2019 IEEE/ACM Workshop on Memory Centric High Performance Computing (MCHPC)

Generalized Sparse Matrix-Matrix Multiplication for Vector Engines and Graph Applications

Jiayu Li
Intelligent Systems Engineering
Indiana University
Bloomington, USA
jl145@iu.edu

Fugang Wang
Intelligent Systems Engineering
Indiana University
Bloomington, USA
fuwang@indiana.edu

Takuya Araki
Data Science Research Laboratories
NEC
Kanagawa, Japan
t-araki@dc.jp.nec.com

Judy Qiu
Intelligent Systems Engineering
Indiana University
Bloomington, USA
xqiu@indiana.edu

***Abstract*—Generalized sparse matrix-matrix multiplication (SpGEMM) is a key primitive kernel for many high-performance graph algorithms as well as for machine learning and data analysis algorithms. Although many SpGEMM algorithms have been proposed, such as ESC and SPA, there is currently no SpGEMM kernel optimized for vector engines (VEs). NEC SX-Aurora is the new vector computing system that can achieve high performance by leveraging high bandwidth memory of 1.2TB/s and long vector of VEs, where the execution of scientific applications is limited by memory bandwidth. In this paper, we demonstrate significant initial work of SpGEMM kernel for a vector engine and implement it to vectorize several essential graph analysis algorithms: Butterfly counting and Triangle counting. We propose a SpGEMM algorithm with a novel hybrid method based on sparse vectors and loop raking to maximize the length of vectorizable code for vector machine architectures. The experimental results show that the vector engine has advantages on more massive data sets. This work contributes to the high performance and portability of the SpGEMM kernel to a new family of heterogeneous computing systems, which is Vector Host (VH) equipped with different accelerators or VEs.**

***Index Terms*—Sparse Linear Algebra Kernel, NEC Vector Engine, Graph**

I. Introduction

Generalized sparse matrix-matrix multiplication (SpGEMM) is a primitive kernel for many high-performance Graph analytics and Machine Learning algorithms. Although many SpGEMM algorithms have been proposed, there is currently no SpGEMM kernel optimized for vector engines. The NEC SX-Aurora TSUBASA is a vector processor of the NEC SX architecture family[1], a CPU Machine with Vector Engine (VE) for accelerated computing using vectorization. The concept is that the full application runs on the high-performance Vector Engine, and the operating system tasks are taken care of by the Vector Host (VH), which is a standard x86 server.

As shown in Figure 1, a NEC SX-Aurora node, also called a ***Vector Island (VI)***, is comprised of a ***Vector Host (VH)*** and one or more ***Vector Engines (VEs)***. The VH is an x86 server with one or more standard server CPU running Linux operating system. One or multiple VEs are connected to each VH CPU. Inside each VE, it has 8 cores and dedicated memory.

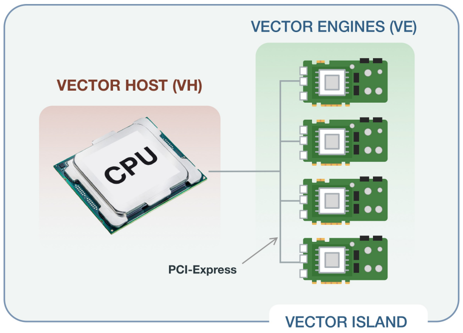


Fig. 1. Hardware configuration of NEC SX-Aurora VH and VEs

Figure 2 shows the detailed architecture of a VE. The Vector Engine Processor integrates 8 vector-cores and 48 GB of high bandwidth memory (HBM2), providing a peak performance of up to 2.45 TeraFLOPS. The computational efficiency is achieved by the unrivaled memory bandwidth of up to 1.2 TB/s per CPU and by the latency-hiding effect of the vector architecture. The single Vector Engine Processor core arithmetic unit can execute 32 double-precision floating-point operations per cycle with its vector registers holding 256 floating-point values. With 3 fused-multiply-add units (FMA), each core has a peak performance of 192 FLOP per cycle or up to 307.2 GigaFLOPS (double precision). The Vector Engine has a peak performance of up to 2.45 TeraFLOPS.

In this paper, we design a new SpGEMM kernel for the vector engine (VE) as an addition to the family of accelerators and further study two subgraph counting algorithms: Triangle counting [2] and Butterfly counting [3]. Our main contributions in this paper are:

978-1-7281-6007-8/19/$31.00 ©2019 IEEE
DOI 10.1109/MCHPC49590.2019.00012

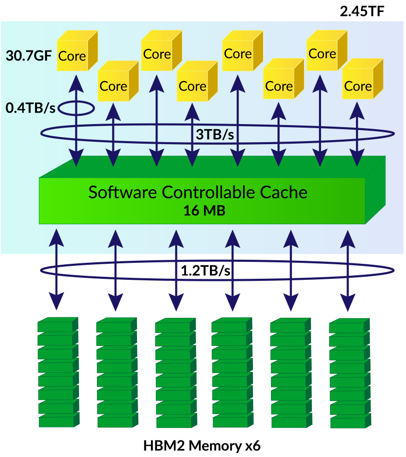


Fig. 2. NEC SX-Aurora VE architecture

- We propose a unique hybrid method that enlarges vector length for non-zeros values and leverages the High Bandwidth Memory (HBM). This enables vector architectures to exert their potentials at 1.2TB/s of HBM and 256 elements of long vector length.
- We deploy loop raking to vectorize a loop and increase the memory access efficiency.
- The SpGEMM kernel is used to implement several important graph analysis algorithms on the vector machine.

The experimental results show that the vector engine achieves high performance on large data sets. We implemented the algorithms in C++ and have made the open-source code available on Github[1][4].

II. Related Work

Counting subgraphs from a large network is fundamental in graph problems. It has been used in real-world applications across a range of disciplines, such as in bioinformatics [5], social networks analysis, and neuroscience [6]. Many graph algorithms have been previously presented in the language of linear algebra [7], [8]. The matrix-based triangle counting algorithm we use is mainly based on the text [2]. [3] proposed a fast butterfly counting algorithm, which we converted into a matrix operation form. We make it highly parallel and take full advantage of vector machines.

SX-Aurora TSUBASA is a vector engine developed by NEC. Comparison of SX-Aurora TSUBASA with other computing architectures includes: [9][10]. The hybrid SpGEMM algorithm we propose is based on the following related work: ESC Algorithm [11], Hash-based SpGEMM [12] [13], and SPA Algorithm [14]. They will be explained in the next section, together with our implementation.

III. Implementation of SpGEMM on Vector Engine

In the Compressed Sparse Row (CSR) format, we represent a matrix $\mathbf{M}_{m\times n}$, by three 1-D arrays or vectors called as `A`, `IA`, `JA`. Let NNZ denote the number of non-zero elements in $\mathbf{M}$.

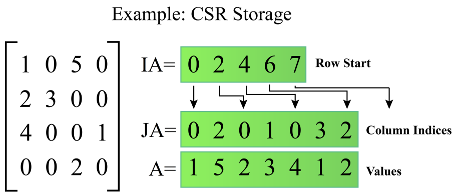


Fig. 3. CSR representation of a sparse matrix

The `A` vector is of size NNZ, and it stores the values of the non-zero elements of the matrix. The values appear in the order of traversing the matrix row-by-row. The `JA` vector stores the column index of each element in the `A` vector. The `IA` vector is of size $m{+}1$ stores the cumulative number of non-zero elements up to (not including) the i-th row. For example, to calculate the number of non-zero elements in row 0, just calculate `IA[1]-IA[0] = 2 =` NNZ_{row0}.

SpGEMM can be used for various kinds of graph algorithms, including triangle counting and butterfly counting that are addressed in this paper. Since both of the matrices are sparse, its implementation is not straightforward; there exist several algorithms called ESC algorithm, the hash-based algorithm, and sparse accumulator (SPA).

Figure 4 shows basic structure of SpGEMM algorithm. It shows $\mathbf{C} = \mathbf{A}\cdot\mathbf{B}$; we assume that the sparse matrices are in CSR format.

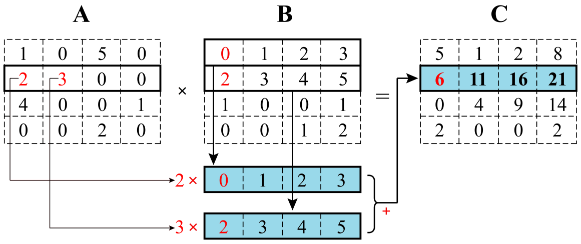


Fig. 4. Basic structure of SpGEMM algorithm

Here, each row of $\mathbf{A}\cdot\mathbf{B}$ creates each row of $\mathbf{C}$; we focus on *kth* row. The *kth* row of $\mathbf{A}$ has two non-zero elements, whose column indices are *i* and *j*. Because other elements are zero, we only need to care about *ith* and *jth* row of the matrix $\mathbf{B}$. They are multiplied by the corresponding non-zero elements of the *kth* row of $\mathbf{A}$ and added to create the *kth* row of $\mathbf{C}$. There are several algorithms to add these sparse vectors.

We implemented these algorithms on SX-Aurora TSUBASA using the technique called loop raking, and propose a novel hybrid method. To the best of our knowledge, this is the first attempt to implement SpGEMM on a vector architecture.

A. Loop raking

Loop raking is a long-forgotten technique that was proposed in the early '90s to implement radix sort [15]. However, it is

[1] https://github.com/dsc-nec/frovedis_matrix

an essential technique to enhance vectorization and enlarge vector length. The key idea of loop raking is viewing each element of a vector register as a virtual processor. Here we take the union of sets as an example to introduce the loop raking technique.

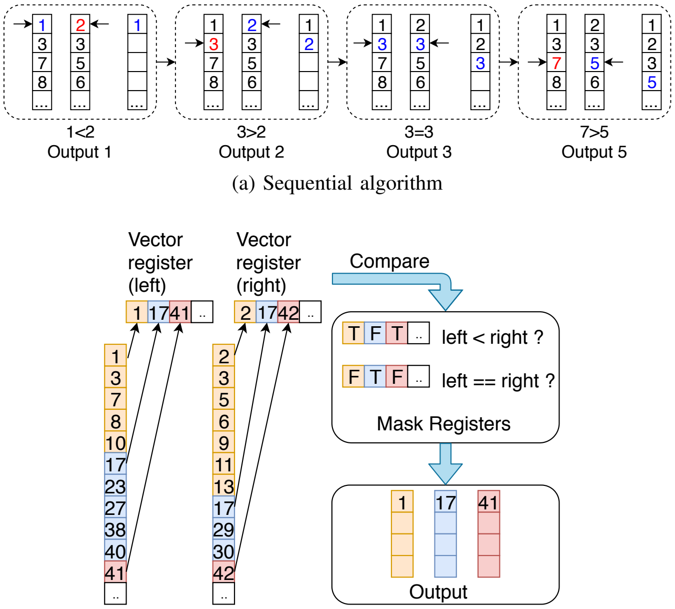
Set operations (union, intersection, merge, difference, etc.) of sorted integer can be easily implemented, but vectorizing them is not trivial.

The traditional algorithm compares two lists one by one, and the result of the comparison will determine which pointer is increased. It contains the following steps:

1) Set the pointers to the first elements of the sets.
2) Compare the data of the pointers.
3) Output smaller data and increase the pointer of it; If the data are the same, output it and increase both pointers.
4) Goto 2).

In contrast, the loop raking method divides the data set into many groups (in our case, 256 groups, since the length of vector register is 256), and comparing the first element in all groups at the same time. It consists of the following steps:

1) Divide the data into groups.
2) The first unused element of each group is placed in the vector register.
3) Compare the two vector registers (vectorized).
4) Goto 2).



(a) Sequential algorithm

(b) Loop Raking

Fig. 5. Comparison of sequential algorithm and loop raking.

Loop raking makes it possible to vectorize a loop that cannot be vectorized otherwise. Besides, it can be used to enlarge vector length. However, it has several drawbacks. One drawback is that memory access becomes non-contiguous. Especially, if the memory access becomes scatter or gather (e.g., access like a[b[i]]), the performance of memory access becomes non-optimal. Another drawback is the performance of the branch. If the computation of each virtual processor becomes complex, it might contain a branch. A loop that contains a branch can be vectorized, but it is implemented using a mask register. That is, the value of the condition is stored in the mask register, and the instruction is executed regardless of the condition; the result is reflected to register or memory according to the mask value. Therefore, if the condition becomes complex, many of the results end up unused.

B. ESC Algorithm

ESC algorithm [11] is proposed by Bell et al. for GPU. It consists of three phases, which are expansion, sorting, and compression. In the expansion phase, it creates sparse vectors multiplied by non-zeros of **A**, as explained above. This phase is done for all the rows of **A** (and **C**) in parallel. In the sorting phase, the resulting non-zeros of the sparse vectors are sorted according to the row and column indices. In the compression phase, the non-zeros that have the same row and column index are added to one non-zero value. The result becomes the matrix **C**. Each phase of this algorithm has a high parallelism. As for the sorting phase, which is the most time-consuming part of this algorithm, we used radix sort based on loop raking that is proposed in [15].

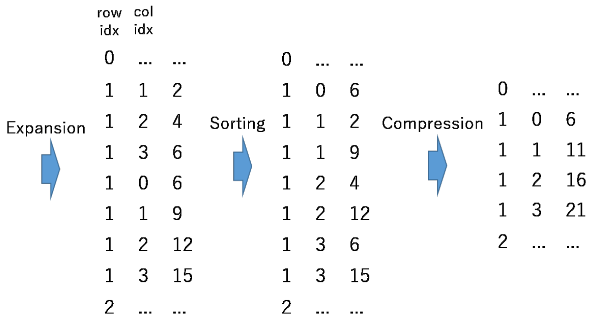


Fig. 6. Esc Algorithm

In the implementation of vector architecture, we separate the matrix A into blocks and do the steps block by block to utilize the cache (LLC), which is similar to the strategy proposed by Dalton et al. [16].

Besides, we added individual case support for matrices that have only 0 or 1 as the values (which means non-zero values are always 1). This is typical if the matrix is an adjacency matrix, and the edge weights of the graph are always 1. In this case, we can speed up the sort phase, because we only sort indices instead of pairs of index and value.

C. Hash based Algorithm

The SpGEMM algorithm in the cuSPARSE library uses the hash table for the addition of sparse vectors [12]. Where the column index becomes the key of the hash table; the value is inserted if the key is not stored in the hash table. Otherwise, the value is accumulated to the already stored value. After this process, we can get the result row by extracting the stored key value pairs from the hash table. We can do this process for all the rows in parallel.

We used loop raking technique to implement hash based algorithm; each virtual processor (element of vector register) processes different rows. In this case, each virtual processor updates the hash table for the corresponding rows sequentially; we do not have to worry about the parallel update of the hash table. To handle the collision, if a collision occurs, the key is stored in a different array; the contents of the array is processed again by adding 1 to the hash value, which realizes open address linear probing.

The column indices of the result matrix are not sorted in the case of the hash-based algorithm. Since some algorithms assume that they are sorted, we added an option to sort them, though it decreases the performance.

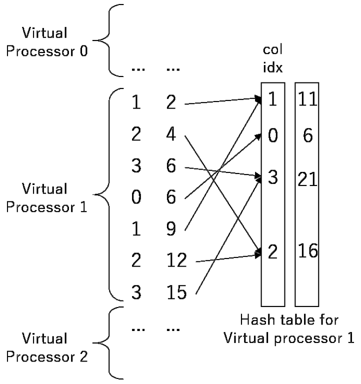


Fig. 7. Hash based Algorithm

D. Sparse Accumulator (SPA)

Sparse accumulator (or SPA) is a classic algorithm proposed by Gilbert et al. [14]. It uses dense vectors whose size is the same as the number of columns of C. The sparse vectors are added into the dense vector. There is another flag vector that contains the information if the index of the vector contains a value or not. If the corresponding index of the flag vector is false, it is set to true, and the index is saved into another vector; this vector stores the non-zero part of the dense vector after the process. By using this vector, the non-zero part can be known without scanning the flag vector or the dense vector that contains the value.

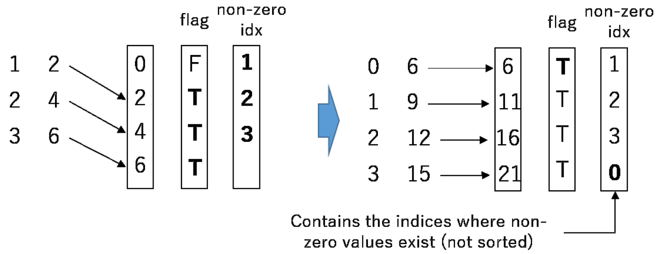


Fig. 8. SPA Algorithm

Though SPA is a quite efficient algorithm, implementations of SpGEMM for highly parallel architectures, such as GPUs, usually avoid it. This is because if multiple rows of **A** are processed in parallel, the required number of the dense vectors is the number of parallelisms, which is too large and not affordable memory size.

In the implementation of vector architecture, adding sparse vectors into the dense vector is processed in a vectorized way without loop raking technique to avoid using too many dense vectors. In this case, parallelism (which corresponds to vector length) is limited by the number of non-zeros of each sparse vector. Saving the non-zero index is done by loop raking manner; separate memory space is assigned to each virtual processor, and indices are stored independently. Like the hash-based algorithm, The column indices of the result matrix is not sorted. We also added an option to sort them.

E. Hybrid Algorithm

As described above, The parallelism of SPA is limited by the number of non-zero elements of each sparse vector. In practical applications, the number of non-zero elements per row will vary greatly, as they usually follow the power-law distribution. Therefore, we propose a novel hybrid method. It combines SPA and other methods according to the average numbers of the non-zeros of intermediate sparse vectors. For example, that of *kth* row of **A** in Figure 4 is (3 + 4) / 2 = 3.5.

First, the average number of non-zeros of each row is calculated, and the rows are sorted according to this value. Then, the matrix is divided by the user-defined threshold; the part with a higher average number of the non-zeros is processed by SPA, another part is processed by ESC or hash-based algorithm.

Liu et al. proposed a method that separates the matrix into bins and uses different algorithms for each bin [17]. It is similar to our method in that it uses multiple algorithms. However, our method is unique in that it uses an efficient SPA algorithm for the part where the average number of the non-zeros is high to enlarge vector length for vector architecture.

F. Parallelization of SpGEMM with Vector Engine

SpGEMM can be parallelized by dividing the matrix by row and assigning them to each processor. However, to get better scalability, it is essential to assign tasks evenly to the processors for load balancing. In our implementation, we count the number of non-zero of intermediate sparse vectors and divide the matrix according to the number, which achieved better load balancing.

Figure 9 illustrates this process. For the two operand matrices of SpGEMM - A and B, we applied 1D decomposition [18] by row for the left side matrix, i.e., we leave the right side matrix B untouched, but for the left side matrix A we split it into row blocks. However, to achieve better load balancing, we do not split the rows evenly but based on the number of non-zeros (denoted by a solid dot). In this example, matrix A is split into 4 slices. Although the number of rows is not the same for each slice, getting the non-zeros evenly distributed can help

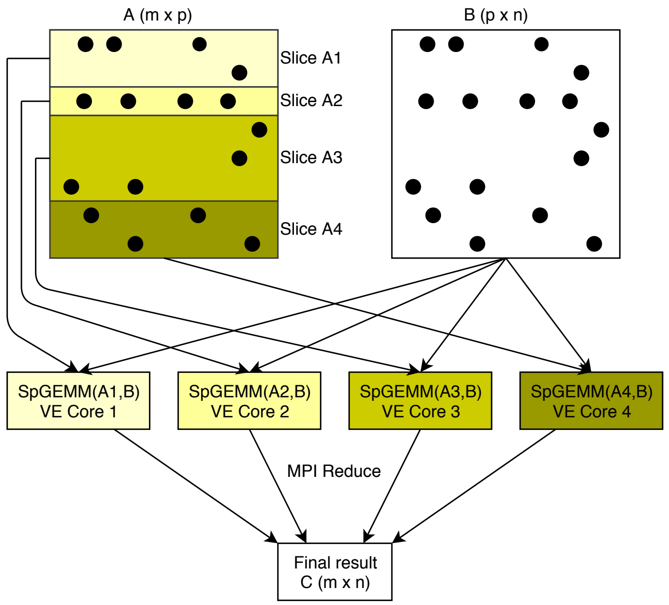


Fig. 9. Parallelizing SpGEMM using MPI with partitioned row blocks

achieve better load balance. We utilize MPI for parallelization, and each slice is assigned to a different MPI process so that the work can be conducted in parallel on the VE cores. For actual datasets, the split number could be in thousands.

G. Evaluation

We evaluated our implementation using sparse matrices from the SuiteSparse Matrix Collection [19] that are commonly used in papers like [13]. Table I shows the matrix used for evaluation.

TABLE I
MATRICES FOR EVALUATION

	nnz/row	max nnz/row	intermed. nnz
Protein	119.3	204	555,322,659
FEM/Accelerator	21.7	81	79,883,385
webbase	3.1	4700	69,524,195
cit-patents	4.4	770	82,152,992
wb-edu	5.8	3841	1,559,579,990
Circuit	5.6	353	8,676,313

The performance is evaluated by $\mathbf{A}^2$. The FLOPS is calculated as (the number of non-zero of intermediate sparse vectors) * 2 / (execution time), which is commonly used as SpGEMM evaluation. The execution time does not contain I/O but contains the counting cost for load balancing.

We measured the performance using 1, 2, 4, and 8 VEs. Since SX-Aurora TSUBASA (A300-4) contains 4 VEs per server. We used two servers for evaluation of 8 VEs, which are connected via InfiniBand. We used MPI also for parallelization within the VE (flat-MPI); in the case of 8 VEs, there are 64 ranks in total.

To compare absolute performance, we also evaluated the performance on Xeon using Intel MKL. We used 1 and 2 sockets of Xeon 6126 Gold. The API we used is `mkl_sparse_spmm` and `mkl_sparse_order`; the API `mkl_sparse_order` is for sorting the index of the result. We utilized shared memory parallelization for MKL that is provided by the library.

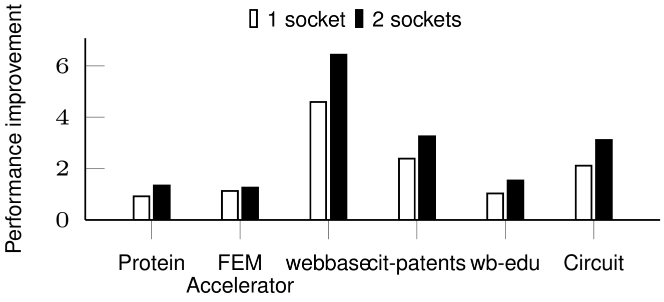


Fig. 10. Performance improvement over CPU. Calculated by $\frac{\text{NEC-Hybrid GFLOPS}}{\text{CPU GFLOPS}}$. NEC-Hybrid has an average performance improvement of 139% over CPU, with a maximum performance improvement of 6.43x.

Figure 11 shows the evaluation result. Hybrid is a hybrid of ESC and SPA method. We used the single-precision floating-point as the value and 32bit integer as the index. All the results include the sorting time of the index.

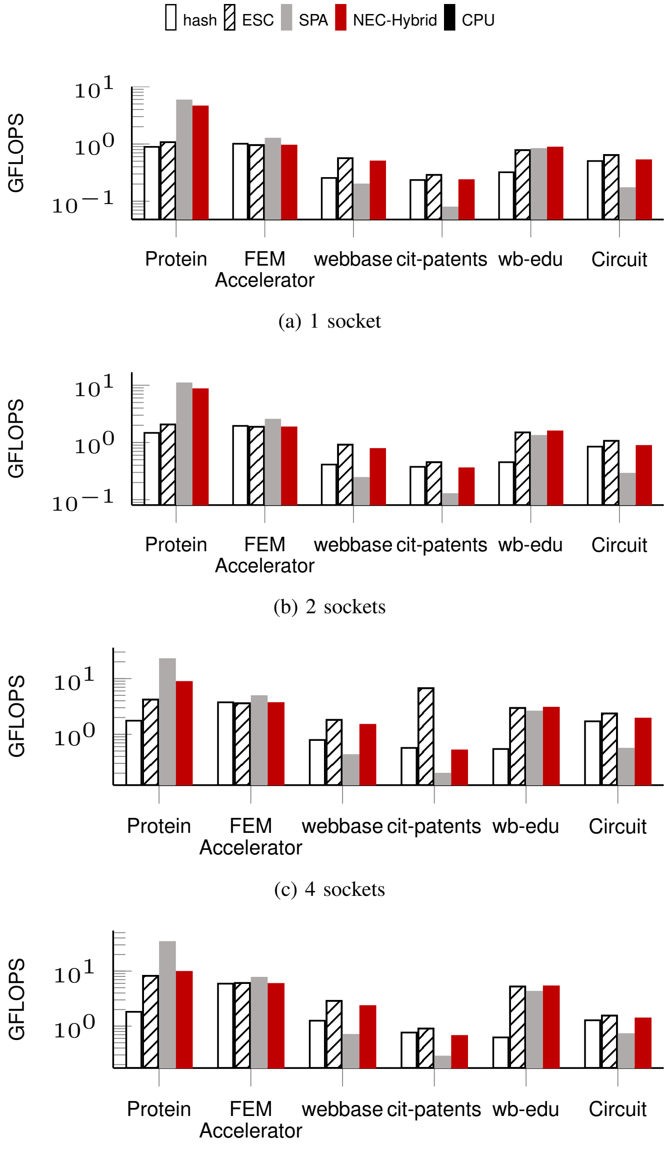
The matrices are grouped into three categories. As for Protein and FEM/Accelerator, the NNZ per row is relatively large. Therefore, SPA performs better than other methods. As for webbase and cit-patents, NNZ per row is small; Therefore, the ESC shows better performance than the SPA. For wb-edu, the network size is relatively large, and the maximum NNZ per row is much larger than average. Therefore, our hybrid method performs the best. Our hybrid method shows stable performance. Although it is not the best performer in all situations, it avoids some of the noticeable shortcomings of ESC and SPA. In all the cases, the hash-based method does not show better performance than other methods. Since it consists of a complex branch in the loop raking technique to implement the hash table; the overhead of loop raking caused poor performance. Our implementation shows better performance than CPU and also shows good scalability.

IV. VECTORIZATION OF GRAPH ALGORITHMS WITH SPGEMM

We have implemented a high-performance linear algebra kernel SpGEMM on the vector engine. In this section, we will introduce two important graph analysis algorithms: triangle counting and butterfly counting. We will use linear algebra operations, mainly SpGEMM, to implement these algorithms so they can take advantage of the NEC vector engine.

A. Triangle Counting

A triangle is a special type of a subgraph that is commonly used for computing important measures in a graph. The triangle counting algorithm consists of the following steps:



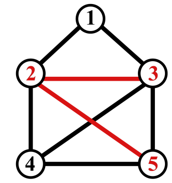
(a) 1 socket

(b) 2 sockets

(c) 4 sockets

(d) 8 sockets

Fig. 11. Evaluation of SpGEMM kernels on VE



(a) Input graph G. There is only one 2-path (3,2,5) between node 3 and node 5 that satisfies $2 < 3$ and $2 < 5$. Path (3,4,5) is not considered since $4 > 3$.

$$\begin{array}{l} Vertex1 \\ Vertex2 \\ Vertex3 \\ Vertex4 \\ Vertex5 \end{array} \begin{pmatrix} 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 & 1 \\ 1 & 1 & 0 & 1 & 1 \\ 0 & 1 & 1 & 0 & 1 \\ 0 & 1 & 1 & 1 & 0 \end{pmatrix}$$

(b) Adjacency matrix $\mathbf{A}$ of G. The lower triangular part $\mathbf{L}$ and the upper triangular part $\mathbf{U}$ are marked in blue and red.

$$\mathbf{B} = \mathbf{LU} = \begin{bmatrix} & & & & \\ 1 & & & & \\ 1 & 1 & & & \\ 0 & 1 & 1 & & \\ 0 & 1 & 1 & 1 & \end{bmatrix} \cdot \begin{bmatrix} & 1 & 1 & 0 & 0 \\ & & 1 & 1 & 1 \\ & & & 1 & 1 \\ & & & & 1 \\ & & & & \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 \\ 0 & 1 & 2 & 1 & \boxed{1} \\ 0 & 0 & 1 & 2 & 2 \\ 0 & 0 & \boxed{1} & 2 & 3 \end{bmatrix}_{5\times5}$$

(c) Line 3 of algorithm 1 : $\mathbf{B} = \mathbf{LU}$ using SpGEMM kernel. $\mathbf{B}_{5,3}$ and $\mathbf{B}_{3,5}$ corresponds to the number of 2-paths between node 5 and 3 satisfying $2 < 3$ and $2 < 5$.

Fig. 12. Adjacency matrix

Algorithm 1: Triangle counting

```
   input : Graph $G = (V, E)$
   output: $number\ of\ triangles\ in\ G$
1  Generate the adjacency matrix $\mathbf{A}$
2  Split $\mathbf{A}$ into a lower triangular $\mathbf{L}$ and an upper triangular $\mathbf{U}$
3  $\mathbf{B} = \mathbf{LU}$ // SpGEMM Kernel
4  $\mathbf{C} = \mathbf{U}.*\mathbf{B}$ // Element wise multiplication
5  return $\sum \mathbf{C}_{i,j}$
```

1) Split $\mathbf{A}$ *into a lower triangular* $\mathbf{L}$ *and an upper triangular* $\mathbf{U}$*:* Given an adjacency matrix $\mathbf{A}$, as shown in figure 12 (a) and (b), the algorithm splits $\mathbf{A}$ into a lower triangular and an upper triangular pieces via $\mathbf{A} = \mathbf{L} + \mathbf{U}$.

2) Calculate $\mathbf{B} = \mathbf{LU}$*:* In graph terms, the multiplication of $\mathbf{L}$ by $\mathbf{U}$ counts all the wedges of (i, j, k) form where j is the smallest numbered vertex. As shown in Figure 12 (c), only one wedge $(5, 2, 3)$ between node 5 and node 3 satisfies $2 < 5$ and $2 < 3$. Correspondingly, $\mathbf{B}_{5,3} = 1$.

3) Calculate $\mathbf{U}.*\mathbf{B}$*, the element-wise multiplication of* $\mathbf{A}$ *and* $\mathbf{B}$*:* The final step is to find if the wedges close by doing element-wise multiplication with the original matrix.

B. Butterfly Counting

Butterfly refers to a loop of length 4 in the bipartite graph. It is the simplest cohesive higher-order structure in a bipartite graph. [3] presented exact algorithms for butterfly counting, as shown in algorithm 2, which can be considered as state-of-the-art. Although this algorithm is fast, it is a loop-based, sequential algorithm. To achieve parallelization and vectorization, we have improved algorithm 2 to algorithm 3 which fully utilize the linear algebra kernels. Our Butterfly counting algorithm consists of the following steps:

1) Create the adjacency matrix $\mathbf{A}$*:* Let $G = (V = (L, R), E)$ be a bipartite graph, where L and R are its left and right parts, respectively. Suppose $L = \{L_1, L_2, \ldots, L_m\}$, $R = \{R_1, R_2, \ldots, R_n\}$. Then we can represent the adjacency matrix of G as $\mathbf{A}_{m\times n}$. $\mathbf{A}_{i,j} = 1$ if and only if L_i and R_j is connected, otherwise, $\mathbf{A}_{i,j} = 0$. In this case, $\mathbf{A}$ is sometimes called the biadjacency matrix.

2) Calculate $\mathbf{AA}^{\intercal}$*:* According to the properties of the adjacency matrix[20], we have: If $\mathbf{B} = \mathbf{AA}^{\intercal}$, then the matrix $B_{i,j}$ gives the number of walks of length 2 from vertex L_i to vertex L_j. As shown in Figure 13, there are three paths of length 2 between L_1 and L_2, which are marked in three colors: red, blue, and green.

3) Set the element on the diagonal of matrix $\mathbf{B}$ *to 0 and add up:* Since each butterfly is made up of two paths of length 2, the two paths share endpoints in L (or R). Thus, if there are $B_{i,j}$ paths between L_i and L_j, then the number of butterflies

Algorithm 2: ExactBFC: Sequential Butterfly Counting

```
input  : Graph $G = (V = (L, R), E)$
output: $Butterfly(G)$
if $\sum_{u \in L}(d_u)^2 < \sum_{v \in R}(d_v)^2$ then
    $A \leftarrow R$
else
    $A \leftarrow L$
for $v \in A$ do
    $C \leftarrow hashmap$
    for $u \in Neighbour(v)$ do
        for $w \in Neighbour(u)$ do
            $C[w] \leftarrow C[w] + 1$
        for $w \in C$ do
            $Butterfly(G) \leftarrow Butterfly(G) + \binom{C[w]}{2}$
return $Butterfly(G)/2$
```

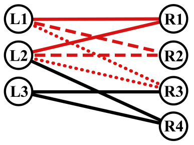
Algorithm 3: Vectorized butterfly counting

```
  input  : Graph $G = (V = (L, R), E)$
  output: $number\ of\ butterflies\ in\ G$
1 Generate the adjacency matrix $\mathbf{A}$
2 $\mathbf{B} \leftarrow \mathbf{A}\mathbf{A}^\intercal$ // SpGEMM kernel
3 Set the element on the diagonal of matrix $\mathbf{B}$ to 0.
4 return $\frac{1}{2}\sum_{i,j}\binom{\mathbf{B}_{i,j}}{2}$
```

with L_i, L_j as the endpoint is $\binom{B_{i,j}}{2}$. Note that we only count 2-paths that differ from the start and endpoints, so we set the elements on the diagonal of matrix B to 0 to exclude those paths where the start and endpoints are the same. For example, the matrix in Figure 13 is transformed into the following form

$$\mathbf{B} = \begin{bmatrix} 3 & 3 & 1 \\ 3 & 4 & 2 \\ 1 & 2 & 2 \end{bmatrix} \rightarrow \begin{bmatrix} 0 & \binom{3}{2} & \binom{1}{2} \\ \binom{3}{2} & 0 & \binom{2}{2} \\ \binom{1}{2} & \binom{2}{2} & 0 \end{bmatrix} = \begin{bmatrix} 0 & 3 & 0 \\ 3 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix}_{3\times 3}$$

Finally, we calculate $\frac{1}{2}\sum_{i,j}\binom{B_{i,j}}{2}$ to get the exact number of “butterflies”.



(a) Input bipartite graph and 2-paths between L_1 and L_2.

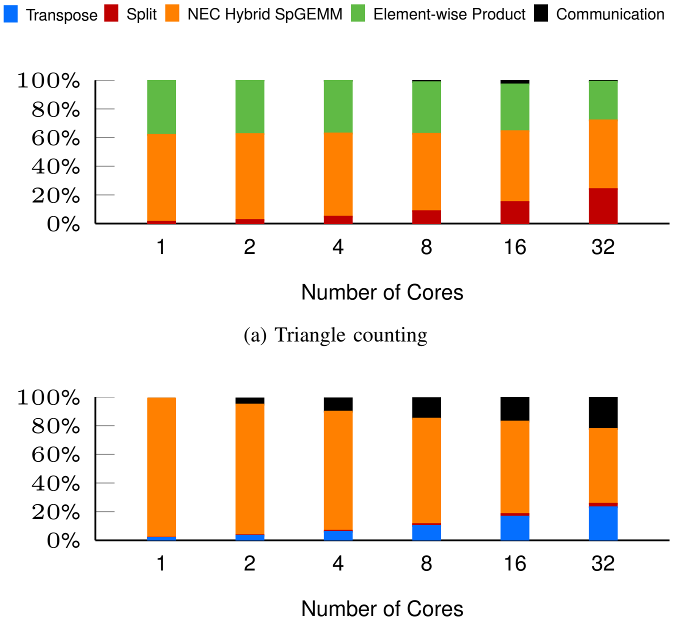
$$\begin{array}{c|cccc} & R_1 & R_2 & R_3 & R_4 \\ \hline L_1 & 1 & 1 & 1 & 0 \\ L_2 & 1 & 1 & 1 & 1 \\ L_3 & 0 & 0 & 1 & 1 \end{array}$$

(b) Biadjacency matrix $\mathbf{A}$ of input graph.

$$\mathbf{B} = \mathbf{A}\mathbf{A}^\intercal = \begin{bmatrix} 1 & 1 & 1 & 0 \\ 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 1 & 0 \\ 1 & 1 & 0 \\ 1 & 1 & 1 \\ 0 & 1 & 1 \end{bmatrix} = \begin{bmatrix} 3 & \boxed{3} & 1 \\ 3 & 4 & 2 \\ 1 & 2 & 2 \end{bmatrix}_{3\times 3}$$

(c) Line 2 of algorithm 3: $\mathbf{B} = \mathbf{A}\mathbf{A}^\intercal$. $\mathbf{B}_{1,2}$ and $\mathbf{B}_{2,1}$ represents the number of 2-paths between L_1 and L_2.

Fig. 13. Adjacency matrix of bipartite graph



(a) Triangle counting

(b) Butterfly counting

Fig. 14. Normalized execution time breakdown.

V. Experiments on Subgraph Counting and Machine Learning Algorithms

A. System Architecture and Implementation

In the experiments, we use: 1) a single node of dual-socket Intel(R) Xeon(R) Gold 6126 (architecture Skylake), 2) a single node of a dual-socket Intel(R) Xeon(R) CPU E5-2670 v3 (architecture Haswell), 3) a single NEC Aurora node with 4 VEs. More details of the testbed hardware can be seen in Table II.

B. Execution Time Breakdown

We have applied the NEC SpGEMM algebra kernels in our triangle counting and butterfly counting implementation. We instrumented the code to show the normalized time spent breakdown on different portions of the code with different datasets. The results are shown in Figure 14 for both triangle counting and butterfly counting. The fact that the two computation kernels (SpGEMM and Element-wise Product) consume the majority of the execution time was the motivation why we have implemented and optimized the algebra kernels on the NEC Aurora platform. We can see from the figure that with the increasing number of cores used, the two kernels are consuming less proportional time which suggests that the parallel execution of the kernels has decreased the overall execution time. On another hand, the proportion of the time spent on non-parallelized code, which includes the preparation of the data and other overhead introduced with handling multi-processes (split, transpose operations, and MPI communication and synchronizations) is increasing.

TABLE II
HARDWARE SPECIFICATIONS AND DATASETS USED

Arch	Model	frequency (GHz)	physical cores	Vector register width (bits)	Vector register	Peak Performance	memory bandwidth (GB/s)	memory capacity (GB)	L1 cache (KB)	L2 cache (MB)	L3 cache (MB)
CPU Skylake	Xeon Gold 6126	2.6	12	512	2	998GF(SP)	119	125	768	12	19.25

CPU Haswell	E5-2670 v3	2.3	12	256	1x24	883GF(SP)
						441GF(DP)
							95	125	768	3	30

Vector Engine	SX-Aurora TSUB-ASA	1.4	8	16384	256	4.91TF(SP)
						2.45TF(DP)
							1200	48	32x8	2	16

TABLE III
GRAPH DATASETS USED IN THE EXPERIMENTS

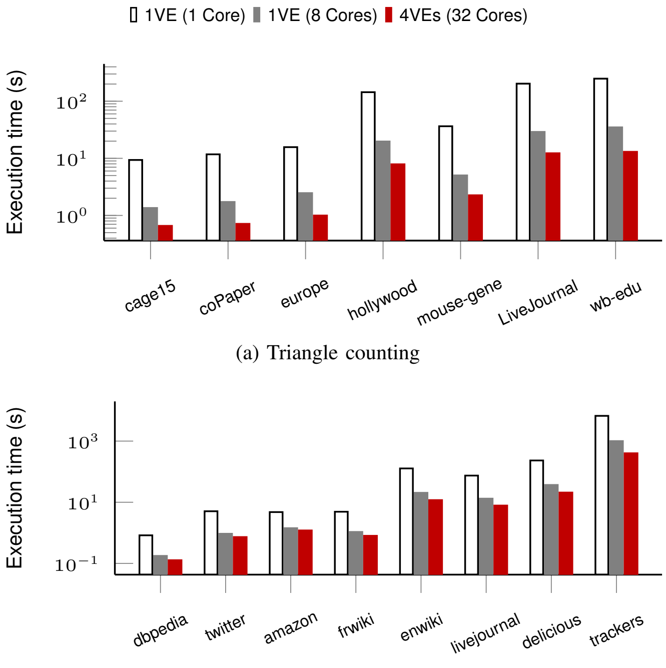
Data	Vertices	Edges	Avg Deg
mouse-gene	29.0M	28.9M	2.00
coPaperDBLP	540k	30.5M	112.83
soc-LiveJournal1	4.8M	85.7M	35.36
wb-edu	9.8M	92.4M	18.78
cage15	5.2M	94.0M	36.49
europe-osm	50.9M	109.1M	4.25
hollywood-2009	1.1M	112.8M	197.83
DBPedia-Location	172K+53K	293K	1.30
Wiki-fr	288K+3.9M	22M	5.25
Twitter	175K+530K	1.8M	2.55
Amazon	2.1M+1.2M	5.7M	1.73
Journal	3.2M+7.4M	112M	10.57
Wiki-en	3.8M+21.4M	122M	4.84
Deli	833K+33.7M	101M	2.92
web-trackers	27.6M+12.7M	140M	3.47

C. Execution Time Evaluation

We used the datasets in Table III to evaluate the performance on a different number of processes utilizing the up to 32 cores of the 4 VEs on the single NEC Aurora node. For butterfly counting, we also compared the execution with the reference BFC_exact [3] running on Haswell CPU. Figure 15 shows the execution time of triangle counting and butterfly counting for the different datasets on single VE (1 core and 8 cores) and 4 VEs (32 cores being utilized). Figure 16 shows the speedup of our algorithm while using one single VE comparing to the BFC_Exact result (normalized to 1). We can see from this figure that for the larger datasets, single VE (with all 8 cores being utilized) achieved better performance with a factor of 3-5 comparing to the BFC_Exact algorithm running on Haswell CPU.

D. Strong Scaling Evaluation

The execution time breakdown shown in Figure 14 has already suggested that the algebra kernels helped achieve good strong scaling. Here we are showing this more clearly in Figure 17 and Figure 18. Figure 17 shows the speedup when utilizing multiple VE cores comparing to one core on the left side; and on the right side, it shows the results from the work in [2]. For the scaling on VE most datasets showing close to linear scalability till the number of processes reached over 10, which is better than the right side chart. Figure 18 shows similar results for Butterfly counting, although it decreases after 8 processes when comparing to the Triangle counting



(a) Triangle counting

(b) Butterfly counting

Fig. 15. Execution Time of Triangle counting and Butterfly counting

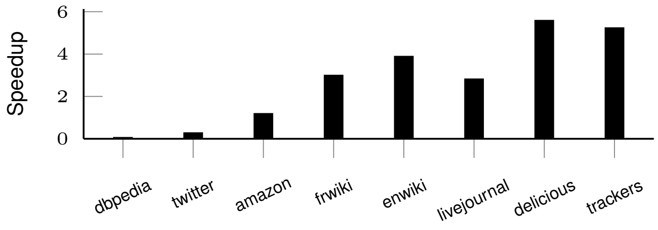
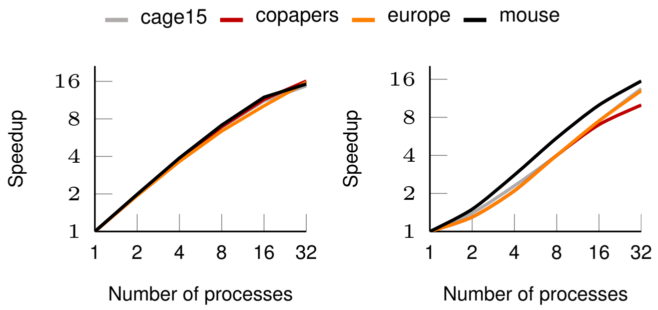


Fig. 16. Butterfly counting speedup (1VE vs BFC_Exact on CPU)

results.



(a) VE (b) CPU

Fig. 17. Triangle counting scalability test on single process

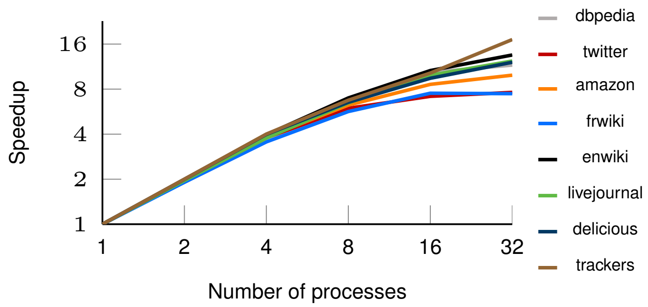


Fig. 18. Butterfly counting scalability with increasing number of processes

E. SpGEMM Kernel Algorithms and Parameter Options

The SpGEMM kernel provided multiple algorithms to choose from, among HASH, SPA, SPA_SORT, ESC, and hybrid of any two of these. For the hybrid algorithm, a user can change the default threshold parameters as part of the function call. The evaluation we have done was using the hybrid algorithm of ESC and HPA/HPA_SORT with default parameters. In this section, we are showing the results of alternative algorithms and parameters. We used triangle counting as an example on three different datasets to show the impact of the algorithms and parameters options. For the algorithms, We tested the ESC alone, SPA_SORT, and the hybrid of the two; For parameters choice of the hybrid mode, a user can specify 3 parameters N1, N2, N, meaning number of columns to process at a time for the first separated matrix, the number of columns to process for the second matrix, and the threshold to separate the two matrices, respectively. The default values for these 3 parameters are 256, 4096, and 64, in that order. We used default values, the increased values by timing 4, and the decreased values by dividing 4 from the default parameters. The results are shown in Figure reffig:spgemm-options. The columns are the SpGEMM breakdown time from the overall execution time of running triangle counting on the testing datasets. The left-most column for each dataset is the hybrid algorithm with default parameters setting. We can see from figure 19 that while this may not be the best-performed choice among all, the hybrid mode with default parameters can produce balanced and good enough results without the need to get more details of the dataset beforehand. As mentioned before in the SpGEMM kernel evaluation section, knowing the characteristics of the dataset, e.g., the distribution of the NNZ per row, could help choose the best algorithm/parameter options, but our experiments were done with hybrid mode and default parameters already show good results. For the SPA and SPA_SORT options, the difference is that with SPA_SORT the column indices of the result matrix are sorted while for SPA, they are not. While sorting added more overhead compared to the non-sorted version, this feature is especially useful if the result matrix is to be chained as input to another algorithm that assumes the column indices are sorted. E.g., in our triangle counting application, we have an element-wise multiplication step after SpGEMM, which utilized another kernel that requires the input matrices to be sorted with column indices. Thus we would need to use the hybrid algorithm of ESC and SPA_SORT. While for butterfly counting, the order of the column indicators does not affect the counting result. Thus we use the hybrid algorithms of ESC and SPA to get better performance.

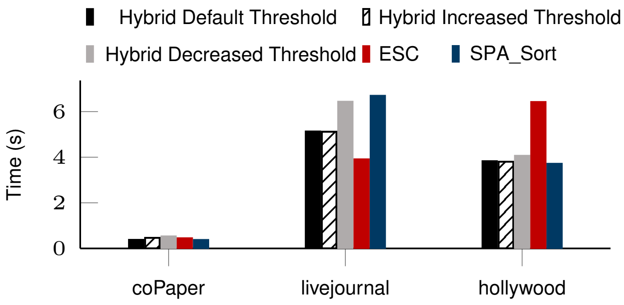


Fig. 19. SpGEMM Time from Triangle Counting Execution Breakdown using Different SpGEMM Algorithms and Parameter Options

VI. Conclusion

In this paper, we have introduced a new vectorized SpGEMM algorithm for butterfly counting in bipartite graphs and also adapted another vectorized triangle counting algorithm, on the NEC Aurora platform. The algorithms are all vectorized, makes it very suitable to run on the vector engine of the NEC Aurora system.

In the SpGEMM kernel evaluation (section III-G), NEC-Hybrid has an average performance improvement of 139% over CPU, with a maximum performance improvement of 6.43x. In the test of the triangle counting algorithm, our implementation shows high scalability compared to related work [2]. In the test of the butterfly counting algorithm, our implementation on large datasets has achieved up to 6 times faster performance even with a single VE, and more than 10 times faster when multiple VEs are used from one node. With the optimized linear algebra kernels such as SpGEMM, both

Graph algorithms demonstrate good performance and scalability. This work can be extended to support other applications and architecture-specific code optimizations in future work.

VII. Acknowledgment

We gratefully acknowledge the support from NEC Corp., NSF CIF21 DIBBS 1443054: Middleware and High Performance Analytics Libraries for Scalable Data Science, Science and NSF EEC 1720625: Network for Computational Nanotechnology (NCN) Engineered nanoBio node, NSF OAC 1835631 CINES: A Scalable Cyberinfrastructure for Sustained Innovation in Network Engineering and Science, and Intel Parallel Computing Center (IPCC) grants. We would like to express our special appreciation to the FutureSystems team.

References

[1] "Nec sx-aurora tsubasa - vector engine." https://www.nec.com/en/global/solutions/hpc/sx/vector_engine.html. Accessed: 2019-09-05.

[2] A. Azad, A. Buluç, and J. Gilbert, "Parallel triangle counting and enumeration using matrix algebra," in *2015 IEEE International Parallel and Distributed Processing Symposium Workshop (IPDPSW)*, pp. 804–811, IEEE, 2015.

[3] S.-V. Sanei-Mehri, A. E. Sariyuce, and S. Tirthapura, "Butterfly counting in bipartite networks," in *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pp. 2150–2159, ACM, 2018.

[4] "Spgemm on nec vector engine." https://github.com/dsc-nec/frovedis_matrix. Accessed: 2019-09-05.

[5] N. Alon, P. Dao, I. Hajirasouliha, F. Hormozdiari, and S. C. Sahinalp, "Biomolecular network motif counting and discovery by color coding," *Bioinformatics*, vol. 24, no. 13, pp. i241–i249, 2008.

[6] F. Battiston, V. Nicosia, M. Chavez, and V. Latora, "Multilayer motif analysis of brain networks," *Chaos: An Interdisciplinary Journal of Nonlinear Science*, vol. 27, no. 4, p. 047404, 2017.

[7] J. Kepner and J. Gilbert, *Graph algorithms in the language of linear algebra*. SIAM, 2011.

[8] L. Chen, J. Li, A. Azad, L. Jiang, M. Marathe, A. Vullikanti, A. Nikolaev, E. Smirnov, R. Israfilov, and J. Qiu, "A graphblas approach for subgraph counting," *arXiv preprint arXiv:1903.04395*, 2019.

[9] I. V. Afanasyev, V. V. Voevodin, V. V. Voevodin, K. Komatsu, and H. Kobayashi, "Analysis of relationship between simd-processing features used in nvidia gpus and nec sx-aurora tsubasa vector processors," in *International Conference on Parallel Computing Technologies*, pp. 125–139, Springer, 2019.

[10] K. Komatsu, S. Momose, Y. Isobe, O. Watanabe, A. Musa, M. Yokokawa, T. Aoyama, M. Sato, and H. Kobayashi, "Performance evaluation of a vector supercomputer sx-aurora tsubasa," in *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 685–696, IEEE, 2018.

[11] N. Bell, S. Dalton, and L. N. Olson, "Exposing fine-grained parallelism in algebraic multigrid methods," *SIAM J. Sci. Comput.*, vol. 34, no. 4, pp. C123–C152, 2012.

[12] J. Demouth, "Sparse matrix-matrix multiplication on the gpu," in *Proceedings of the GPU Technology Conference*, 2012.

[13] Y. Nagasaka, A. Nukada, and S. Matsuoka, "High-performance and memory-saving sparse general matrix-matrix multiplication for nvidia pascal gpu," in *Proceedings of 46th International Conference on Parallel Processing (ICPP)*, 2017.

[14] J. R. Gilbert, C. Moler, and R. Schreiber, "Sparse matrices in MATLAB: Design and implementation," *SIAM J. Matrix Anal. Appl.*, vol. 13, no. 1, pp. 333–356, 1992.

[15] M. Zagha and G. E. Blelloch, "Radix sort for vector multiprocessors," in *Proceedings of the 1991 ACM/IEEE Conference on Supercomputing*, pp. 712–721, ACM, 1991.

[16] S. Dalton, L. Olson, and N. Bell, "Optimizing sparse matrix&mdash;matrix multiplication for the gpu," *ACM Trans. Math. Softw.*, vol. 41, pp. 25:1–25:20, Oct. 2015.

[17] W. Liu and B. Vinter, "A framework for general sparse matrix-matrix multiplication on gpus and heterogeneous processors," *J. Parallel Distrib. Comput.*, vol. 85, pp. 47–61, Nov. 2015.

[18] A. Buluç and J. R. Gilbert, "Highly parallel sparse matrix-matrix multiplication," *arXiv preprint arXiv:1006.2183*, 2010.

[19] T. A. Davis and Y. Hu, "The university of florida sparse matrix collection," *ACM Trans. Math. Softw.*, vol. 38, pp. 1:1–1:25, Dec. 2011.

[20] R. P. Stanley, "Algebraic combinatorics," *Springer*, vol. 20, p. 22, 2013.