ROMANIAN JOURNAL OF INFORMATION SCIENCE AND TECHNOLOGY
Volume 23, Number 3, 2020, 262–273

Sparse Matrix-Vector Multiplication on a Map-Reduce Many-Core Accelerator

Voichiţa DRAGOMIR[1] and Gheorghe M. ŞTEFAN[2]

[1]Politehnica University of Bucharest
Email: voichita.dragomir@upb.ro
[2]Politehnica University of Bucharest
Email: gheorghe.stefan@upb.ro

Abstract. Our proposal for accelerating the computation of Sparse Matrix Vector Multiplication is a Map-Reduce Accelerator as part of a heterogenous computer. We prove that both, structured and unstructured sparse matrices are efficiently multiplied with a dense vector approach using a parallel accelerator structured as a linear array of cells loop connected, through a *log*-depth reduction network, with a controller. The specific algorithms are presented and their implementation is compared with the of-the-shelf solutions. The main advantages of our architectural proposal, compared with the GeForce GTX 280 GPU which is implemented in the same technological node, are: (1) it provides the means to use $5 \div 12\times$ more computation out of the peak computational power, (2) it performs the computation with $2.5\times$ less energy.

Keywords: sparse matrix, matrix-vector multiplication, unstructured sparse matrix, structured sparse matrix, heterogenous computing.

1. Introduction

Real problems are often modeled using linear algebra. Among these, there are a lot of applications leading toward sparse matrix modelling. More precisely, the multiplication of a rare matrix with a dense vector is of great interest. The Sparse Matrix Vector Multiplication (SpMV) is defined on big $n \times n$ matrices (sometimes $n \geq 10^6$) with a high sparsity ($nonzeroes/row\_size \sim 10^{-5}$) [2]. Sometimes the sparsity is random, while sometimes is organized in bands. The algorithms for these sparse matrices are specific. They solve the problems raised by the memory space and data transfer between the memory and the processing engine. The price to pay in this case, compared with the dense matrices, is the reduced acceleration caused by various specific parallel calculation that needs to be done. The non-uniformity of the representations requests sometimes costly solutions. The main effort consists of putting at work all the computation power our accelerator is able to deliver.

In [2] only 15 GFLOPS from the peak performance of 933 GFLOPS [1] are used for unstructured sparse matrices, while for the structured sparse matrices 36 GFLOPS are used. Only 1.6% from peak performance for unstructured matrices or 3.8% from peak performance for structured matrices! The degree of suitability of an architecture, ARCH, for the problem it solves, is given by the *use coefficient*:

$$\alpha_{ARCH} = \frac{used\_number\_of\_FLOPS/sec}{peack\_number\_of\_FLOPS/sec}$$

The architecture we propose for the accelerator of a heterogenous computing engine – Map-Reduce Accelerator (MRA) – provides for the SpMV application domains the use coefficient $\alpha_{MRA} > 20\%$, while the current, GPU-based solutions provide only $\alpha_{GPGPU} < 4\%$ [2].

In the next section our architecture is presented. The third section develops algorithms for SpMV and compares them with current implementations. The last section concludes our work.

2. The Architecture

2.1. The Structure of the Accelerator

The system considered in our approach (see Figure 1) is a heterogenous one. It has a general purpose processor as HOST tightly coupled with an ACCELERATOR, a many-cell array, called MAP array, equipped with a CONTROLLER which sends in each clock cycle, through a *log*-depth DISTRIBUTE network, an instruction to be executed in the active cells, $c_0, c_1, \ldots, c_{p-1}$, and receives back from the MAP array, through a *log*-depth reduction network, REDUCE, scalars, as the result of a reduction function applied to the active components of a vector distributed along the linear array of cells MAP.

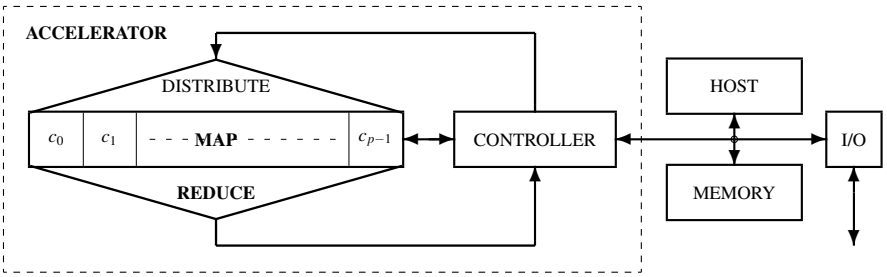

Fig. 1. The heterogenous system equipped with the map-reduce ACCELERATOR.

Each cell, c_i, consists of an accumulator centered execution unit and a big register file of few KWords.

2.2. Architectural resources

The data organization of ACCELERATOR consists of vector resources for the p-cell MAP section and scalar resources for CONTROLLER, as follows:

IX : index vector – `indexVect[0:p-1]` – is a constant vector used to identify each cell

VM : vector memory – `vectMem[0:m-1][0:p-1]` – is the distributed memory along the cells; each cell is endowed with `vectMem[0:m-1][index]`

AV : address vector – `addrVect[0:p-1]` – used for relative addressing in the local register files

ACC : accumulator vector – `accVect[0:p-1]` – is the left source and destination for the arithmetic and logic operations

BV : Boolean vector – `b[0:p-1]` – is used to enable the execution in the current clock cycle

SR : serial register – `serReg[0:p-1]` – is a serial-parallel register used to communicate in parallel with ACC and serially with acc

IOR : input-output register – `ioReg[0:p-1]` – is the register used to transfer data between VM and the system memory

PM : is the program memory having in each location a pair of instructions: one to be executed in the active cells of the MAP array, another for CONTROLLER

SM : scalar memory – `mem[0:s-1]` – used for the control process

pc : program counter

acc : accumulator – `acc` – used for the control process

addr : address pointer – `addr` – used to perform relative addressing SM for the control process.

2.3. Instruction Set Architecture

In each clock cycle two 16-bit instructions are fetched from the program memory of CONTROLLER, one for CONTROLLER and another for the MAP array. The double instruction format is:

```
instruction = {arrayOpCode  , // operation code for array
               arrayOperand , // operand select code for array
               arrayScalar  , // immediate value/address for array
               contrOpCode  , // operation code for controller
               contrOperand , // operand select code for controller
               contrScalar  } // immediate value/address for controller
```

A line command for ACCELERATOR has three fields in assembly:

```
LB(X);     cYYY;     ZZZ;
```

where:

- `X`: is a integer representing a label for jumps and the conditional branch instructions
- `YYY`: is an instruction for CONTROLLER
- `ZZZ`: is a MAP instruction executed in the active cells with a latency in $O(log\ p)$

The operands for the ALUs in CONTTROLLER and in each cell of MAP are mainly the following:

- left operand: `acc` for the control process, and `accVect[i]` in each of the p cells of the map array
- right operand: `op` for the control process, and `op[i]` for each of the p cells of the map array:
  - immediate value: `op = contrScalar`, and `op[i] = arrayScalar`
  - directly addressed in the local memory by `op = mem[contrScalar]` in CONTROLLER or by `op[i] = vectMem[arrayScalar][i]` in MAP
  - indirectly addressed the local memory by `op = mem[addr + contrScalar]` or by `op[i] = vectMem[addrVect[i] + arrayScalar][i]`
  - indirectly addressed the local memory by `op = mem[addr + contrScalar]` or by `op[i] = vectMem[addrVect[i] + arrayScalar][i]` **and** increment: `addr <= addr + contrScalar` in CONTROLLER and `addrVect[i] <= addrVect[i] + arrayScalar` in MAP
  - cooperand provided:
    - for the control process by the *log*-depth REDUCE network of the array:
      - `op = redAdd`: where $redAdd = \sum_0^{p-1}(boolVect[i]\ ?\ accVect[i]\ :\ 0)$
      - `op = redMax`: where $redMax = Max_0^{p-1}(boolVect[i]\ ?\ accVect[i]\ :\ 0)$
      - `op = redMin`: where $redMin = Min_0^{p-1}(boolVect[i]\ ?\ accVect[i]\ :\ 0)$
    - for the array by the control process: `op[i] = acc`

The instructions, performed for control on scalars and in array, on vectors, are mainly the standard unary and binary operations (for a full description of the architecture see [3]). Besides these, some specific operations are implemented in the map-reduce array:

- **spatial selections**: allow predicated executions using, for $i = 0, 1, \ldots, p-1$, the following instructions:

```
WHERE(cond) :b[i] <= cond(arrayScalar[2:0]) ?  1 :  0
ELSEWHERE :b[i] <= !b[i]
ENDWHERE :b[i] <= 1
```

- **global shift** operations:

```
GROTATE :accVect[i] <= acc[(i+1)%(1<<x)]
GLSHIFT :accVect[i] <= (i==0) ?  0 :  accVect[i-1]
GRSHIFT :accVect[i] <= (i<(1<<x)-1) ?  accVect[i+1] :  0
GLSHEXT :accVect[i] <= (i==0) ?  ext :  accVect[i-1];
    ext <= accVect[0]
GRSHEXT :accVect[i] <= (i<(1<<x)-1) ?  accVect[i+1] :  ext;
    ext <= accVect[(1<<x)-1]
```

where `ext` is an extension register used to perform multi-vector shifts.

- **scan** operation: generates the vector FIRST with a latency in $O(log\ p)$ following the evolutions of the values of the vector BV, as follows:

```
BV    = {0 0 ... 0 1 x x ... x}
FIRST = {0 0 ... 0 1 0 0 ... 0}
```

where $x \in \{0,1\}$

- **reduction** operations: generate with a latency in $O(log\ p)$ following the evolutions of the values of the vectors ACC and BV, as follows:

`redAdd` : $\sum_0^{p-1}(boolVect[i]\ ?\ accVect[i]\ :\ 0)$

`redMax` : $Max_0^{p-1}(boolVect[i]\ ?\ accVect[i]\ :\ 0)$

`redMin` : $Min_0^{p-1}(boolVect[i]\ ?\ accVect[i]\ :\ 0)$

- transfer operations are used to load vectors or to store vectors from/to the main memory of the system, MEMORY (see Figure 1).

2.4. Physical Implementation

MRA was initially implemented for various image processing accelerators [4]. We consider for this paper the last silicon implementation, in 65*nm*, which for 1024 32-bit cells comes with 10 $GFLOP/sec/Watt$ for float operations, or with 60 $GOP/sec/Watt$ for 32-bit integer operations (in [7] the whole story leading to MRA is told and the main publications are listed).

3. Algorithms

There are two kinds of sparsity: unstructured and structured. The algorithms are specific for each of the two kinds of sparsity.

3.1. Unstructured Sparse Matrices

An unstructured sparse matrix is represented in various forms. We use here the *coordinate format* (COO) of the matrix which means using three vectors: one for the row indexes (*rowVect*), another for column indexes (*colVect*) and the last one for the non zero values (*valVect*).

Example 1. *For example, the matrix-vector multiplication operation:*

$$SM = \begin{bmatrix} 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} 0 \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \end{bmatrix} = \begin{bmatrix} 2 \\ 12 \\ 7 \\ 6 \\ 1 \\ 5 \\ 15 \\ 8 \end{bmatrix}$$

is performed using the COO representation for the previous 8×8 *sparse matrix:*

$$\begin{bmatrix} rowVector \\ colVector \\ valVector \end{bmatrix} = \begin{bmatrix} 0,1,1,1,2,2,3,3,4,5,5,6,6,6,7,7 \\ 2,1,4,7,2,5,0,6,1,2,3,4,5,6,1,7 \\ 1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1 \end{bmatrix}$$

The dense vector is

$$inVect^T = \begin{bmatrix} 0,1,2,3,4,5,6,7 \end{bmatrix}$$

and the result of the multiplication is

$$outVect^T = \begin{bmatrix} 2,12,7,6,1,5,15,8 \end{bmatrix}$$

◇

Depending on the mean value of the `nonzero/row` parameter, the algorithm is designed for a pure SIMD architecture or for a SPMD architecture. For very small `nonzero/row`, the SIMD-like algorithm is recommended, while for enough big `nonzero/row` the SPMD-like algorithm can be used.

The usual size of the matrices falls within the range in which n is equal to tens of thousands to one million. The current many-core engines accommodates no more than thousands of cells. Therefore, a big $n \times n$ matrix must be tiled in many small ones and delivered in an appropriate way to the accelerator.

3.1.1. The SIMD-like kernel algorithm

When the diversity in the two index vectors – *rowVect* and *colVect* – tends to be maximal (almost all, if not all, the q components are different) the best solution is to load in each cell the 4 vectors associated to one SpMV computation which takes a tile from the matrix and the associated sub-vector.

```
/*************************************************************
SpMV algorithm for a n x n unstructured matrix with q non-zeroes
*************************************************************/
algorithm spmvUnstrucKernel(rowVect[0:q-1],
                            colVect[0:q-1],
                            valVect[0:q-1],
                            inVect[0:n-1] ,
                            outVect[0:n-1]);
    for(i=0; i<n; i=i+1) outVect[i] <= 0;
    for(i=0; i<q; i=i+1) vect[i] <= inVect[colVect[i]];
    for(i=0; i<q; i=i+1) vect[i] <= vect[i] * valVect[i];
    for(i=0; i<q; i=i+1)
        outVect[rowVect[i]] <= outVect[rowVect[i]] + vect[i];
endalgorithm
```

The execution time for $p \times q$ non-zeroes in integer arithmetic is $T_{SpMV}(n,q) = 8+n+26q$, while for floating point arithmetic is $T_{SpMV}(n,q) = 8+n+36q$. Therefore, from the peak floating point performance the algorithm uses 32%.

3.1.2. The SPMD-like kernel algorithm

When the diversity in the two index vectors – *rowVect* and *colVect* – is not so high (almost all, if not all, indexes occur many times) the POSSIBLE solution is to load the 4 vectors associated to one SpMV computation as horizontal vectors in the map array. It is about matrices near to dense ones or not too sparse. For them $nonzeroes/n < 0.5$ but not $nonzeroes/n << 0.5$.

The kernel algorithm for this kind of sparse matrix is defined for a $n \times n$ sparse matrix multiplied with a n-component dense vector in a p-cell engine, for a number of $q \leq p$ nonzero elements in the matrix. The compressed format (COO) of the matrix is used. The algorithm is:

```
/*****************************************************************
SpMV algorithm for unstructured matrix
*****************************************************************/
algorithm spmvUnstrucKernel(rowVect[0:q-1],
                            colVect[0:q-1],
                            valVect[0:q-1],
                            inVect[0:n-1] ,
                            outVect[0:n-1]);
    for(i=0, i<n, i=i+1)          // MULTIPLY STAGE
        where (colVect = i)
            valVect <= multiply(valVect, inVect[i]);
        endwhere
    for(i=0, i<m, i=i+1)          // ADD STAGE
        where (rowVect = i)
            outVect[i] <= reductionAdd(valVect);
        endwhere
endalgorithm
```

The execution time is in $O(n)$. The actual program is listed below in order to show an example of how the accelerator is programmed in assembly language. The program consists in a two-column code, one for CONTROLLER and another for the array of cells.

```
/*****************************************************************************
SpMV program for unstructured matrix:
    vectMem[1] = rawVect
    vectMem[2] = colVect
    vectMem[3] = valVect
    vectMem[0] = inVec

    accVect[i] <= outVect[i]
*****************************************************************************/
        cVLOAD(0);  LOAD(0);      // acc <= 0 = counter; accVect[i] <= inVect[i]
        cSTORE(0);  SRSTORE;      // mem[0] = counter; serialReg[i] <= accVect[i]
                                  // MULTIPLY STAGE
LB(1);  cLOAD(0);   LOAD(2);      // acc <= counter; accVect[i] <= colVect[i]
        cVADD(1);   CSUB;         // acc <= acc + 1; accVect[i] <= accVect[i] - counter
        cSTORE(0);  SRLEFT;       // save counter; shift left serialReg
        cCLOAD(4);  WHEREZERO;    // acc <= inVect[counter]; column select
        cLOAD(0);   CLOAD;        // acc <= counter; accVect[i] <= inVect[counter]
        cVSUB(8);   STORE(4);     // acc <= acc - n; vectMem[4][i] <= accVect[i]
        cBRNZ(1);   ENDWHERE;     // loop if 0; reactivate all cells
        cNOP;       LOAD(4);      // acc[i] <= vectMem[4][i]
```

```
        cNOP;          MULT(3);      // acc[i] <= accVect[i] x vectMem[3][i]
        cVLOAD(8);     STORE(5);     // acc <= n; vectMem[5][i] <= accVect[i]
                                     // ADD STAGE
LB(2);  cVSUB(1);      LOAD(1);      // acc <= acc - 1; accVect[i] <= vectMem[1][i]
        cNOP;          CSUB;         // acc[i] <= accVect[i] - acc
        cNOP;          WHEREZERO;    // line select
        cNOP;          LOAD(5);      // acc[i] <= vectMem[5][i] to drive the reduction net
        cCPUSHL(0);    ENDWHERE;     // redOut is pushed in serialReg; reactivate all cells
        cBRNZ(2);      NOP;          // loop is 0
        cNOP;          NOP;          // latency step
        cNOP;          NOP;          // latency step
        cNOP;          SRLOAD;       // acc[i] <= serialReg[i]
```

The loop labeled with `LB(1)` distributes in one vector (`vectMem[4]`) the elements `i` of `inVector` in the positions associated with the elements of the columns `i`. Then, all multiplications are performed in one parallel operation. The loop labeled with `LB(2)` selects by turn the elements of each line already multiplied and applies them to the reduction network to be added.

SpMV execution time for $n \times n$ sparse matrix is:

$$T(n) = 13n + 8 \in O(n)$$

The acceleration derives from the fast search enabled by our architecture. But, it is meaningful only if there are much more than one nonzero element for each line/column.

3.1.3. Algorithm for n as large as possible

Tiling a matrix is the procedure used to expand the kernel algorithm to a large matrix for a whatever n. If $q/p > 1$ for SPMD algorithm or $3q + n > m$ for SIMD algorithm, then we use the `spmvUnstrucKernel` algorithm to design the `spmvUnstruc` algorithm to work on no matter how big the size of the data structure is. Instead of a matrix of scalars, we deal with a matrix of tiles. Each tile is submitted to the kernel algorithm and the final result is obtained in two steps:

1. the results of each line of horizontally distributed tiles are added, hence providing sub-vectors of the final result
2. the sub-vectors resulting form the first step are concatenated to obtain the final result.

The program which uses the `spmvUnstrucKernel` function, as a component of a kernel library, runs on the host and it is written in a high level language.

3.1.4. Comparison with GPU

The SIMD solution On GeForce GTX 280, which is deployed with GT200 GPU (65 nm, 576 mm^2, $TPD = 230\ W$), from its 933 $GFLOP/s$ peak performance are used 6 $GFLOP/s$ for SpMV multiplication. It uses only $\alpha_{GT200} = 6/933 \rightarrow 0.65\%$ from its peak performance.

For our SIMD solution $\alpha_{MRA} = 12/37 \rightarrow 32\%$ from its peak performance. Therefore,

$$\alpha_{MRA}/\alpha_{GT200} = 50$$

i.e., the use of floating point arithmetic peak performance is $50\times$ more efficient in our architecture [2]. We must add the fact that GT200 provides 4 $GFLOP/sec/Watt$, compared with

MRA which provides 10 $GFLOP/sec/Watt$. Therefore, we use $50\times$ more efficiently the FLOPs obtained, spending $2.5\times$ less energy [1].

The SPMD solution is of interest in applications where a matrix becomes sparse by neglecting the small values. Then, the number of nonzeroes is big enough on each line/column. For these kinds of applications (with weight matrices whose values result from the training process of a Deep Neural Network) we do not have meaningful data yet.

3.2. Structured Sparse Matrices

The structured $n \times n$ sparse matrix is a b-band band matrix represented with vectors for each diagonal. The main diagonal is a n-component vector, while the upper diagonals are represented by u n-component padded vectors with initial 0s, and the DOWN diagonals are represented by d n-component padded vectors ending with 0s. Thus, $b = u + d + 1$.

Example 2. *For example, the band matrix - vector multiplication of a* 8×8 *band matrix with* $u = 1$ *and* $d = 2$*:*

$$BM = \begin{bmatrix} 1 & 2 & 0 & 0 & 0 & 0 & 0 & 0 \\ 3 & 1 & 2 & 0 & 0 & 0 & 0 & 0 \\ 4 & 3 & 1 & 2 & 0 & 0 & 0 & 0 \\ 0 & 4 & 3 & 1 & 2 & 0 & 0 & 0 \\ 0 & 0 & 4 & 3 & 1 & 2 & 0 & 0 \\ 0 & 0 & 0 & 4 & 3 & 1 & 2 & 0 \\ 0 & 0 & 0 & 0 & 4 & 3 & 1 & 2 \\ 0 & 0 & 0 & 0 & 0 & 4 & 3 & 1 \end{bmatrix} \times \begin{bmatrix} 0 \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \end{bmatrix} = \begin{bmatrix} 2 \\ 5 \\ 11 \\ 21 \\ 31 \\ 41 \\ 51 \\ 45 \end{bmatrix}$$

uses the following representation for the band matrix:

$$VM = \begin{bmatrix} UpDiagonal\_1 \\ MainDiagonal \\ DownDiagonal\_1 \\ DownDiagonal\_2 \end{bmatrix} = \begin{bmatrix} 0 & 2 & 2 & 2 & 2 & 2 & 2 & 2 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 3 & 3 & 3 & 3 & 3 & 3 & 3 & 0 \\ 4 & 4 & 4 & 4 & 4 & 4 & 0 & 0 \end{bmatrix}$$

and provides the following vector as result:

$$\begin{bmatrix} 2, 5, 11, 21, 31, 41, 51, 45 \end{bmatrix}$$

$\diamond$

3.2.1. The algorithm

For $n \leq p$ the kernel algorithm uses map functions (multiplication and addition) and shift functions issued under the control of the CONTROLLER unit.

```
/*****************************************************************
SpMV for band matrix
*****************************************************************/
algorithm spmvBandKernel();
    for (k=1, k=<u, k=k+1)
        UpDiagonal_k <= multply(UpDiagonal_k, inVector) << k
    MainDiagonal <=  multply(MainDiagonal, inVector);
    for (k=1, k=<d, k=k+1)
        DownDiagonal_k <= multply(DownDiagonal_k, inVector) >> k
    outVector = sum(UpDiagonal_1, ..., UpDiagonal_u,
                    MainDiagonal,
                    DownDiagonal_1, ..., DownDiagonal_d);
endalgorithm
```

3.2.2. The program for $n \leq p$

The program in assembly language is:

```
/*****************************************************************************
SpMV for structured (band) matrix
    mem[0] = b; // band width
    mem[1] = 0; // vectorAddress in vectMem;
    mem[2] = u; // number of upper diagonals
    mem[3] = d; // number of down diagonals
    mem[4] = counter
    vectMem[0]   = vector;
    vectMem[1]   = diagonal_1;
    ...
    vectMem[b]   = diagonal_b;
    vectMem[b+1] = result
*****************************************************************************/
        cLOAD(0);       VLOAD(0);    // MULTIPLICATIONS
        cVSUB(1);       ADDRLD;
LB(3);  cNOP;           RILOAD(1);
        cNOP;           MULT(0);
        cBRNZDEC(3);    RSTORE(0);
        cLOAD(2);       VLOAD(0);    // LEFT SHIFTS
        cSTORE(4);      ADDRLD;
LB(5);  cLOAD(4);       RILOAD(1);
        cBRZDEC(9);     NOP;
        cSTORE(4);      NOP;
LB(4);  cBRNZDEC(4);    GLSHIFT;
        cJMP(5);        RSTORE(0);
LB(9);  cLOAD(3);       NOP;         // RIGHT SHIFTS
        cSTORE(4);      NOP;
LB(7);  cLOAD(4);       NOP;
        cBRZDEC(6);     NOP;
        cSTORE(4);      NOP;
        cRSUB(3);       RILOAD(1);
LB(8);  cBRNZDEC(8);    GRSHIFT;
        cJMP(7);        RSTORE(0);
LB(6);  cNOP;           VLOAD(0);    // ADDS
        cLOAD(0);       ADDRLD;
        cVSUB(1);       VLOAD(0);
LB(4);  cBRNZDEC(4);    RIADD(1);
        cNOP;           RSTORE(0);
```

The execution time in the worst case scenario, for $d = b-1$, $u = 0$, is:

$$T_{SpMV}(b) = 0.5b^2 + 9.5b + 9$$

for 32-bit integer arithmetic, while for 32-bit floating point arithmetic is:

$$T_{SpMV}(b) = 0.5b^2 + 19.5b + 9$$

3.2.3. The program for $n > p$

For $n > p$ the algorithm is the same. The main differences are in the implementation of the shift, multiply and addition operations. Instead of the lines:

```
...
LB(4);   cBRNZDEC(4);     GLSHIFT;
...
LB(8);   cBRNZDEC(8);     GRSHIFT;
...
```

used for shifts in the previous program, loops of three lines are used for each type of shifts, as follows:

```
...
         cVSUB(1);        GLSHIFT;     // acc[i]<=acc[i+1]; acc[p-1]<=0; ext<=acc[0];
         cNOP;            RISTORE(1);
LB(4);   cBRNZDEC(10);    RILOAD(1);
         cNOP;            GLSHEXT;     // acc[i]<=acc[i+1]; acc[p-1]<=ext; ext<=acc[0];
         cJMP(4);         RSTORE(0);
LB(10);  ...
...
         cVSUB(1);        GRSHIFT;     // acc[i]<=acc[i-1]; acc[i]<=0; ext<=acc[p-1];
         cNOP;            RISTORE(1);
LB(8);   cBRNZDEC(11);    RILOAD(1);
         cNOP;            GRSHEXT;     // acc[i]<=acc[i-1]; acc[0]<=ext; ext<=acc[p-1];
         cJMP(4);         RSTORE(0);
LB(11);  ...
...
```

Another, simpler, difference is provided by the number of multiplications and additions which are multiplied by n/p. Therefore, the execution time, in the worst case, becomes for floating point arithmetic:

$$T_{SpMV}(b) = 1.5b^2n/p + 19.5bn/p + 7b + 9 \in O(b^2n/p)$$

which provides an acceleration $a \in O(p/b)$.

3.2.4. Comparison with GPU

On GeForce GTX 280 (previously described), for $b = 3$, $\alpha_{GT200} = 2.6\%$, while for $b = 27$, $\alpha_{GT200} = 3.9\%$ according to [2].

On our MRA, for $b = 3$, $\alpha_{MRA} = 50\%$, while for $b = 27$, $\alpha_{MRA} = 20\%$

Then the ratio $\alpha_{MRA}/\alpha_{GT200}$ varies from 19.23 to 5.12.

4. Conclusions

The efficiency of a SpMV algorithm is better expressed by the α coefficient, because the actual FLOP/sec performance is only a fraction from its peak, due to the matching process of the operands in performing multiplications and additions. Indeed, we have much less operations from the $n \times n$ multiplications and additions requested by a dense matrix, but putting together the nonzero components of the matrix with the vector's components costs us a lot of non arithmetic operations.

MRA implemented in 65nm achieved $\alpha_{MRA} \in [0.2, 0.5]$ while the Nvidia chip GT200 implemented in the same technology achieved only $\alpha_{GT200} \in [0.0065, 0.039]$.

The energy requested by each FLOP is $2.4\times$ less for the proposed MRA than for GT200 chip.

Future work: integrating the algorithms developed for SpMV in a kernel library used to define in Python the corresponding components of a consecrated linear algebra library.

References

[1] Fedy ABI-CHAHLA: "Nvidia GeForce GTX 260/280 Review", Available at: `https://www.tomshardware.com/reviews/nvidia-gtx-280,1953-12.html`

[2] Nathan BELL and Michael GARLAND: "Efficient Sparse Matrix-Vector Multiplication on CUDA", *NVIDIA Technical Report NVR-2008-004*, Dec. 2008.

[3] Gheorghe M. ŞTEFAN: *Simulation Manual for Configurable MapReduce Accelerator*, Available at: `http://users.dcae.pub.ro/~gstefan/2ndLevel/teachingMaterials/ypoSpec1.pdf`

[4] Gheorghe M. ŞTEFAN, et al.: "The CA1024: A Fully Programable System-On-Chip for Cost-Effective HDTV Media Processing", *Hot Chips: A Symposium on High Performance Chips*, Memorial Auditorium, Stanford University, August 20 to 22, 2006. Available at: `https://youtu.be/HMLT4EpKBAw` at 35:00.

[5] ***: *NVIDIA GeForce GTX 280*, Available at: `https://www.techpowerup.com/gpu-specs/geforce-gtx-280.c216`

[6] Ashu REGE: *An Introduction to Modern GPU ArchitectureOnline*, Available at: `http://download.nvidia.com/developer/cuda/seminar/TDCI_Arch.pdf`

[7] Gheorghe M. ŞTEFAN: *The Connex Project*, Available at: `http://users.dcae.pub.ro/~gstefan/2ndLevel/connex.html`