Reusing Data Reorganization for Efficient SIMD Parallelization of Adaptive Irregular Applications

Peng Jiang, Linchuan Chen, and Gagan Agrawal
Department of Computer Science and Engineering
The Ohio State University
Columbus, OH 43210, USA
{jiang.952, chen.1996, agrawal.28}@osu.edu

ABSTRACT

Applying SIMD parallelization to irregular applications with non-continuous and data-dependent memory accesses is challenging. While an application involving a static pattern of indirect accesses (across iterations) can be accelerated by data transformations, such techniques are no longer feasible if the indirect access patterns change over time. In this paper, we propose an indexing method that facilitates the reuse of data reorganization for efficient SIMD parallelization of dynamic irregular applications. This indexing approach is first applied on a class of vertex-centric graph algorithms where the set of active vertices varies over the execution – the indexing method helps maintain the set of active edges. Next, we focus on unstructured particle interaction applications in which the edges change adaptively, and present an incremental indexing method. In our experimental evaluation, the speedups achieved by utilizing SIMD on graph applications range from 3.04x to 7.19x, and between 2.54x to 4.43x for molecular dynamics.

1. INTRODUCTION

SIMD has been a common feature in popular processors for several years. Regular applications can easily benefit from SIMD execution because of continuous and statically determined memory access patterns. However, applying SIMD efficiently to irregular applications, examples of which include graph algorithms and particle-in-cell or N-body applications, is challenging. In such applications, the first issue is that the memory accesses are non-continuous and data-dependent. Although newer SIMD instruction sets include *gather* (*scatter*) instructions that support non-continuous memory accesses for loading (storing) a vector, performance of applications using these features largely depends on the memory distances [3]. The second issue is that indirection-based writes in certain applications can cause data conflicts, which must be avoided or removed at runtime.

Numerous efforts have been made on transforming irregular memory access patterns to achieve better memory access performance [4, 5, 6, 7, 15]. These methods can be viewed as an application of the well known *inspector-executor* paradigm, previously researched in the context of distributed memory execution [2]. Transforming memory access patterns can effectively improve the cache performance, and thus SIMD utilization, in irregular applications. However, these transformations can be expensive. If an irregular application has the same pattern for the entire execution (for example, interactions between vertices or cells do not change across iterations), the preprocessing costs associated with the transformations are easily amortized, but this may not always be the case.

We now discuss the challenges associated with one particular set of applications. Consider vertex-centric graph algorithms such as Breadth First Search (*BFS*) and Single Source Shortest Path (*SSSP*). The *active* vertices and edges in these applications vary over iterations or *supersteps* [14] – only the vertices that receive messages from the previous superstep or whose values were updated in the previous superstep are considered active. Now, let us consider the potential options for SIMDization of these applications. Simply vectorizing the computation of each vertex leads to poor memory locality, because a vertex may access the values of distant vertices through its adjacency list. In addition, we likely have a poor SIMD utilization ratio because the size of the adjacency list of certain vertices can be (much) smaller than the SIMD vector length. In fact, Harish *et al.* [8] and Hong *et al.* [9] accelerate graph algorithms on GPU in this manner (i.e. processing vertex's neighbors in parallel), and their approach leads to non-coalesced memory accesses and load imbalance. Another approach has been taken by Merrill *et al.* [16], which involves maintaining the *frontiers* based on vertices most recently explored, and applies SIMD on the frontiers. Neither of them consider memory locality as a critical factor that influences the performance. At the same time, improving memory locality is non-trivial, since the memory access patterns are both data-dependent and dynamic in these graph applications. Finally, Chen *et al.* [3] apply tiling on a sparse matrix representation of a graph algorithm to achieve better memory locality for SIMD operations. However, their methods are also not applicable to graph applications where memory access patterns change over time.

Graph algorithms are not the only class of irregular applications with dynamic changes in memory accesses over time. Another example is a particle interaction or N-body application like *Molecular Dynamics*. Here, as particles or molecules move over time, the neighbor lists need to be *rebuilt* every few iterations. Chen *et al.* [3] propose a tiling-and-grouping approach, which can also be seen as an application of the inspector-executor idea, tailored for efficient SIMD execution. Although this approach does improve memory locality and remove the write conflicts during a *static* period of the simulation, their approach requires tiling-and-grouping for the entire set of interaction edges after each neighbor rebuilding step. Though *incremental inspectors* have been proposed in the distributed memory execution context [10], the issues in designing (incremental) inspectors for SIMD execution are very different.

ICS '16, June 01 - 03, 2016, Istanbul, Turkey
© 2016 ACM. ISBN 978-1-4503-4361-9/16/06…$15.00
DOI: http://dx.doi.org/10.1145/2925426.2926285

In this paper, we present a method for efficient SIMD parallelization of irregular applications with *time varying* data access patterns. Our method further builds on the *tiling-and-grouping* idea proposed by Chen *et al.* [3], which was applied only on the static irregular applications. They observe that computations in many irregular reductions and graph algorithms can be seen as operations on a sparse matrix. Based on this observation they show that tiling the sparse matrix, followed by grouping nonzeros in each tile, can improve the memory access performance and resolve the data conflicts caused by indirection-based memory writes. However, their method assumes, or is effective, only when 1) the sparse matrices representing the grids/graphs do not change across iterations or supersteps, and 2) all nonzeros in the sparse matrix are involved in computation in each iteration.

In this work, we show that by introducing a new data structure – an *index* of tiling and grouping information – we can efficiently reuse the data reorganization and achieve good SIMD performance even when the accesses are more dynamic. We have developed a vertex-centric graph processing framework that applies SIMD processing to tiles and groups, leading to better memory locality and SIMD utilization compared with applying SIMD processing directly on vertices. This framework provides an API that is similar to Pregel [14], which allows programmers to easily specify graph applications. In addition, we develop a similar idea for particle interaction applications where the list of interactions (edges) changes over time. We show that by *incrementally* tiling and grouping the interaction lists and maintaining the index, the data reorganization of existing neighbor edges can be efficiently reused.

2. BACKGROUND

This section provides background on the irregular applications we target, a popular graph API, the Intel Xeon Phi coprocessor (latest commercial release of Intel Many Integrated Core (MIC) architecture), and a tiling-and-grouping optimization approach proposed in recent work [3].

2.1 Irregular Applications

Irregular applications are a broad class of applications that involve unstructured data accesses and/or irregular control flows. According to their memory access patterns, we find that most irregular applications belong to one of the three categories: *static irregular*, *dynamic irregular*, and *adaptive dynamic irregular*. We use three examples to demonstrate each of these categories.

2.1.1 Static Irregular

Figure 1 shows a code snippet from *Bellman-Ford* algorithm, which computes the shortest path between a single source vertex and every other vertex in the graph. The inner loop iterates over all edges in the graph, and in each iteration, the current distances of vertices $n1$ and $n2$ are read from array Dis. The memory accesses of array Dis are dependent on the contents of array $Edges$ and are generally non-continuous. There is also a conditional update of vertex $n2$'s distance that leads to irregular control flow and potential conflict writes. In summary, the memory accesses are *non-continuous* and *data-dependent*. It is noteworthy that in the code snippet of *Bellman-Ford* algorithm, each iteration of the outer loop iterates over all the edges in the graph, and furthermore, no edges are added or removed during the execution. So, the inner loop accesses the same positions in memory across the iterations of outer loop. The memory access patterns in these applications are considered as *static irregular*. This kind of irregularity can be eliminated or at least mitigated by a preprocessing phase involving data reorganization [3, 4, 20, 5]. Once the data is reorganized, the optimizations are applicable over the entire execution, and thus, their cost is amortized.

```
int Edges[numEdges][2];
float Dis[numVertices];
float Weights[numEdges];

for(int i = 0; i < numVertices; i++) {
    for(int j = 0; j < numEdges; j++) {
        // get the two vertices connected by edge j
        int n1 = Edges[j][0];
        int n2 = Edges[j][1];
        // get the current distances of vertex n1 and n2
        float d1 = Dis[n1];
        float d2 = Dis[n2];
        // get the weight of edge j
        float w = Weight[j];
        if(d2 > d1 + w) {
            Dis[n2] = d1 + w;
        }
    }
}
```

Figure 1: Code snippet for Bellman-Ford algorithm

```
class SSSPVertex: public Vertex {
 public:
    void Compute(MessageIterator* msgs) {
        if(superstep()==0) *MutableValue() = INF;
        float mindist = IsSource(vertex_id()) ? 0 : INF;
        while(!(msgs->Done())){
            mindist = mindist < msgs->Value() ? mindist
                : msgs->Value();
            msgs->Next();
        }
        if(mindist < GetValue()) {
            *MutableValue() = mindist;
            OutEdgeIterator iter = GetOutEdgeIterator();
            for(;!iter.Done();iter.Next()) {
                SendMessageTo(iter.Target(), mindist +
                    iter.GetValue());
            }
        } else {
            VoteToHalt();
        }
    }
};
```

Figure 2: Code snippet for Vertex-Centric SSSP

2.1.2 Dynamic Irregular

Not all irregular applications have static data access patterns. Figure 2 shows a code snippet from the vertex-centric SSSP algorithm, which is essentially a *wavefront* based Bellman-Ford implementation, and turns out to be more efficient for graphs with no negative cycles. We will show that the memory access patterns in these graph applications are not static. Before that, we first introduce vertex-centric graph processing models and their typical API functions.

Vertex-centric graph processing frameworks, such as Pregel [14], GPS [18] and CuSha [11], provide easy development and efficient execution of graph algorithms. To summarize [14], the key aspect of these frameworks is that *"Programs are expressed as a sequence of iterations, in each of which a vertex can receive messages from the previous iteration, send messages to neighbor vertices, and modify its own state and that of its outgoing edges"*. The underlying parallelization is applied to the *wavefront* (set of vertices that are active) in every synchronization phase, which is also referred to as a *superstep*.

To make the idea concrete, we show the typical set of functions in these frameworks in Table 1. Programmers need to define a class

API	Description
`void Compute(MessageIterator *msgs);`	Implemented by programmer to express computation in a vertex
`OutEdgeIterator GetOutEdgeIterator();`	Returns all the outgoing edges of a vertex
`void SendMessageTo(int target_id, MsgType val);`	Sends message to a target vertex
`int superstep();`	Returns the superstep the execution is in
`NodeType *MutableValue();`	Returns a pointer to the vertex's state
`NodeType GetValue();`	Returns the value of the vertex's state
`void VoteToHalt();`	Informs the framework not to execute `Compute` for this vertex in next superstep

Table 1: Common API functions in Vertex-Centric graph frameworks

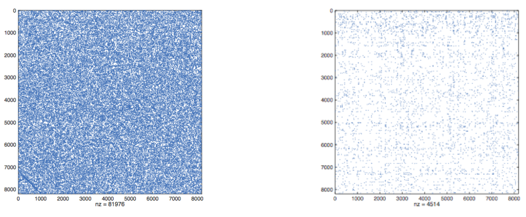
inherited from `Vertex` and implement the `Compute` interface to express the computation on each vertex. During a *superstep*, the framework invokes user-defined `Compute` function for each vertex in parallel. A vertex can get all of its outgoing edges by calling `GetOutEdgeIterator`. It can send messages to neighbor vertices by calling `SendMessageTo`, which accepts two parameters – the first is the target vertex identifier and the second the message value. The framework maintains an incoming message buffer for each vertex and passes a handler of that buffer to `Compute` as a parameter. `MutableValue` is used to access a vertex's own state.

The vertex-centric SSSP in Figure 2 is a wavefront based version of the *Bellman-Ford* algorithm. Iterations of the outer loop can be seen as supersteps. The difference from *Bellman-Ford* is that in this vertex-centric view, not all edges in the graph are processed in every superstep – only edges associated with vertices in the wavefront (i.e., vertices whose values were updated in the previous superstep) are processed. Also the number of supersteps does not need to be the number of vertices as is the case in the *Bellman-Ford* algorithm – instead, the execution stops as soon as the wavefront becomes empty during an entire superstep.

The wavefront based *Bellman-Ford* does not have a static memory access pattern across the supersteps, as the set of active vertices is changing. The memory access pattern in such applications are considered to be *dynamic*. This behavior has a significant consequence to our ability to optimize the processing, as was done for the *static* irregular applications in [3]. Reorganizing the data in every superstep will be expensive and can outweigh the benefits from SIMD execution.

2.1.3 Adaptive Dynamic Irregular

There is yet another set of irregular applications where the topology of the graphs/grids can change over the execution. In Molecular Dynamics, for example, one needs to rebuild the *neighbor edges* every few iterations – only two nodes that are within a certain cutoff radius are considered neighbors and have interaction force between each other. As the simulation proceeds, the coordinates of the molecules adjust, and the distances between two molecules can change. Thus, new edges need to be added and old edges have to be removed. However, one interesting point here is that the difference between two successively built neighbor edge lists is small. Figure 3 shows the edges between 8192 nodes, as well as the edges that are changed (either removed or newly added) after rebuilding. The number of edges before rebuilding is 81,976, while the number of modified edges is 4,514, i.e, only 5% of the total number of edges. The memory access pattern here can be considered as *adaptive dynamic*.



(a) Edges before Rebuilding (b) Edges Changed in Rebuilding

Figure 3: Adaptive variation of edges in Molecular Dynamics

2.2 Intel Xeon Phi Architecture

Intel Xeon Phi coprocessor is the latest commercial release of the Intel Many Integrated Core (MIC) architecture, and it has been incorporated in 9 of the top 100 supercomputers at the time of writing this paper [1]. The coprocessor is designed to be x86-compatible, so that it can leverage existing x86 software, and thus, benefit from traditional multi-core parallel programming models, libraries, and tools. There are 60 or 61 x86 cores in the coprocessor, each of which has a separate 32 KB L1 cache and a coherent 512 KB L2 cache, with L2 cache for all cores interconnected in a ring. Each core supports as many as 4 hardware threads. Each core also has 32 512-bit vector registers, providing large-scale SIMD parallelism. Because of the large number of cores and coherent cache, L2 cache misses on Intel Xeon Phi coprocessor are expensive compared to those on multi-core CPUs. A miss on L2 cache needs to be forwarded on the ring, and if the requested address is found on another core's cache, the data need to be forwarded on the ring afterwards. The cost in the worst case is of the order of hundreds of cycles.

Unlike the previous SSE that was supported on Intel CPUs, Intel Xeon Phi coprocessor has a new 512 bit SIMD instruction set referred to as IMCI orIntel Initial Many Core Instructions. It has a class of *gather/scatter* primitives for bulk operations that load/-store data from unaligned and non-continuous memory addresses. Another important feature in IMCI is the *mask* data type, which is associated with a class of *mask* operations. These operations facilitate operating on certain specific elements within a SIMD vector.

2.3 Tiling and Grouping Approach

Chen *et al.* [3] have recently shown that *tiling* and *grouping* input data is an effective approach for improving the SIMD performance of irregular applications with gather and scatter operations. Their work applies to graph algorithms, irregular reductions, and sparse matrix computations.

Their approach include three steps. In first step, *Sparse Matrix View*, a sparse matrix is used as a unifying structure – for example, since a graph can be represented by a sparse matrix (edges in the graph being represented as non-zeros in the sparse matrix), the operations can be seen as operations on sparse matrices. Similarly, for irregular reduction (like the ones arises with unstructured grids), edges that the computation iterates over can be seen as non-zeros in a sparse matrix. The computation associated with processing any given edge usually involves reading and writing values associated with the nodes that are the end-points of the edge.

The idea behind the second step is as follows. With the sparse matrix view of the computation, dividing and storing *tiles* that have a high concentration of non-zeroes, and processing the non-zeros *tile-by-tile*, is an effective method for improving the memory locality in these applications. In each tile, the memory accesses utilize the *gather/scatter* operations provided by Intel IMCI. Since each tile spans a small number of cache lines, the SIMD performance is good.

The third step, *Grouping*, addresses another challenge, which is that IMCI instruction set (unlike GPUs) does not support atomic operations among the lanes in a vector. Thus, if the same address is used in more than one lane at any given time, we have a write conflict. In order to remove the write conflicts, Chen *et al.* [3] group the edges in a tile into *conflict-free vectors*. Groups with less than 16 elements are padded and the padded values are *masked* during the execution.

As an example, it is straightforward to vectorize the inner loop of *Bellman-Ford* algorithm by unrolling 16 iterations and combining the scalar operations into vector operations. After applying tiling and grouping, the inner loop iterates over the edges in the graph tile-by-tile, leading to better data locality. Because the edges are reordered into conflict-free vectors within a tile, the scatter operation does not lead to any write conflicts.

3. MOTIVATION AND OVERVIEW

Consider the vertex-centric SSSP algorithm described in Section 2.1.2. There are many difficulties in applying SIMD to this algorithm, as compared to achieving SIMD parallelization for the *Bellman-Ford* algorithm. The computation is expressed in the `Compute` function, which is applied to each vertex, and this can limit our ability to apply any data reorganization. More specifically, optimizations based on reorganizing the edge list (such as those presented by Chen et al. [3]) cannot be applied to this computation model. Second, each vertex stores its neighbor list locally, and once the distance is updated, it needs to send messages to all of its neighbors. Since the neighbors can be stored far away from each other in the vertex list, there can be poor memory locality. Finally, a static data reorganization, applied before the execution, will fail because the memory access patterns change in each superstep.

Now, let us consider a particle interaction application such as Molecular Dynamics. The tiling-and-grouping approach proposed earlier [3] does improve data locality and removes data conflicts for SIMD execution during a static period of simulation. However, the only way it can handle changes in the neighbor list during execution is by running an inspector after every modification to the neighbor list. Clearly, this will be extremely expensive.

We aim at facilitating SIMD execution for graph algorithms for the vertex-centric computing model. This includes being able to modify data layout, making tiling-and-grouping optimization [3] applicable, and facilitating SIMD execution. Our goal also includes making the tiling-and-grouping approach practical for adaptive particle interaction applications, by creating an incremental version of our transformation approach.

4. UTILIZING SIMD IN A VERTEX-CENTRIC GRAPH PROCESSING FRAMEWORK

In this section, we present a framework that maps vertex-centric algorithms to SIMD execution as an example to show how dynamic irregular applications can be SIMDized efficiently. We first introduce the API of our framework, and then give details of the implementation.

4.1 Programming Interface

```
class SSSPVertex: public Vertex {
 public:
  void InitMsg() {
    Aggregate();
    SetMsgValue(INF);
  }

  void ProduceMsg(vfloat *edge, vfloat *source, vfloat *
      message) {
    mask m = *message > *source + *edge;
    conditional_update(message, *source + *edges, m);
  }

  void Compute(MessageIterator* msgs) {
    ......
  }
};
```

Figure 4: Vertex-centric SSSP with our framework

Besides the API functions listed in Table 1, our framework provides two more functions for users to implement. `InitMsg` initializes the values of messages, while `ProduceMsg` calculates the messages based on the values of the source vertices and/or the edges. Our framework calls `InitMsg` before the first superstep and invokes `ProduceMsg` at the end of each superstep – with the latter, the idea is to delay the message production till the end of a superstep, when all messages can be processed in an edge-centric manner. This style of computation, which also requires omitting message production in `Compute` and storing edges in `SendMessageTo`, enables our framework to apply SIMD instructions and even the locality transformations.

Figure 4 shows an example of vertex-centric SSSP algorithm implemented with our framework. In the example, `InitMsg` sets all the messages' value to INF, and `ProduceMsg` compares the messages sent in previous superstep with the sum of the source vertices' values and the corresponding edges' weights. If the sum is smaller than the initial message, which means there is a chance for the values in the destination vertices to be relaxed, then the values of the messages need to be updated. `Compute` is almost identical to the function shown in Figure 2, except that the message production is sliced out and only the target vertex identifier is passed to `SendMessageTo`.

4.2 Implementation

Our implementation includes four steps as illustrated in Figure 5. First, the edges in a graph are preprocessed by the tiling-and-grouping method reviewed in Section 2.3. These edges are also then stored in an index data structure. Next, to overcome the vertex-by-vertex computing constraint, we convert the vertex-centric computation to *edge-centric* form by saving the edges when messages when normally they would be sent, and delaying message production till the end of each superstep. Then, we activate the edges saved in the index data structure in the second step. Last, we apply SIMD to the edge-centric computation based on the activated edges in the index data structure. We explain each of four steps below.

4.2.1 Preprocessing

In the tiling-and-grouping method described in Section 2.3, the edges in a graph are represented by non-zeros in a sparse matrix, and the edges in each tile are grouped into conflict-free vectors. These vectors are stored in two arrays, $Xcoord$ and $Ycoord$ – $Xcoord$ stores the x-coordinates of edge groups in a tile, whereas $Ycoord$ stores the y-coordinate. Every set of 16 consecutive elements in the two arrays represents a conflict-free group of edges in a tile.

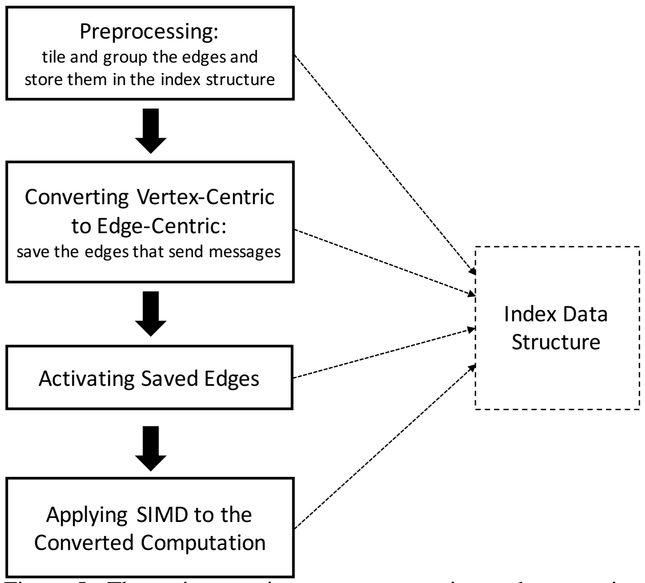
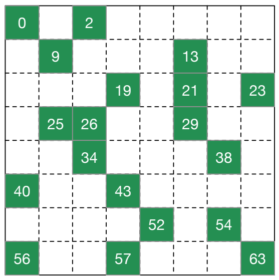
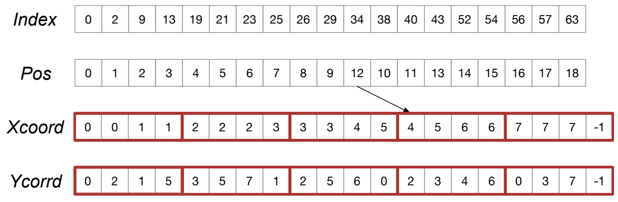

Preprocessing:
tile and group the edges and store them in the index structure
Converting Vertex-Centric to Edge-Centric:
save the edges that send messages
Activating Saved Edges
Applying SIMD to the Converted Computation
Index Data Structure

Figure 5: The main steps in our vertex-centric graph processing framework for reusing data reorganization and applying SIMD execution



(a) Edges in a Tile



(b) Index Data Structure

Figure 6: Example of a tile and its edges stored in index data structure

Index Data Structure We now present an index structure which is the key data structure used in our framework. All the edges in a tile are stored in an array $Index$ as its relative positions in a tile. The relative position of an edge at the row r and column c in a tile is $r \times tilesize + c$. The absolute coordinates of an edge in a sparse matrix, which are the positions of it two endpoints in the vertex list, are determined by the coordinates of the tile and the relative position of that edge. Suppose the coordinates of a tile is (p, q) and the relative position of an edge in the tile is w, then the absolute coordinates of the edge is $(p \times tilesize + w/tilesize, q \times tilesize + w\%tilesize)$. $Xcoord$ and $Ycoord$ in our data structure store the row and column number (not the absolute coordinates) of grouped edges. In addition to $Xcoord$, $Ycoord$, and $Index$, there is another array Pos – $Pos[i]$ is the position of the edge $Index[i]$ in $Xcoord$ and $Ycoord$. Initially, all the edges in a tile are marked as inactive by setting the values in $Xcoord$ to be -1.

Figure 6(a) shows a 8×8 tile where the filled cells represent the edges in the tile, and the numbers in the cells are their relative positions. Figure 6(b) shows the index data structure to store the tile. In Figure 6(b), the edges are grouped into vectors of length 4 (the actual vector length is 16 in our test platform, and here we choose 4 for demonstration). The edge 34, for example, is stored at the position 12 in $Xcoord$ and $Ycoord$. The value -1 at the end of $Xcoord$ happens to be a padding here, but it can also indicate that an edge is not active.

4.2.2 *Converting Vertex-Centric to Edge-Centric*

The reason for the conversion is that edge-centric computation does not have the vertex-by-vertex computing constraint as in vertex-centric model, which makes it much easier to improve data locality in SIMD execution.

There are two key requirements for a vertex-centric computation to be converted to edge-centric fashion. First, *the produced messages are sent to the adjacent vertices of a given vertex.* For example, in the vertex-centric SSSP algorithm, each vertex first reads the messages received from others. From these messages, it selects the one with the minimum value. If this value is less than its own distance, it will update its own distance with this value. If its own distance is updated, it needs to send the updated distance to all its neighbors via out-going edges. Second, *only one message is sent through every edge in a superstep.* For example, in SSSP, the distance of a vertex is updated only once in a superstep, and the updated message will also be sent through the adjacency list only once. A broad class of vertex-centric algorithms meet the two requirements, and thus, can be viewed as *edge-centric*.

In our framework, this conversion does not require any code transformation; instead, it is conducted by simply recording the edges when messages are sent. More specifically, it is achieved by modifying `SendMessageTo` function to save the adjacency edges in different tiles instead of producing and writing the messages. The saved edges are later activated in the index data structure by a *Search-and-Activate* procedure which will be describe in next subsection.

4.2.3 *Activating Saved Edges in Index Data Structure*

By leveraging the *mask* operations supported in Intel IMCI, we represent the active status of an edge by its value in $Xcoord$: a value of -1 in $Xcoord$ indicates that the corresponding edge is not active. Non-negative values in $Xcoord$ represent the x-coordinates of the edges. After reading values from $Xcoord$, the program first compares the values with zero to produce a *mask*. If the value is greater than zero, the corresponding bit in the *mask* is set; otherwise, the bit is unset. Only the edges whose values in $Xcoord$ are non-negative actually participate in the computation. Thus, activating an edge simply requires updating the value of an edge in $Xcoord$ from -1 to its x-coordinate.

With the help of the index data structure described above, in order to activate the edge e, we first perform a binary search in $Index$ to obtain the index i such that $Index[i] == e$. Then, we can get e's position in $Xcoord$ from $Pos[i]$. We next change the value in $Xcoord[Pos[i]]$ from -1 to the x-coordinate of e. The algorithm is named *Search-and-Activate* and is shown in Algorithm 1.

```
Algorithm 1: Search-and-Activate an edge
  Input: $e$: the edge that need to be activated
         $Index$: ordered array of edge indices
         $Xcoord$: x-coordinates of edge groups
         $Ycoord$: y-coordinates of edge groups
         $Pos$: positions of edges in edge groups
         $x$: x-coordinate of the tile
         $y$: y-coordinate of the tile
  $inx = binary\_search(e, Index)$;
  if $inx >= 0$ then
      $position = Pos[inx]$;
      // calculate the $x - coordinate$ of $e$
      $coordinate = x * tilesize + e/tilesize$;
      $Xcoord[position] = coordinate$;
```

The complexity of this procedure is $O(log(n))$, where n is the size of $Index$. To further accelerate the procedure, the binary search can traverse $Index$ for several edges simultaneously using SIMD lanes. More specifically, vectors of left, right, and middle positions are maintained and the process stops when all edges are found in $Index$. The update of $Xcoord$ is also in SIMD.

Now, the vertex-centric computation has been converted into edge-centric fashion – our framework iterates over all of the active edges and performs message producing by invoking `ProduceMsg`.

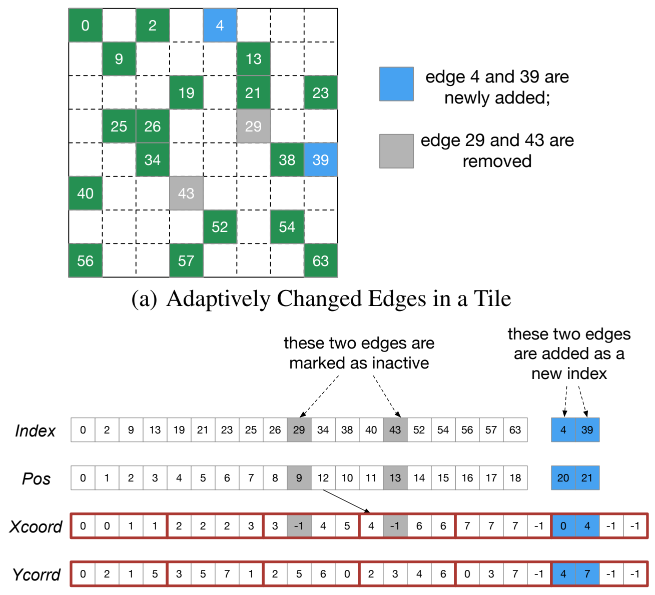
4.2.4 *Applying SIMD to the converted Computation*

We define three default SIMD vectors, `_message`, `_source` and `_edge`, in our framework to implement SIMD execution. `_message` stores message values that need to be produced, `_source` stores values for the source vertices, and `_edge` stores edge weights. The operations on these three vectors are implemented as overloaded functions with SIMD instructions. As shown in the vertex-centric SSSP code in Figure 4, three parameters are passed to `ProduceMsg` – *edge*, *source* and *message*, with which the programmers can reference the three default SIMD vectors. When iterating over the active edges, the framework first loads coordinates of the edges from $Xcoord$ and $Ycoord$. Next, based on the the values in $Xcoord$, the framework produces a *mask* which indicates the active edges as non-zero bits. Last, the framework invokes `ProduceMsg` to do the actual message producing with SIMD where the framework is responsible for loading and storing the values of `_message`, `_source` and `_edge`, which are implemented by the *gather/scatter* instructions on Intel MIC architecture.

4.3 Overheads vs. Benefits

Most of the overheads incurred in our approach are in the transformation from vertex-centric to edge-centric computation. More specifically, one source of overheads is that the framework needs to save all the active edges. Compared with directly producing the messages and sending to the destination, saving the active edges does not involve reading or writing the values in the vertices, neither does it involve computing the values of the messages. Instead, we just save the indices of edges in a temporary array. Another form of overhead is due to the step of searching-and-activating the stored edges.

On the other hand, the benefits of our approach come from data locality and SIMD processing. The original vertex by vertex execution can be seen as processing the non-zeros in the sparse matrix row by row, whereas our approach processes the non-zeros in the sparse matrix tile by tile, leading to significant memory locality improvement when accessing the values in vertices. 16 lanes vector processing is applied to the active edges globally with SIMD utilization rates range from 0.18 to 0.88. It turns out that the benefits outweigh the overheads, bringing substantial speedups in vertex centric graph algorithms, as we will show in the evaluation.



(a) Adaptively Changed Edges in a Tile

(b) Index Data Structure Collection

Figure 7: Example of a tile with adaptively changed edges stored in index data structure collection

5. SIMD EXECUTION FOR ADAPTIVE DYNAMIC APPLICATIONS

In a particle interaction application like *Molecular Dynamics*, the *neighbor lists* need to be rebuilt in every few iterations. During the iterations between two rebuilding steps, the interactions are static and the *tiling-and-grouping* method proposed by Chen *et al.* [3] is effective in improving SIMD performance. However, once a neighbor list is rebuilt, this method would require the tiling procedure to be invoked again, which can have a high overhead.

This section describes our approach for incrementally applying the solution. The key observation is that the interactions among the particles (or neighbor lists) change gradually over time. Every time the neighbor list is rebuilt, only a small fraction of edges are removed and a similar fraction is typically added. This property provides the opportunity to perform tiling-and-grouping only once and reusing the grouping information, somewhat similar to our approach in supporting the vertex-centric graph framework. The key difference from the approach for the vertex-centric graph framework arises because new edges can be added to the list of interactions.

5.1 Index Data Structure Collection

Consider a specific neighbor list rebuilding step, where some edges are removed from the interaction, and some are newly added. The removed edges do not participate in computation in the following iterations, which are indicated by -1 in the array $Xcoord$ and will be masked out in the SIMD operations. The newly added edges are appended to $Index$. They also need to be grouped and stored in $Xcoord$ and $Ycoord$ arrays for SIMD execution in the following iterations. In addition, their positions in the $Xcoord$ array need to be stored in the Pos array.

Since the edges in the array $Index$ are ordered by their indices for binary search, the newly added edges cannot be directly appended. We modify the index data structure presented in Section 4.2.1 and create a *index data structure collection*. There are multiple $Index$ arrays, each of which stores ordered edges. The binary search is conducted to all of the $Index$ arrays until the element is found in one of them. Similarly, the Pos array is extended to a collection of Pos arrays, each of which corresponds to an $Index$ array. The arrays $Xcoord$ and $Ycoord$ remain unchanged and have groups of new edges appended to the end of them.

5.2 Search and Update the Edges

Because the graph/grid structure is changing, newly added edges are not pre-reorganized. Thus, they need to be incrementally added to the index data structure. The task is accomplished by a *Search-and-Update* procedure that includes three steps.

In the first step, the procedure executes a modified version of the Algorithm 1, with the following changes: (1) the input of the new algorithm is an index data structure collection, (2) it creates a new $Index$ for the new edges that will be added to the multiple index data structure, (3) when an edge is found in a certain $Index$ array, it will be activated as normal as in Algorithm 1, and (4) if an edge is not found in any existing $Index$ arrays, the algorithm will add it to the new $Index$. Step two is to group the newly added edges – specifically, after all the new edges are added to the new $Index$ array, we use the grouping method proposed by Chen *et al.* [3] to group these new edges into conflict-free groups. These conflict-free groups are subsequently appended to $Xcoord$ and $Ycoord$ arrays. Finally, the newly added edges in $Index$ and Pos arrays are sorted to facilitate binary search, which will be required the next time the edges are rebuilt.

Figure 7(a) shows an example of adaptively changed edges in a tile – edge 4 and edge 39 are newly added to the tile, while edge 29 and edge 43 are removed from the tile. Figure 7(b) shows the index data structure collection storing the tile. The coordinates of edge 29 and 43 in $Xcoord$ are set to -1, indicating these two edges are no longer active in the following execution. A new index is created for edge 4 and 39, with their coordinates appended to $Xcoord$ and $Ycoord$. Pos records the positions of the two newly added edges in $Xcoord$ and $Ycoord$.

6. EVALUATION RESULTS

In this section, we evaluate the effectiveness and efficiency of our method using five graph algorithms and two particle interaction applications. For comparison, we used a version that incorporated our techniques, as well as serial implementations and *naive* use of SIMD (which for graph algorithms implied the use of SIMD instructions while processing all edges for a given vertex). The goals of our experiments included evaluating: 1) the benefits of our indexing and grouping method for graph algorithms – to this end, we compare the performance of the optimized SIMD framework against serial and naive SIMD executions, and 2) efficiency of incremental indexing and grouping, for which we compare the overheads of this method with a naive method we call *all regrouping*, and evaluated performance against serial and naive version.

Our experiments are conducted on an Intel Xeon Phi SE10P coprocessor, which contains 61 cores running at 1.1 GHz, each with four hyperthreads. The GDDR5 main memory is 8GB. We use Intel ICC compiler 13.1.0, to compile all the codes, with -O3 optimization enabled.

6.1 Application Used

Table 2 shows the applications and datasets used in our experiments. For each application, datasets of different sizes are used. Five popular vertex centric graph algorithms are used: Breadth First Search (*BFS*), Single Source Shortest Path (*SSSP*), Single Source Widest Path (*SSWP*), Topological Sort (*TS*), and Weakly Connected Component (*WCC*). We select these algorithms because their set of active edges changes over supersteps, and thus they match the pattern we are optimizing. Conversely, We exclude graph algorithms such as *PageRank* that have static active edges.

	App	Dataset	Diminutions	NNZ
Graph Algorithms	BFS	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
	TS	small-dag	412K*412K	850K
		medium-dag	350K*350K	6.8M
		large-dag	1M*1M	30M
Molecular Dynamics	Moldyn	16-3.0r	131K*131K	11M
		32-3.0r	365K*365K	30M
	MiniMD	16-3.0r	131K*131K	11M
		32-3.0r	365K*365K	30M

Table 2: Applications and datasets used in the experiments

Elaborating on these applications – *SSSP* has been discussed in earlier sections, and not discussed here. *BFS* is a widely used graph traversal algorithm, which proceeds as follows. Initially, the flags in each vertex are set to *unvisited* except for the starting vertex, which set to *just_visited*. In each superstep, if a vertex is *unvisited* and receives a message from other vertices, it will change its flag to *just_visited*, send messages to all of its neighbors, and then set its flag to *visited*. If a vertex is *visited*, despite the received message, it will go inactive by calling `VoteToHalt`. *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. Initially, the values in all vertices are set to 0 except the source vertex, which is set to INF. This indicates that the source vertex has an infinite width path to itself, but zero width path to all other vertices. In each superstep, a vertex finds the message with largest width and compares it with its own value. If the message's value is greater than its own value, it will update the value to the message's value. If its own values is greater than the weight of edge directing to the neighbor, the edge weight will be sent, otherwise, the vertex updated value will be sent. *TS* is an algorithm for getting a topological sort of vertices in a directed acyclic graph or a DAG. Initially, every vertex sets its values to 0 and sends 1 to its neighbors. In each superstep, a vertex adds all messages it receives to its own value. If the sum is 0, it means there is no edge pointing to this vertex, and thus it can be removed from the graph. When a vertex is removed, it sends -1 to all its neighbors, indicating that its outgoing edges are deleted from the graph. The algorithm stops when all vertices are removed from the graph. Finally, *WCC* finds a maximal subgraph of a directed graph such that for every pair of vertices there is a path from one to another. Initially, all vertices set their value to its *vertex_id*, indicating that they belong to the weakly connected component of themselves only. In each superstep, a vertex picks the vertex with smallest index among those that send message to it, then it sets its own value to this smallest index (except for in the superstep 0) – this implies that they are merging the vertex into the weakly connected component of the sending vertex. Next, the vertex sends its own value to all its neighbors. If the smallest index of the received messages are greater than the vertex value, the vertex does not belong to any other component and goes inactive by calling `VoteToHalt`. The graphs used for *BFS*, *SSSP*, *SSWP*, and *WCC* are from the *SNAP*[13] graph datasets, while the input for *TS* are direct acyclic graphs (DAGs) that are synthetically generated with varying size and sparsity.

A typical particle-interaction or molecular dynamics application involves irregular reductions. The goal is to simulate the interactions and motion of molecules based on Newton's law. In each iteration, the coordinates of the molecules are first updated. Next 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 force computing procedure is conducted on the interaction edges. The interaction edges are initialized from input, but need to be constantly updated as the coordinates of the molecular change over time. The edge list updates are done by a *neighbor rebuilding procedure* that needs to be executed in every few iterations. We use two versions of *Molecular Dynamics*: *Moldyn* (used in many studies in this area) and *MiniMD*, which is a DOE mini-app. The major difference between these two implementations is in the neighbor rebuilding procedure. *Moldyn* computes the distances between every pair of molecules in the grid, while *MiniMD* bins the molecules and only computes the distances among molecules in a bin and its stencil bins. The inputs are generated by the program that was distributed with the original serial code of *Moldyn*.

6.2 Results from Graph Algorithms

The experiments are conducted with three versions. The serial version (`Serial`) is based on an API similar as Pregel [14] and does not attempt any SIMD parallelization. The straightforward version (`Naive-SIMD`) utilizes SIMD to process the edges vertex-by-vertex, using SIMD instructions for loading/storing and also for message production. Finally, our tiling and indexing approach described in Section 4.2.2 is referred to as `Opt-SIMD`, where the edges are processed tile by tile with SIMD instructions. All of the three versions use only a single thread running on a single core of Intel Xeon Phi, so as to focus (only on) SIMD parallelization.

6.2.1 *Overall Performance*

Figure 8 shows the overall performance of the graph algorithms. The execution time in Figure 8 is the total running time of the algorithms. It shows that `Naive-SIMD` runs barely faster than `Serial`, and in several cases even slower. This is largely because of poor memory locality, leading to slow execution of *gather* and *scatter* instructions. For *BFS*, the speedup of `Opt-SIMD` version over `Serial` version is 7.19 on amazon0312, and 4.26 and 4.72 on higgs-twitter and soc-Pokec, respectively. These two speedups, which are results of SIMD processing (and improved data locality due to tiling and grouping) show that the benefits of data reorganization and SIMD parallelization outweigh the overheads of indexing used for maintaining the set of active edges. The relative speedup for *BFS* are better on amazon0312, which also happens to be sparser (and thus absolute performance of other versions is even worse). For *SSSP* and *SSWP*, the speedups are similar to those in *BFS*. For *TS*, the speedup of `Opt-SIMD` over `Serial` is 4.43 on small-dag, and 5.30 and 5.26 on medium-dag and large-dag, respectively. For *WCC*, the speedup of `Opt-SIMD` over `Serial` is 3.04, 4.95, and 4.59 on on amazon0312, higgs-twitter, and soc-Pokec, respectively.

6.2.2 *SIMD Utilization*

SIMD utilization is an important factor that influences the efficiency of the vectorization. This factor is calculated by dividing the number of active edges over the entire execution by the product of SIMD width and the number of times vector operations are used .

$$simd\_utilization = \#active\_edges/(\#vector\_ops \times 16)$$

We report the SIMD utilization of the graph algorithms (`Opt-SIMD` version) in Table 3, the SIMD utilization of *BFS* range from 0.18 to 0.29, which means 3 to 5 lanes in the vector are utilized in computation. We find that the SIMD speedups are even a little bit greater than the number of utilized lanes, which are attributed to better memory access locality brought by tiling. *SSSP* and *SSWP* have a similar result. The SIMD utilization in these three algorithms are determined by the graph topology, the more active edges on the traversal frontier, the higher SIMD utilization. *TS* has SIMD utilization range from 0.22 to 0.35 because only the adjacency edges of the removed vertex need to be calculated in each superstep. *WCC* has a higher SIMD utilization ranging from 0.70 to 0.88 because most of the vertices are active in each superstep.

App	amazon0312	higgs-twitter	soc-Pokec
BFS	0.18	0.25	0.29
SSSP	0.18	0.25	0.28
SSWP	0.18	0.25	0.29
WCC	0.88	0.70	0.79
TS	0.35	0.25	0.22

Table 3: SIMD utilization of Opt-SIMD: BFS, SSSP, SSWP, WCC, TS (three datasets each)

6.3 Results from Molecular Dynamics

In this section, we will show detailed results from the evaluation of *Moldyn* and *MiniMD*.

6.3.1 *Overheads of Neighbor Rebuilding*

We compare three versions each of *Moldyn* and *MiniMD*. The first version is the original serial code without any data reorganization (`Non-Group`). The second and the third versions are vectorized code with tiled and grouped input. The difference is that the second version (`Regroup-All`) regroups all of the edges when the neighbor edges are rebuilt, while the third version (`Inc-Group`) uses our incremental grouping and indexing method. All of the three versions use only a single thread running on a single core of Intel Xeon Phi, so as to focus (only on) SIMD parallelization.

Table 4 shows the neighbor rebuilding *overheads* of the three versions. The numbers in the table are the overhead time measured in seconds, which is defined as follows – for `Non-Group`, it is just the time for neighbor list rebuilding; for `Regroup-All`, it includes the execution time of neighbor rebuilding and the time for regrouping all the edges; and finally, for `Inc-Group`, it includes neighbor rebuilding and incrementally grouping and indexing of the new edges. As shown in Table 4, the overhead of `Regroup-All` is much higher than that of `Non-Group`. For `Moldyn`, the overhead costs of `Regroup-All` is 121.94x times higher than that of `Non-Group` on 16-3.0r, and 47.02x times higher on 32-3.0r. For `MiniMD`, the overheads for `Regroup-All` are 538.66x times higher than `Non-Group` on 16-3.0r, and 130.99x higher on 32-3.0r. These results imply that regrouping all edges is likely impractical. On the other hand, the overhead costs of `Inc-Group` are comparable to that of `Non-Group`. For `Moldyn`, the overhead time of `Inc-Group` is 1.19x that of `Non-Group` on 16-3.0r, and even smaller on 32-3.0r For `MiniMD`, the overhead costs of `Regroup-All` are 4.52x higher than that of `Non-Group` on 16-3.0r, and 2.31x on 32-3.0r. The results show that our indexing method is efficient in reusing tiling and grouping.

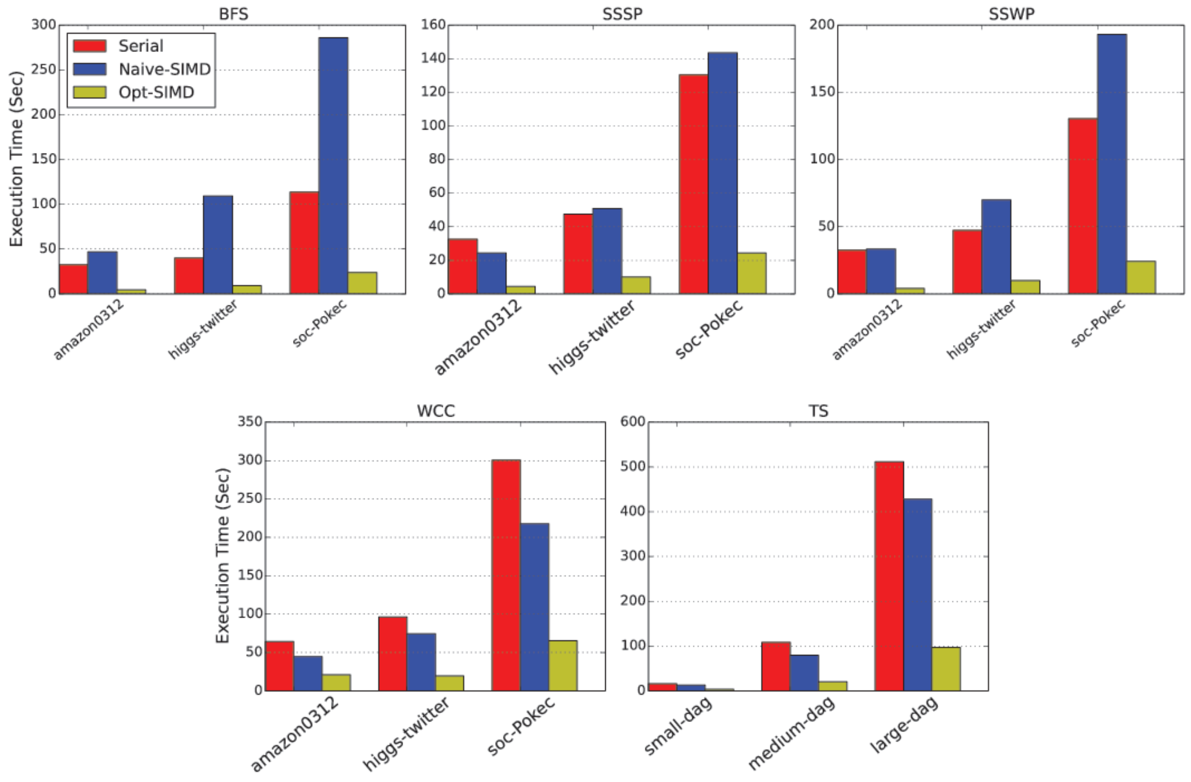


Figure 8: Performance of Serial code (Serial), Naive SIMD processing (Naive-SIMD), and SIMD processing with our indexing and tiling (Opt-SIMD): BFS, SSSP, SSWP, WCC, TS with three datasets each

App	Dataset	Non-Group	Regroup-All	Inc-Group
Moldyn	16-3.0r	4.93	601.14	5.89
	32-3.0r	20.21	950.23	8.43
MiniMD	16-3.0r	1.21	651.78	5.47
	32-3.0r	8.37	1096.35	20.20

Table 4: Overhead of neighbor rebuilding in original serial code (Serial), neighbor rebuilding with regrouping all new edges (Regroup-All), neighbor rebuilding with incremental grouping and indexing (Inc-Group): Moldyn, MiniMD with two datasets each

6.3.2 Overall Performance

To demonstrate that our incremental grouping and indexing method is beneficial in SIMD parallelization, we evaluate the total execution time of *Moldyn* and *MiniMD* with 100 iterations. Neighbor list rebuilding takes place every 20 iterations. As shown in Figure 9, we compare the execution time of two versions of each program. The first version is original serial code (`Serial`). The second version `Inc-Group+SIMD` computes the forces in SIMD manner with tiled and grouped input, and uses incremental grouping and indexing in neighbor rebuilding.

For *Molydn*, the speedups of `Inc-Group+SIMD` over `Serial` are 4.43 and 3.44 on 16-3.0r and 32-33.0r, respectively. For *MiniMD*, the speedups of `Inc-Group+SIMD` over `Serial` are 4.35 and 2.54 on 16-3.0r and 32-33.0r, respectively. With tiling and grouping method, the speedup of SIMD force computation over serial code in each iteration is 5.53 on 16-3.0r, and 3.67 on 32-3.0r, which is the same as what Chen *et al.* report in [3] for non-adaptive or static irregular applications. Overall, we have shown that our incremental grouping and indexing method imposes little overheads and makes it feasible to accelerate applications using modern SIMD features.

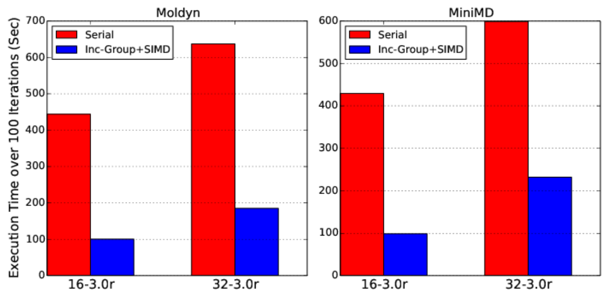


Figure 9: Overall performance of serial code (Serial) and SIMD force computation with incremental grouping and indexing (Inc-Group+SIMD): Molydyn, MiniMD with two datasets each

7. RELATED WORK

Vectorizing irregular applications is a challenging problem that has received considerable interest in recent years. Many efforts focus on accelerating irregular applications on various parallel platforms, including the GPUs and the Intel MIC. Among these, a subset focuses on a specific application such as *SpMV* and *Molecular Dynamics*, while others focus on vectorizing a category of applications (for example, graph algorithms).

Pennycook *et al.* [17] accelerate *Molecular Dynamics* on Intel Xeon Phi. They compare the efficiency of different gather/scatter implementations (software and hardware). The neighbor edges are processed node by node in their work, leading to poor cache performance, i.e., optimizing memory locality has not been a part of their work. Thébault *et al.* [19] parallelize unstructured 3D mesh computations at both multi-core and SIMD levels. The memory performance is optimized by recursive bisection with METIS, and the vectorization is done by a coloring scheme. The cost of these two data reorganization steps is high. Thus, their work is only applicable for static irregular patterns, and not the dynamic irregular applications we have targeted.

A number of research efforts have focused on optimizing graph processing. Kyrola *et al.* [12] propose a graph storage format (*shard*) to achieve efficient large graph processing on a PC with disk-based computations – with the idea that improved data locality can reduce the number of disk operations. The main steps of their graph storing process involve tiling the sparse matrix of the graph and storing the edges tile-by-tile. However, they do not apply this idea for SIMD processing. CuSha [11] modifies the *shard* format for efficient vertex-centric graph processing on GPUs. GPUs support atomic operations on shared memory, and thus there is not need to remove conflicts among the threads. Also, once the graphs are stored in *shard* format, there is no need to reorganizing the data anymore. In comparison, we have targeted SIMD instruction sets that do not support atomic operations among the lanes in a vector, so conflict removal is necessary for correctness. Merrill *et al.* [16] focus on parallelizing BFS on GPUs by fine-grain task management. Hong *et al.* [9] propose a novel virtual warp centric programming method to address the work imbalance problem in graph algorithms for GPUs. Chen *et al.* [3] use tiling and grouping to improve memory localities for SIMD gather/scatter operations and to remove the write conflicts among lanes in a SIMD vector. Their methods target irregular applications that have static memory access patterns. As stated throughout the paper, our work builds on their ideas but optimizes for dynamic or adaptive irregular patterns.

Data reorganization for locality improvement of irregular memory accesses has been studied for many years on CPUs. The most recent work on GPUs is done by Wu *et al.* [20]. They find that searching an optimal data reorganization (in terms of minimizing non-coalesced data access on GPUs) for irregular memory accesses is NP-complete. They propose two data reorganization methods (padding and sharing), which are effective in reducing non-coalesced data accesses. The sharing method utilizes a graph partitioning procedure to improve the data locality. In our work, we use tiling instead of partitioning since tiling appears to be cheaper than graph partitioning.

8. CONCLUSIONS

This paper has presented methods that facilitate reusing data reorganization for efficient SIMD parallelization of dynamic and adaptively dynamic irregular applications. We have proposed an index data structure and a search-and-activate procedure to maintain the active edges for a given iteration or superstep of a graph application. The active edges are also organized in tiles – thus our method incurs low overheads and the applications can benefit from improved data locality while utilizing SIMD features. The methods are applied to a vertex-centric graph framework and particle-in-cell applications. Experimental results show that our methods bring substantial SIMD speedups. The speedups on graph applications range from 3.04x to 7.19x, and those on particle-in-cell applications vary from 2.54x to 4.43x.

Acknowledgements

This work was partially supported by NSF Award CCF-1526386.

9. REFERENCES

[1] http://www.top500.org/lists/2015/11/.

[2] G. Agrawal and J. Saltz. Interprocedural compilation of irregular applications for distributed memory machines. SC '95, 1995.

[3] L. Chen, P. Jiang, and G. Agrawal. Exploiting recent simd architectural advances for irregular applications. CGO '16, 2016.

[4] C. Ding and K. Kennedy. Improving cache performance in dynamic applications through data and computation reorganization at run time. *SIGPLAN Not.*, may 1999.

[5] E. Gutiérrez, O. Plata, and E. L. Zapata. Balanced, locality-based parallel irregular reductions. In *Languages and Compilers for Parallel Computing*. 2003.

[6] H. Han and C.-W. Tseng. Improving compiler and run-time support for irregular reductions using local writes. In *Languages and Compilers for Parallel Computing*. 1999.

[7] H. Han and C.-W. Tseng. Efficient compiler and run-time support for parallel irregular reductions. *Parallel Computing*, 2000.

[8] P. Harish and P. J. Narayanan. Accelerating large graph algorithms on the gpu using cuda. HiPC'07, 2007.

[9] S. Hong, S. K. Kim, T. Oguntebi, and K. Olukotun. Accelerating cuda graph algorithms at maximum warp. *SIGPLAN Not.*, 46(8), Feb. 2011.

[10] Y.-S. Hwang, B. Moon, S. D. Sharma, R. Ponnusamy, R. Das, and J. H. Saltz. Runtime and language support for compiling adaptive irregular programs. 25(6):597–621, June 1995.

[11] F. Khorasani, K. Vora, R. Gupta, and L. N. Bhuyan. Cusha: Vertex-centric graph processing on gpus. HPDC '14.

[12] A. Kyrola, G. Blelloch, and C. Guestrin. Graphchi: Large-scale graph computation on just a pc. OSDI'12, 2012.

[13] J. Leskovec and A. Krevl. SNAP Datasets: Stanford large network dataset collection, June 2014.

[14] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: A system for large-scale graph processing. SIGMOD '10, 2010.

[15] J. Mellor-Crummey, D. Whalley, and K. Kennedy. Improving memory hierarchy performance for irregular applications using data and computation reorderings. *Int. J. Parallel Program.*, June 2001.

[16] D. Merrill, M. Garland, and A. Grimshaw. Scalable gpu graph traversal. *SIGPLAN Not.*, Feb. 2012.

[17] S. J. Pennycook, C. J. Hughes, M. Smelyanskiy, and S. A. Jarvis. Exploring simd for molecular dynamics, using intel® xeon® processors and intel® xeon phi coprocessors. IPDPS '13.

[18] S. Salihoglu and J. Widom. Gps: A graph processing system. 2013.

[19] L. Thébault, E. Petit, and Q. Dinh. Scalable and efficient implementation of 3d unstructured meshes computation: A case study on matrix assembly. PPoPP 2015, 2015.

[20] B. Wu, Z. Zhao, E. Z. Zhang, Y. Jiang, and X. Shen. Complexity analysis and algorithm design for reorganizing data to minimize non-coalesced memory accesses on gpu. PPoPP '13, 2013.