Exploiting Recent SIMD Architectural Advances for Irregular Applications

Linchuan Chen Peng Jiang Gagan Agrawal

The Ohio State University, Columbus, OH 43210, USA

{chen.1996,jiang.952,agrawal.28}@osu.edu

Abstract

A broad class of applications involve indirect or data-dependent memory accesses and are referred to as irregular applications. Recent developments in SIMD architectures – specifically, the emergence of wider SIMD lanes, combination of SIMD parallelism with many-core MIMD parallelism, and more flexible programming APIs – are providing new opportunities as well as challenges for this class of applications. In this paper, we propose a *general optimization methodology*, to effectively optimize different subclasses of irregular applications. Based on the observation that all applications with indirect memory accesses can be viewed as sparse matrix computations, we design an optimization methodology, which includes three sub-steps: 1) locality enhancement through tiling, 2) data access pattern identification, and 3) write conflict removal at both SIMD and MIMD levels. This method has been applied to unstructured grids, molecular dynamics, and graph applications, in addition to sparse matrix computations. The speedups achieved by our single threaded vectorized code over serial code is up to 9.05, whereas the overall speedup while utilizing both SIMD and MIMD (61 cores in Intel Xeon Phi) with our approach is up to 467.1. Further optimization using matrix reordering on irregular reductions and graph algorithms is able to achieve an incremental speedup of up to 1.69, though at a relatively high preprocessing cost. Moreover, SpMM using our approach outperforms routines from a highly optimized commercial library by up to 2.81x.

Categories and Subject Descriptors D.1.3 [*Software*]: Programming Techniques—Concurrent Programming

Keywords SIMD, Irregular Applications, Intel MIC

CGO'16, March 12–18, 2016, Barcelona, Spain
© 2016 ACM. 978-1-4503-3778-6/16/03...$15.00
http://dx.doi.org/10.1145/2854038.2854046

1. Introduction

Several classes of applications involve *irregular*, *data-dependent*, and/or *indirection-based* data accesses. Examples of such applications include sparse matrix computations, irregular reductions (arising from unstructured grid and molecular interaction applications) and graph algorithms. Because the memory access patterns of such applications are unknown at compile-time, we have at least two sets of challenges to address: 1) *efficiency* – non-continuous memory accesses that can lead to poor locality, and thus low memory access efficiency, and 2) *correctness* – indirect memory writes that lead to potential conflicts at the runtime. Over the last three decades, the above challenges have led to considerable research.

SIMD parallelism has been commonly available in popular processors for at least 15 years now. However, till recently, this type of parallelism has considered suitable only for regular applications. Recent SIMD architectures are changing this. The developments include wider SIMD lanes (up to 512 bits), combination of SIMD parallelism with massive shared memory MIMD parallelism (e.g., Intel Xeon Phi has 61 cores), and more flexible programming APIs, which include *gather* and *scatter* instructions. These trends have the following implications. First, with increasing SIMD width, there is an increasing performance penalty associated with not exploiting such parallelism – for applications that use integers and/or single precision floating point data, a potential speedup of up to 16 will not be achieved if SIMD potential is ignored. Second, effectively combining SIMD parallelism with massive MIMD parallelism requires that memory bandwidth limitations and write conflicts must be addressed. Finally, with support for gather and scatter operations, applications with non-contiguous accesses can be mapped to SIMD hardware, but performance characteristics of the hardware need to be understood and then exploited.

This paper investigates mapping applications involving data-dependent memory accesses to modern SIMD features. Though different class of applications that fit this pattern (i.e., unstructured grid computations, molecular dynamics, graph applications and sparse solvers) have different characteristics, we develop a general optimization methodology applicable to each of them. We observe that different types of irregular applications can all be viewed as sparse matrix computations – a representation that captures both the data accesses and the computation patterns, and makes the optimization opportunities apparent. The optimization method we propose aims to reduce the distance between the elements that are processed concurrently in SIMD lanes, so that the use of gather and scatter operations is efficient. In addition, we avoid inter-lane and inter-thread write conflicts, while eliminating the need for locking across lanes and cores.

We show the generality and effectiveness of our optimization methodology with three important irregular application subsets: *irregular reductions*, *graph algorithms* and *sparse matrix matrix multiplication (SpMM)*. These three subsets cover a broad range of cases, including the cases that access only one sparse matrix, as well as the case that accesses multiple sparse matrices. Despite the different data access patterns of these subsets, our optimization methodology effectively accelerates each of them.

Our implementation and evaluation are specific to the Intel Xeon Phi, which supports 512 bit SIMD lanes in each of the 61 cores on each node, and the KNC intrinsic instruction set. It should be noted that even with greater flexibility in the instruction set, many aspects of the original SSE instruction set remain (e.g., lack of locking among SIMD cores and lack of automatic support for dealing with control-flows), and thus, the challenges to SIMDization are very different from those arising in mapping irregular applications to GPUs. We have experimented with different applications and different datasets for each subclass of applications. We compare our approach against a method reported in a recent publication, analyze the SIMD lane utilization and the cache miss rates, and evaluate overall speedups. The results are as follows. First, our optimization methodology leads to a high SIMD utilization efficiency. For irregular reductions and graph algorithms, a *SIMD utilization ratio* of over 80% is achieved with tile sizes over 4096. For SpMM, clustered matrix inputs can keep the *SIMD utilization ratio* to be higher than 50%, on the average. Single thread vectorization delivers speedups of up to 9.05, 7.5, and 8.07, for irregular reductions, graph algorithms, and SpMM (for clustered inputs), respectively. The MIMD execution also leads to a high scalability: utilizing both SIMD and MIMD can speed up the execution by up to 290, 467.1, and 392.49, for the three subclasses, respectively. Moreover, SpMM using our approach is able to outperform the Intel MKL library routines by up to 2.81x. The artifact of our work is publicly available online[1].

2. Irregular Applications

Applications with indirect memory accesses typically arise because the interactions among physical entities lead to a sparse data structure, such as a graph, unstructured mesh, or a sparse matrix. We focus on three important subclasses of irregular applications: irregular reductions, graph algorithms, and sparse matrix matrix multiplication (SpMM). In this section, we will show how they can all be viewed as sparse matrix computations.

2.1 Irregular Reductions

Irregular reductions arise from unstructured grids or molecular (or particle) interactions. Unlike structured grids, nodes in unstructured grids are connected by edges explicitly, since

```
Real X(numNodes), Y(numEdges); // node/edge arrays
Integer IA(numEdges, 2);       // indirection array
for (e = 0; e < numEdges; e++) {
    val = f(X(IA(e,1)), X(IA(e,2)), Y(e));
    X(IA(e,1)) = X(IA(e,1)) + val;
    X(IA(e,2)) = X(IA(e,2)) - val;
}
```

Figure 1. An Irregular Reduction Loop

the connectivity of nodes cannot be determined by node positions (coordinates) [5]. Many implementations of molecular dynamics also lead to a similar pattern [11]. Figure 1 shows a typical irregular reduction loop. Each iteration updates two elements in reduction array X, indexed by an indirection array (edges) IA. The nodes connected by the edges are random, and thus the accesses to the node array are typically very irregular. Because each node may be connected by multiple edges, the update of the elements in X involves reductions.

Sparse Matrix View of Irregular Reductions: The indirection array IA is essentially a matrix M with dimensions of $N \times N$, where N is the number of nodes: an interaction (`IA(e, 1), IA(e, 2)`) in IA corresponds to a non-zero at the position (`IA(e, 1), IA(e, 2)`) in M. The value of each non-zero can either be 1, or the corresponding edge value (if edge values are used, as in the example above, originally stored in Y). Because an interaction only occurs at most once between each node pair, the matrix M is sparse and triangular. For each non-zero at the position (i, j) in M, computations are conducted using the values of elements i and j in the array X, as well as the value of the non-zero at position (i, j).

2.2 Graph Algorithms

Graphs have become an important data structure for representing relationships among people or other entities, and there has been a considerable interest in graph mining, especially in context of social network graphs [2, 32, 17].

Sparse Matrix View of Graph Algorithms: A graph can be viewed as (and if often stored as) an adjacency matrix. Because graphs are usually sparse, the corresponding adjacency matrix for a graph is also sparse, and thus, graphs are usually stored in compressed formats, e.g., *CSR*. Similar with irregular reductions, graph algorithms can also be summarized as a computation around a sparse matrix (the graph), and a vertex array. The work of project PEGASUS [13] has already shown how a number of graph mining applications can be viewed as a generalization of sparse matrix vector multiplication (SpMV). Specifically, for each non-zero at the position (i, j) in the sparse matrix, three major steps are followed: 1) reading the value v_i of Vertex i, 2) conducting computation using v_i and the value of the edge $e_{i,j}$ from the sparse matrix, and 3) updating the value v_j of Vertex j using the computation result from the Step 2.

2.3 SpMM

SpMM is an important kernel, widely used in many fields including scientific computation, data mining, and machine learning. It conducts a multiplication operation between two sparse matrices: $C = A \times B$. In general, A, B and C are stored in a compressed format, such as *CSR*. Because of the

[1] http://lcchen008.github.io/irreg-simd

sparsity of the matrices, the data accesses are irregular, in the sense that each row of the product matrix is accumulated according to the non-continuous column indices of the elements in the matrix B. SpMM is different from the other two application subsets in that it accesses multiple sparse matrices, instead of one. Common approaches for implementing this algorithm will be introduced in later sections.

2.4 Sparse Matrix Storage Formats

For memory efficiency, typically only the non-zeros in a sparse matrix are stored. We review two of the most popular representations, as a background for presenting our scheme.
CSR: CSR or *compressed row storage*, comprises a tuple of arrays: $(row\_idx, cols, vals)$. row_idx is the array whose size is equal to the total number of rows in the sparse matrix, and contains the indices where each row starts. $cols$ contains the column indices corresponding to each of the non-zero values, and similarly, $vals$ stores the values of the non-zeros.
COO: Another more intuitive format, *coordinate list (COO)*, stores a tuple of arrays $(rows, cols, vals)$. The only difference of the COO from the CSR is that in COO, the row index of every non-zero is explicitly stored in the array $rows$.

3. Intel Xeon Phi Architecture

While our proposed solution addresses challenges associated with any architecture with wide SIMD lanes, combination of SIMD and MIMD parallelism, and/or gather/scatter instructions, our implementation and evaluation has been in the context of the Intel Xeon Phi (co)processor. This x86-compatible (co)processor, which is a latest commercial release of the Intel Many Integrated Core (MIC) architecture, has already been incorporated in 11 of the top 100 supercomputers at the time of writing this paper [1]. MIC is designed to leverage existing x86 experience and benefit from traditional multi-core parallelization programming models, libraries, and tools.

In the available MIC systems, there are 60 or 61 x86 cores organized with a shared memory. These cores have a low frequency and in-order execution, and each supports as many as 4 hardware threads. Additionally, there are 32 512-bit vector registers on each core for SIMD operations. The main memory sizes vary from 8 GB to 16 GB. This memory is logically shared by all cores. The L1 cache is 32 KB, entirely local to each core, whereas each core has a coherent L2 cache, 512 KB, where cache for different cores are interconnected in a ring.

The features that are important for this study are:
Large Number of Concurrent Threads: Each Xeon Phi core allows up to 4 hyper-threads, in another word, we can have as many as 240/244 hardware threads sharing the same memory on Xeon Phi. This provides us with massive Multiple Instruction Multiple Data (MIMD) parallelism with shared memory, which has not been common in the past. Particularly, such a large number of threads can lead to a significant amount of contention if the execution involves frequent locking operations.
Wide SIMD Registers and Vector Processing Unit (VPU): VPU has been treated as the most important feature of Xeon Phi by previous studies [19, 33, 24, 8]. The reason is that the Intel Xeon Phi coprocessor has doubled the SIMD lane width compared to Intel Xeon processor, i.e., 256-bit to 512-bit, which means that it can process 16 (8) identical floating point (double precision) operations at the same time.
More Flexible SIMD Programming with Gather/Scatter: Unlike earlier SSE instructions supported by Intel CPUs, we have a new 512-bit SIMD instruction set called the Intel Initial Many Core Instructions (Intel IMCI). The most important feature is the support for irregular accesses through built-in *gather* and *scatter* operations. These primitives support load (store) of elements that are unaligned and non-continuous, according to a predefined *index vector*. This provides significant flexibility for irregular applications. Other important features include a hardware supported *mask* data type, and *write-mask* operations that allow operating on some specific elements within the same SIMD register.

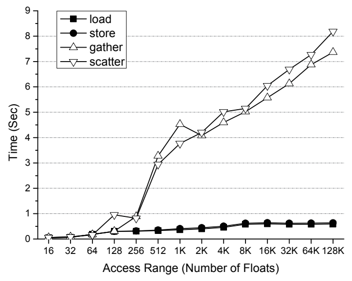


Figure 2. Time Taken for Accessing 16M floats using Load/Store and Gather/Scatter with Different Data Access Ranges

3.1 Understanding the Performance of Scatter/Gather

As we stated above, gather/scatter operations are an important and recent development, which can allow the use of SIMD parallelism for irregular applications. However, the performance characteristics associated with the use of these operations are not yet well understood. Prior to developing our schemes, we conducted experiments to understand these features. Specifically, we conducted the following study.

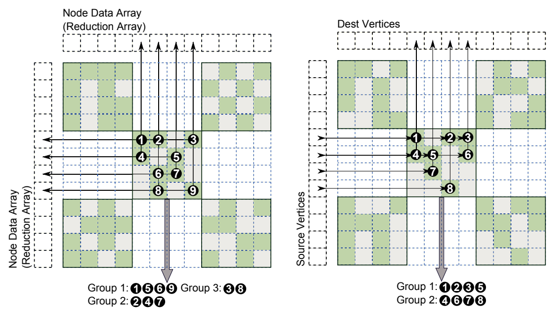
We carried out read/write operations from/to a floating point array of size 100 million, using single-thread SIMD execution. In the process, we varied the *access range size*, by which we imply the range of memory addresses that each `gather/scatter` operation spans – from 16 to 128K. For each access range size, two types of accesses are performed iteratively along the array (1 million each). For a `gather/scatter` operation, we use a randomly generated index vector, consisting of 16 integers, each between 0 and the *access range size*. As a baseline, a simple `load/store` operation is conducted to the consecutive 16 elements (starting position randomly chosen within the access range size, but aligned to 64 bytes). The first type of accesses are designed to measure the cost of accessing non-contiguous elements through a gather/scatter operation, with an increasing range that the elements are accessed by one operation span. The second type of accesses are designed to measure the cost of contiguous accesses, but with increasing distance between two consecutive accesses.

The results are shown in Figure 2. Each of the four operations, load, store, gather, or scatter, is repeatedly conducted on the 1M access ranges, which means that each data point in Figure 2 represents the time spent on accessing 16M float numbers. From the figure, we could see that the performance of `gather/scatter` is close to that of `load/store` till the access range size of 256. With the access range becoming larger, the performance gap becomes much larger. This is because the 16 elements accessed by each `gather/scatter` operation become farther from each other, and thus each operation involves an increasing number of cache lines. The figure also indicates a performance drop for `load/store` as the access range gets larger. This is because as distances between the different consecutive operations are becoming larger, the likelihood of a cache miss is increasing.

Overall, this experiment indicates the importance of limiting the access range of `gather/scatter` operations for obtaining effective SIMD parallelization. Our proposed execution method focuses on improving the `gather/scatter` performance, as well as the cache performance, through locality enhancement.

4. Exploiting New SIMD Architectural Features for Irregular Applications

We present our proposed methodology initially focusing on irregular reductions. Later, we will show how the same basic idea can be applied on other irregular application classes.


Node Data Array (Reduction Array)
Node Data Array (Reduction Array)
Group 1: ❶❺❻❾ Group 3: ❸❽
Group 2: ❷❹❼
Dest Vertices
Source Vertices
Group 1: ❶❷❸❺
Group 2: ❹❻❼❽

(a) Irregular Reduction (b) Graph Algorithm

Figure 3. Access Patterns of Irregular Reductions and Graph Algorithms

4.1 Irregular Reductions

As analyzed in Section 2.1, the computations in an irregular reduction can be summarized as reading and updating *data arrays*, indexed by the non-zeros from a sparse matrix. Because of the sparsity and randomness of the matrix, the accesses to the data arrays are non-continuous, and in most cases, the stride between accesses can be very large, leading to very low memory access efficiency. At the SIMD level, though `gather/scatter` enable the irregular bulk read-/write, the performance is not expected to be high, as can be inferred from Figure 2. In addition, cache performance is also negatively impacted by the large access strides.

Our main idea is what we refer to as *hierarchical tiling*, where the sparse matrix is stored as tiles of several different *tile sizes*. A tile is a square portion of the imaginary dense matrix – however, only the non-zeros in each tile are actually stored. Thus, a tile is represented as a COO format sparse matrix, including three aligned arrays: $rows$, $cols$, and $vals$. Figure 3(a) shows the tiling result of the sparse matrix involved in an irregular reduction kernel – for simplicity, we have tiles of only one size, which is 4×4 in this example.

Algorithm 1: Transforming a Tile to Conflict-free Groups

```
Input: $tile$: A tile in COO format
Output: A vector of *conflict-free* COO groups
$vector < COO > groups$;
foreach $nnz$ in $tile$ do
    $grouped = false$;
    foreach $g$ in $groups$ do
        // $g$ is not full and $nnz$ is compatible with $g$
        if $g.size < 16$ && $nnz.row \notin g.rows$ &&
        $nnz.col \notin g.cols$ then
            $g.add(nnz)$;
            $grouped = true$;
    // $nnz$ is incompatible with any existing group
    if !$grouped$ then
        // create a new group and add $nnz$ to it
        $groups.push\_back(empty\_coo)$;
        $groups.back().add(nnz)$;
return $groups$;
```

During the execution, a tile is processed exclusively by one thread, focusing on exploiting the SIMD instructions. Our goal is to have sufficient number of non-zeros in each tile, and thus to have a high SIMD utilization ratio. At the same time, unnecessarily large tile sizes can negatively impact locality in accessing the data arrays. Certain portions of the sparse matrix can have a large concentration of non-zeros as compared to others – thus, dividing the entire sparse matrix into tiles of one size is unlikely to be effective. This is the motivation behind our idea of hierarchical tiling. Specifically, we tile the sparse matrix iteratively, with the tile size doubled after each iteration. For example, we first extract 128×128 tiles, with the number of non-zeros in each tile being no less than a certain predefined threshold n, from the original sparse matrix. We next extract 256×256 tiles, each containing at least n non-zeros, from the remaining sparse matrix. We limit the total number of distinct tile sizes to 3 in our implementation, as very large tiles only harm the data access locality. At the last iteration, the remaining sparse matrix is already very sparse, and we set the threshold n as 1: any tile with at least 1 non-zero is extracted. Typically, the first tiling iteration extracts most of the non-zeros, if the threshold n is set to a not too large value, e.g., 32.

The next issue is of *write conflicts*. To motivate this issue, consider again Figure 3(a), which also shows the node data and reduction array accesses for each tile. Non-zeros in a same row/column lead to reads on the same node data, and similarly, they also lead to writes to the same position in the reduction array. This access pattern is challenging for both SIMD and MIMD executions, because of the write conflicts. At the SIMD level, write conflicts among SIMD lanes should be completely avoided, as no locking mechanism is supported across SIMD lanes. Though locking is able to protect concurrent read/write conflicts among threads at the MIMD level, it is critical to avoid these. This is because not only the locking operation itself is expensive, but the contention among the large number of threads can also lead to a significant overhead.

In our approach, we remove write conflicts by further partitioning the non-zeros in each tile into *conflict-free* groups. The simplest way of organizing conflict-free groups is to partition the tile by diagonals, as no two non-zeros along one diagonal can have the same row ID or the same column ID. But due to the sparsity of the tile and the short diagonals in the upper right and lower left sides, the length of each group might be too short to efficiently utilize the SIMD lanes. In view of these, we derive a more effective greedy conflict removal method, which is shown through Algorithm 1. For a non-zero nnz, we try to pack it to the first conflict-free group g that it is *compatible* with. By *compatible*, we imply that there is no non-zero in g that has either same row ID or column ID with nnz. The overall idea is to improve SIMD efficiency by merging multiple sparse diagonals into groups. We limit the maximum size of each conflict-free group to be the vector length of the SIMD lane, i.e., 16 for `float`. Greedy grouping is performed within each tile, which has constant number of non-zeros. And thus the overhead of the grouping is eventually linear to the total number of tiles, and thus linear to the total number of non-zeros.

The output of the algorithm is a vector of conflict-free COO groups. The arrays ($rows$, $cols$, and $vals$) in each COO are padded to the SIMD vector length (i.e., 16). Each COO group is executed in SIMD, with each lane processing one non-zero. The $rows$ and $cols$ are used as indices in `gather` and `scatter` for accessing node/reduction arrays. $vals$ is read using aligned `load` operations.

Similar to SIMD processing, we also need to consider write conflicts among threads that are processing different tiles. If two tiles involve the same row/column, write conflicts during the reduction stage can arise. To avoid locking (and associated contention) overheads, we group the tiles into *conflict-free* tile groups, using a same logic as Algorithm 1 (though without the maximum group size (16) limit). Each parallel step processes one such tile group using all available threads. The time complexity of the tile-level grouping is $O(t^2)$, where t is the total number of tiles. The grouping cost is small, due to the fact that tiles are usually large and thus the number of tiles is very small.

4.2 Graph Algorithms

We had previously summarized in Section 2.2 how graph algorithms are also computations involving sparse matrices and vertex arrays. They have both similarities and differences with the irregular reductions, and thus, though we can broadly use the same methodology, we need to make certain modifications as well.

We now use a representative application, `Bellman-Ford`, to show the typical properties of graph algorithms – especially those that can be processed by one of the popular data parallel graph frameworks like Pregel [20]. `Bellman-Ford` is an iterative kernel conducted over a graph $G = (V, E)$. It computes the shortest distance from the source s to each vertex $v \in V$. In each iteration, it conducts a *relaxation* over each edge $e(u \rightarrow v) \in E$, by decreasing the distance value of v, if $v.d > u.d + e.w$. Overall, each iteration loops on the edges (non-zeros) in the graph (sparse matrix), and the relaxation steps use each source vertex's value ($u.d$), together with the value of the non-zero element in the sparse matrix ($e.w$), to perform a computation ($u.d + e.w$), and update the destination vertex's value ($v.d$).

The set of edges can again be stored as a sparse matrix, and we can apply the same hierarchical tiling to the sparse matrix. However, the data access pattern of graph algorithms has one critical difference over those in irregular reductions. As is shown in Figure 3(b), the data access for computing associated with one edge (non-zero in sparse matrix) involves the following three steps: 1) Reading source vertex data from the array on the left, 2) Loading edge data from the sparse matrix, and 3) Updating the destination vertex on the top. Specifically, the left array is read-only, and only the top array is written. This introduces a critical difference in the algorithm for finding conflict-free groups – i.e., we only need to group the non-zeros according to the column ID, and not both row and column IDs. Two non-zero values in a tile can be grouped in the same *conflict-free* group as long as they have distinct column IDs. An example of grouping is shown by Figure 3(b): the non-zeros in the central tile are grouped into two groups. SIMD processing can again be applied to each of the groups. For MIMD parallelization, write conflicts exist only among the tiles in the same column, so the tile columns are scheduled as basic task units to threads.

4.3 Sparse Matrix-Matrix Multiplication

Unlike irregular reductions and graph algorithms, Sparse Matrix Matrix Multiplication (SpMM) is an irregular application that accesses multiple sparse matrices. As a result, the data access pattern is quite different. Nevertheless, our methodology is still broadly applicable, as we will show here. As a background, a sequential kernel for computing SpMM involves iterating over the non-zeros in the first matrix A. For each non-zero e that is inspected, according to its column ID $e.col$, we *scale* the non-zeros in the row of $e.col$ from the second matrix B. The scaling results are accumulated to a hash table corresponding to the row $e.row$.

Before presenting our proposed method, we discuss similarities and differences from the work done in the context of GPUs. Demouth *et al.* proposed an implementation of SpMM on the GPU [7]. The idea is to load a number of non-zeros from a row of A in a SIMD fashion, and use these elements to scale the elements in the corresponding row in B, based on the column ID of each element from A. The scaling operation is a SIMDized multiplication, and the multiplication result has to be accumulated back to a hash table corresponding to the specific row in the result matrix C. This method is not going to be efficient for the modern SIMD machines, as the accumulation of the multiplication result is challenging – because the column IDs of the non-zeros in the row from B are not continuous, one cannot do the reduction in a SIMD manner. Instead, one needs to unpack the SIMD multiplication result (from a vector) and reduce the scalar values to the hash table. Through our preliminary experiments, we determined that the unpack step is very time consuming (taking 85% of the total execution time).

We now describe our proposed method, which is based on the fact that most sparse matrices (especially the ones from real world data, e.g., social network graphs) exhibit clustered distribution of non-zeros. Among the 2,547 sparse matrices in the UFL Sparse Matrix Collection [6], which arise from

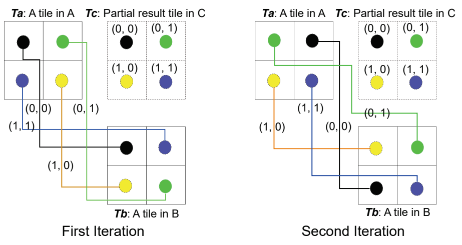


Figure 4. SIMDized Multiplication between Two 2×2 Tiles

real world applications and are also being used in our evaluations, 84% demonstrate clustered distribution of non-zeros, according our examination on a randomly sampled subset (with 100 samples). Accelerating these implicit regular components (clustered non-zeros) with SIMD can provide impressive speedups for the entire computation. Our methodology involves a *hybrid* storage format for a sparse matrix. We tile the sparse matrices A and B into 4×4 *tiles*, with each non-empty tile being stored as a dense matrix. Such 4×4 tiles fit the modern SIMD lane width for the `float` type. These dense matrices (tiles) are indexed by CSR. The processing will be similar to the sequential code, and the only difference is that multiplications need to be performed between these dense tiles instead of the scalar non-zeros. Because of the size and storage of the tiles, each such operation can be performed using SIMD lanes. To divide the workload among the threads, each thread is responsible for a subset of rows of the tiled matrix A. The computation results from different threads are reduced to independent hash tables corresponding to the different rows in the result, and thus, there are no write conflicts between the threads. The values stored in the hash table are dense tiles, instead of scalars, and thus the reduction operations are conducted in a SIMD fashion.

The challenge of eliminating write conflicts while maintaining a high SIMD utilization ratio still remains. We show that we can avoid the inter-lane conflicts by permuting the multiplications and the SIMD lane mapping. Figure 4 shows how SIMD multiplication is conducted between two dense tiles (T_a and T_b). For simplicity of illustration, we use 2×2 tiles, so that 2 iterations are needed, each iteration producing and accumulating 4 multiplication results. Similarly, for 4×4 tiles (as is used in our implementation on Intel MIC), 4 iterations are involved, each iteration producing and accumulating 16 multiplication results. Each shade in the figure corresponds to one SIMD lane. In each iteration, a corresponding one-to-one mapping between the elements in T_a and T_b is used for SIMD multiplication, and the multiplication result is reduced to the corresponding elements in the partial result tile T_c, in a SIMD fashion, using respective index mappings. After the tile-wise multiplication, T_c is reduced to the hash table, also in a SIMD fashion. For the example in Figure 4, the multiplication for the first iteration is:

$$
\begin{aligned}
T_c\left(0,0\right) += T_a\left(0,0\right) * T_b\left(0,0\right),\\
T_c\left(0,1\right) += T_a\left(0,1\right) * T_b\left(1,1\right),\\
T_c\left(1,0\right) += T_a\left(1,1\right) * T_b\left(1,0\right),\\
T_c\left(1,1\right) += T_a\left(1,0\right) * T_b\left(0,1\right);
\end{aligned}
$$

and the multiplication for the second iteration is:

$$
\begin{aligned}
T_c\left(0,0\right) += T_a\left(0,1\right) * T_b\left(1,0\right),\\
T_c\left(0,1\right) += T_a\left(0,0\right) * T_b\left(0,1\right),\\
T_c\left(1,0\right) += T_a\left(1,0\right) * T_b\left(0,0\right),\\
T_c\left(1,1\right) += T_a\left(1,1\right) * T_b\left(1,1\right).
\end{aligned}
$$

The mapping above ensures that the multiplication in each iteration does not have inter-lane write conflicts, as the SIMD lanes accumulate multiplication results to different elements in T_c. As the multiplication between any two dense tiles follows a same pattern, the mapping indices are hard coded.

The 4×4 tiles contain 16 elements, and can match the SIMD lane perfectly if the elements are of type `float` or `int`, as the size of the tile equals the SIMD lane width (64 bytes). In each of the above steps, only one SIMD instruction is executed. On the other hand, if the element type is `double`, in each step, the same SIMD statement is executed twice, each for 8 `doubles`. Also, one can see that the `gather/scatter` intrinsics used here are very efficient, as the elements that are accessed together within a tile are physically stored close to each other.

4.4 Matrix Reordering for Irregular Reductions and Graph Algorithms

Graphs and unstructured meshes often have (large) variance in connectivity, resulting in irregular distributions of non-zeros. This can make it hard to select a sufficient number of dense slides, leading to low SIMD utilization ratio and poor data access locality. Sparse matrix reordering, using software like Metis [14], has been widely used for irregular applications [22, 26, 31]. In our context, we try to apply such a reordering as a precursor to tiling, to reduce the irregularity. Specifically, we use the software Metis to divide the nodes into a number of partitions. The goal in creating these partitions is to maximize the number of edges within each partition, and minimize the number of edges crossing the partitions. The nodes are renumbered according to the partitioning result. We do note that such partitioning can be quite expensive, and the cost/benefit needs to be evaluated.

4.5 Discussion

Preprocessing Overheads: Our proposed methods introduce certain preprocessing overheads, as is the case with any other existing optimization techniques. The tiling overhead is linear to the input size, as only one pass over the input is required. If we choose to reorder the data before tiling – Multilevel k-way Partitioning [14] used in Metis also has a linear time complexity, though it has a much larger constant factor and execution time. However, these overheads can be amortized because they need to be applied only once. For example, most of the irregular application kernels used in scientific or data intensive computations are iterative kernels. Similarly, a graph application can involve a large number of iterations, and/or a number of different algorithms may be applied on the same dataset.
Other Irregular Applications: Our work is also related to existing work in the context of Sparse Matrix Vector Multiplication (SpMV) [4, 19, 30]. Our method, if applied to

SpMV, will create very similar schemes as the one reported in these studies. Thus, the main contribution of our work is generalizing these ideas to be applicable to irregular reductions, graph applications, and SpMM.

5. Evaluation Results

A series of experimental studies were conducted to evaluate our methodology for each application subclass we have targeted. Specifically, the goals were to evaluate: 1) Impact of vectorization – we compare the performance of our approach against serial code, serial code with tiled inputs, as well as a *straightforward* SIMD implementation for each application. For irregular reductions and graph algorithms, we also evaluate the impact of tile sizes. 2) Overall performance that is delivered by using combined MIMD and SIMD parallelism on Intel Xeon Phi, and 3) Performance one can obtain from matrix reordering prior to tiling, for irregular reductions and graph algorithms.

5.1 Evaluation Environment

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

5.2 Applications Used

	Application	Dataset	Dimensions	NNZ
Irregular Reductions	Moldyn	32-3.0r	131K*131K	11M
		45-3.0r	365K*365K	30M
	Euler	gsm_106857	589K*589K	11M
		kron_g500-logn19	524K*524K	22M
Graph Algorithms	Bellman-Ford	soc-Pokec	1.6M*1.6M	31M
		higgs-twitter	457K*457K	15M
	PageRank	soc-Pokec	1.6M*1.6M	31M
		higgs-twitter	457K*457K	15M
SpMM	SpMM	msc10848	11K*11K	620K
		conf5_4-8x8-05	49K*49K	2M
		shipsec5	180K*180K	5.1M
		crankseg_1	53K*53K	5.3M
		appu	14K*14K	1.9M
		598a	111K*111K	742K

Table 1. Applications and Datasets Used in the Experiments

Table 1 shows the applications and datasets that are used for each target subclass. For each application, multiple datasets of different sizes are used. The evaluation of irregular reductions is based on `Moldyn` and `Euler`, two popular kernels that are widely used. `Moldyn` simulates the interaction and motion of molecules in a period of time, and involves an irregular reduction kernel for computing the forces among molecules based on the interactions, which are stored in an interaction list. `Euler` is a Computational Fluid Dynamics (CFD) simulation on an unstructured mesh. The input matrices for `Euler` come from the UFL Sparse Matrix Collection [6]. In applying our methodology, the interaction list of each of these applications are transformed into standard sparse matrices. For both applications, the computation was conducted with type `float`.

We also use two popular graph applications – the first application, `Bellman-Ford`, was described in Section 4.2. The other application that is used is `PageRank` [2]. This application assigns weights to each element in a hyper linked set of documents (represented as a graph). It is widely used in search engines to estimate the importance of a web page and is an iterative application. In each iteration, each vertex distributes its rank evenly to all the vertices it directs to. After several iterations, the rank values of the vertices will precisely reflect the relative importance. The graphs used in both applications are stored as sparse matrices. The two graphs used are from the *SNAP* [18] graph dataset. `float` is the basic type for the computation for both the applications.

For `SpMM`, we only evaluate the kernel itself, with different matrices as the input. Each matrix is multiplied with itself. The matrices we use vary in sizes and in the properties of non-zero distributions – specifically, we tried both clustered and randomly distributed matrices. We show test results with both single and double precision executions. We compare our optimized approach with different other implementation methods, including the straightforward approach mentioned in Section 4.3, as well as the highly optimized Intel MKL Sparse BLAS Level 3 routines `mkl_scsrmultcsr` (for float) and `mkl_dcsrmultcsr` (for double). The parallel MKL routines benefit from both MIMD and SIMD. To the best of our knowledge, these are the only SpMM routines in MKL available at the time of writing this paper. The matrices used for the tests are also from the UFL Sparse Matrix Collection.

For both irregular reductions and graph algorithms, we compare against a recently published method, proposed in the context of `Moldyn`, by Pennycook *et al.* [24]. The approach here is based on an adjacency list view, so that each element is associated with a row of neighbors. The adjacency list is processed row by row, and for each row, the data of the row element is loaded into a vector V_r. The processing of the neighbors is conducted in a SIMD manner: for every SIMD step, data for 16 neighbors are loaded using `gather` into a vector V_n, according to the IDs of these neighbors. SIMD computation is conducted between V_r and V_n. Each SIMD computation result is scattered back to the reduction array for the neighbors, again according to the neighbor IDs. For MIMD execution, this method is not expected to scale well due to the fact that it uses local reductions to avoid inter-thread (inter-row) conflicts, and an expensive final combination has to be conducted among threads. Thus, we only compare the single thread SIMD execution with this method.

The results reported here (including the execution times and speedups) are the average value for a single iteration (averaged over 100 iterations).

5.3 SIMD Utilization Ratio

The average SIMD utilization ratio is calculated as $orig\_nnz/padded\_nnz$, where $orig\_nnz$ stands for the number of non-zeros in the original input, and $padded\_nnz$ is the number of non-zeros after the tiling. The latter is larger than $orig\_nnz$, as padding is used in both conflict-free groups (for irregular reductions and graph algorithms), and the 4×4 tiles in SpMM. Before showing the execution results for any application, we analyze the SIMD utilization ratios for all the datasets that are used in our study. Figure 5 shows the SIMD utilization ratio for datasets used in irreg-

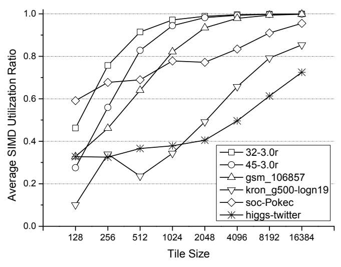


Figure 5. SIMD Utilization Ratios of Datasets (Used by Irregular Reductions and Graph Algorithms) under Different Tile Sizes

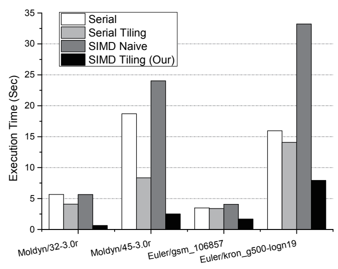


Figure 7. Single Thread Performance of Moldyn and Euler with Different Approaches

ular reductions and graph algorithms, under different tile sizes. The tile size here indicates the size used in the first tiling iteration – the subsequent tiling iterations just double the tiling size of the previous one.

The utilization ratio keeps increasing with the tile size getting larger, due to the fact that more non-zeros are included in each tile, and that less padding is used. For four out of the six datasets shown in the figure, SIMD utilization ratios is above 80% for tile sizes larger than 4096. Moreover, three datasets have utilization ratios above 90% at the tile sizes of larger than 2048. SIMD utilization ratios for the four clustered matrices used in SpMM vary between 47.2% and 55.8%, which are acceptable since the multiplications between tiles are completely regular, and combined with the optimized access locality, the vectorization performance is acceptable, as will be shown in later sections. The SIMD utilization ratios for the two random matrices (*appu* and *598a*) are very low, only around 7% for both, and these matrices do not fit our approach.

5.4 Results from Irregular Reductions

5.4.1 SIMD Parallelism

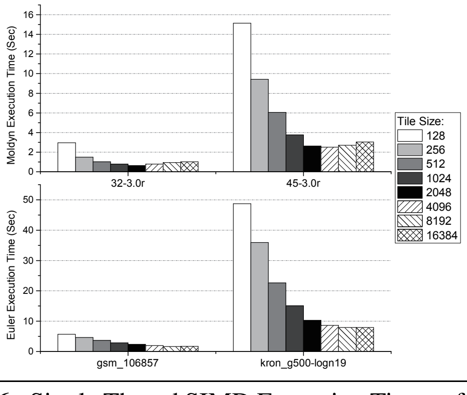


Figure 6. Single Thread SIMD Execution Times of Moldyn and Euler under Different Tile Sizes

Figure 6 shows the execution times of `Moldyn` and `Euler`, under different tile sizes, with single-thread SIMD execution using our approach. We also measured L1 cache miss rates with Intel VTune. With the tile size increasing, the L1 cache miss rates first decrease and then increase. This is because very small tile size imply that `gather/scatter` operations switch among the tiles frequently. On the other hand, very large tiles also have bad data locality inside each tile. The overall performance is impacted by SIMD utilization ratio (increasing with tile sizes) and the cache miss rate (decreasing and then increasing), and the increasing access range of `gather/scatters`.

To see the relative performance of different SIMD parallelization schemes, we plot the performance of different execution configurations that all use only one thread. The versions are *Serial* (serial code on original input, baseline), *Serial Tiling* (serial code on tiled input, with the tile size that gives the best performance), *SIMD Naive* (straightforward vectorization used in [24]) and *SIMD Tiling (Our)* (vectorization using our method, with the tile size that gives the best performance). The results are shown in Figure 7. We can see that compared with serial codes, straightforward vectorization does not achieve better performance (slows down in most cases). There are several reasons behind this – large data access ranges of `gather/scatter`, less efficient cache usage due to poor data locality, as well as inefficient inter-lane reductions. On the other hand, tiling significantly optimizes the data locality, which can be seen from the fact that *Serial Tiling* is obviously faster than *Serial*, with 1.03x to 2.24x improvement. The cache miss rates of *SIMD Naive* are all much higher than those of *SIMD Tiling (Our)*, based on the VTune profiling results. The overall performance of *SIMD Tiling (Our)* is significantly better. For `Moldyn`, the speedup over *Serial* is 9.05, and 7.47, on the two different inputs, respectively. For `Euler`, the speedups are 2.09 and 2.01. The reasons why `Euler` has smaller speedups are that the inputs are much sparser than those used by `Moldyn` and that `Euler` has a much lower computation to memory access ratio.

5.4.2 Overall Performance of MIMD+SIMD

The performance of versions that used different number of cores and performed SIMD execution on each core is shown in Figure 8. The speedups shown in the figure are over the serial executions. For each of the two irregular reduction computations, we plot the scalability under different tile sizes. The overall trend indicates that smaller tiles have better scalability than larger tiles, with the number of threads increasing – the reason being that smaller tiles result in higher inter-thread parallelism, as well as better load balance. For example, the speedups using tile size of 128 keeps increasing till over 200 threads, for all the cases. However, smaller tiles have low SIMD utilization ratios. Thus, the best overall performance is typically achieved at a medium tile size, where the right trade-off between SIMD and MIMD parallelism is

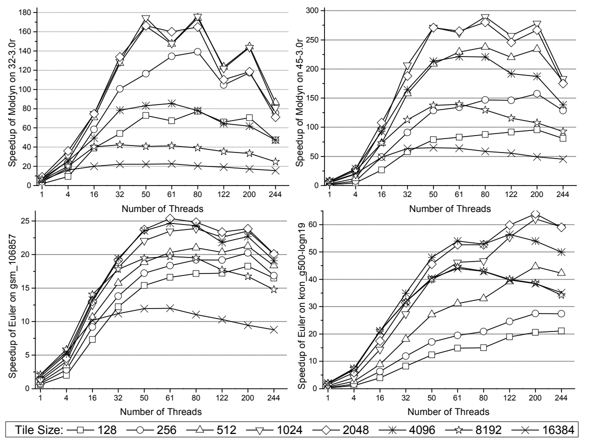


Figure 8. SIMD+MIMD Speedups of Irregular Reductions over Serial (Single Thread Scalar) Execution

achieved. The best speedups of `Moldyn` are 176 and 290, for *32-3.0r* and *45-3.0r*, respectively, both with 1024 tile size and 80 threads. For `Euler`, a 25.37x speedup is achieved for the small dataset, *gsm_106857*, using tile size of 2048 and 61 threads. The larger dataset, *kron_g500-logn19*, achieves a higher speedup of 64 at the thread of 200, using a tile size of 2048.

5.4.3 Matrix Reordering Optimization

For each dataset that is used above, we also generated a reordered version with Metis, using the approach described in Section 4.4. The same optimization methodology was then applied to reordered matrix. The reordered datasets are able to provide further speedups for certain applications and datasets. For `Moldyn`, speedups of 1.3 and 1.69 are achieved with datasets *32-3.0r* and *45-3.0r*, respectively, according to the best performance obtained with MIMD+SIMD executions. For `Euler`, slowdowns are observed for both inputs, due to that the reordering lowers the SIMD utilization ratios – specifically, the write conflicts within each tile is more severe after reordering. In addition, it should be noted that reordering adds significant preprocessing overhead (which is not directly reported here).

5.5 Results from Graph Algorithms

5.5.1 SIMD Parallelism

Figure 9 shows the execution times of the two graph applications under different tile sizes. Same as irregular reductions, we also investigated the L1 cache miss rates, with the help of VTune. The cache miss rates nearly keep decreasing as tile sizes increase, with an exception for very large tile sizes. The execution time follows a similar trend.

We also compare the performance of different single thread execution approaches shown in Figure 10 (same naming convention as for irregular reductions). *SIMD Naive* has a small speedup over *Serial* for both `Bellman-Ford` and `PageRank`. The low speedup is due to the poor memory access performance: the cache miss rates of *SIMD Naive* are much higher than those of *SIMD Tiling (Our)*. Moreover, large access ranges for the `gather/scatter` operations also negatively impact the performance of *SIMD Naive*. *Serial Tiling* has speedups of between 2.29 and 4.1 over *Serial*, clearly showing the benefit of increasing data locality with tiling. *SIMD Tiling (Our)* achieves speedups of between 4.9 and 7.5 over *Serial*.

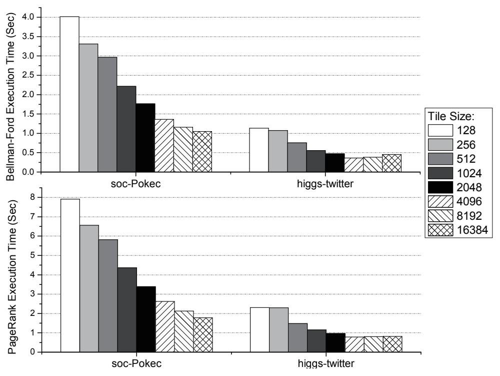


Figure 9. Single Thread SIMD Execution Times of Bellman-Ford and PageRank under Different Tile Sizes

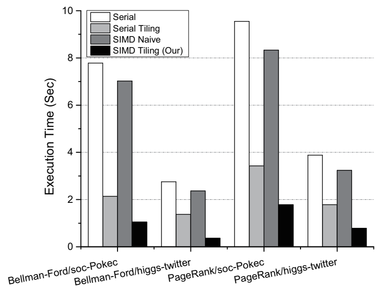


Figure 10. Single Thread Performance of Bellman-Ford and PageRank with Different Approaches.

5.5.2 Overall Performance of MIMD+SIMD

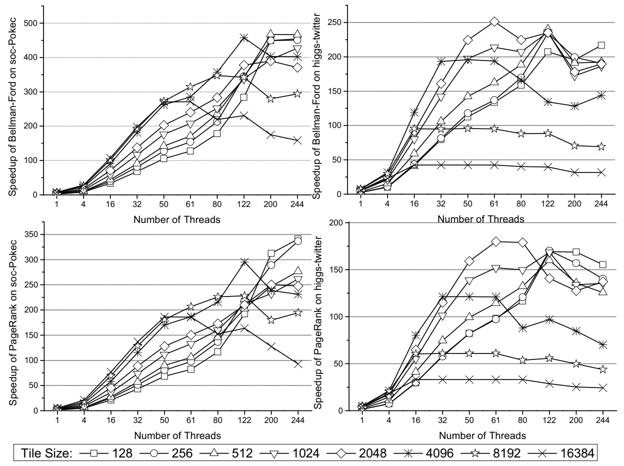


Figure 11. SIMD+MIMD Speedups of Bellman-Ford and PageRank over Serial (Single Thread Scalar) Execution

Figure 11 shows the overall speedup of SIMD+MIMD over the serial code. Similar with irregular reductions, smaller tile sizes scale better than larger tile sizes, due to higher degree of parallelism and better load balance among the threads. For `Bellman-Ford`, the best speedup achieved for *soc-Pokec* is 467.1, using tile size of 512 and 200 threads, whereas for *higgs-twitter*, the best speedup is 251, achieved at the tile size of 2048 and 61 threads. For `PageRank`, on dataset *soc-Pokec*, the highest speedup is 341, at the tile size of 128, and 244 threads. Finally, for dataset *higgs-twitter*, the highest speedup is 180, with a tile size of 2048 and 61 threads.

5.5.3 Matrix Reordering Optimization

Similar to irregular reductions, we conduct matrix reordering for better non-zero distributions. According to the best performance with MIMD+SIMD executions, `Bellman-Ford` obtains further speedups of 1.42 and 1.25, and `PageRank` obtains speedups of 1.08 and 1.05 with *soc-Pokec* and *higgs-twitter*, respectively.

5.6 Results from SpMM

We evaluate the performance of SpMM using different data sets and different data types. The sparse matrices that are used are of two types: *clustered* (involving a number of dense blocks) and *random* (randomly distributed non-zeros). As noted in the previous section, a vast majority of matrices in the most popular collection of sparse matrices are clustered, and our technique focuses on exploiting such clustering of non-zeroes. For completeness, we have also experimented with matrices without such clustering, referred to as random matrices. Since the tile size is fixed for SpMM, the results from our approach are all based on 4×4 tiling.

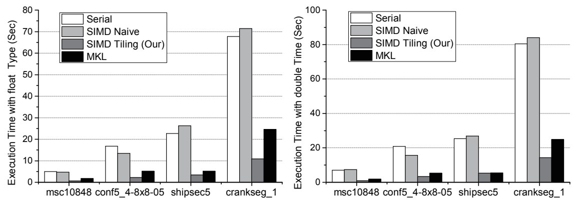


Figure 12. Single Thread Performance of SpMM with Clustered Datasets using Different Data Types.

Execution with Clustered Matrices: Figure 12 shows the single thread performance of SpMM on four clustered matrices. We compare four different versions. *Serial* (single thread scalar execution, baseline), *SIMD Naive* (single thread row by row SIMD processing, as is described in Section 4.3), *MKL* (MKL single thread execution, which also utilizes SIMD), and *SIMD Tiling* (our tiling based single thread SIMD execution). For `float` type execution, *SIMD Tiling* is much faster than the other two versions, achieving speedups of between 6.58 and 8.07, over *Serial*. Similarly, for `double` type executions, the speedups vary from 4.75 to 7.70. *SIMD Naive* does not have noticeable speedups, and even have a slowdown for certain datasets, due to the inefficient memory accesses and scalar result accumulation. Moreover, *SIMD Tiling* also outperforms *MKL* by 1.50x to 2.89x, and 1.01x to 1.97x, for `float` and `double` types, respectively. MKL implementation is not as efficient as our approach. This is because MKL directly uses CSR format for input matrices, and thus it is not able to benefit from the enhanced data locality and efficient SIMD processing.

Execution with Random Matrices: We used two matrices, *appu* and *598a* from Table 1 to evaluate the effect of our approach on random matrices. As is expected, obvious slowdowns of 6.48 and 3.88 (over *Serial*) were observed. This is due to that the poor clustering property leads to a significant amount of padding to the tiles, and thus very low SIMD utilization ratio.

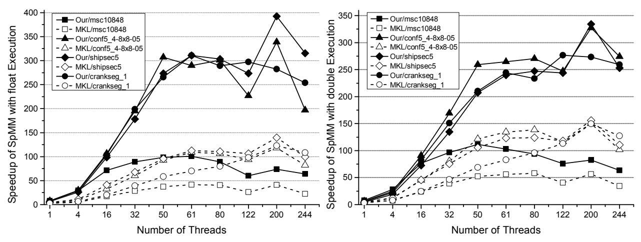


Figure 13. Speedups of SpMM with MIMD+SIMD Parallelism over Single Thread Scalar Execution

Overall Performance of MIMD+SIMD: The scalability of multi-thread execution using SIMD is shown in Figure 13. The test was conducted using clustered matrices, as random matrices are not benefiting from our optimization. Since the computation of different rows do not have data dependency, we simply parallelize the rows using OpenMP, for multi-threaded execution. The results include both `float` and `double` executions. Speedups are against the single thread scalar executions. To show the overall efficiency of our approach over commercial libraries, we plot both the speedups of our approach (in solid lines) and the speedups of Intel MKL (in dashed lines), by varying the number of threads. For the small dataset, *msc10848*, our approach is able to achieve the maximum speedup of 100.64 with 61 threads for `float` execution, and the maximum speedup of 111.77 with 50 threads for `double` execution. More threads do not improve the overall performance, due to the limited amount of data and parallelism, as well as possible load imbalance because of the varying row lengths. Higher speedups are seen from the other three datasets. For `float` execution, the maximum speedup of 392.49 is achieved for dataset *shipsec5*, with 200 threads. For `double` execution, the maximum speedup of 334.34 is achieved using the same dataset and same number of threads. Compared with MKL, our approach is much faster, in terms of both performance with different thread configurations and peak performance. The maximum speedups of both approaches are achieved using dataset *shipsec5*, with 200 threads. Our approach is 2.81x and 2.14x faster than MKL, for `float` and `double` executions, respectively.

6. Related Work

Many efforts in recent years have focused on accelerating irregular applications on various parallel platforms, including multi-core machines, GPUs, as well as vector hardware. Some of these studies are focusing on specific applications, and a few are focusing on proposing a generalized approach for a subclass.

Application-specific Solutions: Strout *et al.* [29, 28] improved the data locality for irregular kernels like Gauss-Seidel through rows/columns permutation and tiling, on traditional CPU architectures, but did not consider SIMD parallelism. Pennycook *et al.* implemented *Moldyn* on the Xeon Phi, using both MIMD and SIMD parallelism [24]. This is a very initial investigation, using a relatively straightforward implementation specific to a single application. We have shown how we significantly outperform their implementation. Iwashita *et al.* [12] proposed an approach that uses blocked coloring scheme to accelerate ICCG solver for multi-threaded execution. Their work was specific for ICCG and does not involve SIMD parallelism. Park *et al.* [23] utilized the same coloring approach to vectorize the computation of CG. Similarly, Thébault *et al.* used a similar approach for vectorizing unstructured 3D mesh computations. The conflict removal step in our method involves a grouping algorithm. While it has some similarities with the coloring approach, it is lighter-weight. Our work is distinct in not only proposing methods for efficient vectorization and parallelization across multiple application classes, but we are also introducing a formal way of identifying data access patterns for different irregular application subsets.

Studies Targeting Subclasses: Recent work for optimizing graph processing using SIMD-like instruction set have mostly been in the context of GPUs. Merrill *et al.* [21] parallelized breadth-first search on the GPUs by focusing on fine-grained task management. Hong *et al.* [10] developed a novel virtual warp centric programming method to address the work imbalance problem in graph algorithms for GPUs. CuSha [15] optimized graph processing on GPUs with intensive usage of shared memory, by re-organizing the graph data in shards. Saule and Catalyurek [27] provided a preliminary evaluation on graph applications on the Xeon Phi architecture, but without particular optimizations specific to the applications and the hardware. Choi *et al.* [4] proposed a way of optimizing SpMV on GPUs by storing sparse matrices into small subblocks, each represented as a dense matrix. Their method aims to reduce the amount of column indices stored for the non-zeros, and thus increasing the computation to memory access ratio. Similar with our tiling-based approach for SpMM, the method is also based on the observation that most sparse matrices have the clustered property. The key difference in our work is to tile the two sparse matrices, and to vectorize the computation between every pair of the tiles, using the wide SIMD lanes on the Intel MIC. Liu *et al.* [19] used ELLPACK sparse block format, to optimize SpMV kernel on the Intel MIC. Similarly, Tang *et al.* [30] utilized a hybrid storage format with jagged partitioning to optimize SpMV, also for the Intel MIC. Ren *et al.* [25] focused on code transformation for SIMDizing recursive task parallel programs on vector hardware. As we have stated throughout, our work is proposing a much more general scheme applicable to a number of application classes.

Compiling for SIMD: Compilation for automatically generating SIMD code has been a topic of much investigation, though the focus has been almost entirely on regular applications. For example, Henretty *et al.* [9] designed a domain-specific language and compiler for vectorizing stencil computations. To the best of our knowledge, the only existing work on compiling irregular application on SIMD instruction sets is from Kim *et al.* [16], who focused on irregular applications. Their target hardware is the Cell processor, with vector units that do not support modern SIMD operations such as `gather/scatter`. Their approach involves software based `gather/scatter` support, through packing and unpacking. By conducting benefit-cost ratio analysis, they determine if a certain part of the code is worth the packing/unpacking. They did not conduct optimizations based on access patterns, which are performed in our work.

On the more theoretical side, Wu *et al.* [34] concluded that searching an optimal data reordering (in terms of minimizing non-coalesced data access on GPUs) for irregular data references is an NP-complete problem. Burtscher [3] *et al.* quantitatively studied the irregularity in the applications and showed that code optimizations are able to reduce the irregularity from either memory level or control-flow level.

7. Conclusions and Future Work

We have presented a general methodology for SIMD parallelization of several distinct classes of irregular applications, by representing them as sparse matrix computations. Our proposed technique includes three general steps: locality enhancement, data access pattern identification, and conflict removal. Within the general framework, we are able to modify the specific steps to account for the differences between subclasses of applications. Results from detailed evaluation show that not only we obtain significant speedups, but we also outperform other recently published methods.

Our future work will focus on applying our methodology to other classes of irregular applications, such as the adaptive methods and pointer-traversal applications. Auto-tuning for parameters such as the tiling sizes also needs to be investigated.

Acknowledgements

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

References

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

[2] Sergey Brin and Lawrence Page. The Anatomy of a Large-scale Hypertextual Web Search Engine. *Comput. Netw. ISDN Syst.*, 30(1-7):107–117, April 1998.

[3] Martin Burtscher, Rupesh Nasre, and Keshav Pingali. A quantitative study of irregular programs on gpus. IISWC '12, pages 141–151. IEEE, 2012.

[4] Jee W. Choi, Amik Singh, and Richard W. Vuduc. Model-driven Autotuning of Sparse Matrix-vector Multiply on GPUs. PPoPP '10, pages 115–126, New York, NY, USA, 2010.

[5] Raja Das, DJ Mavriplis, J Saltz, S Gupta, and R Ponnysamy. Design and Implementation of a Parallel Unstructured Euler Solver using Software Primitives. *AIAA journal*, 32(3):489–496, 1994.

[6] Timothy A Davis and Yifan Hu. The University of Florida Sparse Matrix Collection. *ACM Transactions on Mathematical Software (TOMS)*, 38(1):1, 2011.

[7] Julien Demouth. Sparse Matrix-Matrix Multiplication on the GPU, GPU Technology Conference, GTC'12.

[8] Jiri Dokulil, Enes Bajrovic, Siegfried Benkner, Martin Sandrieser, and Beverly Bachmayer. HyPHI - Task Based Hybrid Execution C++ Library for the Intel Xeon Phi Coprocessor. ICPP '13, pages 280–289, 2013.

[9] Tom Henretty, Richard Veras, Franz Franchetti, Louis-Noël Pouchet, J. Ramanujam, and P. Sadayappan. A stencil compiler for short-vector simd architectures. ICS '13, pages 13–24, New York, NY, USA, 2013. ACM.

[10] Sungpack Hong, Sang Kyun Kim, Tayo Oguntebi, and Kunle Olukotun. Accelerating CUDA Graph Algorithms at Maximum Warp. PPoPP '11, pages 267–276, New York, NY, USA.

[11] Yuan-Shin Hwang, Raja Das, Joel H Saltz, Milan Hodoscek, and Bernard R Brooks. Parallelizing Molecular Dynamics Programs for Distributed-memory Machines. *Computing in Science and Engineering*, 2(2):18–29, 1995.

[12] Takeshi Iwashita, Hiroshi Nakashima, and Yasuhito Takahashi. Algebraic Block Multi-color Ordering Method for Parallel Multi-threaded Sparse Triangular Solver in ICCG Method. IPDPS '12, pages 474–483. IEEE, 2012.

[13] U. Kang, Charalampos E. Tsourakakis, and Christos Faloutsos. PEGASUS: A Peta-Scale Graph Mining System Implementation and Observations. ICDM '09, pages 229–238, Washington, DC, USA, 2009. IEEE Computer Society.

[14] George Karypis and Vipin Kumar. A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs. *SIAM J. Sci. Comput.*, 20(1):359–392, December 1998.

[15] Farzad Khorasani, Keval Vora, Rajiv Gupta, and Laxmi N. Bhuyan. CuSha: Vertex-centric Graph Processing on GPUs. HPDC '14, New York, NY, USA. ACM.

[16] Seonggun Kim and Hwansoo Han. Efficient simd code generation for irregular kernels. PPoPP '12, pages 55–64, New York, NY, USA, 2012. ACM.

[17] Brian Kulis, Sugato Basu, Inderjit Dhillon, and Raymond Mooney. Semi-supervised Graph Clustering: A Kernel Approach. ICML '05, pages 457–464. ACM.

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

[19] Xing Liu, Mikhail Smelyanskiy, Edmond Chow, and Pradeep Dubey. Efficient Sparse Matrix-vector Multiplication on x86-based Many-core Processors. ICS '13, pages 273–282, New York, NY, USA, 2013. ACM.

[20] Grzegorz Malewicz, Matthew H. Austern, Aart J.C Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: A System for Large-scale Graph Processing. SIGMOD '10, pages 135–146. ACM.

[21] Duane Merrill, Michael Garland, and Andrew Grimshaw. Scalable GPU Graph Traversal. PPoPP '12.

[22] Leonid Oliker, Xiaoye Li, Parry Husbands, and Rupak Biswas. Effects of ordering strategies and programming paradigms on sparse matrix computations. *Siam Review*, 44(3):373–393, 2002.

[23] Jongsoo Park, Mikhail Smelyanskiy, Karthikeyan Vaidyanathan, Alexander Heinecke, Dhiraj D. Kalamkar, Xing Liu, Md. Mosotofa Ali Patwary, Yutong Lu, and Pradeep Dubey. Efficient Shared-memory Implementation of High-performance Conjugate Gradient Benchmark and Its Application to Unstructured Matrices. SC '14, Piscataway, NJ, USA. IEEE Press.

[24] Simon J Pennycook, Chris J Hughes, M Smelyanskiy, and Stephen A Jarvis. Exploring SIMD for Molecular Dynamics, Using Intel® Xeon® Processors and Intel® Xeon Phi Coprocessors. IPDPS '13, pages 1085–1097. IEEE, 2013.

[25] Bin Ren, Youngjoon Jo, Sriram Krishnamoorthy, Kunal Agrawal, and Milind Kulkarni. Efficient execution of recursive programs on commodity vector hardware. PLDI 2015, pages 509–520, New York, NY, USA, 2015. ACM.

[26] Semih Salihoglu and Jennifer Widom. Gps: A Graph Processing System. SSDBM '13, page 22. ACM, 2013.

[27] Erik Saule and Ümit V. Catalyurek. An Early Evaluation of the Scalability of Graph Algorithms on the Intel MIC Architecture. IPDPSW '12, Washington, DC, USA. IEEE Computer Society.

[28] Michelle Mills Strout, Larry Carter, Jeanne Ferrante, Jonathan Freeman, and Barbara Kreaseck. Combining performance aspects of irregular gauss-seidel via sparse tiling. In *Languages and Compilers for Parallel Computing*, pages 90–110. Springer, 2005.

[29] Michelle Mills Strout, Larry Carter, Jeanne Ferrante, and Barbara Kreaseck. Sparse Tiling for Stationary Iterative Methods. *Int. J. High Perform. Comput. Appl.*, 18(1):95–113, February 2004.

[30] Wai Teng Tang, Ruizhe Zhao, Mian Lu, Yun Liang, Huynh Phung Huynh, Xibai Li, and Rick Siow Mong Goh. Optimizing and Auto-tuning Scale-free Sparse Matrix-vector Multiplication on Intel Xeon Phi. CGO '15, pages 136–145, Washington, DC, USA, 2015. IEEE Computer Society.

[31] Loïc Thébault, Eric Petit, and Quang Dinh. Scalable and Efficient Implementation of 3D Unstructured Meshes Computation: A Case Study on Matrix Assembly. PPoPP 2015, pages 120–129, New York, NY, USA, 2015. ACM.

[32] Takashi Washio and Hiroshi Motoda. State of the Art of Graph-based Data Mining. *SIGKDD Explor. Newsl.*, 5(1):59–68, July 2003.

[33] Samuel Williams, Dhiraj D. Kalamkar, Amik Singh, Anand M. Deshpande, Brian Van Straalen, Mikhail Smelyanskiy, Ann Almgren, Pradeep Dubey, John Shalf, and Leonid Oliker. Optimization of Geometric Multigrid for Emerging Multi- and Manycore Processors. SC '12, pages 96:1–96:11, 2012.

[34] Bo Wu, Zhijia Zhao, Eddy Zheng Zhang, Yunlian Jiang, and Xipeng Shen. Complexity Analysis and Algorithm Design for Reorganizing Data to Minimize Non-coalesced Memory Accesses on GPU. PPoPP '13, pages 57–68, 2013.