TOWARDS REALISTIC PERFORMANCE FOR ITERATIVE METHODS ON SHARED MEMORY MACHINES[*]

SHENGXIN ZHU[†]

Abstract. The paper proposes a random linear model to investigate the memory bandwidth barrier effect on current shared memory computers. Based on the fact that floating-point operations can be hidden by implicit compiling techniques, the runtime for sparse matrix vector multiplications and other memory intensive applications can be modelled by memory reference time plus a random term. The random term due to cache conflicts, data reuse and other environmental factors is proportional to memory reference volume. The model is verified and validated by comparing the performance of sparse matrix vector multiplications in different formats. Various numerical results based on thousands representative matrices are presented, compared, analysed and validated to confirm the proposed model. The model shows that a realistic and fair metric for performance of iterative methods and other memory intensive applications should consider the memory bandwidth capability and memory efficiency.

Key words. Memory bandwidth, memory intensive applications, performance evaluation.

AMS subject classifications. 65F10, 65Y99, 68Q99, 68W40.

1. Introduction.

A new benchmark, High Performance Conjugate Gradient (HPCG), was introduced recently for the Top500 list and the Green500 list [4]. It motivates us to improve the performance of the widely used iterative methods in many real applications on all kinds of computing platforms. In particular, as multi-core techniques and accelerators are becoming more accessible and more universal, more people will pay closer attention to the performance of iterative methods on shared memory machines. It was noticed that the realistic performance of iterative methods on high performance computers was limited by memory bandwidth decades ago [6], when high performance computing was limited to a fraction of people. Memory bandwidth bottleneck effect for performance of iterative methods and other *memory intensive applications* still remains on modern chips and becomes increasing significant [1, 7, 10]. Here proposes a random linear model to describe the phenomena and unravel the underling “mystery”. Presented experimental design is closely related to iterative methods: comparing the performance of symmetric sparse matrix vector multiplications(SpMVs) in different formats. Algorithms for symmetric SpMVs in compressed-column(CSC), compressed row(CSR), and compressed packed form are investigated. It turns out that the performance of SpMVs significantly linearly depends on the memory reference volume, its realistic performance in the classical floating-point operations(flops) metric is limited by the underlying memory bandwidth capability, and time for flops almost can be neglected. Various numerical results are processed, analysed, validated and presented in a convincing way so that everyone who reads the results and hasn’t yet pay enough attention to such a phenomena, will notice that a realistic performance of iterative methods should be paid attention to. The remainder of the paper is organized as follows: §2 proposes the model to evaluate the performance, and details the difference of the presented algorithms. Numerical verifications and validation are presented in §3. Finally presents some discussion.

2. Model and Performance Analysis.

The time for matrix vector multiplications was assumed to be the sum of time for computation and time for memory reference(or data

[*]The research is supported by Award no KUK-C1-013-04, made by King Abdullah University of Science of Technology. Part of the work was supported by a short knowledge transfer project(EPSRC-sKTP100852).

[†]Oxford Center for Collaborative and Applied Mathematics & Numerical Analysis Group, Mathematical Institute, The University of Oxford. Andrew Wiles Building, Radcliffe Observatory Quarter, Woodstock Road, Oxford, OX2 6GG United Kingdom. (Email: zhus@maths.ox.ac.uk).

```
      for (col = 0;col<n;++col)
      {   for(prow = Ap[col];prow <Ap[col+1];++prow)
              y[Ai[prow]]+=Ax[prow]*x[col] ;
      } //  stand sparse matrix multiplication code
```

Algorithm 1: A C/C++ snippet for sparse matrix vector multiplication in compressed column(CSC) format, where `Ap,Ai,Ax` are the column pointers, row index, elements for the sparse matrix.

```
      for (row = 0;row<n;++row)
      {   for(pcol = Ap[row];p <Ap[row+1];++pcol)
              y[row]+=Ax[pcol]*x[Aj[pcol]] ;
      } //  stand sparse matrix multiplication code
```

Algorithm 2: A C/C++ snippet for sparse matrix vector multiplication in compressed row(CSR) format, where `Ap,Aj,Ax` are the column pointers, column index, elements for the sparse matrix.

```
     for (col=0; col<n;++col)    //  note n=A->n-1
      {
         if(Ap[col]==Ap[col+1]) continue; // skip empty columns
         tempx=x[col];                    //reset yu(col)=0;
         temp=x[Ai[Ap[col]]]*Ax[Ap[col]]; // yu part
         if (Ai[Ap[col]]!=col)            // lower part yl part
            y[Ai[Ap[col]]] +=tempx*Ax[Ap[col]];
         for(prow=Ap[col]+1;prow<Ap[col+1];++prow)
           {
              temp+=Ax[prow]*x[Ai[prow]];       // yu part
              y[Ai[prow]]+=tempx*Ax[prow];      // yl part
           }
         y[col]+=temp;                        // yl+yu
      }
      if(Ap[n]!=Ap[n+1]) y[n]+=x[n]*Ax[Ap[n]]; // A(end,end) !=0
  //update yu part to y
```

Algorithm 3: A C/C++ snippet for symmetric sparse matrix multiplication in packed format. The matrix A is split into three three part $A = L_A + D_A + U_A$, $y_\ell = L_A x$ and $y_u = (D_A + L_A^T)x$.

movement). For memory intensive kernels like SpMVs with low ratio of flops per memory reference, the time for flops can be neglected, because modern compilers are intelligent enough to hid most of the flops by *pipeline* and *pre-fetching* techniques. Here we assume the total time is dominated by the memory reference time and neglect the time for computations. Hierarchy memory architecture is simplified to be only two levels, the fast memory where computations take place and the slow memory where the data is. A simple linear model for the relationship between the observed runtime T and the memory reference volume is proposed as follows,

$$T = t_m M + \epsilon \qquad (1)$$

where t_m is the time required for unit memory reference, M is the total memory reference volume, ϵ is a random noise due to cache misses, data reuse and other possible environmental factors. Since both data conflicts and data reuse are proportional to M, thus the random term ϵ should be proportional to M. The key point is to assume that each unknown noise data satisfies

$$\epsilon_i/M_i = T_i/M_i - t_m \sim \mathcal{N}(0, \sigma^2), \tag{2}$$

where $\mathcal{N}(0,\sigma^2)$ is a random normal distribution. In this way the observed data $\frac{T_i}{M_i}$ is a random normal distribution $\mathcal{N}(t_m, \sigma^2)$, where t_m and σ^2 are unknown parameters to be quantified.

2.1. Parameter estimation. Since the uncertainty term ϵ comes in, an individual result makes little sense and more test results produce less biased estimation. Here uses various numerical results to estimate the parameter t_m and the uncertainty σ^2. Apply the maximum likelihood estimation to the transformed model (2), we obtain the *best linear unbiased estimator* for t_m, which will be denoted as $\bar{t}_m = E(\frac{T_i}{M_i})$; unbiased refers to the expectation of the random variable $\bar{t}_m$ satisfies $E(\bar{t}_m) = t_m$. Let N be the number of observed results, the unbiased sample variance is used as the estimation of σ^2, i.e. $\hat{\sigma}^2 = \sum_{i=1}^N (T_i/M_i - \bar{t}_m)^2/(N-1)$, which can be computed by the Matlab function `var(·,0)` or `var(·)`, or the estimation of the unbiased *standard derivation* $\hat{\sigma}$ can be computed by the Matlab function `std(·)` or `std(·,0)`. The estimation of the *standard error* of the mean is $\hat{se} = \frac{\hat{\sigma}}{\sqrt{N}}$. For a 95 percent confident interval for the estimation of $\bar{t}_m$ is $[\bar{t}_m - 2\hat{se}, \bar{t}_m + 2\hat{se}]$ [11, p.100]. The memory reference volume M for the presented algorithms is listed in Table 1, where nnz_A is the number of non-zero elements of a $n \times n$ matrix A, for the packed form, the diagonal elements of the matrix A are assumed to be non-zeros.

Table 1: Memory reference volume for the three algorithms

Method	Alg1: CSC	Alg2 :CSR	Alg3: packed form
M	$28nnz_A + 12n$	$20nnz_A + 20n$	$14nnz_A + 18n$

2.2. Performance comparison. If there is no random effect, the speed-up of the packed form over the CSC format is

$$S_{3,1}^{\text{ideal}} = \frac{T^{\text{Alg1}}}{T^{\text{Alg3}}} = \frac{(28nnz_A + 12n)t_m}{(14nnz_A + 18n)t_m} = \frac{14nz_A + 6}{7nz_A + 9} = 2 - \frac{12}{7nz_A + 9}, \tag{3}$$

where nz_A is the average number of non-zero elements per column. With the assumption that all the diagonal elements of A are non-zero, then $nz_A \geq 1$ and $1.25 \leq S_{3,1}^{\text{ideal}} < 2$. Similarly the speed-up of the compressed row format over the compressed column format is

$$S_{2,1}^{\text{ideal}} = \frac{T^{\text{Alg1}}}{T^{\text{Alg2}}} = \frac{(28nnz_A + 12n)t_m}{(20nnz_A + 20n)t_m} = \frac{7nz_A + 3}{5nz_A + 5} = \frac{7}{5} - \frac{4}{5nz_A + 5}. \tag{4}$$

Here comes the delicate part of comparing performance of two methods based on the model in(1), the speed-up is $S_{b,a} = \frac{T^a}{T^b} = \frac{t_m M^a + \epsilon^a}{t_m M^b + \epsilon^b}$. Because the two different random distributions are correlated with the different memory reference volume M^i, $i = a, b$, thus an individual observed speed-up can be far away from the expected interval, for $S_{3,1}$, the interval is $[1.25, 2)$. A trick used here to reduce the the large random noise is viewed $\log S_{b,a} = \log(\frac{T^a}{T^b})$ as a random variable, or fit the model

$$\log(T^a) = \log(T^b) + \log S_{b,a} + \epsilon. \tag{5}$$

Table 2: Ideal speed-up for typical sparse matrices

nz_A	3	5	7	9	15.6	27
$S_{3,1}^{\text{ideal}}$	1.6	1.73	1.79	1.83	1.90	1.94
$S_{2,1}^{\text{ideal}}$	1.2	1.27	1.3	1.32	1.35	1.37

The expected speed-up is then $S_{b,a} = \exp(E(\log(T^a) - \log(T^b)))$.

Two efficiency are defined, the first is defined as the traditional metric in flops count.

$$\eta_{flops} = T_{flops}/T = 2t_f n nz_A/T, \tag{6}$$

where t_f is the theoretical time for one flop. If the observed time T is linearly correlated with the memory reference volume, then $E(\eta_{flops}^{\text{Alg3}}/\eta_{flops}^{\text{Alg1}}) = E(T^{\text{Alg1}}/T^{\text{Alg3}}) = E(S)$. Similarly the memory efficiency is defined as

$$\eta_m = T_M/T = t_m^t M/T = t_m^t M/(\,\bar{t}_m M + \epsilon), \tag{7}$$

where t_m^t is the theoretical time for unit memory reference, when t_m^t is close to $\bar{t}_m$, the efficiency may be bigger than one, for example, when the random term ϵ corresponds to large proportion of data reuse. With additional computation, it follows that $E(\eta_m^{\text{Alg3}}/\eta_m^{\text{Alg1}}) \approx 1$.

3. Numerical verification and validation. Sequential algorithms are implemented so that the results can be easily repeated.

3.1. Hardware specification. Numerical examples are carried on an AMD Phenom II x4 925 CPU. For one single core, the frequency of the CPU is 2.8 Ghz, which results in a $2.8 \times 4 = 11.2$ G flops per seconds, where 4 is the number of instructions for each clock cycle. The theoretical time for one floating-point operation is $t_f = 10^{-9}/11.2 = 8.9 \times 10^{-11}$ seconds (or 89 *picoseconds*). The theoretical memory bandwidth for one core is 21.3/4 GB/s, thus theoretical time for unit memory reference is $t_m^t = 4 \times 10^{-9}/21.3 = 1.88 \times 10^{-10}$(188 picoseconds). It has a 4×512 KB L2 cache and 6MB L3 cache.

3.2. Test matrices. The first group of test matrices consists of 996 symmetric sparse matrices from University of Florida(UF) sparse matrix collection [3]. Various of types of matrices are included so that the sample matrices are representative. The test matrices are obtained with the `UFget` function in the CXSparse package [2]. Here is a Matlab snippet to select the 1001 symmetric matrices.

```
idex=UFget;               % 2650 total test matrices
sym1=find(idex.numerical_symmetry==1.0); % 1004 symmetric matrices
sym2=find(idex.isReal(sym1));      % 1001 real symmetric matrices
sym=sym1(sym2);           % UF id of the 1001 chosen matrice
```

5 Out of the 1001 chosen symmetric matrices are skipped due to the size is too big, the UF id of the skipped matrices are `sym([698,699,700,880,883])`. Besides, 300 Wathen matrices are generated by the Matlab function `A=gallery('wathen',k*nx,k*ny)`, where `k`$= 1, 2, \ldots, 300$, `nx=2` and `ny=3`. These are sparse, random, $n \times n$ finite element matrices where $n = 3\mathtt{k}^2 \cdot \mathtt{nx} \cdot \mathtt{ny} + 2\mathtt{k} \cdot \mathtt{nx} + 2\mathtt{k} \cdot \mathtt{ny} + 1$. These matrices share the same structure and satisfy the assumption that all the diagonal elements are non-zeros.

3.3. Implementation details. Algorithms are implemented in C/C++ and compiled with `gcc 4.4`. The maximum speed compiling flag `-O2` is opened. A Matlab wrapper function is used so that all the experiments are carried out in Matlab. High resolution timer for C/C++

Table 3: STREAM benchmark results

Function	Rate(MB/s)	t_m(ps)	η_m(%)
Copy	4741.5031	210	89
Scale	4459.9498	224	84
Add	5052.1102	198	95
Triad	4937.1375	202	93

Table 4: Estimations in this paper

	UF collections		Wathen matrices	
	$\bar{t}_m$(ps)	η_m(%)	$\bar{t}_m$(ps)	η_m(%)
Lob	196	96	158	119
Avg	221	85	160	117
Upb	245	78	163	115

computation kernels is used so that the overhead due to timing is minimized. The shortest time in the testing results is 0.525 microseconds, whereas `tic;toc` in Matlab has about 5 microseconds overhead in the author's machine. When fitting parameters, those matrices for which the memory reference volume in Algorithm 1 are smaller than 10^3 bytes or larger than 10^9 bytes are excluded, because timing results for extremely small matrices are less accurate and timing results for large matrices tend to be longer than expected due to the physical memory constraint which can result in more cache waiting.

3.4. Validate results. Numerical results are illustrated in Figure 1 to Figure 7, details are described in the corresponding captions. Results are validated in three ways. First, for any fitted parameters, a 95% confidence interval is supplied so that comparison is more reasonable when considering the random effect. Second, we compare the fitted speed-up of Algorithm 3 over Algorithm 1 with the ideal speed-up computed by (3), see Figure 3 to Figure 6. The Wathen matrices satisfy all the assumptions for (3), and thus the fitted speed-up matches the ideal speed-up almost perfectly. Finally and most convincingly, Table 3 and Table 4 compare the memory efficiency η_m and the fitted expected unit memory reference time $\bar{t}_m$ obtained in this paper with that from the STREAM benchmark, an established synthetic benchmark program that measures sustainable memory bandwidth for simple vector kernels [9]. Table 4 list the 95% percentage confidence interval for the estimation of $\bar{t}_m$ and η_m, where the lower bound(Lob), upper bound(Upb) and the expected value (Avg) for $\bar{t}_m$ are listed.

4. Discussion. Presented results show that the runtime for SpMVs is dominated by memory reference volume and can be modelled by memory reference time plus a random effect. The real hierarchical memory architecture is much more complicated than the simplification here, whereas the present results show the proposed model is reasonable to some extent and validated by various results and comparisons. A key point for the successful simplification is based on the reasonable assumption that the uncertainty ϵ in (1) is proportional to the memory reference volume M, without realizing this point and fitting the observed results directly does not result in satisfactory results.

Figure 3 and Figure 4 show that algorithms with different flop efficiency share almost the same memory efficiency. In particularly, for the Wathen matrices which satisfies all the assumptions, the ratio of the flop efficiency equals to the reciprocal of memory reference volume. This can show the performance of SpMVs in the classical flops metric is limited by the memory bandwidth. In this paper the memory efficiency achieves as high as 78% to 120%, while the traditional flops efficiency can only reach to a range from 7% to 13%, a range similar to the range of ratio of flops per memory reference $(1/14, 1/7)$. In contrast, the runtime for dense matrix matrix kernels and other computation intensive applications with high ratio of flops per memory reference can be modelled by the amount of floating-point operations [5, 8]. Because there are huge disparities between the memory intensive applications and the computation intensive applications, using traditional benchmark based on flops for memory intensive applications is unfair and not valid, therefore, the HPCG benchmark appears [4]. A realistic performance for sparse iterative methods on shared memory computing platforms should consider the memory bandwidth capability; a fair and reasonable metric for performance of sparse iterative methods on shared memory machines should based on the memory efficiency rather than the flops efficiency. For performance of sparse iterative methods on distributed computing system, situation becomes more complicated because of the global communications and synchronizations [12]. In that case, priorities are given to avoiding or minimizing synchronizations. Investigation on performance of sparse iterative methods on distributed computers with up to thousands computing nodes can be find in recent paper [13].

Without any difficulty, one can show that the memory reference consumes far more energy than flops do. Because the energy W, is proportional to the time for the underlying operations and the underlying power of the CPU, P, precisely, $W = Pt$. In this paper, 1 floating-point operation takes about 89 picoseconds, while one unit memory reference takes about 188 picoseconds. More important, computations and data reference can be concurrency. Thus memory efficiency algorithms are in fact energy efficient algorithms. Efficient memory management will be challenging for iterative methods and other memory intensive applications on current and future shared memory computing platform.

REFERENCES

[1] Shekhar Borkar and Andrew A Chien, *The future of microprocessors*, Communications of the ACM, 54 (2011), pp. 67–77.

[2] Timothy A Davis, *Direct Methods for Sparse Linear Systems (Fundamentals of Algorithms 2)*, Society for Industrial and Applied Mathematics, 2006.

[3] Timothy A Davis and Yifan Hu, *The university of florida sparse matrix collection*, ACM Transactions on Mathematical Software (TOMS), 38 (2011), p. 1.

[4] Jack Dongarra and Michael A. Heroux, *Toward a new metric for ranking high performance computing systems*, tech. report, Sandia National Laboratories, June 2013. SAND2013-4744.

[5] NICHOLAS J HIGHAM GIMA, *Matrix computations in BASIC on a microcomputer*, IMA Bulletin, (1986), pp. 13–20.

[6] WD Gropp, DK Kaushik, DE Keyes, and BF Smith, *Toward realistic performance bounds for implicit CFD codes*, in Proceedings of parallel CFD, vol. 99, 1999, pp. 233–240.

[7] Peter Kogge, Keren Bergman, Shekhar Borkar, Dan Campbell, W Carson, William Dally, Monty Denneau, Paul Franzon, William Harrod, Kerry Hill, et al., *Exascale computing study: Technology challenges in achieving exascale systems*, (2008).

[8] Piotr Luszczek, Jakub Kurzak, and Jack Dongarra, *Looking back at dense linear algebra software*, Journal of Parallel and Distributed Computing, (2013), pp. –.

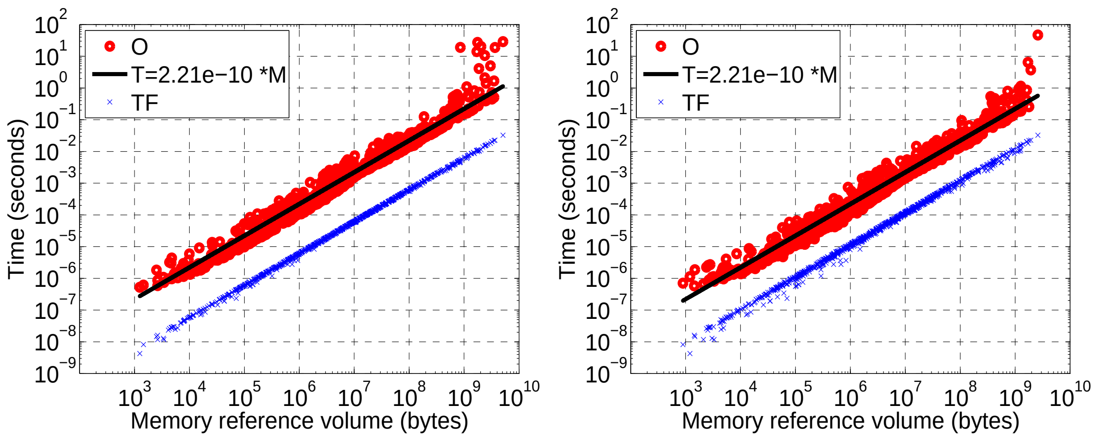
[9] John D. McCalpin, *STREAM: Sustainable memory bandwidth in high performance computers*, tech. report, University of Virginia, Charlottesville, Virginia, 1991-2007. A continually updated technical report. http://www.cs.virginia.edu/stream/.

[10] John Shalf, Sudip Dosanjh, and John Morrison, *Exascale computing technology challenges*, in High Performance Computing for Computational Science–VECPAR 2010, Springer, 2011, pp. 1–25.

[11] Larry Wasserman, *All of statistics: a concise course in statistical inference*, Springer, 2004.

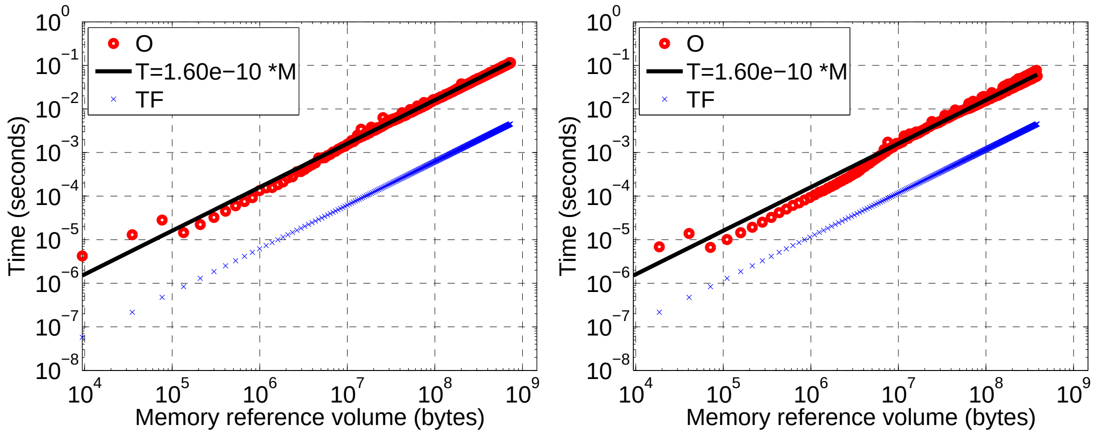
[12] Sheng-Xin Zhu, *Small dots, big challenges*, tech. report, April, 2013. Abstract for ExaMath13 Workshop, https://collab.mcs.anl.gov/display/examath/Submitted+Papers; an updated slide: Small dots, big challenges: on the new benchmark of top500 and Green500. Nov, 2013. http://people.maths.ox.ac.uk/ zhus/pub/Smalldot.pdf, http://www.maths.ox.ac.uk/node/23741.

[13] Sheng-Xin Zhu, Tong-Xiang Gu, and Xing-Ping Liu, *Minimizing synchronizations in sparse iterative solvers for distributed supercomputers*, Computers & Mathematics with Applications, 67 (2014), pp. 199 – 209.



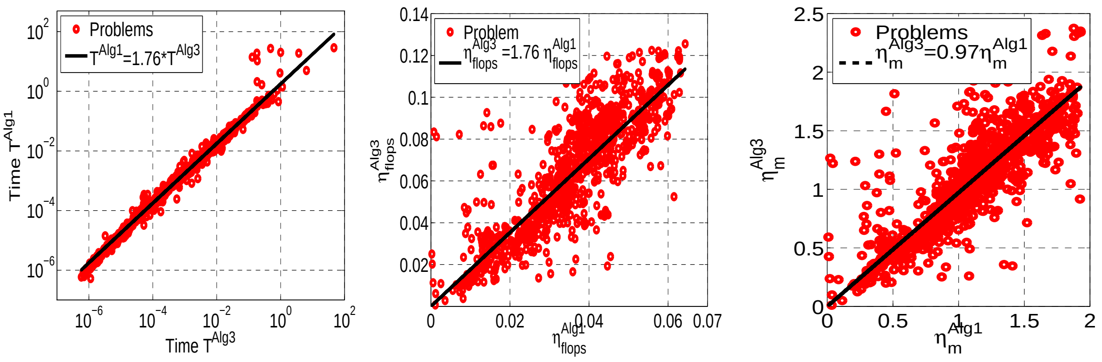
(a) Algorithm 1 (b) Algorithm 3

Fig. 1: Results for the 996 matrices from UF sparse matrix collections. In both panels, the red circles(O) are observed timing results, the blue $\times$ are theoretical floating-point operations time(TF), $2nnz_A t_f$, and the black solid line(T) are fitted results. In both figures all the 996 problems are plotted. The coefficient 2.21×10^{-10} (221 picoseconds) is the expected time, $\bar{t}_m$, for unit memory reference. A 95% confidence interval for $\bar{t}_m$ is $(196, 245)$ picoseconds. The theoretical unit memory reference time is $t_m^t = 188$ picoseconds.



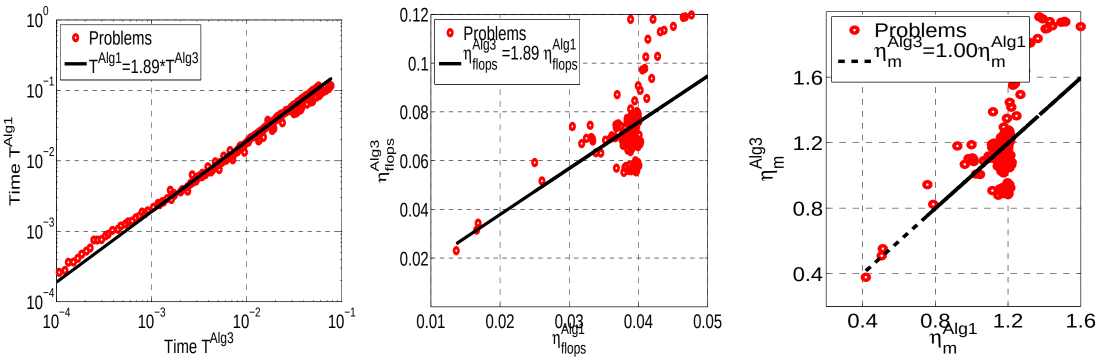
(a) Algorithm 1 (b) Algorithm 3

Fig. 2: Results for the 300 Wathen matrices. The legends are the same as those in Fig. 1. The 300 Wathen matrices share the same structure and thus observed timing results are less divergent away from the fitted line. The 95% confidence interval for the fitted value $\bar{t}_m$ is $(158, 163)$ picoseconds, which results in a memory efficiency between 115% to 119%. This is likely due to the nice block dense structure of the underlying matrices(see Fig 6b)which may result in a higher cache hitting rate than usual.



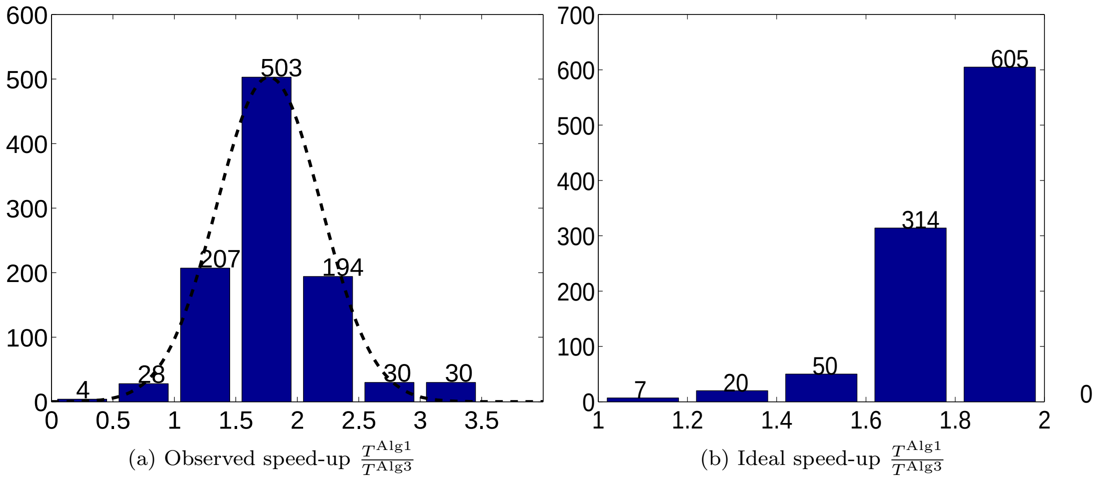
(a) Observed time (b) Flops efficiency (c) Memory efficiency

Fig. 3: Comparison between Algorithm 1 and Algorithm 2 for the 996 matrices from UF sparse matrices collection. In (a) and (b) the fitted coefficient is the expected speed-up of Algorithm 3 over Algorithm 1. A 95% confidence interval for the expected speed-up is $(1.73, 1.80)$. The memory efficiency of Algorithm 3 is slightly worse than Algorithm 1, this is likely because there are two `if` branches in Algorithm 3. The 95% percentage confidence interval of the coefficient is $(0.95, 0.99)$.



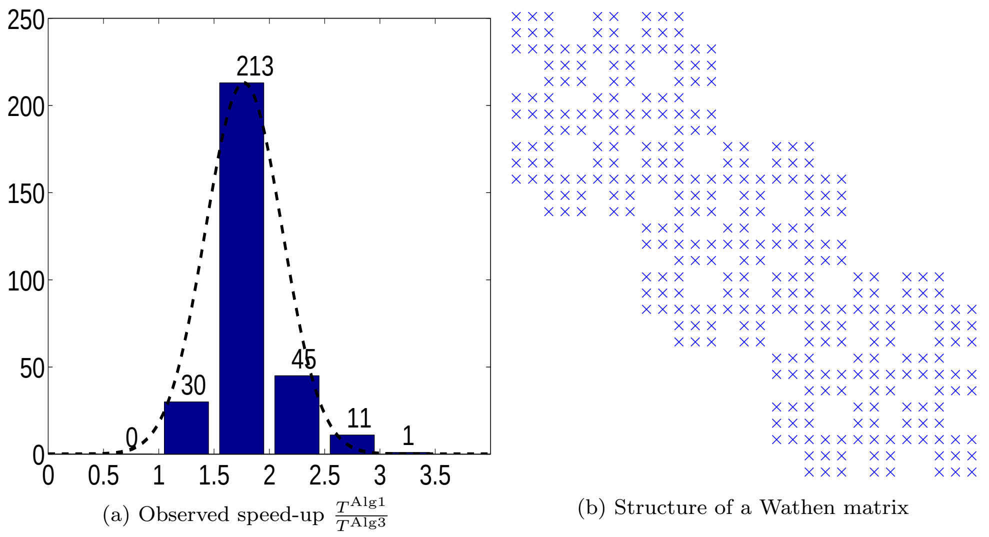
(a) Observed time (b) Flops efficiency (c) Memory efficiency

Fig. 4: Comparison between Algorithm 1 and Algorithm 2 for the 300 Wathen matrices. For these matrices with nice structure (see Fig. 6b), the speed-up of Algorithm 3 over Algorithm 1 is higher than average, with a 95% confidence interval $(1.86, 1.92)$. For such fixed problems share the same structure, both the memory efficiency and floating-point efficiency tend to be at a fixed level, there are only about 30 cases divergent away from the clusters in panel (b) and (c). The 95% confidence interval for the coefficient in (c) is (0.98, 1.01).



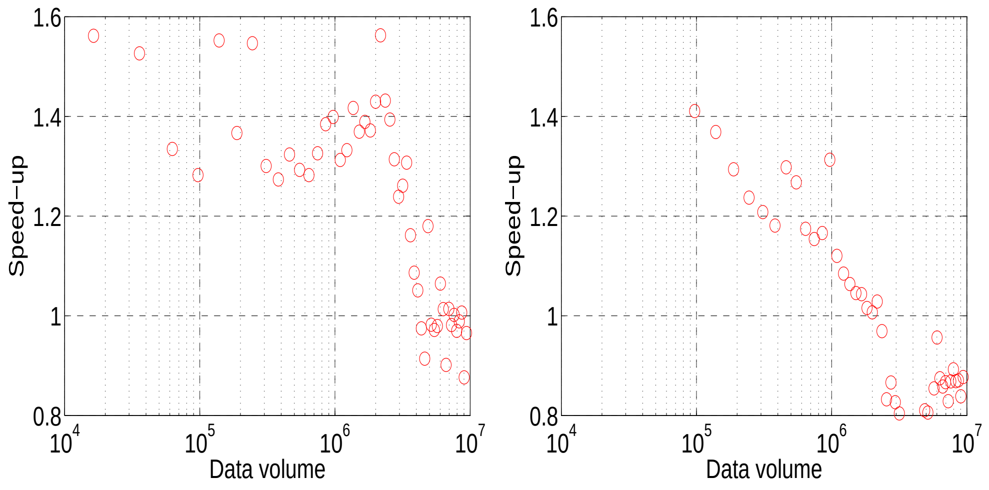
(a) Observed speed-up $\frac{T^{\text{Alg1}}}{T^{\text{Alg3}}}$ (b) Ideal speed-up $\frac{T^{\text{Alg1}}}{T^{\text{Alg3}}}$

Fig. 5: Comparison between observed speed-up and ideal speed-up for the 996 matrices from UF sparse matrices collection. The figures show the count of speed-up which locates in a certain interval. For observed speed-up, those bigger than 3.5 are assumed belong to the group 3.0 to 3.5. The ideal speed-up is computed by (3).



(a) Observed speed-up $\frac{T^{\text{Alg1}}}{T^{\text{Alg3}}}$ (b) Structure of a Wathen matrix

Fig. 6: Observed speed-up for the 300 Wathen matrices and the structure of `gallery('wathen',2,3)` matrix. For observed speed-up, those bigger than 3.5 are assumed belong to the group 3.0 to 3.5. The average non-zero elements per column of these Wathen matrices is 15.6, the average ideal speed-up by formula (3) is 1.899, which is very close to the fitted value 1.89 in Fig 4 and locates in the 95% percentage confidence interval (1.86, 1.92).



(a) Speed-up $\frac{T^{\text{Alg1}}}{T^{\text{Alg2}}}$ (b) Speed-up $\frac{T^{\text{Matlab}}}{T^{\text{Alg2}}}$

Fig. 7: Comparison of the speed-up of the CSR format over CSC format and CSR format over Matlab `y=A*x`. Only the first 50 Wathen matrices are used. Overheard due to `tic;toc` is removed for Matlab timing, several cases in (b) are outside of the range due to larger speed-up. The data volume here is the memory size for matrix A, vectors x and y, not the memory reference volume. A turning point in both case is when the data volume equals 2MB, the L2 cache size. The Matlab `y=A*x` have a similar performance with that of the CSC format for small matrices (enough to fit into the L2 cache). When the data volume is larger than the L2 cache size, the performance of SpMVs in CSC format Algorithm 1 and CSR format Algorithm 2 tends to have the same performance, this is because the L3 cache are much slower on mother board, the data movement from the lower level memory dominates the whole time. For larger matrices MATLAB possibly uses auto turning and blocking techniques to improve the performance of SpMVs.