Efficient Dense And Sparse Matrix Multiplication On GP-SIMD

Amir Morad
Dept. of Electrical Engineering,
Technion, Haifa 32000, Israel
amirm@tx.technion.ac.il

Leonid Yavits
Dept. of Electrical Engineering,
Technion, Haifa 32000, Israel
yavits@tx.technion.ac.il

Ran Ginosar
Dept. of Electrical Engineering,
Technion, Haifa 32000, Israel
ran@ee.technion.ac.il

***Abstract*—We present efficient Dense and Sparse Matrix Multiplication on GP-SIMD, a hybrid general purpose SIMD computer architecture that eliminates synchronization by in-memory computing, combining data storage and massively parallel processing. Cycle-accurate simulation of on a large set of matrices shows enhanced power efficiency relative to conventional architectures.**

Keywords— Sparse Linear Algebra, GP-SIMD, Associative Processor, Memory Intensive Computing, In-Memory Computing.

I. Introduction

Large scale machine learning tasks require extensive dense and sparse matrix multiplications. We explore the efficiency of dense matrix multiplication (DMM) and sparse matrix multiplication (SpMM) on the GP-SIMD architecture. The GP-SIMD is a hybrid general purpose SIMD computer architecture that combines data storage and massively parallel processing in order to eliminate the need to synchronize data between the general purpose processor and its accelerators [23]. Figure 1 shows the architecture of the GP-SIMD, comprising a sequential CPU, a shared memory array, instruction and data caches, a SIMD coprocessor, and a SIMD sequencer. The SIMD coprocessor contains a large number of fine-grain processing units, each comprising a single bit ALU, single bit function generator and a 4-bit register file. The GP-SIMD processor is thus a large memory with massively parallel processing capability. No data synchronization between the sequential and parallel segments is required since both the general purpose sequential processor and the SIMD co-processor access the very same memory array. Thus, no time and power penalties are incurred for synchronization.

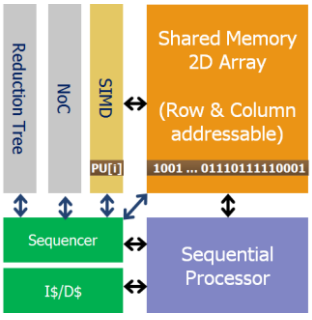

Figure 1. GP-SIMD architecture

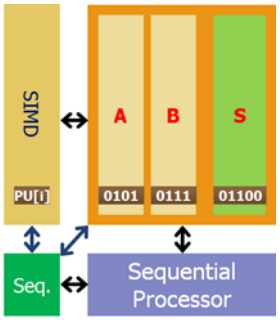

Figure 2. Memory array containing three operands

The GP-SIMD architecture has been discussed in [2]. The GP-SIMD delivers a number of advantages over conventional SIMD architectures:

- Data processing and data storage are unified. There is no need for data transfer between sequential memory and SIMD PUs;
- GP-SIMD allows concurrent operation of the sequential processor and SIMD co-processors on the shared memory, allowing the sequential processor to offload a task to the SIMD while continuing to process some other sequential functions.
- The number of GP-SIMD fine grain processing units matches the number of memory rows, striving to match the entire dataset.
- The GP-SIMD architecture enables the sequential processor to associatively address the memory array [2]. It may thus allow reduction of software complexity for certain sequential algorithms.
- GP-SIMD power dissipation is distributed uniformly over the entire processing array rather than being concentrated around a smaller number of large, power hungry processing cores. Thus, there are fewer hotspots leading to further reduction of temperature dependent leakage power [18].

In this paper, we present GP-SIMD algorithms for dense and sparse matrix multiplication (DMM, SpMM).

The rest of this paper is organized as follows. Section II discusses related work. Section III presents GP-SIMD architecture. Section IV presents GP-SIMD algorithms for dense and sparse matrix multiplication. Section V details the evaluation methodology and presents cycle-accuarte simulation results. Section VI concludes this paper.

II. Related Work

Vector machines and SIMD architectures are a class of parallel computers with multiple processing units performing the same operation on multiple data points simultaneously [1][26][3]. Such machines exploit data level parallelism, and are thus well suited for machine learning over Big Data [7]. The concept of mixing memory and logic has been around since the 1960s [16]. Similar to DAP, STARAN, CM-2, GAPP, and Associative Processor (AP) [21] computer architectures (comprehensive reference is provided in [2]), GP-SIMD belongs to a Processing-In-Memory (PiM) class of architectures that use a large number of Processing Units (PUs) positioned in proximity to memory arrays to

implement a massively parallel SIMD computer. To differentiate between GP-SIMD and other works, and since keywords like PiM and SIMD are often used with different meanings in mind, [2] studies the GP-SIMD, cites an exhaustive list of studies, and presents a taxonomy categorizing previous works in the processing-in-memory (PiM) and SIMD fields.

Previous studies on SpMM target sparse matrix by dense vector multiplication (SpMV) or sparse matrix by dense matrix multiplication (SpMM). For simplicity, in this section we apply the term SpMM to both SpMM and SpMV. A comprehensive review of sparse matrix multiplication techniques is provided by R. Vuduc [35]. Considering hardware aspects rather than software implementation [22], previous work can be divided into three categories (TABLE 1).

TABLE 1: SPMM RELATED WORK SUMMARY

Category	Existing Work
General Purpose Computers	Off-the-shelf [4][9][37][41] Advanced multicore [38] Manycore supercomputer [6]
GPU	[11][14][25][28][29][36]
Dedicated Hardware Solutions	FPGA [17][24] Manycore Processor [27] Distributed Array Processor [13] Systolic Processor [32] Coherent Processor [5] TCAM / PIM [12] Heterogeneous platform[30][31] 3D LiM [33]

The key contribution of the present work is the efficient implementation of dense and sparse matrix multiplication on a GP-SIMD processor, verified by extensive cycle-accurate GP-SIMD simulation using a large collection of sparse matrices [39].

III. THE GP-SIMD PROCESSOR

In this Section we describe GP-SIMD, focusing on relevant aspects of the architecture, arithmetic, logic and associative processing capabilities. Further details are given in [2].

A. Top Level Architecture

The GP-SIMD is a hybrid general purpose and SIMD computer architecture that resolves the issue of synchronization by in-memory computing, through combining data storage and massively parallel processing. References to on-chip memory 'row' (*r*) and 'column' (*c*) are physical. Each row may contain many words of software programmable width (*w*) (if *w* is constant for all words, the number of words is thus r· c/w). The number of rows typically matches the dataset elements, *N*.

- Sequential processor accesses either one word at a time, or multiple words. Typically, such a transaction accesses one physical row at a time.
- The SIMD reads/writes a bit-slice (having *r* bits) comprising the same bit-number from all words in some partition of the memory. Physically, it may access multiple bits in a physical row and all rows per access, namely accesses multiple columns of the physical array.

Figure 1 details the architecture of a GP-SIMD processor. The sequential processor schedules and operates the SIMD processor via the sequencer. In a sense, the sequential processor is the Master controlling a slave SIMD co-processor. The SIMD coprocessor contains a number of fine-grain processing units (PUs), as depicted in Figure 3, each containing a single bit Full Adder (*FA*), single bit Function Generator (*FG*) and a 4-bit register file, *RA*, *RB*, *RC* and *RD*. A single PU is allocated per row of the shared memory array, and physically resides close to that row. The PUs are interconnected using an interconnection network. The set of all *r* registers of the same name constitute a *register slice*. Note that the length of the memory row (e.g., 256 bits) may be longer than the word length of the sequential processor (e.g., 32 bits), so that each memory row may contain several words.

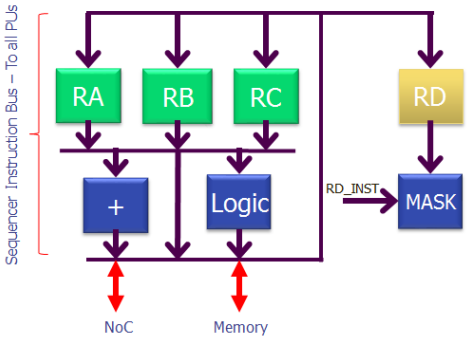

Figure 3. GP-SIMD Processing unit

When the SIMD reads data from the shared memory, the contents of a bit slice of the memory are transferred into the register slice (*RAs*, *RBs* or *RCs*). Upon writing to the shared memory, the contents of one of the register slices are transferred into the GP-SIMD memory array. A conditional register (*RD*) is utilized to enable special/masked microinstructions as depicted in TABLE 2.

The *RD_INST* is a part of the SIMD co-processor instruction bus driven by the sequencer, and each RD_INST value specifies an operation (depicted as a bus going from the sequencer to the SIMD co-processor in Figure 3). While the first four operations are self-explanatory, the last two operations allow the sequential processor to perform associative commands on the memory array, as detailed in section C below.

TABLE 2: CONDITIONAL/MASKED PU MICROINSTRUCTIONS

RD_INST	***RD Value***	***Operation***
00	0	Memory access (read/write) by the sequential processor or the SIMD co-processor.
00	1	Conditional read command, according to a RD: (*RD*>0 ? *RB*=0 : *RB*=Memory output)
01	0	SIMD co-processor memory-write of RA
01	1	Conditional write command, according to *RD*: (*RD*>0 ? Memory input=*RB* : Memory input=*RA*).
10	0	Conditional (masked) read / write: Disable row for memory access by sequential processor. Used during associative access
10	1	Conditional (masked) read / write: Enable row for memory access by sequential processor. Used during associative access

B. Arithmetic / Logic Operations

GP-SIMD can implement a wide range of arithmetic and logic

processing tasks. Consider a workload using two datasets, *A* and *B*, each containing *N* elements, where each element is m bits wide. These vectors are mapped into the GP-SIMD memory array such that two m bit adjacent column-groups hold vectors *A* and *B*. Assume that we need to add the two vectors and place the results into *m+1* bit column-group *S*, as illustrated in Figure 2 (where *m=4*). The addition is performed in m single-bit addition steps:

$$c[*] \mid s[*]_i = a[*]_i + b[*]_i + c[*] \quad \forall\, i = 0, \ldots, m-1 \tag{1}$$

where i is the bit index and '$*$' is the vector index (corresponding to a PU and memory row). Since addition is carried out simultaneously for all vector elements, fixed point m bit addition consumes $3m \in O(m)$ cycles, independent of the size of the vectors *N*.

Using the same logic, subtracting or performing logic AND, OR, XOR via the function generator on the two operand sets entails $O(m)$ cycles as well. *Compare immediate* operation between set A and a fixed word sourced from the sequential processor requires only $O(m)$ cycles since the second operand is sourced from the sequencer, not from the memory array.

Fixed point multiplication and division in GP-SIMD are also implemented bit-serially but word-parallel, consisting of a series of add-shift and subtract-shift vector operations. Shift is implemented by appropriate column addressing and therefore requires no extra cycles. Thus, fixed point $m \times m\ bit$ vector multiplication requires $3m * m \in O(m^2)$ cycles, regardless of the vector size, *N*. Floating-point arithmetic for GP-SIMD is somewhat more complex to implement. Different exponents require shifting mantissas by different lengths, resulting in a sequence of bit-serial vector operations. IEEE single precision floating-point vector multiplication takes close to 2500 cycles, regardless of the length of the dataset, *N*.

C. Associative Operations

GP-SIMD, besides being a massively parallel SIMD accelerator, can implement classical CAM operations such as associative search, sorting and ordering. The CAM allows comparing all data words to a key, tagging the matching words, and possibly reading some or all tagged words one by one. Consider a large vector, where each element is *m* bits wide, illustrated by column *A* in Figure 2. The Sequential processor wishes to find all elements in vector *A* matching a certain *Key* of *m* bits, and reset the matched values of *A* (that is, *A*[*i*|*A*[*i*]==*Key*]=0). The sequential CPU issues a *compare immediate* of *Key* on column *A*, storing the single bit-slice compare results output in register *RD*. At this point, register *RD* has logic *one* in all rows where *A* matches the *Key* and *zero* elsewhere. Next, a masked write is performed by the sequential processor *only* to flagged rows of the memory array. To that end, the output of *RD* enables writing of each memory row (*RD_INST* bus is set to *'10'*), and the sequential processor writes '0' to the *A* column of the memory array. Only the matching rows are enabled for writing, and the A values of only these rows are reset. Elsewhere, in non-matching rows, the A values are left unaffected.

Content-addressable access is achieved as follows. Assume that the memory array contains a vector of unique indices (*A*), adjacent to a vector of data (*B*). Comparing vector A with a key, followed by setting the *RD_INST* Bus to '10' while issuing *read* to the memory array, allows the sequential processor to fetch a single value of vector *B*, corresponding to the row in which vector *A* matched the *Key* (that is, Output=*B*[*i*|*A*[*i*]==*Key*]). When multiple rows match the key, the values must be read one by one. Further, a portion of GP-SIMD memory grid may be programed to mimic bit-serial TCAM [2].

D. Interconnection Network

Since GP-SIMD processing operation is mainly bitwise, the interconnection can be a relatively simple circuit-switched network. An example of an efficient network is a logarithmic ±k nearest neighbor, forming *N*-bit shift register. Assuming each PU has a single bit direct access to its $\pm Y$ neighbors, where $Y \in \{1,2,4,\ldots,log_2 N\}$, transferring in parallel an entire vector of *N* rows (a slice of the shared array) by *H* rows up/down entails a maximum of $O(m + m\log_2(H))$ cycles, independent of the vector size, *N*. Note that if $H \subset Y$, the transfer time entails $O(2m) \in O(m)$ cycles.

E. Reduction Tree

A common reduction operation sums up a large array of values. Consider a vector *A* of *N* fixed point *m*-bit elements, as illustrated in Figure 2. Further, consider a hardware reduction tree implemented using a pipelined binary adder tree. The first level of the tree sums two single bits from two adjacent PUs. Following $\log_2 N$ levels, the scalar sum of the entire array becomes available. The fixed precision summation of vector *A* entails reading a single column slice of vector *A*, LSB first, and summing this column via the reduction tree. The addition is carried out simultaneously for all vector elements, column-slice at a time until all *m* columns have been processed. In a similar manner, the reduction tree having floating-point adders, sums up a large array of floating-point values. Further, rather than waiting for the reduction tree operation to complete, the tree can be operated in pipeline fashion. In such a case it takes *O(m)* cycles to store a set of *m*-bit values into the tree, and from that point the GP-SIMD can start working on the next set.

F. GP-SIMD Performance Summary

Consider a data set having two *m*-bit *N*-element vectors *A* and *B*. TABLE 3 summarizes the arithmetic/logic performance of the GP-SIMD processor, as analyzed in the previous sections.

TABLE 3: ARITHMETIC/LOGIC PERFORMANCE

Command	*Performance (cycles)*
Read/Write(address)	$O(1)$
Cmp(A, Immediate)	$O(m)$
Conditional Write (address)	$O(1)$
FP Add/Sub/Mult(A, B)	2500
HW Reduction tree	O(m)

Write and read delays in GP-SIMD are identical to those of a conventional SRAM. Since both SRAM write and read delays in contemporary technologies are well under 300ps [19], the GP-SIMD can be operated at or above 3GHz. The matrix multiplication algorithms presented in this paper utilize the

sequential processor. The sequential processor is a baseline microprocessor, similar to that of [12], with a simple 4 stage pipeline and a typical instruction set including arithmetic/logic, memory access and control instructions. A single-precision floating-point addition and multiplication in the CPU is assumed to be performed in a single pipeline stage.

As described in [2], 8 million PU array (256 bits per memory row) would occupy close to 200mm^2 in 22nm technology.

IV. Matrix Multiplication on GP-SIMD

In this section we describe the implementation of dense and sparse matrix multiplication algorithms on GP-SIMD. We assume two input matrices, the $N\times M$ multiplier matrix A and $M\times L$ multiplicand matrix B, are stored in the GP-SIMD memory in the Coordinate List (COO) format, where nonzero elements are stored along with their row and column indices. The output $C = A \times B$ is of dimension $N\times L$. Figure 4 illustrates the COO storage format of two 2×2 dense matrices; the PUs of the GP-SIMD are shown to the left. Note that the GP-SIMD storage may have large number of memory columns (typically, 256). These memory columns may be grouped together into fields, each representing a variable, flag, etc. Note also that, for efficient implementation, the multiplicand matrix B is stored in a transposed form. Further, each column of the multiplicand matrix B is stored in a number of memory rows round up to the nearest power of two (2 memory rows in the figure).

PU	Val	Row	Col	Unallocated
1	$A_{1,1}$	1	1	
2	$A_{1,2}$	1	2	
3	$A_{2,1}$	2	1	
4	$A_{2,2}$	2	2	
5	$B_{1,1}$	1	1	
6	$B_{2,1}$	1	2	
7	$B_{1,2}$	2	1	
8	$B_{2,2}$	2	2	

Figure 4. GP-SIMD COO storage format

In this example, the value field (*Val*) is allocated 32 bit-slices (memory columns 0:31) to accommodate a single precision floating-point number. The *Row* and *Col* fields are allocated *1* bit-slice each (memory columns 32:33).

A. Dense Matrix Multiplication

Dense matrix multiplication is explained by means of the example of Figure 4. Element $A_{1,1}$ is to be multiplied by all elements of the first row of matrix B, to form two singleton products. In the same manner, $A_{1,2}$ is to be multiplied by all elements of the second row of the multiplicand matrix B, to form two singleton products, and so on. We thus have three main procedures:

- *Broadcast*: match the row-elements of B with the appropriate column-elements of A (namely, match $A_{*,i}$ with $B_{i,*}$).
- *Multiply*: multiply pairs of matched elements
- *Reduce*: add the singleton products together

The *broadcast* procedure is performed as a sequence of associative operations on the sequential processor, while *Multiply* and *Reduce* are performed as parallel operations on the SIMD processor, as follows. In GP-SIMD, the sequential processor can associatively match rows of memory array with a Key of k bits (in $O(k)$ cycles, cf. Sect. C). Once executed, all rows matching the Key are tagged (*RD* of the tagged rows contain '1'). The sequential processor can then execute associative read/write from/to the tagged rows (in O(1) cycles), achieving *Broadcast*.

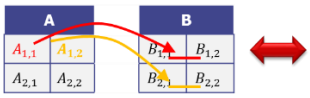

Figure 5. Broadcast, 2x2

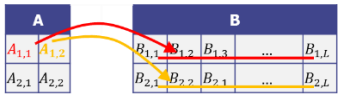

Figure 6. Broadcast, 2xL

Figure 5 and Figure 6 illustrate two general cases of broadcast, while Figure 7 and Figure 8 demonstrate *Broadcast* and *Multiply* on the example of Figure 4. Starting from the first row of A, the sequential processor fetches $A_{1,1}$ and $A_{1,2}$, broadcasting them one by one to the appropriate rows of B (into temporary field T_1), as follows:

1. Element $A_{1,1}$ is read by the sequential processor ($O(1)$ cycles),
2. The *Row* index fields of B are compared with the column index of $A_{1,1}$ ('1') and matching memory rows are tagged ($O(1)$ cycles),
3. $A_{1,1}$ is written into field T_1 of the tagged rows ($O(1)$ cycles).

Thus, broadcasting a single element of A into matching pairs of B takes three cycles. In the general case, for $N\times M$ multiplier matrix, *Broadcast* entails $O(N\ logM)$ cycles. Once all elements of the first row of matrix A are broadcast into the appropriate places (as depicted in Figure 7), a single floating-point multiplication is performed, and $M\times N$ singleton products are stored (as illustrated in Figure 8).

PU	Val	Row	Col	T1	Unallocated
1	$A_{1,1}$	1	1		
2	$A_{1,2}$	1	2		
3	$A_{2,1}$	2	1		
4	$A_{2,2}$	2	2		
5	$B_{1,1}$	1	1	$A_{1,1}$	
6	$B_{2,1}$	2	1	$A_{1,2}$	
7	$B_{1,2}$	1	2	$A_{1,1}$	
8	$B_{2,2}$	2	2	$A_{1,2}$	

Figure 7. *Broadcast*

PU	Val	Row	Col	T1	T2	Unallocated
1	$A_{1,1}$	1	1			
2	$A_{1,2}$	1	2			
3	$A_{2,1}$	2	1			
4	$A_{2,2}$	2	2			
5	$B_{1,1}$	1	1	$A_{1,1}$	$A_{1,1}\times B_{1,1}$	
6	$B_{2,1}$	2	1	$A_{1,2}$	$A_{1,2}\times B_{2,1}$	
7	$B_{1,2}$	1	2	$A_{1,1}$	$A_{1,1}\times B_{1,2}$	
8	$B_{2,2}$	2	2	$A_{1,2}$	$A_{1,2}\times B_{2,2}$	

Figure 8. *Multiply*

Since the multiplicand matrix B is transposed, the sum of the singletons residing in rows 5 and 6 yields $C_{1,1}$, the first element of the output matrix, and the sum of singletons residing in rows 7, 8 yields $C_{1,2}$. The *Reduction* process is effectively implemented by the hardware reduction tree. For single precision floating-point vector summation, the hardware reduction tree takes about *32* cycles (bit-serially feeding the 32 bits of the numbers into the tree, which subsequently reduces these number off-line in a pipeline fashion) and following the transfer of these singletons into the tree, processing of the next matrix row (*Broadcast* and *Multiply*) is initiated. The outputs of reduction tree are fed directly to the sequential processor, that stores the computed elements of C into the designated memory addresses.

Although we detailed a single precision floating-point *Broadcast*, *Multiply*, and *Reduce* procedures, any operand wordlength may be considered. Further, although the example shows matrices A and B stored in separate PUs and memory rows,

typically they would be stored side by side to enable handling larger matrices.

The hardware reduction tree of floating-point operands entails considerable area penalty. For effective implementation, note that *Reduce* follows a very long *Multiply* procedure of 2500 cycles. Pipelined *Multiply-Reduce* is thus preferable, in which a single floating-point accumulator can serially add up to 2500 singleton products, before piping its results to the next level of the tree. With that, the area and power of the reduction tree is kept at bay.

The *Broadcast*, *Multiply*, and *Reduce* procedures are now repeated for the next row of the multiplier matrix *A*, until all rows have been processed. The pseudo code of the algorithm is depicted in Figure 9. It includes two nested loops. The external loop goes over the rows of matrix *A*. The internal loop performs *Broadcast* of all row elements of *A*.

```
Dimensions:
        A: Multiplier, N×M single precision floating-point matrix
        B: Multiplicand, M×L single precision floating-point matrix
Data structure:
        COO: each element accompanied by its row and col indices
Memory column fields:
        A and B: 32b single precision
        ROW_INDEX: COO row index, max(log(N),log(L)) bits
        COL_INDEX: COO column index, log(M) bits
        T: 32b single precision temporary. Used to store singletons

GP-SIMD-DMM(A, B)
        Clear column-field T ☞ ~32 cycles
        For all A rows i=1:N
            Broadcast: For all A columns j=1:M
                Read A[i,j] by sequential CPU ☞ ~1 cycle
                Tag all rows in B that have ROW_INDEX == j ☞ O(log(M))
                Conditional write A[i,j] to T in tagged rows ☞ ~1 cycle
            Multiply: Multiply B by T, store the results into T
            Reduce: Reduction tree on column-field T
            Store outputs in designated space
end
```

Figure 9. GP-SIMD DMM pseudo code

The complexity of the algorithm is as follows:

$$N\left[M\left(1+\log_2 M+1\right)+C_{Mult}+C_{Reduction}\right] \qquad (2)$$

where $C_{Mult}=2500$ cycles and $C_{Reduction}=32$ cycles.

Note that for large *N*, *M*, the complexity of GP-SIMD DMM approaches $O(NM\log_2 M)$ and $O(N^2\log_2 N)$ for square matrices. For example, the estimated complexity of multiplying a *10,000×10,000* matrix *A* exceeds 1B cycles.

The GP-SIMD DMM algorithm may also be used for multiplying dense matrix by dense vector, but it takes the same number of cycles regardless of the number of columns (*L*) of *B* (as illustrated in Figure 6). Thus, the efficiency of the GP-SIMD DMM algorithm grows with the number of columns of the multiplicand matrix *B* (efficiency is defined as the number of actually performed arithmetic operations divided by the number of PUs times total cycle count, namely the maximum number of operations possible during the execution time).

B. *Sparse Matrix Multiplication*

In this section we describe the implementation of multiplication of sparse matrix *A* by dense matrix *B*. The algorithm is similar to dense matrix multiplication (Sect. A) but instead of processing all *N*×*M* elements of *A*, we only process the nonzero elements:

- *Broadcast* is executed only for nonzero elements of A,
- *Multiply* and *Reduce* are performed only for nonzero rows of A.

The pseudo code of the GP-SIMD SpMM algorithm is depicted in Figure 10.

```
Dimensions:
        A: Multiplier, N×M single precision floating-point sparse matrix
        B: Multiplicand, M×L single precision floating-point dense matrix
Data structure:
        COO: each element accompanied by its row and col indices
Memory column fields:
        A and B: 32b single precision
        ROW_INDEX: COO row index, max(log(N),log(L)) bits
        COL_INDEX: COO column index, log(M) bits
        T: 32b single precision temporary. Used to store singletons

GP-SIMD-SpMM(A, B)
        Clear column-field T ☞ ~32 cycles
        Last_row=1;
        While (not end of matrix A)
            Broadcast: Read the next value of A ☞ ~1 cycle
                    Store the element's row index into current_row_index
            If(Last_row<current_row_index)
                Multiply: Multiply B by T, store the results into T
                Reduce: Reduction tree on column-field T
                Store outputs in designated space
                For i=1: current_row_index- Last_row-1  //skip empty rows
                    Store zero outputs in designated space
                Last_row=current_row_index;
                Clear column-field T ☞ ~32 cycles
            Broadcast: Tag all rows in B that have ROW_INDEX == j ☞ O(log(M))
            Broadcast: Conditional write A[i,j] to tagged column T ☞ ~1 cycle
end
```

Figure 10. GP-SIMD SpMM pseudo code

The complexity of GP-SIMD SpMM is approximately

$$N_{NNZ}\left[M_{NNZ}\left(1+\log_2 M+1\right)+C_{Mult}+C_{Reduction}\right] \qquad (3)$$

where N_{NNZ} is the number of nonzero rows of matrix *A* and M_{NNZ} is the average number of nonzero elements per row (equal to number of nonzero elements of the multiplier matrix *A*, denoted by A_{NNZ}, divided by N_{NNZ}). Thus, for large N_{NNZ}, M_{NNZ}, the complexity of the algorithm approaches $O(A_{NNZ}\log_2 M)$. For example, given sparse *10,000×10,000* matrix *A* with only 1,000 nonzero elements (10^{-5} density) and large M_{NNZ}, the complexity of GP-SIMD SpMM is about 16,000 cycles, 10^{-5} times shorter than GP-SIMD DMM. Similar to the GP-SIMD DMM algorithm, GP-SIMD SpMM efficiency grows with the number of columns in the multiplicand matrix *B*.

Note that GP-SIMD SpMM is similar to GP-SIMD DMM, except for several cycles spent by the sequential processor handling matrix indices. It is thus preferable to employ the GP-SIMD SpMM algorithm even for dense matrices (with a minor modification for skipping zero elements). Hence, in the following section, only the results of GP-SIMD SpMM cycle accurate simulations are presented.

V. Cycle Accurate Simulations

The GP-SIMD simulator [2] is used to quantify performance and power of the GP-SIMD SpMM. The experimental setup, matrix statistics and simulation results are described in this section.

A. Experimental Setup

To simulate sparse matrix multiplication, we use 1,000 floating-point square matrices with the number of nonzero elements spanning from one hundred thousand to eight million, randomly selected from the collection of sparse matrices from the University of Florida [39]. Figure 11 presents the selected test-set.

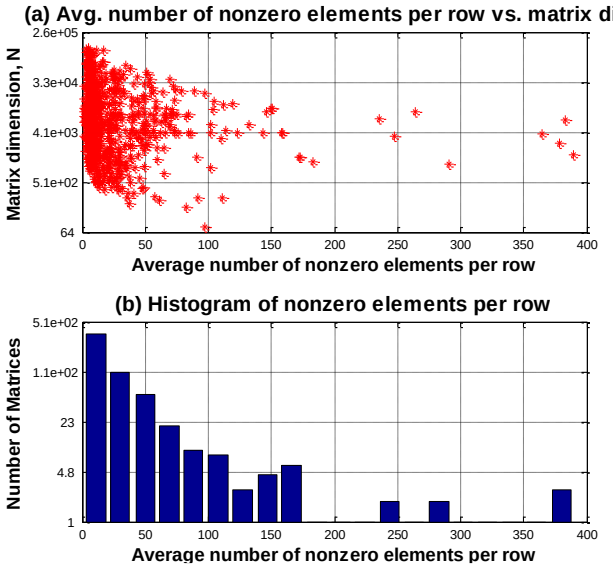

Figure 11. University of Florida Sparse Matrix Collection, (a) Matrix dimension vs. average number of nonzero elements per row, (b) Histogram of the average number of nonzero elements per row

We simulate the dense and sparse matrix multiplication using the GP-SIMD simulator [2]. Each pair of matrix elements and the resulting singleton product are processed by a single GP-SIMD processing unit. Simulations are performed on Intel® XEON™ C5549 processor with 32GB RAM, and simulation times for the 100K—8M nonzero element matrices range between few minutes and few tens of hours. The simulator is cycle based, keeping record of the state of each register of each PU and of the memory row assigned to it. Each command (for example, floating-point multiply) is broken down to a series of fine-grain single bit PU operations. In a similar manner to SimpleScalar [8], the simulator also keeps track of the registers, buses and memory cells that switch during execution. With the switching activity and area power models of each baseline operation detailed in [2], the simulator tracks the total energy consumed during workload execution.

As detailed in earlier sections, GP-SIMD performance depends on the data wordlength rather than on data set size. If matrix elements are presented in a floating-point format, the wordlength is 32 bit (IEEE754 single precision). Data set size in SpMM typically equals the number of nonzero elements in the sparse matrix.

B. SpMM Cycle Accurate Simulations

In this section we compare our cycle accurate simulations results with those of nVidea K20 [10], Intel XEON PHI [10] and Associative Processor (AP) [22]. Although the efficiency of the presented GP-SIMD SpMM algorithm grows with the number of columns L of matrix B, for fair comparison we will limit our analysis to multiplicand (B) matrices having 16 columns, as used in [10]. Further, we assume a GP-SIMD having 8 million PUs having an area of approximately 200mm^2 in standard 22nm technology [2]. We consider test-case sparse multiplier matrices A having 1M columns or less and up to 8M nonzero elements, and dense multiplicand matrices B with 16 columns. Figure 12 presents the sparse by dense matrix multiplication execution times employing the GP-SIMD SpMM algorithm of Section IV for these matrices.

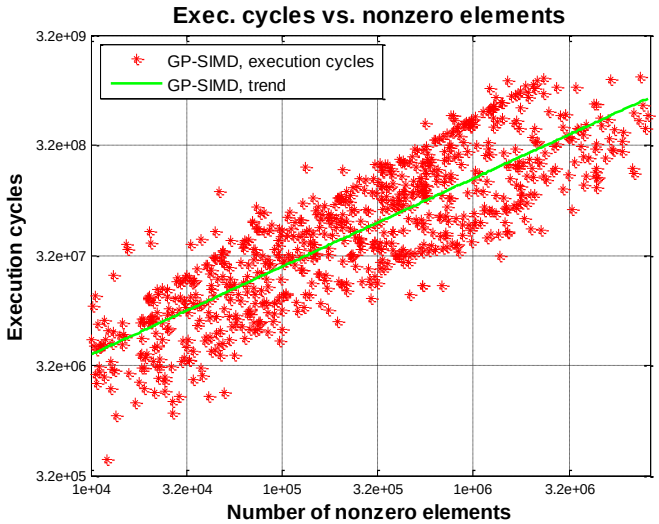

Figure 12. Execution cycles vs. number of nonzero elements

Note the spread in execution times (per each number of nonzero elements) caused by the sensitivity of the GP-SIMD SpMM algorithm to the average number of nonzero elements per row. For two matrices with a similar number of nonzero elements, the difference of two orders of magnitude in the average number of nonzero elements per row results in a similar difference in the execution time. This sensitivity of performance to the average number of nonzero elements per row is shared, although possibly to a lesser extent, by conventional SpMV and SpMM implementations (on GPU or multicore) [15][40]. Since the average of nonzero elements per row in our test-set is somewhat

capped (see Figure 11(b)), as the number of nonzero elements grows, so does the average number of rows and columns of matrices. Execution time of *Broadcast* depends on the number of columns of matrix A (cf. Sect. IV).

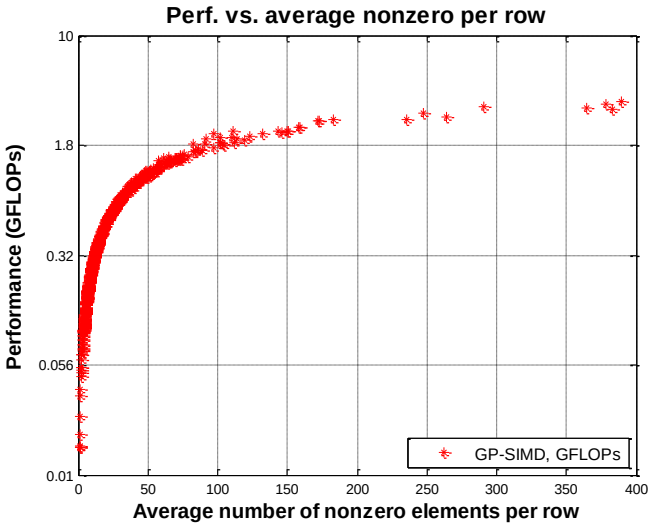

Figure. 13. Performance (GFLOPs) vs. average number of nonzero elements per row

The performance the GP-SIMD SpMM algorithm as a function of the average number of nonzero elements per row is presented in Figure. 13. The figure demonstrates a close to logarithmic dependency of the GP-SIMD SpMM algorithm performance on the average number of nonzero elements per row. Hence, if the average number of nonzero elements per row is small (which is consistently the case in the University of Florida collection matrices), the effectiveness of the GP-SIMD SpMM algorithm is limited. GP-SIMD SpMM algorithm is least efficient for diagonal matrices, where there is only about one multiplication per nonzero row. On the other end of the efficiency scale is dense matrix multiplication, where the *Multiply* procedure is applied to $M \cdot L$ elements of the multiplicand matrix B in parallel, per each matrix row.

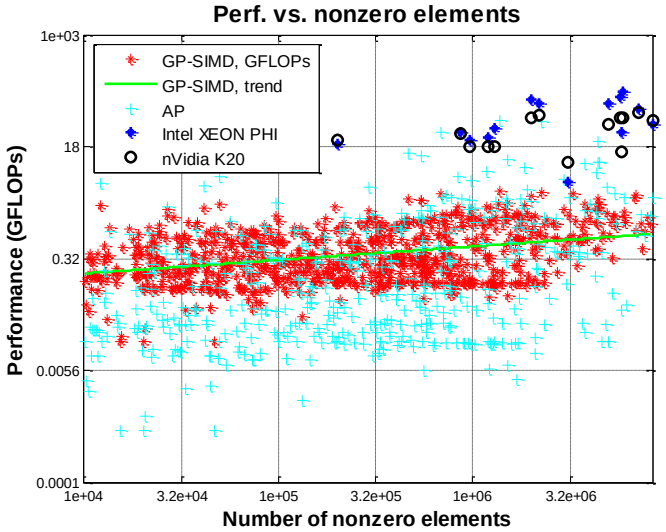

Figure 14. Performance (GFLOPs) vs. number of nonzero elements

The performance the GP-SIMD SpMM, AP, and two commercial processors (Intel XEON-PHI and K20 [10]) as functions of the number of nonzero elements is presented in Figure 14, and the simulated power consumption of the GP-SIMD SpMM, and AP (as well as reported power of NVidia K20 [34]) is presented in Figure 15(a).

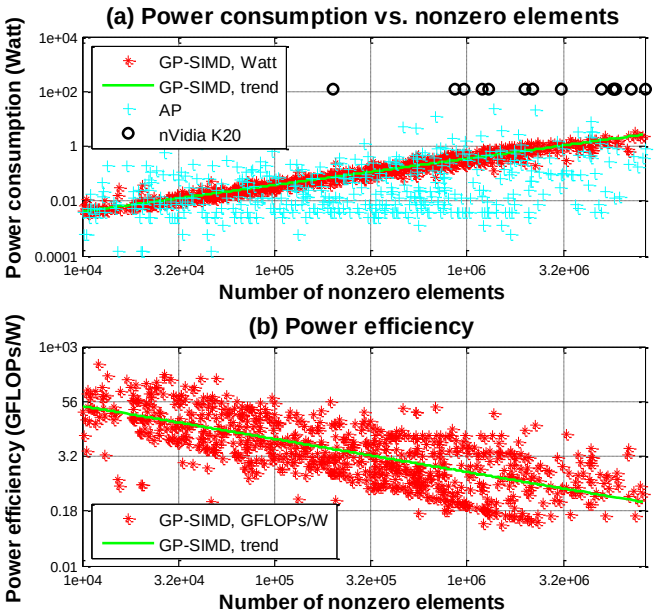

Figure 15. (a) Power consumption (Watt) vs. average number of nonzero elements per row. (b) Power efficiency (GFLOPs/W) vs. average number of nonzero elements per row

The GP-SIMD SpMM power efficiency is in the range of 0.1 to 100 GFLOPS/W (see Figure 15(b)). The power efficiency declines with the number of nonzero elements (requiring higher power consumption). The SpMM/SpMV power efficiency of advanced contemporary GPUs such as NVidia's K20 and GTX660 is in the 0.1-0.5 GFLOPS/W range [34]. A wide variety of multicore processors such as quad-core AMD Opteron 2214, quad-core Intel Xeon E5345, eight-core Sun UltraSparc T2+ T5140 and eight-SPE IBM QS20 Cell reportedly reach the SpMM power efficiency of up to 0.03 GFLOPS/W [38]. Several FPGA SpMV and SpMM implementation were proposed (for example [24] [20]), however these studies focused on optimization of performance or energy-delay, and power dissipation figures were not reported. The GP-SIMD and AP power efficiency advantage stem from in-memory computing (there are no data transfers between processing units and memory hierarchies) and from low-power design made possible by the very small size of each processing unit.

A noticeable limitation of the GP-SIMD SpMM algorithm is the sequential processing of matrix rows (the outer loop of Figure 10). A parallelization of matrix row processing may significantly improve the performance of the GP-SIMD SpMM algorithm. For example, diagonal matrices can easily be processed in a row-parallel manner when there is only one nonzero singleton product per each matrix row.

VI. Conclusions

We investigate an efficient implementation of dense and sparse matrix multiplication for the GP-SIMD, and simulate sparse matrix multiplication using a large variety of sparse matrices. Dense $N \times M$ matrix multiplication algorithm has a computational complexity of $O(NM \log_2 M)$ and the efficiency grows with the number of columns of the multiplicand matrix. Further, sparse matrix multiplication has a computational complexity of $O(A_{NNZ} \log_2 M)$ (where A_{NNZ} is the number of nonzero elements of the multiplier matrix), and the efficiency grows with the number of multiplier matrix's average nonzero elements per row, and with the number of multiplicand matrix's columns.

Lastly, we show that GP-SIMD sparse matrix multiplication is more power-efficient than conventional GPU or multicore based solutions.

Acknowledgment

This research was partially funded by the Intel Collaborative Research Institute for Computational Intelligence and by Hasso-Plattner-Institut.

References

[1] "The Intel® Xeon Phi™ Coprocessor". Available at: http://www.intel.com/content/www/us/en/high-performance-computing/high-performance-xeon-phi-coprocessor-brief.html

[2] A. Morad, L. Yavits, R. Ginosar, "GP-SIMD Processing-in-Memory", 2014, http://webee.technion.ac.il/~ran/papers/GP-SIMDProcessing-in-Memory-2014.pdf.

[3] ARM® NEON™ general-purpose SIMD engine, http://www.arm.com/products/processors/technologies/neon.php

[4] A. Pinar, M. Heath. "Improving performance of sparse matrix-vector multiplication." In Proceedings of the 1999 ACM/IEEE conference on Supercomputing (CDROM), p. 30. ACM, 1999.

[5] C. Stormon, "The Coherent Processor: an associative processor architecture and applications." In IEEE Compcon, Digest of Papers, pp. 270-275., 1991.

[6] D. Bowler, T. Miyazaki, M. Gillan. "Parallel sparse matrix multiplication for linear scaling electronic structure calculations." Computer physics communications 137, no. 2 (2001): 255-273.

[7] D. Steinkraus D., L. Buck, P. Simard, "Using GPUs for machine learning algorithms," IEEE ICDAR 2005.

[8] D. Burger, T. Austin. "The SimpleScalar tool set, version 2.0", ACM SIGARCH Computer Architecture News 25.3 (1997): 13-25.

[9] E. Im, K. Yelick. Optimizing the performance of sparse matrix-vector multiplication. University of California, Berkeley, 2000.

[10] E. Saule, *et al.* "Performance Evaluation of Sparse Matrix Multiplication Kernels on Intel Xeon Phi." arXiv preprint arXiv:1302.1078 (2013).

[11] G. Blelloch, "Vector Models for Data-Parallel Computing", MIT Press, 1990.

[12] G. Qing, X. Guo, R. Patel, E. Ipek, E. Friedman. "AP-DIMM: Associative Computing with STT-MRAM," ISCA 2013.

[13] J. Andersen, G. Mitra, D. Parkinson. "The scheduling of sparse matrix-vector multiplication on a massively parallel DAP computer." Parallel Computing 18, no. 6 (1992): 675-697.

[14] J. Bolz, I. Farmer, E. Grinspun, and Peter Schröoder. "Sparse matrix solvers on the GPU: conjugate gradients and multigrid." In ACM Transactions on Graphics, vol. 22, no. 3, pp. 917-924. ACM, 2003.

[15] J. Kurzak, D. Bader, J. Dongarra, "Scientific Computing with Multicore and Accelerators", CRC Press, Inc., 2010.

[16] J. Potter, W. Meilander. "Array processor supercomputers", Proceedings of the IEEE 77, no. 12 (1989): 1896-1914.

[17] J. Sun, G. Peterson, O. Storaasli. "Sparse matrix-vector multiplication design on FPGAs." In Field-Programmable Custom Computing Machines, 15th Annual IEEE Symposium on FCCM, pp. 349-352, 2007.

[18] K. Banerjee et al., "A self-consistent junction temperature estimation methodology for nanometer scale ICs with implications for performance and thermal management," IEEE IEDM, 2003, pp. 887-890.

[19] K. Eshraghian, et al. "Memristor MOS content addressable memory (MCAM): Hybrid architecture for future high performance search engines", IEEE Transactions on VLSI Systems, 19.8 (2011): 1407-1417.

[20] L. Colin Yu, et al. "Design space exploration for sparse matrix-matrix multiplication on FPGAs." International Journal of Circuit Theory and Applications 41.2 (2013): 205-219.

[21] L. Yavits, A. Morad, R. Ginosar, "Computer Architecture with Associative Processor Replacing Last Level Cache and SIMD Accelerator", IEEE Transactions on Computers, 2014.

[22] L. Yavits, A. Morad, R. Ginosar, "Sparse Matrix Multiplication on Associative Processor", http://webee.technion.ac.il/people/ran/papers/YavitsSpMMonAP.pdf

[23] L. Yavits, A. Morad, R. Ginosar, "The effect of communication and synchronization on Amdahl's law in multicore systems", Parallel Computing Journal, 2013.

[24] L. Zhuo, V. Prasanna. "Sparse matrix-vector multiplication on FPGAs." In Proceedings of the 2005 ACM/SIGDA 13th international symposium on Field-programmable gate arrays, pp. 63-74. ACM, 2005.

[25] M. Baskaran, R. Bordawekar. "Optimizing sparse matrix-vector multiplication on GPUs using compile-time and run-time strategies." IBM Research Report, RC24704 (W0812-047) (2008).

[26] M. Gschwind et. al., "Synergistic processing in Cell's multicore architecture", IEEE Micro 26 (2), 2006, pp. 10–24.

[27] M. Misra, D. Nassimi, V. Prasanna. "Efficient VLSI implementation of iterative solutions to sparse linear systems." Parallel Computing 19, no. 5 (1993): 525-544.

[28] N. Bell, M. Garland. "Implementing sparse matrix-vector multiplication on throughput-oriented processors." In Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis, p. 18. ACM, 2009.

[29] N. Bell, M. Garland. "Efficient sparse matrix-vector multiplication on CUDA", Vol. 20. NVIDIA Technical Report NVR-2008-004, NVIDIA Corporation, 2008.

[30] O. Beaumont, et al. "A proposal for a heterogeneous cluster ScaLAPACK (dense linear solvers)", IEEE Transactions on Computers, 50.10 (2001): 1052-1070.

[31] O. Beaumont, et al. "Matrix multiplication on heterogeneous platforms", IEEE Transactions on Parallel and Distributed Systems, 12.10 (2001): 1033-1051.

[32] O. Wing, "A content-addressable systolic array for sparse matrix computation." Journal of Parallel and Distributed Computing 2, no. 2 (1985): 170-181.

[33] Q. Zhu, *et al.* "Accelerating Sparse Matrix-Matrix Multiplication with 3D-Stacked Logic-in-Memory Hardware", IEEE HPEC 2013

[34] R. Dorrance *et al.*, "A scalable sparse matrix-vector multiplication kernel for energy-efficient sparse-BLAS on FPGAs", 2014 ACM/SIGDA international symposium on Field-programmable gate arrays.

[35] R. Vuduc, "Automatic performance tuning of sparse matrix kernels." PhD diss., University of California, 2003.

[36] S. Sengupta, M. Harris, Y. Zhang, J Owens. "Scan primitives for GPU computing." In Graphics Hardware, vol. 2007, pp. 97-106. 2007.

[37] S. Toledo, "Improving the memory-system performance of sparse-matrix vector multiplication." IBM Journal of research and development 41, no. 6 (1997): 711-725.

[38] S. Williams et al., "Optimization of sparse matrix–vector multiplication on emerging multicore platforms." Parallel Computing 35, no. 3 (2009): 178-194.

[39] T. Davis, Y. Hu, "The University of Florida sparse matrix collection," ACM Transactions on Mathematical Software (TOMS), 38, no. 1 (2011): 1.

[40] X. Liu, M. Smelyanskiy, "Efficient sparse matrix-vector multiplication on x86-based many-core processors", International conference on supercomputing, ACM, 2013.

[41] Y. Saad, A. Malevsky. "PSPARSLIB: A portable library of distributed memory sparse iterative solvers", Tech. Rep. UMSI 95/180, University of Minnesota, 1995.