four

Algorithm 2 Invector_Reduction2 ($active$, $vindex$, $vdata$)

```
1: mret1 ← v_get_conflict_free_subset(active, vindex)
2: msafe1 ← mret1
3: mret2 ← v_get_conflict_free_subset(active&~msafe1,
       vindex)
4: msafe2 ← mret2
   ▷ Iterates over all conflicting lanes
5: for each unset bit i in msafe2 do
       ▷ Get the lanes that have an identical index with
   vdata[i], excluding those in the second subset
6:     mreduce ← v_compare_eq(~msafe2, vdata, vdata[i])
       ▷ Reduce these lanes
7:     res ← v_horizontal_reduce(mreduce, vdata)
       ▷ Find the first one of these lanes
8:     res_pos ← mreduce&(~mreduce + 1)
       ▷ Store the reduction result to that first lane
9:     vdata[res_pos] ← res
       ▷ Invalidate the reduced lanes
10:    msafe2 ← msafe2 | mreduce
11: end for
   ▷ Store part of the results into the auxiliary array
12: v_mask_scatter(aux_array, mret2, vindex, vdata)
13: return mret1
```

iterations, all of the conflicting lanes in the data vector have been processed.

3.3 Overheads

As discussed in Section 3.1, in our approach, the SIMD processing of the computation part of associative irregular applications is assured to have 100% SIMD utilization. Thus, the key factor to the overall vectorization performance is the overhead of in-vector reduction. In our implementation, since we sum up all of the lanes with an identical index in each iteration, the number of iterations for Algorithm 1 to process all conflicting lanes is equal to the number of *distinct* conflicting lanes, which is no more than half of the total number of lanes in the vector. According to the actual implementation, there are about eight instructions in each iteration of Algorithm 1 and the first line takes two instructions, so an invocation of Algorithm 1 takes no more than $2 + 8 \times D_1$ instruction where D_1 is the number of distinct conflicting lanes. We will further evaluate and compare the overheads of Algorithm 1 experimentally with a similar functionality (*reduce_by_key*) provided by an optimized C++ library in Section 4.5.

3.4 Optimization

On an Intel Xeon Phi, a SIMD vector can accommodate 16 integers or single-precision floats. In the worst case, there are up to eight distinct conflicting lanes in a vector, costing Algorithm 1 eight iterations to complete an in-vector reduction, or up to 66 total instructions. This overhead may still

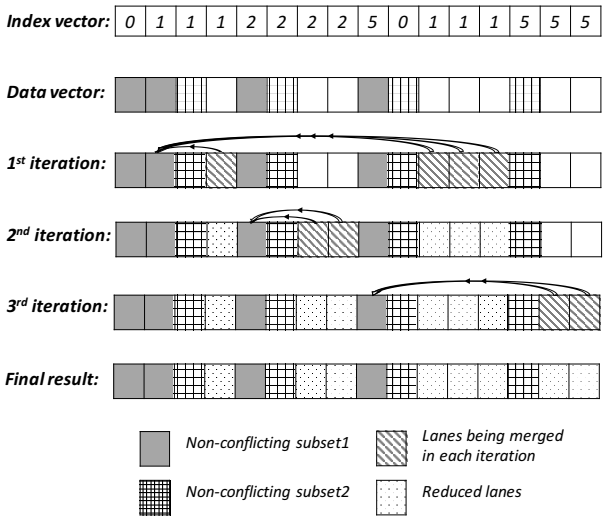


Figure 6. A running example of Algorithm 2

be high compared with the cost of actual computation. Although the worst case rarely happens in practice, in general, more iterations in Algorithm 1 incur more overhead.

The overhead of Algorithm 1 can be reduced at the cost of more memory consumption, as we now describe. The idea is to have an auxiliary reduction array. Then, instead of resolving all of the conflicting lanes in a SIMD vector by merging them with the non-conflicting lanes, we can treat the subset of conflicting and non-conflicting lanes separately and reduce them to two different arrays. Algorithm 2 shows an optimized implementation of in-vector reduction. It first obtains non-conflicting subset of lanes in the index vector, represented by set bits in $msafe1$. Then, it gets non-conflicting lanes within the subset of conflicting lanes obtained in the first step and represent these non-conflicting lanes as set bits in $msafe2$. The two masks, $msafe1$ and $msafe2$, represent two non-conflicting subsets of lanes that will be updated to two different reduction arrays. The two reduction arrays need to be merged later to achieve the final results. The algorithm then iterates over the unset bits of $msafe2$, which represent the conflicting lanes in the conflicting subset. These conflicting lanes are merged with other lanes that have identical indices, excluding the lanes in the subset of $msafe2$ – the latter because otherwise some of the lanes in $msafe2$ will be invalidated during the process. The merged result is written to the first lane that is represented by the first bit in $mreduce$ from the least significant bit. Finally, $msafe2$ is updated with $mreduce$ to invalidate the reduced lanes.

Figure 6 illustrates how Algorithm 2 works on the data vector and index vector of Figure 5. Initially, the algorithm identifies the non-conflicting subset of lanes in the entire vector, as shown by the shadowed cells in Figure 6. It then identifies the non-conflicting lanes among the conflicting subset of the first step, as shown by the small-grid cells. The remaining lanes are the lanes need to be merged to

```
// iterate over all edges
for(int j=0;j<nedges;j+=16) {
  // obtain vertex indices
  vint vnx, vny;
  vnx.load(n1+j);
  vny.load(n2+j);
  // add up rank values
  vfloat vrankx, vnnx, vsumy, vadd;
  vrankx.load(rank, vnx, 4);
  vnnx.load(nneighbor, vnx, 4);
  vsumy.load(sum, vny, 4);
  vadd = vrankx / vnnx;
  // in-vector reduction of vadd,
  // m indicates the conflict-free lanes
  mask m = invec_add(0xFFFF, vny, vadd);
  vsumy += vadd;
  // store the conflict-free lanes safely
  // to memory
  Mask::set_mask(m, vzero);
  vsumy.mask().store(sum, vny); }
```

Figure 7. Vectorized inner loop of PageRank with our API

other lanes excluding the small-grid ones. As shown in the figure, the merge can be done in three iterations, which is one fewer than Algorithm 1. The final results are stored in the shadowed and small-grid cells that will be updated to two separate arrays.

Algorithm 2 ensures that the number of iterations is no more than one-third of the total number of lanes in a vector. Since the algorithm require one more $get\_conflict\_free\_subset$ (two SIMD instructions) and three more instructions for merging the two copies of reduction results, an invocation of Algorithm 2 takes about $7 + 8 \times D_2$ instructions where D_2 is the number of distinct conflicting lanes in the conflicting subset. When the number of lanes in a SIMD vector is 16, Algorithm 2 takes no more than 47 instructions as $D_2 <= \lfloor 16/3 \rfloor$. As an extreme example, if a vector has two identical groups of eight distinct lanes, Algorithm 1 needs 8 iterations to finish the in-vector reduction, while Algorithm 2 needs none since the two subsets of eight conflict free lanes can be safely updated to two arrays.

In our framework, we decide the underlying implementation of in-vector reduction between Algorithm 1 and 2 based on the average number of distinct conflicting lanes in the first few iterations of an application. Specifically, Algorithm 2 is invoked only when $2 + 8 * D_1 > 7 + 8 * D_2$ which implies $D_1 > D_2 + 0.625$. Though the worst case D1 is 8, it rarely happens in real applications. In our evaluation, the graph applications have a very small D_1 of 10^{-4} in average. Only for hash-based aggregation, D_1 can reach 4, and in this case, Algorithm 2 has clear advantage over Algorithm 1 and achieves D_2 of about 1. Overall, Algorithm 1 is more efficient than Algorithm 2 in practice. So we use Algorithm 1 as default implementation and simply change the invocation to Algorithm 2 when D_1 is greater than 1.

3.5 Programming Interface

To facilitate the vectorization of associative irregular applications, we introduce an API to simplify the use of in-vector reduction. The API is built on a framework[1] for SIMD programming on Intel Xeon Phi processors without explicitly writing Intel Intrinsics [7].

Figure 7 shows the vectorized code of the inner loop of PageRank with the API. The *vint* and $vfloat$ represent SIMD vectors of integer and floating-points. The *load* methods load or gather the data from main memory to SIMD vectors. The *store* methods store or scatter the data from SIMD vectors to memory. We embed our in-vector reduction as functions in the framework. The functions have a prototype of

$$mask\ invec\_op(mask\ active,\ vint\ idx,\ vtype\ data)$$

Here, *op* are different types of reduction operations, *active* is the mask of active lanes, *idx* is the vector storing reduction indices, and *data* are vectors of either integers or floating-points that store the real data values, which are reduced in-place by the function. The return value of the function is a mask indicating the conflict-free lanes that store the partial reduction results.

As shown in Figure 7, *invec_add* sums up the lanes in *vadd* according to the indices in *vny*. The returned value *m* indicates the active lanes that need to be written to array *sum*. Because the lanes indicated by *m* have distinct index values, no conflicting writes to *sum* will occur. As the SIMD lanes are fully utilized in computing *vadd* and the number of iterations for the loop to finish processing all of the edges is $nedges/16$, we can see that the computation in the vectorized code has 100% SIMD utilization.

The wave-frontier SSSP in Figure 2 can also be vectorized with our API in a similar way. In each iteration, *invec_min* is invoked to reduce the new distances in the SIMD vector and then the minimum new distances of vertices are scattered to memory without conflicts. As we will show in the next section, a broad class of associative irregular applications can be efficiently expressed and vectorized with our API.

4 Evaluation

In this section, we evaluate the efficiency of *in-vector reduction* with a diverse set of irregular applications, including graph algorithms, particle simulation, and hash-based data aggregation. We compare the performance of in-vector reduction with two existing approaches, an *inspector/executor* approach (specifically one that uses *tiling-and-grouping* [1]), and *conflict-masking*. Specifically, we focus on (1) the data reorganization overhead saved by in-vector reduction and how it affects the overall performance, and (2) how in-vector reduction performs compared with conflict-masking.

Our experiments are conducted on an Intel Xeon Phi 7250 (Knight Landing) processor, which comprises 68 cores running at 1.4 GHz, each with four hardware threads. The main memory is a traditional DDR4 RAM with 96GB. We use Intel

[1] https://github.com/lcchen008/irreg-simd/tree/master/SSE_API_Package/

ICC compiler 17.0.0 to compile all the codes, with -O3 optimization enabled. Because our focus is on vectorization, all programs are executed on a single-core of an Intel Xeon Phi KNL. MIMD parallelization is a tangential issue and beyond the scope of our current work.

4.1 Applications and Datasets

Table 1 lists the applications and datasets used in our experiments. We select four graph algorithms: PageRank, Single Source Shortest Path (SSSP), Single Source Widest Path (SSWP), and Weakly Connected Component (WCC). The essential computation in PageRank has been described in Section 1. SSSP, SSWP, and WCC are all implemented in an edge-centric manner with wave-frontiers. The computation in SSSP was also discussed earlier in Section 2.2. SSWP is an algorithm for finding paths between a source vertex to other vertices in a weighted graph, maximizing the weight of the minimum-weight edge in the paths. SSWP has a similar computation pattern to SSSP, with the difference that it relaxes the width of a vertex if an incoming edge adds a wider path to that vertex. WCC is an algorithm for finding maximal subgraphs in a directed graph such that for every pair of vertices there is a path from one to another. The inner loop of WCC iterates over all active edges in the graph. In each iteration, it sends the index of the incoming vertex to the outgoing vertex, and furthermore, adds the outgoing vertex to the active list if the incoming index is smaller. The graphs used in the four applications are from the SNAP [14] graph datasets, and they are stored as sparse matrices.

Molecular Dynamics (Moldyn) is chosen as our test case of particle simulation. It simulates the interaction and motion of molecules in a period of time based on Newton's law. The simulation consists of a sequence of iterations. In each iteration, the coordinates of the molecules are updated first, and then the forces among the molecules are computed according to their distances. Finally, the velocities are computed based on the forces, which are then used in the next iteration for updating the coordinates. The inputs are generated by the program that was distributed with the original serial code of Moldyn.

Aggregation is an important and expensive operation for summarizing large amounts of data in databases, MapReduce frameworks, and statistical languages. The input data is usually in the form of key/value pairs. Aggregation involves dividing the data into groups of identical keys and merging the values in each group. We use Hash-based aggregation, which is a commonly implemented algorithm for data aggregation. The inputs in our evaluation are randomly generated under three skewed data distributions [3]: heavy-hitter, Zipf, and moving cluster. In the *heavy hitter* input (`HHitter`), one value account for 50% of the group-by keys, while the other values are chosen uniformly from the other group-by keys. The Zipf uses an exponent of 0.5. In the moving cluster input (`MovCluster`), the size of the window of data locality (which gradually shifts) is 64.

Table 1. Applications and datasets used in the experiments

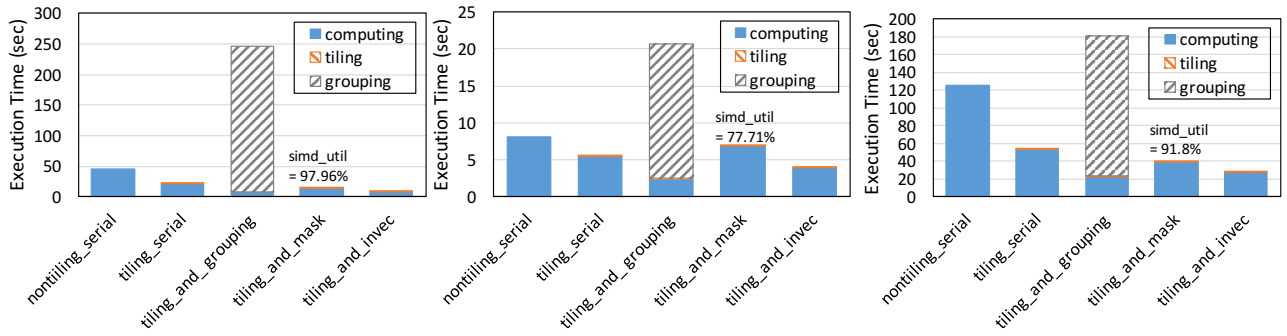
	App	Dataset	Dimensions	NNZ
Graph Algorithms	PageRank	higgs-twitter	457K*457K	15M
		soc-Pokec	1.6M*1.6M	31M
		amazon0312	401K*401K	3.2M
	SSSP	higgs-twitter	457K*457K	15M
		soc-Pokec	1.6M*1.6M	31M
		amazon0312	401K*401K	3.2M
	SSWP	higgs-twitter	457K*457K	15M
		soc-Pokec	1.6M*1.6M	31M
		amazon0312	401K*401K	3.2M
	WCC	higgs-twitter	457K*457K	15M
		soc-Pokec	1.6M*1.6M	31M
		amazon0312	401K*401K	3.2M
Particle Simulation	Moldyn	16-3.0r	131K*131K	11M
		32-3.0r	365K*365K	30M
Data Aggregation	Hash-based Aggregation	Zipf	1*32M	32M
		heavy-hitter	1*32M	32M
		move-cluster	1*32M	32M

For each of these applications, we implemented a serial version as the baseline. For graph algorithms and particle simulation, the serial version is executed on both original data (`nonTiling_serial`) and tiled data (`tiling_serial`) to show the benefit as well as overheads of tiling. To compare the different vectorization approaches, we implemented multiple vectorized versions for each application. One version uses tiling-and-grouping (`tiling_and_grouping`), one version uses conflict-masking (`tiling_and_mask`), and one version uses our in-vector reduction (`tiling_and_invec`). For hash-based aggregation, our implementations are based on two types of hash tables: a linear probing hash table and a bucketized hash table that is designed to mitigate the conflicts in SIMD vector [10]. We use the serial code based on the linear probing hash table `linear_serial` as the baseline, and compare the performance of conflict-masking on both types of hash table implementations (`linear_mask` and `bucket_mask`) and in-vector reduction on both types of hash table (`linear_invec` and `bucket_invec`).

4.2 Results from Graph Algorithms

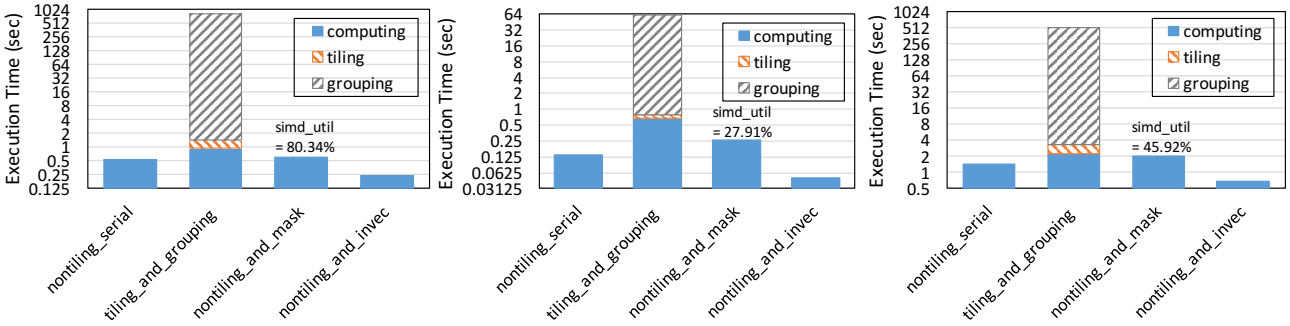
All of the four graph applications we evaluate have a termination condition. Since the ranks of the vertices converge to a stable value in PageRank, we set the terminating criteria as the change of rank values being less than 0.1%. SSSP, SSWP and WCC stop when the there is no active vertex in the graph.

Figure 8 shows the overall execution time of different versions of PageRank on different input graphs. The numbers of iterations before convergence is also listed. As we can see from the figure, tiling has a very tiny (almost unobservable) overhead compared with the total execution time, and yet `tiling_serial` runs 1.5x - 2.5x faster than `nontiling_serial`, indicating that tiling is cost-effective in improving data locality. Among the vectorized versions, `tiling_and_grouping` has the shortest computing time, which is reasonable since the edges are processed with high SIMD utilization and without conflicts, but the grouping overhead



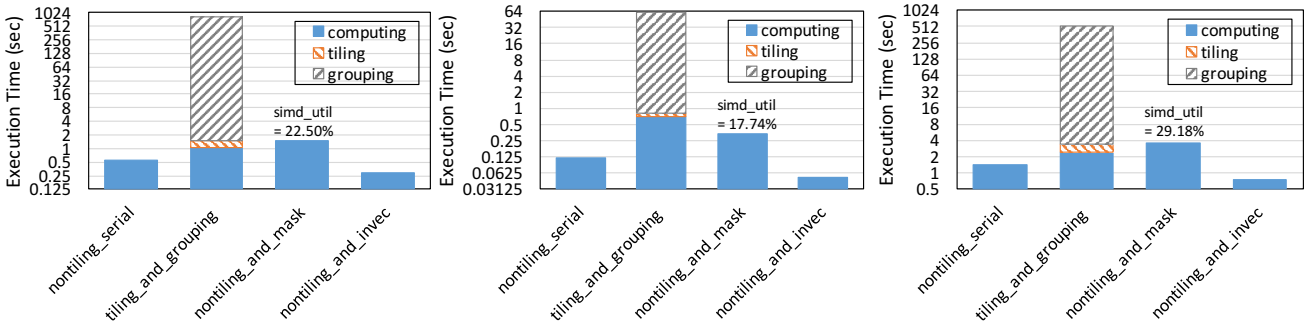
(a) higgs-twitter (conv_iter=116) **(b)** amazon0312 (conv_iter=125) **(c)** soc-pokec (conv_iter=110)

Figure 8. Overall performance of of different versions of PageRank on different inputs



(a) higgs-twitter (conv_iter=13) **(b)** amazon0312 (conv_iter=42) **(c)** soc-pokec (conv_iter=13)

Figure 9. Overall performance of of different versions of SSSP on different inputs

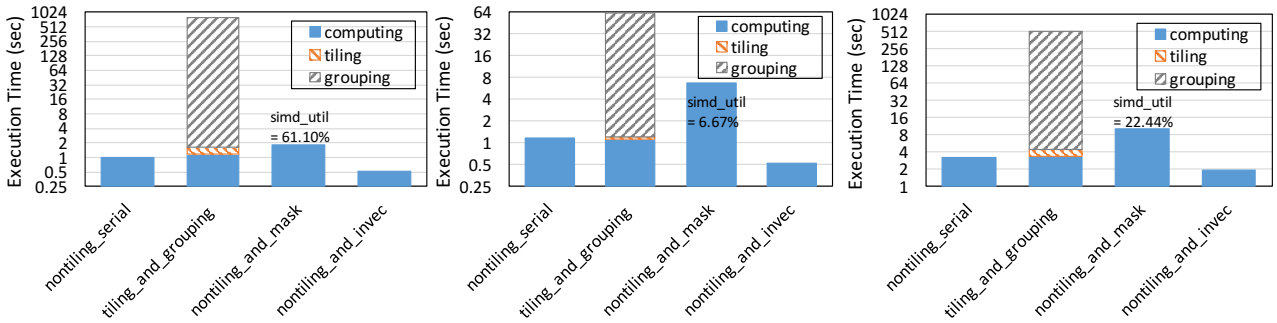


(a) higgs-twitter (conv_iter=13) **(b)** amazon0312 (conv_iter=42) **(c)** soc-pokec (conv_iter=13)

Figure 10. Overall performance of of different versions of SSWP on different inputs

incorporated is so high that it denies the benefit of vectorization, ending up with an overall execution time even much longer than the original serial version. The performance of conflict-masking approach depends on the input distribution, as `tiling_and_mask` achieves about 1.5x speedup against `tiling_serial` on higgs-twitter and soc-pokec datasets but runs slower than `tiling_serial` on amazon0312 due to a relatively low SIMD utilization. Our approach `tiling_and_invec` outperforms conflict-masking by 1.4x to 1.8x, and achieves 1.5x and 2.3x speedups against `tiling_serial`, which are close to the speedups from `tiling_and_grouping` when the grouping overhead is ignored.

Figure 9 shows the overall execution time of different versions of wave-frontier SSSP on different input graphs. Note that because the grouping overhead is much larger than the computing time in this case, the y-axis in this figure uses a base-2 log scale. The serial version `nontiling_serial`, conflict-masking version `nontiling_and_mask` and



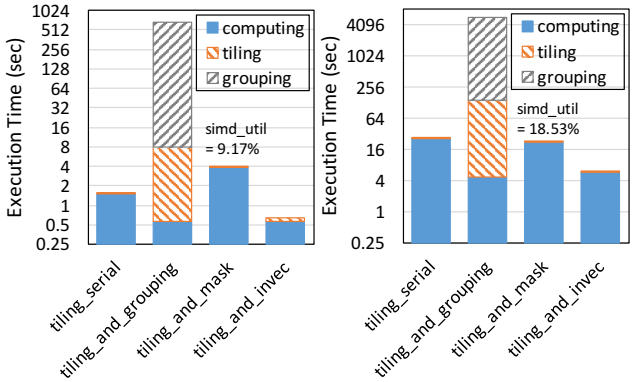
(a) higgs-twitter (conv_iter=13) **(b)** amazon0312 (conv_iter=42) **(c)** soc-pokec (conv_iter=13)

Figure 11. Overall performance of of different versions of WCC on different inputs

in-vector reduction version `nontiling_and_invec` are all evaluated with the original graphs without tiling for the comparison of SIMD performance, because the active edges in the wave-frontier SSSP algorithm change over iterations and an initial tiling does not work for following iterations. We can see that due to the poor SIMD utilization, `nontiling_and_mask` runs even slower than the serial code – only on the higgs-twitter dataset, `nontiling_and_mask` has a similar computing time with `nontiling_serial`. Our in-vector reduction `nontiling_and_invec`, on the other hand, achieves 2.2x - 2.7x speedups against the serial version, showing a clear advantage over the conflict-masking approach – it outperforms `nontiling_and_mask` by 2.3x to 11.8x. Another version in our comparison is the tiling-and-grouping approach, which is a technique of reusing tiling-and-grouping proposed by Jiang *et al.* [11]. Despite the huge amount of overhead for grouping, the SIMD computation of this version does not even have any speedups in our evaluation. We also test the `tiling_and_grouping` code on a Knight Corner coprocessor, and it does achieve 2x - 3x speedups against the serial code on the older version of Intel Xeon Phi. The performance difference may come from the significant improvement of sequential performance from KNC to KNL. The tiling overhead here is also larger than that for PageRank, because an index building procedure is associated with tiling in order to reuse the data reorganization [11].

Figure 10 and 11 show the execution time of SSWP and WCC on different datasets. The execution time patterns are similar to that of SSSP. Our in-vector outperforms the serial code by 1.9x - 2.2x in SSWP and by 1.6x - 2.1x in WCC, and it is the only one among the approaches we are comparing that can deliver SIMD speedups to the wave-frontier graph algorithms on the new Intel KNL processor.

From the performance results, we can see that our in-vector reduction approach save a large part of the overhead of data reorganization, and it outperforms conflict-masking approach even when the SIMD utilization is more than 90%. When the SIMD utilization becomes lower due to adverse input distribution, in-vector reduction shows more advantage over conflict-masking.



(a) 16-3.0r (iter=20) **(b)** 32-3.0r (iter=20)

Figure 12. Execution time of of different versions of Molecular Dynamics running 20 iterations on different inputs

4.3 Results from Molecular Dynamics

Figure 12 shows the execution time of four versions of Molecular Dynamics running 20 iterations. Note that the y-axis here uses a log-2 scale. Because the neighbor list rebuilding is set to take place in every 20 iterations and every neighbor list rebuilding is associated with tiling in all of the four versions, we evaluate the computing time of 20 iterations. The overhead of tiling once is added to the reported execution times. As we can see from the figure, `tiling_and_grouping` has the lowest computing time, achieving high SIMD utilization, and obtaining 2.69x and 5.46x speedups against the serial code on the two inputs. However, it requires a time-consuming grouping procedure – the simulation needs to run nearly 1000 iterations to amortize the overhead of an initial grouping. Also, compared with other versions, `tiling_and_grouping` takes a longer tiling time because it needs to update the index data structure for reusing the initial grouping information [11]. Conflict-masking `tiling_and_mask` runs even slower than the serial code on both of the two inputs due to the poor SIMD utilization. Our in-vector reduction approach `tiling_and_invec` has a SIMD performance that

is close to that of `tiling_and_grouping`, outperforming the serial version by 2.59x and 4.43x, respectively, on the two inputs.

4.4 Results from Hash-based Aggregation

The query we choose for evaluating aggregation performance is "`Select G,count(*),sum(V),sum(V*V) From R GroupBy G`" where `R` is a two-column table consisting of a group-by key `G` and an aggregation value `V`. As is the common practice in this domain, we report *throughput* and not the execution time – Figure 13 shows the throughput values of different versions of hash-based aggregation computing this query on different input data. For all the inputs, the straightforward implementation of conflict-masking approach on a linear probing hash table `linear_mask` has the lowest throughput, even lower than the serial baseline `linear_serial`. For most of the inputs, in-vector reduction on a bucket hash table `bucket_invec` has the highest throughput, delivering up to 3.26x speedups against the serial code. However, it does not deliver speedup in certain cases when the grouping cardinality is getting close to the size of hash table and bucket hash table requires more probing time. A bucket hash table tends to have longer probing time than a linear probing hash table of the same size because it has a smaller hashing range. In these cases, `linear_invec` achieves the best performance, obtaining speedups against the serial version from 1.3x to 1.8x. Conflict-masking with the bucket hash table `bucket_mask` does achieve some speedups in several cases, but its performance is dominated by `bucket_invec`, suggesting that our in-vector reduction is a better approach to dealing with conflicts among SIMD lanes in irregular reductions.

4.5 Comparison with *reduce_by_key*

Libraries such as Boost[2] and Thrust[3] have optimized implementations of *reduce_by_key* which has a similar functionality of in-vector reduction. The difference is that in-vector reduction supports reductions on an arbitrary set of lanes that are specified by the *active mask*, while the *reduce_by_key* provided by these libraries have to reduce all of the array elements. To further show the efficiency of our implementation, we compare its performance with the *reduce_by_key* in the latest version of Thrust library (the function available in Boost could not be compiled with ICC on the Intel Xeon Phi processor we target). Because not all of the irregular applications in our experiments have full active lanes in each iteration, *reduce_by_key* cannot be directly used to resolve data conflicts in these applications. To compare the performance, we conduct a simulation of the reductions occurring in many real graph algorithms, where reductions are conducted on the columns of the sparse matrices of the graphs. We test the performance on three graphs from SNAP dataset that was also used in experiments reported earlier in the section. Table 2 lists the execution time of performing 1000 iterations of reductions on all edges of the graphs. Our implementation is about 8.5 times faster than the optimized *reduce_by_key* in Thrust library. The results suggest that, besides it does not have the full functionality of in-vector reduction, *reduce_by_key* is not likely to deliver speedups for vectorizing the associative irregular applications.

[2] http://www.boost.org/doc/libs/1_61_0/libs/compute/doc/html/boost/compute/reduce_by_key.html
[3] https://thrust.github.io/doc/group__reductions.html

Table 2. Execution time of 1000 iterations of reductions on all edges of the graphs

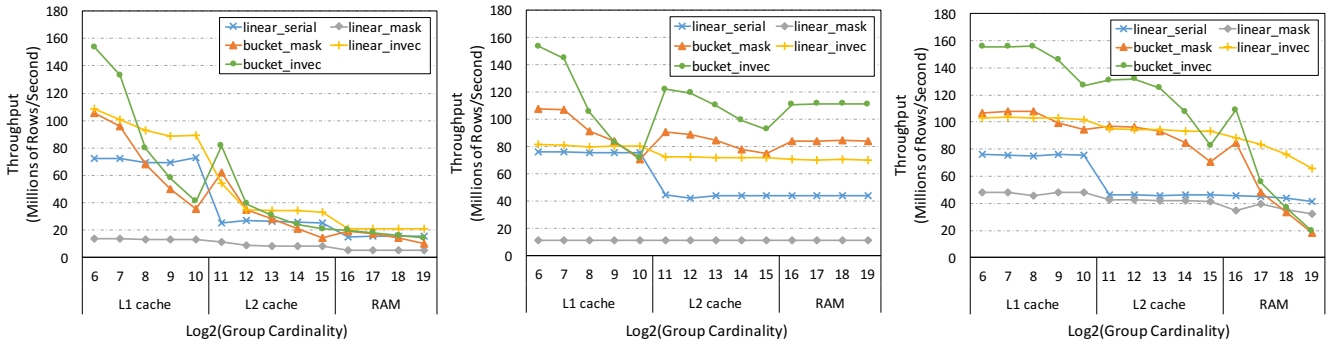
	In-vector Reduction	*Thrust*
`higgs-twitter`	6.99s	57.97s
`amazon0312`	14.73s	123.77s
`soc-pokec`	1.52s	13.59s

5 Related Work

Due to the advances in SIMD architectures, exploiting SIMD to accelerate different classes of irregular applications has received considerable interest in recent years.

Unstructured Mesh and Graph Applications Thébault *et al.* [25] vectorize unstructured 3D mesh computations on an Intel Sandy Bridge and an Intel Xeon Phi KNC. They use METIS to recursively bisect the input to improve data locality and apply a coloring scheme to construct non-conflicting groups for SIMD processing. Pennycook *et al.* [17] compare the efficiency of different gather/scatter implementations (software and hardware) and accelerate Molecular Dynamics on an Intel Xeon Phi KNC. They focus on utilizing the gather/scatter instructions, but do not consider the impact of memory locality to the performance. Saule and Catalyurek [22] provided a preliminary evaluation on graph applications on the Xeon Phi architecture, but without particular optimizations specific to the applications and the hardware. Liu *et al.* [15] use ELLPACK sparse block format to optimize SpMV kernel on the Intel Xeon Phi. Similarly, Tang *et al.* [24] utilize a hybrid storage format with jagged partitioning to optimize SpMV, also for the Intel Xeon Phi. Chen *et al.* [1] target irregular applications that have static memory access patterns. They use tiling-and-grouping to improve memory locality for SIMD gather/scatter operations and to remove the write conflicts among lanes in a SIMD vector. They further propose a reusing technique to extend tiling-and-grouping approach for irregular applications with dynamic and adaptive memory access patterns [11]. Though the computation in these irregular applications is successfully accelerated by SIMD, the work is based on the assumption that the overhead can be amortized over iterations.

There are also many efforts put on GPUs for accelerating these applications. For example, Merrill *et al.* [16] parallelize breadth-first search on the GPUs by focusing on fine-grained task management. CuSha [12] optimizes graph processing on GPUs with intensive usage of shared memory by reorganizing the graph data in shards. Choi et al. [2] propose a way of optimizing SpMV on GPUs by storing sparse matrices into small subblocks, each represented as a dense matrix.



(a) heavy hitter **(b)** Zipf **(c)** moving cluster

Figure 13. Overall performance of of different versions of hash-based aggregation on different inputs

Database Operations Polychroniou *et al.* [18] utilize the SIMD instructions on an Intel Xeon Phi KNC to accelerate most of the basic database operations. They propose a *gather-after-scatter* technique to detect conflicts among SIMD lanes and use it to vectorize hash table operations. Jha *et al.* [9] vectorize hash join on an Intel Xeon Phi KNC, leveraging the gather/scatter instructions and data prefetching features on the coprocessor. However, they fall back to serial code when dealing with the conflicts among SIMD lanes. Inoue *et al.* [8] effectively reduce the branch mispredictions in set intersection and accelerate the operation with SIMD. Compared to these efforts, we have focused on a strategy that can effectively resolve data conflicts for SIMD processing of irregular reductions by exploiting new SIMD instructions.

6 Conclusion

In this paper, we propose an in-vector reduction technique for vectorizing irregular applications with associative irregular reductions. We implement in-vector reduction with conflict detection and horizontal reduction instructions in the new Intel AVX-512 instruction set. The evaluation results show that our in-vector reduction eliminates the expensive grouping overhead and reduces the impact of adverse input distributions while keeping/improving the speedups by SIMD. Compared with serial code, in-vector reduction achieves 1.5x to 2.5x speedups for graph applications, 2.7x to 5.5x for Moldyn, and 1.3x to 3.26x for hash-based aggregation on a single core of Intel Xeon Phi KNL. It also outperforms conflict-masking on these irregular applications by 1.4x to 11.8x.

A Artifact Description

A.1 Abstract

The artifact includes all of the programs that are needed to reproduce the evaluation results in the paper "Conflict-free Vectorization of Associative Irregular Applications with Recent SIMD Architectural Advances". The programs should be executed on an Intel Xeon Phi 7250 (Knight Landing) processor. An Intel ICC compiler is required for compiling. The artifact will print out the execution time or throughput of different versions of the evaluated applications on different datasets, which can be expected to be close to the results reported. The artifact will also print out the SIMD utilizations rate of different applications as reported in the paper.

A.2 Description

A.2.1 Checklist

- **Program:** (1) PageRank, (2) Single Source Shortest Path, (3) Single Source Widest Path, (4) Weakly Connected Component, (5) Molydn, (6) Hash-based Aggregation
- **Compilation:** Intel ICC compiler 17.0.0
- **Data set:** (1) three graphs from SNAP [14], (2) two generated inputs for Moldyn, (3) generated key/value pairs of different group-by cardinalities and distributions for hash-based aggreagation
- **Hardware:** an Intel Xeon Phi 7250 Knight Landing processor
- **Output:** program results including the execution time or throughputs are printed out to the console
- **Experiment workflow:** a bash script is provided in each directory to automatically start the compilation and execution
- **Publicly available?:** Yes

A.2.2 How Software Can Be Obtained

The artifact is publicly available at `https://github.com/jiangohiostate/cgo2018_artifact.git`. Use "git clone" to download the package. The programs are about 12KB.

A.2.3 Hardware Dependencies

An Intel Xeon Phi 7250 (Knight Landing) processor is required.

A.2.4 Software Dependencies

Intel ICC compiler 17.0.0 is required.

A.2.5 Datasets

There is a 'get_data.sh' script in the top directory. It will automatically download and initialize all the datasets described above.

A.3 Installation

In each of the directories, there is a 'run.sh' script, which builds the programs and runs the tests automatically. You can also compile the programs with the Makefile provided in each directory.

A.4 Experiment Workflow

There are three steps to conduct the experiments:

1) Download the datasets. This can be done by executing the 'get_data.sh' script in the top directory.

2) Compile the programs. This can be done by make in each directory.

3) Execute the codes. Each of the application has a baseline named "*_ serial", a tiling-and-grouping version named "*_grouping", a conflict-masking version named "*_mask", and our in-vector reduction version "*_invec". The programs are executed with command line argument that specifies the input files. Detailed usage can be found in the 'run.sh' script in the directory.

Alternatively, after downloading the datasets, you can simply execute the 'run.sh' script in each directory. It will automatically compile the programs and execute the programs.

A.5 Evaluation and Expected Result

The execution time (or throughputs) of the programs will be printed out in the console, and they should be very close to those reported in the paper. The programs will also print out the SIMD utilization for conflict-masking approach. They should be the same as reported in the paper. Some of the computation results (e.g. rank values in PageRank, shortest distance in SSSP) are also printed out to check the correctness of the programs. For hash-based aggregation, there might be some discrepancy in the exact throughputs. This is because the input data are generated randomly. However, the trend and the relative performance of different versions should be the same as reported in the paper.

A.6 Notes

The grouping procedure for some inputs can take as long as one hour. Please be patient when the execution seems to be stuck at these points.

Acknowledgements

This work was supported by NSF award CCF-1526386.

References

[1] Linchuan Chen, Peng Jiang, and Gagan Agrawal. 2016. Exploiting Recent SIMD Architectural Advances for Irregular Applications. In *Proceedings of the 2016 International Symposium on Code Generation and Optimization (CGO '16)*. ACM, 47–58.

[2] Jee W. Choi, Amik Singh, and Richard W. Vuduc. 2010. Model-driven Autotuning of Sparse Matrix-vector Multiply on GPUs. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '10)*. ACM, New York, NY, USA, 115–126.

[3] John Cieslewicz, Kenneth A. Ross, Kyoho Satsumi, and Yang Ye. 2010. Automatic Contention Detection and Amelioration for Data-intensive Operations. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data (SIGMOD '10)*. ACM, New York, NY, USA, 483–494.

[4] Tom Henretty, Kevin Stock, Louis-Noël Pouchet, Franz Franchetti, J. Ramanujam, and P. Sadayappan. 2011. *Data Layout Transformation for Stencil Computations on Short-Vector SIMD Architectures*. Springer Berlin Heidelberg, Berlin, Heidelberg, 225–245.

[5] Sungpack Hong, Tayo Oguntebi, and Kunle Olukotun. 2011. Efficient Parallel Graph Exploration on Multi-Core CPU and GPU. In *Proceedings of the 2011 International Conference on Parallel Architectures and Compilation Techniques (PACT '11)*. IEEE Computer Society, 78–88.

[6] Xin Huo, Vignesh Ravi, Wenjing Ma, and Gagan Agrawal. 2011. An Execution Strategy and Optimized Runtime Support for Parallelizing Irregular Reductions on Modern GPUs. In *Proceedings of the International Conference on Supercomputing (ICS '11)*. ACM, 2–11.

[7] Xin Huo, Bin Ren, and Gagan Agrawal. 2014. A Programming System for Xeon Phis with Runtime SIMD Parallelization. In *Proceedings of the 28th ACM International Conference on Supercomputing (ICS '14)*. ACM, New York, NY, USA, 283–292. https://doi.org/10.1145/2597652.2597682

[8] Hiroshi Inoue, Moriyoshi Ohara, and Kenjiro Taura. 2014. Faster Set Intersection with SIMD Instructions by Reducing Branch Mispredictions. *Proc. VLDB Endow.* 8, 3 (Nov. 2014), 293–304.

[9] Saurabh Jha, Bingsheng He, Mian Lu, Xuntao Cheng, and Huynh Phung Huynh. 2015. Improving Main Memory Hash Joins on Intel Xeon Phi Processors: An Experimental Approach. *Proc. VLDB Endow.* 8, 6 (Feb. 2015), 642–653.

[10] Peng Jiang and Gagan Agrawal. 2017. Efficient SIMD and MIMD Parallelization of Hash-based Aggregation by Conflict Mitigation. In *Proceedings of the International Conference on Supercomputing (ICS '17)*. ACM, New York, NY, USA, Article 24, 11 pages. https://doi.org/10.1145/3079079.3079080

[11] Peng Jiang, Linchuan Chen, and Gagan Agrawal. 2016. Reusing Data Reorganization for Efficient SIMD Parallelization of Adaptive Irregular Applications. In *Proceedings of the 2016 International Conference on Supercomputing (ICS '16)*. ACM, Article 16, 10 pages.

[12] Farzad Khorasani, Keval Vora, Rajiv Gupta, and Laxmi N. Bhuyan. 2014. CuSha: Vertex-centric Graph Processing on GPUs. In *Proceedings of the 23rd International Symposium on High-performance Parallel and Distributed Computing (HPDC '14)*. ACM, 239–252.

[13] Jakub Kurzak, Wesley Alvaro, and Jack Dongarra. 2009. Optimizing matrix multiplication for a short-vector SIMD architecture – CELL processor. *Parallel Comput.* 35, 3 (2009), 138 – 150. Revolutionary Technologies for Acceleration of Emerging Petascale Applications.

[14] Jure Leskovec and Andrej Krevl. 2014. SNAP Datasets: Stanford Large Network Dataset Collection. http://snap.stanford.edu/data.

[15] Xing Liu, Mikhail Smelyanskiy, Edmond Chow, and Pradeep Dubey. 2013. Efficient Sparse Matrix-vector Multiplication on x86-based Many-core Processors. In *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing (ICS '13)*. ACM, New York, NY, USA, 273–282.

[16] Duane Merrill, Michael Garland, and Andrew Grimshaw. 2012. Scalable GPU Graph Traversal. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '12)*. ACM, New York, NY, USA, 117–128.

[17] Simon J. Pennycook, Chris J. Hughes, M. Smelyanskiy, and S. A. Jarvis. 2013. Exploring SIMD for Molecular Dynamics, Using Intel Xeon Processors and Intel Xeon Phi Coprocessors. In *Proceedings of the 2013 IEEE 27th International Symposium on Parallel and Distributed Processing (IPDPS '13)*. IEEE Computer Society, 1085–1097.

[18] Orestis Polychroniou, Arun Raghavan, and Kenneth A. Ross. 2015. Rethinking SIMD Vectorization for In-Memory Databases. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data (SIGMOD '15)*. ACM, 1493–1508.

[19] James Reinders. 2017. Intel AVX-512 Instructions. https://software.intel.com/en-us/blogs/2013/avx-512-instructions.

[20] Bin Ren, Youngjoon Jo, Sriram Krishnamoorthy, Kunal Agrawal, and Milind Kulkarni. 2015. Efficient Execution of Recursive Programs

on Commodity Vector Hardware. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '15)*. ACM, New York, NY, USA, 509–520. https://doi.org/10.1145/2737924.2738004

[21] Bin Ren, Tomi Poutanen, Todd Mytkowicz, Wolfram Schulte, Gagan Agrawal, and James R. Larus. 2013. SIMD Parallelization of Applications That Traverse Irregular Data Structures. In *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO) (CGO '13)*. IEEE Computer Society, Washington, DC, USA, 1–10. https://doi.org/10.1109/CGO.2013.6494989

[22] Erik Saule and Ümit V. Catalyurek. 2012. An Early Evaluation of the Scalability of Graph Algorithms on the Intel MIC Architecture. In *Proceedings of the 2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW '12)*. IEEE Computer Society, Washington, DC, USA, 1629–1639.

[23] Avinash Sodani. 2016. Knight Landing Intel Xeon Phi CPU, Path to Parallelism with General Purpose Programming. http://cgo.org/cgo2016/wp-content/uploads/2016/04/sodani-slides.pdf.

[24] Wai Teng Tang, Ruizhe Zhao, Mian Lu, Yun Liang, Huynh Phung Huynh, Xibai Li, and Rick Siow Mong Goh. 2015. Optimizing and Auto-tuning Scale-free Sparse Matrix-vector Multiplication on Intel Xeon Phi. In *Proceedings of the 13th Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO '15)*. IEEE Computer Society, Washington, DC, USA, 136–145.

[25] Loïc Thébault, Eric Petit, and Quang Dinh. 2015. Scalable and Efficient Implementation of 3D Unstructured Meshes Computation: A Case Study on Matrix Assembly. In *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP 2015)*. ACM, 120–129.