Simulation technique for cache performance analysis

Abhijeet Kumar
amk7371@psu.edu

Abhishek Kumar
azk60385@psu.edu

***Abstract*—Computer architecture simulation play an important role in implementation and evaluation of new designs. Along with that it also act as a bridge between hardware and software world and contribute significantly to the success of the design. Most of the recent simulator used in computer architecture community are either not well maintained or have a complex design to do any kind of modification and perform performance study which limits their usage in real world. We believe ideal simulator should balance between it's alignment with it's counter part and feasibility to provide quick simulation result for real-world workload.**
For our experimentation, we mainly target matrix multiplication operator because of it's usage in multiple domains like Artificial Intelligence (AI), Computer vision, network theory etc. Since the real world matrices tend to be sparse [6] and one particular part of any processor which is mostly effected with random access is cache hierarchy. We developed a Intel PIN framework and python based trace drive simulator for cache analysis. We have given user the full access to configure cache hierarchy. It is developed in such a way that it can be easily integrated with any core simulator to do performance analysis of full system.

I. Introduction

A cache simulator is a tool that mimics the cache subsystem with low percentage of error rate. In the processor cache is mainly used to reduce the average cost of accessing main memory. In recent years, performance gap between processor and memory system has increased which makes it even more important for architects to develop an advance cache system in order to get the needed performance gain from the ASIC. Evaluation of design can be done in many ways. For example, One can develop a hardware prototype and use it with production software stack to run benchmarks. However, this process is very costly and time consuming [2]. An alternated method is do develop analytical model but it's usage is limited and it is difficult to give meaningful stats for the complex workload mainly because the level abstraction is very high and it is not able to model features necessary to get close to ASIC performance number. In recent years, focus of computer architecture community and industry shifted toward simulation tool and infra because of it's high potential and relatively faster development cycle. In addition to being extensively used in pre-silicon stage, simulator can potentially be used for validation and verification task.
Cache simulator are mainly of two types i.e., functional or cycle Approximate. A functional simulator does not have notion of cycle and logically simulates the operations performed to run the scenario [2]. It is mainly used for generating traces of the program and doing functional check of the subsystem. From performance evaluation point of view, functional simulator add minimum value. So from now on, when we say simulator we are mainly referring to cycle approximate simulator unless being specified. Simulation methodologies can further be divided in three parts:

- Execution Driven - Runs the input program on the host platform directly
- Emulation Driven - Simulator creates a virtual platform where the input program is run.
- Trace Driven - User pass a trace file which is parsed by simulator and consume cycles needed to mimic the scenario on the given architecture.

In recent year, many simulator is developed and open-sourced [9] [1] for being used for study and exploration. In addition to Cache only simulation tool-set, some of the general purpose simulation tool also supports cache simulation to a certain extent. The main issue behind them not being used extensively is because of it not being user friendly and also difficulties faced by the researcher in doing modification to add new features. In addition to that some of them also used deprecated library which add more challenges to maintain it for long-run. In order to over come the challenges, we proposed a trace driven simulation system which create the memory hierarchy at each instance after parsing the YAML file passed by user. It has been written with minimum dependency on any external package to make it easy to maintain and ramp-up on. It also gave full control to user to change with timing parameter and level of cache-subsystem making it a generalised cache simulator.
In this paper we have also discussed briefly about the matrix multiplication algorithms and used it as a workload to develop and analyze our simulation technique. Matrix multiplication is widely being used in various domains like Artificial Intelligence, Computer Vision, Statistic and Network theory. As it's importance is increasing day by day it is crucial for computer architects to develop their system with keeping matrix multiplication operator in mind. Recently it is also being noticed that real world matrices are sparse in nature which have a huge impact on cache hierarchy because of random access to memory. As part of this project we also developed a bunch matmul kernel (which is discussed briefly in section III of this paper) for sanity check and performance evaluation of our sample three level cache system. We tested our simulator on matrices with variable size and sparsity to gain confidence on performance number generated by simulator.

II. Cache Simulator Overview

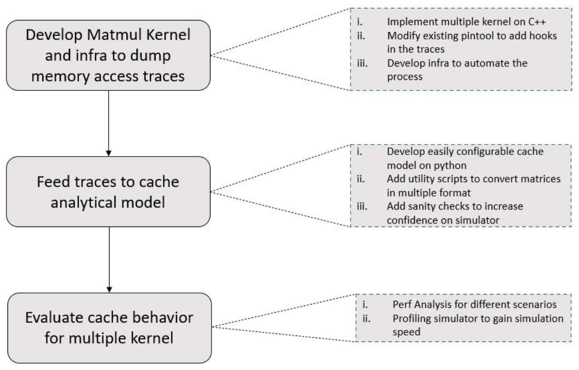

Develop Matmul Kernel and infra to dump memory access traces
i. Implement multiple kernel on C++
ii. Modify existing pintool to add hooks in the traces
iii. Develop infra to automate the process
Feed traces to cache analytical model
i. Develop easily configurable cache model on python
ii. Add utility scripts to convert matrices in multiple format
iii. Add sanity checks to increase confidence on simulator
Evaluate cache behavior for multiple kernel
i. Perf Analysis for different scenarios
ii. Profiling simulator to gain simulation speed

Fig. 1. Simulation flow charts

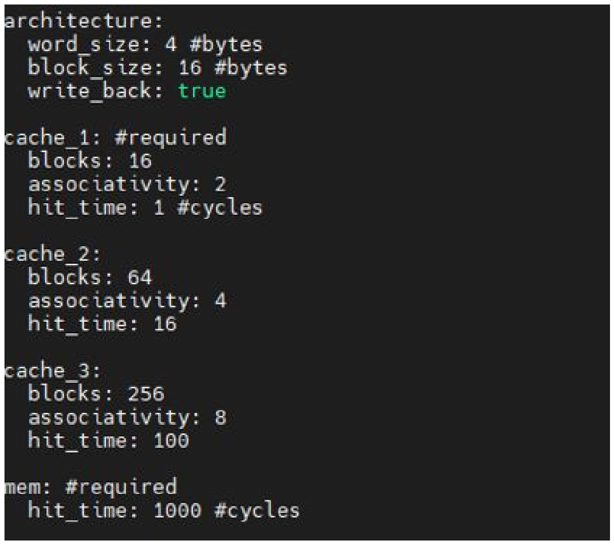
As shown in fig. 1, our simulator has two main components:

- Trace Generator
- Simulator instantiation

A. Trace Generator

In order to generate memory access traces from the application (i.e. matrix multiplication kernel) in our case we used Intel Pin [7]. It is a dynamic binary instrumentation framework [4] that have the capability to add hooks to the C/C++ application execution for dumping the memory read/write traces for a given set of function call. Please note that we developed our own pin tool by keeping example library (pinatrace) as baseline. In our case we generated traces from the application execution on Intel Xeon E5 CPU.

B. Cache Hierarchy creation



```
architecture:
  word_size: 4 #bytes
  block_size: 16 #bytes
  write_back: true

cache_1: #required
  blocks: 16
  associativity: 2
  hit_time: 1 #cycles

cache_2:
  blocks: 64
  associativity: 4
  hit_time: 16

cache_3:
  blocks: 256
  associativity: 8
  hit_time: 100

mem: #required
  hit_time: 1000 #cycles
```

Fig. 2. Sample Config file used by simulator

In order to run simulator, user need to pass configuration file as YAML file, fig. 2, describing how the cache system looks like. It mainly divided in three parts:-

- Cache Architecture : Describe the architecture detail of cache and provide value to architecture knobs
- Cache Level : Describe the level of cache used in the desired system and number of cycle consumes in case of cache hit.
- Memory Attributes : Describe the number of cycle need to consume in case read/write is coming from memory.

As shown in fig. 3, After memory hierarchy creation, simulator call the respective operation method to consume required cycle.

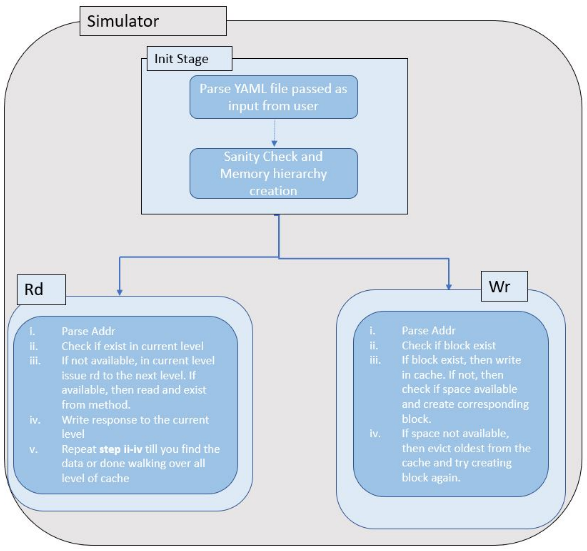

Simulator
Init Stage
Parse YAML file passed as input from user
Sanity Check and Memory hierarchy creation
Rd
i. Parse Addr
ii. Check if exist in current level
iii. If not available, in current level issue rd to the next level. If available, then read and exist from method.
iv. Write response to the current level
v. Repeat step ii-iv till you find the data or done walking over all level of cache
Wr
i. Parse Addr
ii. Check if block exist
iii. If block exist, then write in cache. If not, then check if space available and create corresponding block.
iv. If space not available, then evict oldest from the cache and try creating block again.

Fig. 3. Simulator Design overview

C. Read method

When the input sequence to the simulator is of type ”Rd”, then simulator will first check if it is in current cache level or not. If in case it is then simulator will consume hit cycle corresponding to that level and return the pointer to main thread if not then it will issue read from the next level which will iterate the same flow. Once the initiator receive the response to read request it will write it to the corresponding level and return the pointer to main thread for execution of next instruction.

D. Write method

When the input sequence to the simulator is of type ”Wr”, at first simulator check if block exists or not. If in case block exist then it will check if the write transaction for the current cache architecture supports wrtie back or not if in case it doesn’t it will simply set the dirty bit and return the pointer to main thread after consuming cycle and if in case it does then it will issue a write to the next cache level. Write method also take care of eviction of older block from the memory and flushing the full system in case of control command of type reset.

III. Dense Matrix Multiplication

In recent years with rise of usage of matrix multiplication in almost all compute and memory bound workload, there were multiple advancement on the operators in both Hardware and Software level. In the data-centeric world, most of the HPC computing involves matrices and its algebra. By definition, matrix multiplication computes a m x p matrix C from matrix A of size m x n and mat B of size n x p. And its each elements are calculated according to equation ().

$$c_{ij} = \sum_{i=1}^{N} a_{ik} b_{kj} \tag{1}$$

In this section we try to give a brief overview of different matrix multiplication kernel and try to study its characteristic from the cache performance point of view.

A. ijk algorithm

This is the most simple, and pretty straight forward implementation of multiplication. It's time complexity is O(N*M*P). In this algorithm, we iterate row-wise for matrix A and column-wise for matrix B and compute the fixed point for matrix C. The pseudo code of the algorithm is as follows:-

Algorithm 1: ijk algorithm for matrix multiplication

```
  Input: matrices A and B
1 C be a new matrix of the appropriate size
    For i from 1 to n:
    For j from 1 to p:
    Let sum = 0
    For k from 1 to m:
    Set sum ← sum + A[i][k] × B[k][j]
    Set C[i][j] ← sum
    Return C
```

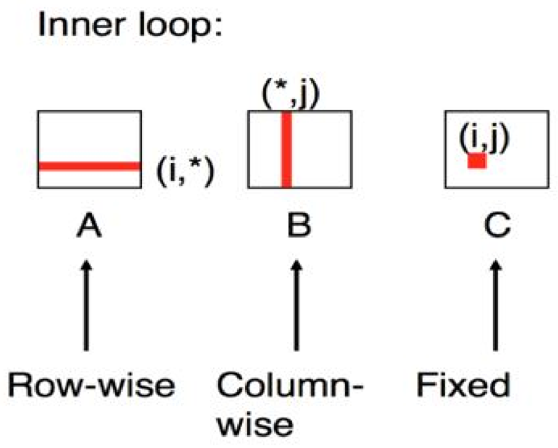


Fig. 4. Cache behaviour of ijk algorithm

Despite having asymptotic running time, the algorithm's practical time drastically change because of cache use of algorithm and change in memory access time. Based on the fig. 5, we can develop an intuition the row iteration in matrix A will cause one compulsory miss and majorly a cache hit, where as column iteration in mat B will cause 100 % miss and C being a fixed point, we can assume it to be 0% miss. Memory bottleneck always being the bottleneck, the cost of cache misses increases significantly the over all running time of the matrix multiplication.Thus, making the ijk kernel less cache friendly.

B. ikj algorithm

Algorithm 2: ikj algorithm for matrix multiplication

```
  Input: matrices A and B
1 C be a new matrix of the appropriate size
    For i from 1 to n:
    For k from 1 to p:
    Let const = A[i][k]
    For j from 1 to m:
    Set C[i][j] ← const × B[k][j]
    Return C
```

In this algorithm, we swapped the order of iteration in all the three matrices without affecting the correctness or asymptotic running time of the prior algorithm. The motivation behind swapping the iteration order is to remove column-wise iteration so that it ultimately removes the 100% misses and thus improves the cache performance of the matrix multiplication kernel. From running the kernel on our simulator, we observed

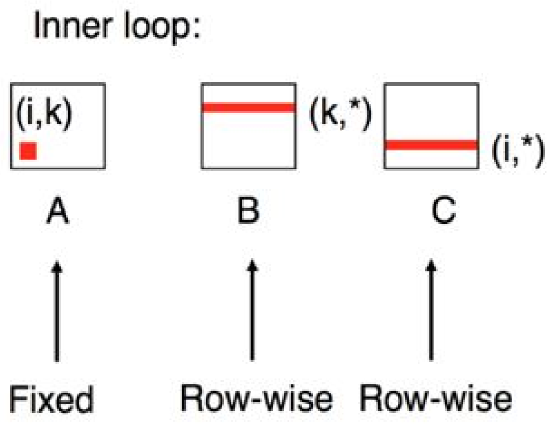
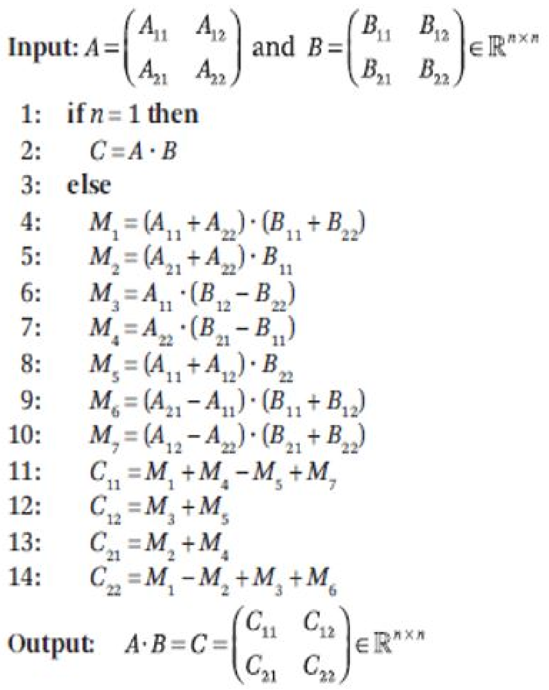


Fig. 5. Cache behaviour of ikj algorithm

that we improve the performance of any program by writing it more cache friendly manner. In case of ikj, we can infer that there will be 0% miss for matrix A as we would constantly access fixed point and just compulsory misses for matrix B and C.

C. Strassen's algorithm

After performing cache study on address traces generated from different dimension of matrices, the architecture community observed that blocking the data structure to fit in a cache, tend to reduce the memory bandwidth requirement. The whole idea of cache blocking was inspired from the classical computer algorithm of divide and conquer rule.
Strassen Algorithm [5] reduce the number of multiplication by 1 (i.e. instead of 8 multiplier, it uses only 7). In this algorithm, we divide the matrices in sub-block of 2x2 and perform the multiplication based on given psuedo-code. Initially it was estimated to perform well with matrix size of 32 and above but as Huang et. al. [5] in the modern architecture Strassen outperform standard algorithm only in case of 1000+ matrix size.



```
Input: $A=\begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}$ and $B=\begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix} \in \mathbb{R}^{n\times n}$
 1:  if $n=1$ then
 2:      $C=A\cdot B$
 3:  else
 4:      $M_1=(A_{11}+A_{22})\cdot(B_{11}+B_{22})$
 5:      $M_2=(A_{21}+A_{22})\cdot B_{11}$
 6:      $M_3=A_{11}\cdot(B_{12}-B_{22})$
 7:      $M_4=A_{22}\cdot(B_{21}-B_{11})$
 8:      $M_5=(A_{11}+A_{12})\cdot B_{22}$
 9:      $M_6=(A_{21}-A_{11})\cdot(B_{11}+B_{12})$
10:      $M_7=(A_{12}-A_{22})\cdot(B_{21}+B_{22})$
11:      $C_{11}=M_1+M_4-M_5+M_7$
12:      $C_{12}=M_3+M_5$
13:      $C_{21}=M_2+M_4$
14:      $C_{22}=M_1-M_2+M_3+M_6$
Output:  $A\cdot B=C=\begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix} \in \mathbb{R}^{n\times n}$
```

Fig. 6. Pseudo code of strassen algorithm

The time complexity of the strassen algorithm is given by recurrence. Theoretically, it should increase the performance as it translate the matrix multiplication from cubic time to quadratic time.

$$T(1) = \Theta(1)$$

$$T(n) = 8T(n/2) + \Theta(n^2)T(n) = 8T(n/2) + \Theta(n^2)$$

IV. Sparse Matrix Multiplication

After analyzing the recent workloads, it is observed that matrices involved in recommendation systems and graph analytics are more sparse in nature. Thus, multiple literature suggested different storage mechanism. Coordinate Format (COO), Compressed Sparse Row Format (CSR), Diagonal Format (DIA), ELLPACK Format (ELL) [8] shares the characteristics of encoding non-zero values along with their positions.

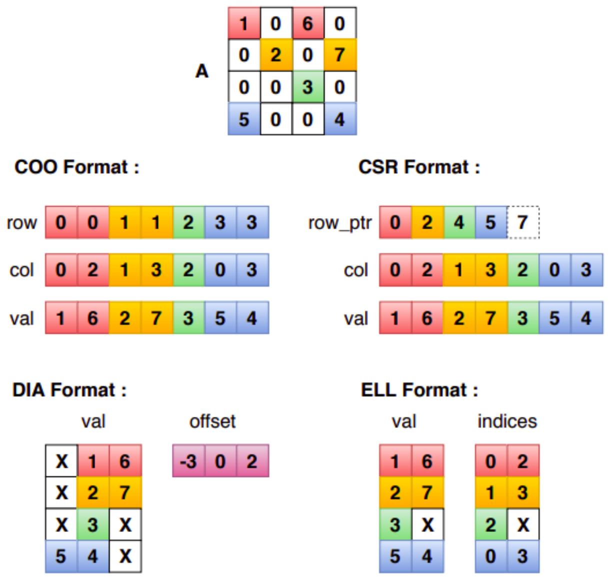


Fig. 7. various storage format for sparse

In COO format, we tend to store value and it's co-ordinates in 3 1-D vector. It is very easy to construct and provides considerable compression ratio and random access time. where as ELI and DIA are special format which can not be generalized to any sparse matrix.

A. Compressed Sparse Row Format(CSR)

It is the state of the art storage format for sparse matrix and it massively used in Intel MKL, TACO libraries for sparse matrix computation. It consists of three 1D array i.e., row_ptr, col_ind and values to store non-zero elements of the sparse matrix.

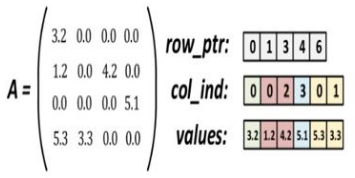


Fig. 8. visual representation of CSR format

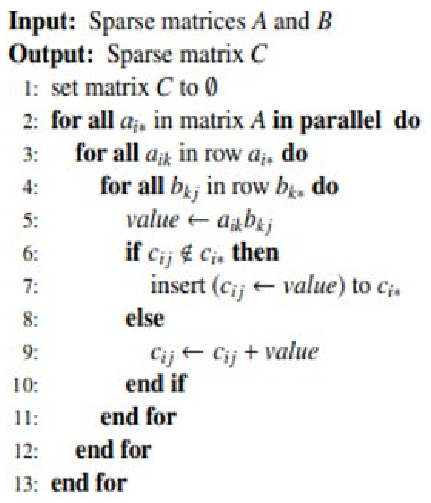
The row_ptr stores accumulative number of non-zero elements instead of row indices thus provide higher compression ratio than COO with similar traversal time.

B. Sparse Matrix Multiplication (SpGEMM)

As a first step, in SpGEMM [3] we find all the nnz entry of matrix A and B. And then row of output matrix is constructed

Algorithm 3: Psuedo code for CSR traversal

```
Input: matrices A and B
1 C be a new matrix of the appropriate size
  For i from 1 to size_of(row_ptr):
  Set num_elements_corr_row = Abs(row_ptr[i] -
  row_ptr[i+1]) For j from 1 to
  num_elements_corr_row:
  Set row_index = i
  Set col_index = j
```



```
Input: Sparse matrices A and B
Output: Sparse matrix C
1: set matrix $C$ to $\emptyset$
2: for all $a_{i*}$ in matrix $A$ in parallel do
3:   for all $a_{ik}$ in row $a_{i*}$ do
4:     for all $b_{kj}$ in row $b_{k*}$ do
5:       $value \leftarrow a_{ik}b_{kj}$
6:       if $c_{ij} \notin c_{i*}$ then
7:         insert ($c_{ij} \leftarrow value$) to $c_{i*}$
8:       else
9:         $c_{ij} \leftarrow c_{ij} + value$
10:      end if
11:    end for
12:  end for
13: end for
```

Fig. 9. Psuedo code for SpGEMM

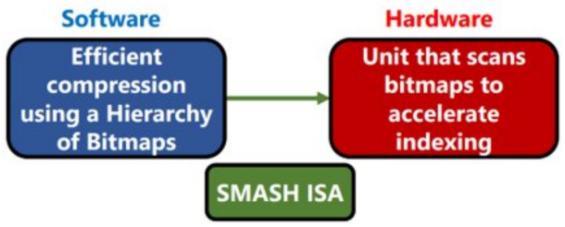
by accumulating the partial output from each iteration as shown in fig. 9.

Any matrix operator works on three 1D array that means processor need to perform multiple instruction to find the position of non-zero element in the matrix. Also, these instructions are dependent on each other i.e we need to find row and column before access element from the original matrix. Konstantinos et. al., proposed a new method [6] to store and operate on sparse matrix. It is HW/SW co-operative mechanism. His mechanism is divided in two parts:

1) Software – Aims at preparing hierarchy bit map to store the information about non-zero elements of the matrix.
2) Hardware (BMU)- Develop a HW block to accelerate the process of interpreting the bitmap and sends the position of non-zero element to CPU for further operation.

V. SMASH

Konstantinos et. al. in MICRO 2019 proposed SMASH [6] which efficiently compresses the sparse matrix to one 1D-vector and few extra bits for bitmap hierarchy. It delivers higher compression ratio than CSR and also leverage special hardware unit to accelerate the indexing of the element and speedups the multiplication operation.



Software
Efficient compression using a Hierarchy of Bitmaps
Hardware
Unit that scans bitmaps to accelerate indexing
SMASH ISA

Fig. 10. SMASH architecture

A. Software Compression Scheme

User defines the compression ratio for each bitmap level and SMASH software converts the CSR format to SMASH specific format accordingly. It's software compression scheme retains the value vector of CSR and iterate through each row to form the bitmap level. For example as mentioned in figure 13, SMASH SW creates a bitmap for each row as level 0 and further compresses it by 4:1 ratio, by using 1 bit to represents each row. And it is further compressed by applying 2:1 compression ration on level 2.

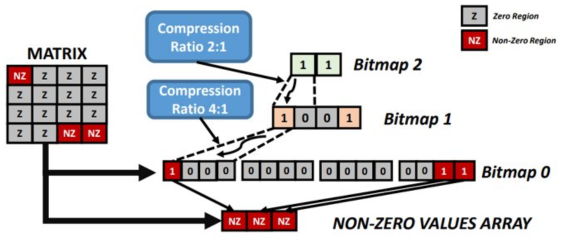


Fig. 11. Software compression scheme in SMASH

B. Bitmap Management Unit (BMU)

Users configures the parameters and stores the bitmap hierarchy in SRAM buffer using SMASH ISA. Once BMU is configured, the HW scans the buffer and update the row and column index accordingly for each group. And upon request from CPU transfer the element value by accessing the memory hierarchy. Thus accelerates the indexing mechanism along with achieving higher compression ratio.

VI. Results

In our research project, We performed a cache study for all the matrix multiplication kernel mentioned in the above sections. For simulator sanity purpose we analyzed the performance for different dense matrix multiplications method.And then finally compared the performance of CSR based as well as SMASH based multiplication. Our work also helped us validating the claim that strassen algorithm outperform only in case of 1000+ matrix dimension [5].

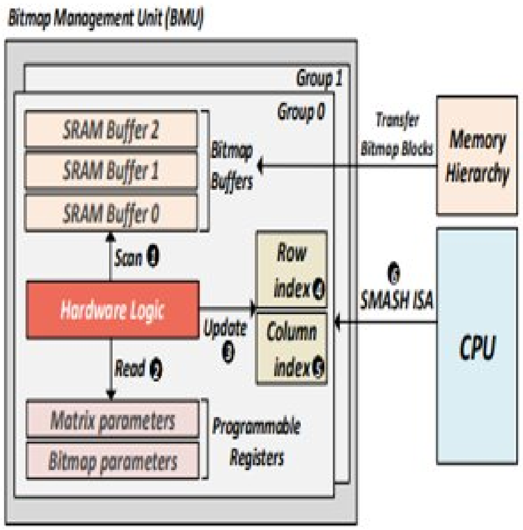


Fig. 12. Bitmap Management Unit (BMU)

A. 10x10 matrix with different sparsity

We tried different kernels with 10x10 matrix with sparsity 10%. Based upon on our analysis, our simulation results also showed the similar output, ijk being the least cache friendly and SMASH speeding up the operator by accelerating the indexing process.

MatMul 10x10 with 10% sparsity				
kernel	L1 Hit %	L2 Hit %	L3 Hit %	Cycles(in millions)
ijk algo	75.56	65	78	11.9
ikj algo	85	81	64.94	1.8
Boost lib	81	96	77.35	4.5
Strassen	86	84	76	4.9
SpGEMM	85	50	42	1.7
SMASH	73.79	73.3	46.3	1.06

$Table1 : MatMul10x10with10\%sparsity$

MatMul 10x10 with 50% sparsity				
kernel	L1 Hit %	L2 Hit %	L3 Hit %	Cycles(in millions)
ijk algo	75	65.8	77.3	12.1
ikj algo	85	80	65	1.8
Boost lib	80.63	96.7	74.84	4.1
Strassen	86.8	84	75.36	4.9
SpGEMM	85	51.1	42	1.08
SMASH	73.1	75.38	43.98	1.07

$Table2 : MatMul10x10with50\%sparsity$

Despite SMASH is delivering speed-up in comparison to ijk, we observe there is a scope of improvement by understanding the memory access pattern or critical word/early restart. As sparse matrix generally doesn't showcase spatial locality we can improve the performance by reducing miss latencies instead of miss rate.

MatMul 10x10 with 80% sparsity				
kernel	L1 Hit %	L2 Hit %	L3 Hit %	Cycles(in millions)
ijk algo	75.3	64.83	78.73	11.9
ikj algo	85.4	80.19	64.54	1.8
Boost lib	80.87	96.2	74.84	4.1
Strassen	86.78	83.46	76	4.9
SpGEMM	83.08	50.65	42.71	1.06
SMASH	73.77	73.4	43.4	1.06

$Table3 : MatMul10x10with80\%sparsity$

B. 50x50 matrix with different sparsity

First set of experiments helped us gain confidence in the simulation technique and also helped us sanitize our model. In the next set of experiments we tried our simulator with higher size of matrices.

MatMul 50x50: with 10% sparsity				
kernel	L1 Hit %	L2 Hit %	L3 Hit %	Cycles(in millions)
SpGEMM	96.82	94.30	55.42	4.2
SMASH	80.92	97.64	51.76	2.9
MatMul 50x50: with 10% sparsity				
kernel	L1 Hit %	L2 Hit %	L3 Hit %	Cycles(in millions)
SpGEMM	96.6585	95.23	51.49	4.3
SMASH	81.60	93.89	73.28	2.9
MatMul 50x50: with 80% sparsity				
kernel	L1 Hit %	L2 Hit %	L3 Hit %	Cycles(in millions)
SpGEMM	96.59	97.18	45.46	4.2
SMASH	81.7	97.57	47.83	2.7

$Table4 : MatMul50x50withdifferentsparsity$

After comparing CSR based SPGEMM and SMASH based SPGEMM we came to a conclusion that we gain significant speedup with SMASH in memory path. SMASH provides significant boost in storage efficiency but and gain in speedup in simple cache architecture.

C. 100x100 matrix with different sparsity

As a third set of experiment, we tried both sparse matrix kernel with 100x100 matrix. We observed that speed up increases drastically when with increase in dimension. Our simulator also helped us in concreting our belief that we can stil improve the cache performance by minimizing the miss latencies instead of miss rate as there is less spatial locality in higher dimensions.

MatMul 100x100: with 10% sparsity				
kernel	L1 Hit %	L2 Hit %	L3 Hit %	Cycles(in millions)
SpGEMM	96.85	97.35	72.2	23.06
SMASH	86.15	90.42	96.75	23.57
MatMul 100x100: with 10% sparsity				
kernel	L1 Hit %	L2 Hit %	L3 Hit %	Cycles(in millions)
SpGEMM	96.98	97.05	75.5	23.06
SMASH	86.95	91	95	15.56
MatMul 100x100: with 80% sparsity				
kernel	L1 Hit %	L2 Hit %	L3 Hit %	Cycles(in millions)
SpGEMM	96.24	97.24	68.5	23.06
SMASH	86.71	90.6	92	7.85

$Table5 : MatMul100x100withdifferentsparsity$

VII. Future Works

In future we would like to make simulator more robust by adding support of passing trace sequence on-the-fly as well as adding modes to support execution based simulation. We would also like to implement features like bag of word in our cache so that user can get access to advance feature. We would also like to work on profiling and increasing speed for our simulator and analyse it with respect to other cache only simulation infra.

Acknowledgements

We would like to thank Dr. John Sampson for their continuous support and guidance. Also we would like to thank Machine Design Lab, Penn State for the sharing the resources need for making development.

References

[1] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The gem5 simulator," *SIGARCH Comput. Archit. News*, vol. 39, no. 2, p. 1–7, Aug. 2011. [Online]. Available: https://doi.org/10.1145/2024716.2024718

[2] H. Brais, R. Kalayappan, and P. R. Panda, "A survey of cache simulators," *ACM Comput. Surv.*, vol. 53, no. 1, Feb. 2020. [Online]. Available: https://doi.org/10.1145/3372393

[3] F. G. Gustavson, "Two fast algorithms for sparse matrices: Multiplication and permuted transposition," *ACM Trans. Math. Softw.*, vol. 4, no. 3, p. 250–269, Sep. 1978. [Online]. Available: https://doi.org/10.1145/355791.355796

[4] K. Hazelwood, in *Dynamic Binary Modification: Tools, Techniques and Applications*, 2011.

[5] J. Huang, T. M. Smith, G. M. Henry, and R. A. Van De Geijn, "Strassen's algorithm reloaded," in *SC '16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2016, pp. 690–701.

[6] K. Kanellopoulos, N. Vijaykumar, C. Giannoula, R. Azizi, S. Koppula, N. M. Ghiasi, T. Shahroodi, J. G. Luna, and O. Mutlu, "Smash: Co-designing software compression and hardware-accelerated indexing for efficient sparse matrix operations," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '52. New York, NY, USA: Association for Computing Machinery, 2019, p. 600–614. [Online]. Available: https://doi.org/10.1145/3352460.3358286

[7] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: Building customized program analysis tools with dynamic instrumentation," *SIGPLAN Not.*, vol. 40, no. 6, p. 190–200, Jun. 2005. [Online]. Available: https://doi.org/10.1145/1064978.1065034

[8] Y. Saad, "Sparskit: a basic tool kit for sparse matrix computations," 1994.

[9] D. Sanchez and C. Kozyrakis, "Zsim: Fast and accurate microarchitectural simulation of thousand-core systems," in *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ser. ISCA '13. New York, NY, USA: Association for Computing Machinery, 2013, p. 475–486. [Online]. Available: https://doi.org/10.1145/2485922.2485963