A Generic High-Performance Method for Deinterleaving Scientific Data

Eric R. Schendel[1,3,4], Steve Harenberg[1,4], Houjun Tang[1,4], Venkatram Vishwanath[3], Michael E. Papka[2,3], and Nagiza F. Samatova[1,4,⋆]

[1] North Carolina State University, Raleigh, NC 27695, USA
[2] Northern Illinois University, DeKalb, IL 60115, USA
[3] Argonne National Laboratory, Argonne, IL 60439, USA
[4] Oak Ridge National Laboratory, Oak Ridge, TN 37830, USA
samatova@csc.ncsu.edu

Abstract. High-performance and energy-efficient data management applications are a necessity for HPC systems due to the extreme scale of data produced by high fidelity scientific simulations that these systems support. Data layout in memory hugely impacts the performance. For better performance, most simulations interleave variables in memory during their calculation phase, but deinterleave the data for subsequent storage and analysis. As a result, efficient data deinterleaving is critical; yet, common deinterleaving methods provide inefficient throughput and energy performance. To address this problem, we propose a deinterleaving method that is high performance, energy efficient, and generic to any data type. To the best of our knowledge, this is the first deinterleaving method that 1) exploits data cache prefetching, 2) reduces memory accesses, and 3) optimizes the use of complete cache line writes. When evaluated against conventional deinterleaving methods on 105 STREAM standard micro-benchmarks, our method always improved throughput and throughput/watt on multi-core systems. In the best case, our deinterleaving method improved throughput up to 26.2x and throughput/watt up to 7.8x.

1 Introduction

Emerging extreme-scale high performance computing (HPC) systems enable high fidelity scientific simulations that generate data at an increasing rate [1]. Yet, these HPC systems and data-intensive applications they support consume energy at an ever-increasing amount [2,3]. Thus, the need for performance and energy efficient data management applications is of utmost importance to maximize throughput/watt while achieving improved scalability and sustainability [4].

To improve performance during scientific data analysis, which is critical for gaining insights from the simulations, simulations often have to *deinterleave* data variables. Upon deinterleaving, the data set for each variable of the simulation is contiguous in memory and storage. This deinterleaved layout is beneficial since most data analyses span multiple time steps of a particular variable [5].

⋆ Corresponding author.

F. Wolf, B. Mohr, and D. an Mey (Eds.): Euro-Par 2013, LNCS 8097, pp. 571–582, 2013.

In contrast, most simulations perform calculations using instances of many variables from a current/previous time step. Hence, an *interleaved* layout in memory provides better data locality during simulation runs by keeping each group of variables together in memory for the active time steps, see Figure 1.

Deinterleaving data is frequently necessary after the completion of a simulation step before data analysis and storage. For example, simulations such as FLASH [6], S3D [7] and Nek5000 [8] have variables that are interleaved in memory while most storage and analysis, such as data compression [9,10] and variable precision analytics [11], are performed using a deinterleaved layout. Through performing numerous micro-benchmarks, we found that common deinterleaving methods have poor throughput and energy performance.

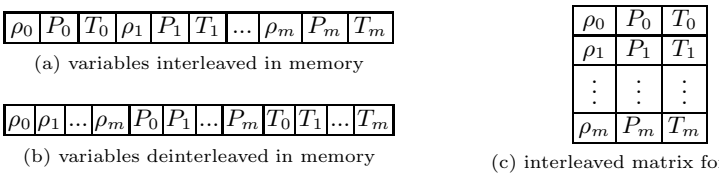
To address this problem, we propose a deinterleaving method that is high performance, energy efficient, and generic to any variable data type. To the best of our knowledge, this is the first deinterleaving method that 1) exploits data cache prefetching, 2) reduces memory accesses, and 3) optimizes the use of complete cache line writes. As a result, our method increases the throughput performance, reduces memory latency, and improves energy utilization.

Specifically, we compare the throughput performance and energy utilization of our deinterleaving method to two common deinterleaving methods. We assessed our method with 105 STREAM standard micro-benchmarks including 84 throughput and 21 energy performance test cases of varying input sizes and data types. In all cases tested, our method achieved better throughput and energy performance than the other two methods. In the best case, our method improved throughput up to 26.2x and throughput/watt up to 7.8x, when compared to the next best deinterleaving method.

2 Background

Simulations such as FLASH, S3D, and Nek5000 have variables that are interleaved in memory. These interleaved variables can be thought of as a matrix of data stored in row major format where each column corresponds to a particular variable. For multidimensional variables, each dimension has a separate column. Consider an example of FLASH simulation data with a sample of three variables ρ, P, and T corresponding to gas density, pressure, and temperature, respectively. The interleaved layout of these variables in memory can be seen in Figure 1a. Representing this data in matrix form would give an $m \times 3$ matrix where the three columns correspond to the three variables and the rows correspond to different steps of the simulation, see Figure 1c. With this interpretation, deinterleaving the data is equivalent to performing a matrix transposition, which would change the layout of the variables in memory, see Figure 1b.

There are two common techniques for deinterleaving data by performing an out-of-place matrix transposition. We refer to these techniques as *standard transposition* and *strided transposition*. These two techniques, along with our proposed method in the following section, are considered *out-of-place* due to the use of an output memory space equal to the size of the original matrix where the elements are copied. In contrast, *in-place* transposition methods use a bounded



(a) variables interleaved in memory

(b) variables deinterleaved in memory

(c) interleaved matrix format

Fig. 1. FLASH data in interleaved and deinterleaved layouts; each ρ_f, P_f, and T_f for $f = 0$ to m refers to the value of ρ, P, and T of the simulation at the f^{th} matrix row

amount of memory space and, in some cases, can slightly outperform out-of-place methods. However, in-place methods are often complex and can be performance constraining for simulations requiring variable interleaving, such as FLASH, S3D and Nek5000, to continue from where it left off in the calculation phase.

The standard and strided out-of-place transposition methods differ from each other in how they copy elements into an output memory buffer. The standard transposition method uses two loops to iterate row-wise and writes out the elements in a strided manner [12]. Alternatively, the strided transposition method uses two loops to iterate column-wise and writes out the elements contiguously.

3 Method

Our deinterleaving method performs an out-of-place transposition to transform a matrix of data stored in row major format to one stored in column major format. During the transposition process, our method combines the strength of both the standard transposition and strided transposition techniques.

In this section, we describe our deinterleaving method in detail. The method section is divided into three subsections corresponding to the three major components of our method: 1) cache prefetching on blocks of data, 2) using the registers as a vector transposition buffer, and 3) optimizing for full cache line writes. In addition, we provide a simple example for clarity.

3.1 Cache Prefetching on Blocks of Data

The benefit of cache prefetching is to hide latency time sinks associated with accessing main memory [13]. The standard transposition method, as discussed in Section 2, is able to take advantage of these benefits due to the sequential data reads inherent in its method. In contrast, the major weakness of the strided transposition method is that cache prefetching is not guaranteed and its effectiveness is dependent on the input buffer size. The cache prefetching benefits of the standard transposition method were the motivation for performing cache prefetching in our method.

Given an $m \times n$ (m rows and n columns) matrix of elements, A, stored in row major format, the first step of our deinterleaving method is to partition A into a block matrix where the blocks correspond to submatrices of A that

$$A = \begin{pmatrix} \begin{pmatrix} e_{1,1} & e_{1,2} & \cdots & e_{1,n} \\ e_{2,1} & e_{2,2} & \cdots & e_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ e_{m_b,1} & e_{m_b,2} & \cdots & e_{m_b,n} \end{pmatrix} \\ \begin{pmatrix} e_{(m_b+1),1} & e_{(m_b+1),2} & \cdots & e_{(m_b+1),n} \\ e_{(m_b+2),1} & e_{(m_b+2),2} & \cdots & e_{(m_b+2),n} \\ \vdots & \vdots & \ddots & \vdots \\ e_{2m_b,1} & e_{2m_b,2} & \cdots & e_{2m_b,n} \end{pmatrix} \\ \vdots \\ \begin{pmatrix} e_{(M-1)m_b+1,1} & e_{(M-1)m_b+1,2} & \cdots & e_{(M-1)m_b+1,n} \\ e_{(M-1)m_b+2,1} & e_{(M-1)m_b+2,2} & \cdots & e_{(M-1)m_b+2,n} \\ \vdots & \vdots & \ddots & \vdots \\ e_{Mm_b,1} & e_{Mm_b,2} & \cdots & e_{Mm_b,n} \end{pmatrix} \end{pmatrix} = \begin{pmatrix} B_1 \\ B_2 \\ \vdots \\ B_M \end{pmatrix}$$

Fig. 2. Matrix A being partitioned into M blocks of size $m_b \times n$

will be consecutively prefetched into cache. As illustrated in Figure 2, matrix A is partitioned as an $M \times 1$ block matrix where each block is of size $m_b \times n$. Partitioning A in this manner creates M blocks each of which we label as B_k for $k = 1$ to M. The number of rows in each block, denoted m_b, is chosen so a block column can fill the entire cache line, discussed in Section 3.3.

For example, suppose the cache line is size of C bytes, which on most modern architectures is 64 or 128 bytes [14]. Suppose the elements of matrix A are each β bytes. Then, for m_b elements to fill the cache line as full as possible we want $m_b\beta = C$, and therefore make $m_b = \lfloor C/\beta \rfloor$. It is plausible that the last block will have fewer elements than the other blocks because m_b may not evenly divide the m elements. In this case, $M = \lceil m/m_b \rceil$. To process the smaller block, the matrix can be padded with values that will be disregarded [15].

The blocks, B_k for $k = 1$ to M, correspond to the submatrices of A that will be consecutively prefetched into the cache. Block of data B_{k+1} will be prefetched into cache while the block B_k is being further processed, as described in the following subsections. By prefetching blocks of elements in this manner, our method can reduce memory latency associated with loading blocks from memory.

3.2 Using the Registers as a Vector Transposition Buffer

Each block B_k can further be partitioned into submatrices using the columns as dividers, making B_k into a $1 \times n$ block matrix, referred to as a *column vector*, as seen in Figure 3a. With both partitions applied, matrix A can be viewed as a matrix of column vectors as shown in Figure 3b. Each column vector of B_k, which we denote as $V^c_{k,j}$ for $j = 1$ to n, consists of elements that are currently non-contiguous in memory due to the row major storage format of A.

The goal of our deinterleaving method is the elements of the column vectors to be contiguous in memory or, equivalently, the elements to belong to the same row in the matrix. To make the elements contiguous, each column vector gets transposed and temporarily stored in CPU registers until it is written out to a full cache line. The general notation for each transposed column vector, now referred to as a *row vector*, is denoted: $V^R_{k,j} = [e_{(k-1)m_b+1,j}, e_{(k-1)m_b+2,j}, \cdots, e_{km_b,j}]$.

$$B_k = \left(\begin{bmatrix} e_{(k-1)m_b+1,1} \\ e_{(k-1)m_b+2,1} \\ \vdots \\ e_{km_b,1} \end{bmatrix} \begin{bmatrix} e_{(k-1)m_b+1,2} \\ e_{(k-1)m_b+2,2} \\ \vdots \\ e_{km_b,2} \end{bmatrix} \cdots \begin{bmatrix} e_{(k-1)m_b+1,n} \\ e_{(k-1)m_b+2,n} \\ \vdots \\ e_{km_b,n} \end{bmatrix} \right) = \begin{pmatrix} V^C_{k,1} & V^C_{k,2} & \cdots & V^C_{k,n} \end{pmatrix}$$

(a) Partitioning of block B_k into column vectors $V^C_{k,j}$ for $j = 1$ to n

$$A = \begin{pmatrix} B_1 \\ B_2 \\ \vdots \\ B_M \end{pmatrix} = \begin{pmatrix} V^C_{1,1} & V^C_{1,2} & \cdots & V^C_{1,n} \\ V^C_{2,1} & V^C_{2,2} & \cdots & V^C_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ V^C_{M,1} & V^C_{M,2} & \cdots & V^C_{M,n} \end{pmatrix}$$

(b) Matrix A partitioned into submatrices of column vectors

Fig. 3. Each block of matrix A partitioned into n column vectors

For clarity, consider a specific example. Suppose block B_1 is currently being partitioned into n column vectors, namely $V^c_{1,j}$ for $j = 1$ to n. The elements of a column vector $V^c_{1,j}$ consist of the elements $e_{1,j}, e_{2,j}, ..., e_{m_b,j}$ from A as seen in Figure 3a. Starting with the first column vector ($j = 1$), the elements must be loaded into a buffer of registers and in the next step written into the extra memory space that was created for the transposition matrix. Using CPU registers as a buffer to store these elements constitutes a transposition of the column vector as the elements will now be contiguous instead of strided.

The motivation for using the registers as a temporary buffer is that each column vector must be transposed into some storage location in order to achieve full cache line writes, which is the strength of the strided transposition method. The registers provide the most efficient location to store the row vectors due to their minimal CPU cycles per operation [16]. In addition, using a buffer of registers in this manner is a viable option since typically a CPU provides enough hardware registers where the buffer size is at least equal to the cache line size.

3.3 Optimizing for Full Cache Line Writes

Once the elements of a row vector are loaded into the register buffer, our method then writes out this data into the memory space that was created for the deinterleaved output. During the write process, our method utilizes the full cache line due to the row vector containing m_b elements, where m_b was chosen to fill the cache line. By utilizing full cache line writes, our method emulates the strength of the strided transposition method [17], while avoiding the inefficient write process of the standard transposition method.

During the write process, our method must leave enough room for m elements of A (an entire column) between the start of each column vector, meaning there will be a stride of m between the memory storage offset of each column vector. So, for a given row vector $V^R_{k,j}$, the elements get mapped consecutively into the new memory storage location offset starting at $(k-1)m_b + (j-1)m$.

After this process is completed and all the row vectors have been written, the process is repeated. The next block, which should already reside in cache, is partitioned into column vectors that are consecutively loaded into the register buffer and written out. The entire process is completed for each block B_k for $k = 1$ to M. Once every block has gone through this process, the output location will

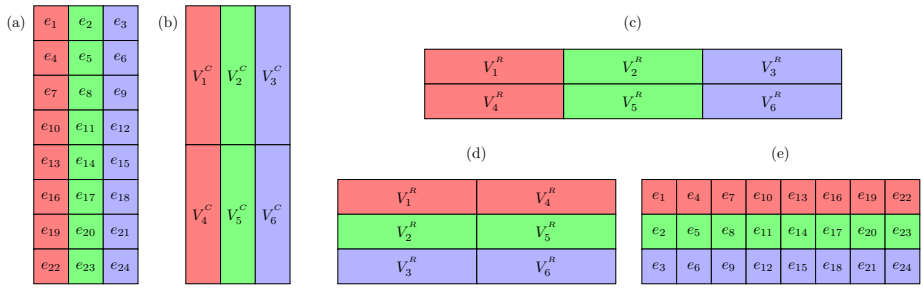


Fig. 4. The partition and transposition steps of our deinterleaving method performed on a simple 8×3 matrix of 8-byte elements optimized for cache line writes of 32 bytes

contain the transpose of matrix A. The entire deinterleaving process is illustrated by the example given in the following section.

3.4 A Simple Example of Our Deinterleaving Method

For clarity, consider a simple example of 24 data elements consisting of three different variables interleaved in memory. Figure 4a shows the matrix representation of these interleaved variables, with each column of the matrix storing data corresponding to a particular variable. For the sake of this example, suppose the elements are 8-byte doubles (common in simulation data) and the cache line size of the system is 32 bytes. The elements of the matrix are initially stored in row major format, meaning the elements are ordered as $e_1, e_2, e_3, e_4, ..., e_{24}$ in memory. The goal of our deinterleaving method is to obtain the transpose of the matrix, illustrated in Figure 4e, so that the elements of each column will be contiguous in memory and thus deinterleaved.

The initial step of our deinterleaving method is to create a new output memory space to hold the transposed matrix. Next, the matrix is partitioned into two 4×3 block matrices, B_1 and B_2 consisting of elements e_1 through e_{12} and e_{13} through e_{24}, respectively. The number of rows in each block was chosen as $m_b = 4$ so that each column within a block will entirely fill the cache line, as four 8-byte doubles is exactly the cache line size of the system.

With the matrix partitioned into two blocks, the next step is to load B_1 into the cache. The block itself is then partitioned into the three column vectors V_1^C, V_2^C, and V_3^C, as depicted in Figure 4b. After this partition, the first column vector of B_1, meaning the elements e_1, e_4, e_7, and e_{10}, is transposed into a row vector and temporarily stored in the register buffer, see Figure 4c. The full cache line is then utilized to write out the elements of the row vector into the output memory space that was created for the transposed matrix, see Figure 4d. This process is repeated on the remaining column vectors of B_1 until all of them have been written into the output memory space.

After B_1 has finished transposing and writing each of its column vectors, the same process is repeated on the second block, B_2. This block would have been

prefetched into cache during the time B_1 was being processed, thus saving the time of retrieving B_2 from memory. After B_2 is processed, the matrix will be transposed and the variables deinterleaved, as illustrated in Figure 4e.

4 Performance Evaluation

In this section, we present the empirical evaluations of our deinterleaving method via a set of micro-benchmarks to evaluate throughput and energy performance. We compare the results of our deinterleaving method against those of the standard and strided transposition methods. For brevity, we will refer to our Out-of-Place Deinterleaving method as *OPD method* in the remainder of the paper.

4.1 Experimental Setup

Performance measurements were collected on the Lens Linux cluster at Oak Ridge National Laboratory and on a dedicated Intel server. The Lens cluster is primarily used for data analysis and high-end visualization. Each cluster node consists of four quad-core 2.3 GHz AMD Opteron processors and 128GB of memory. Each processor has three cache levels: L1 cache is 64KB, L2 cache is 512KB, and the shared last level cache (LLC) is 5118KB. The Intel server consists of a quad-core i7 2.93 GHz processor and 16GB of memory running CentOS-6.3. The Intel processor has three cache levels: L1 is 32KB, L2 is 256KB, and LLC is 8MB. All multi-core evaluations for both the throughput and energy experiments were done utilizing all available processors and computational cores.

For collecting performance metrics, we added micro-benchmarks of all deinterleaving methods into the STREAM [18] framework, compiled with GNU Compiler Collection (GCC) version 4.7.1. STREAM is useful for evaluating memory throughput performance of single- and multi-core I/O-intensive functions that are sensitive to system architecture characteristics [19]. We compared the throughput performance metrics collected from 105 STREAM micro-benchmarks tested across the AMD and Intel systems. The test cases spanned a diverse set of data including multiple data types, column sizes, and input buffer sizes. Specifically, the data types evaluated were bytes, single-precision floating-points, and double-precision floating-points. For each data type, the variables interleaved (columns) were $2, 4, 8$, and 16. The input buffer sizes ranged from $64, 128, \cdots, 4096$ kilobytes per core. To obtain the performance measurements seen in Figure 5 and Figure 6, each micro-benchmark was run 100 times for each deinterleaving method. The highest throughput of the 100 runs was recorded.

For our set of micro-benchmarks, we restricted our input buffer size between 64KB and 4096KB. The reason this lower bound was chosen is due to the precision of the timer used in the STREAM benchmark, which states at what point the clock measurement becomes unreliable. For input sizes less than 64KB, our deinterleaving technique ran too fast for a reliable throughput measurement. However, at sizes of 64KB and higher, the throughput could be measured accurately. The upper bound of 4096KB was chosen to represent an input size that was beyond the size of the LLC for multi-core evaluations.

4.2 Deinterleaving Throughput Performance

In all multi-core evaluations, our deinterleaving method performed better than the standard and strided transposition methods, see Figure 5 and Figure 6. In the best case, our deinterleaving method performed at a 26.2x faster throughput, when compared to the next best method. In addition, our method consistently reported gains of over 40GB/s on smaller input sizes (corresponding to lower cache levels). The performance gains of our deinterleaving method were more pronounced on smaller input buffer sizes because memory latency starts to become a significant factor on larger buffer sizes.

Another characteristic seen in our results is that neither the standard transposition nor the strided transposition was consistently better than the other. In some cases, the standard transposition would significantly outperform the strided transposition and vice versa, irrespective of the instruction set architecture being used, see Table 1. The performance inconsistency of these two techniques is another strength of our deinterleaving method, as ours consistently outperformed the other two methods.

Although not depicted in throughput performance figures, our method was also compared against the other methods when all were utilizing only a single core of the system. In this case, our method reported similar, but scaled down trends to those seen in multi-core evaluations. Even in this case, our method always had better throughput performance than the other two methods.

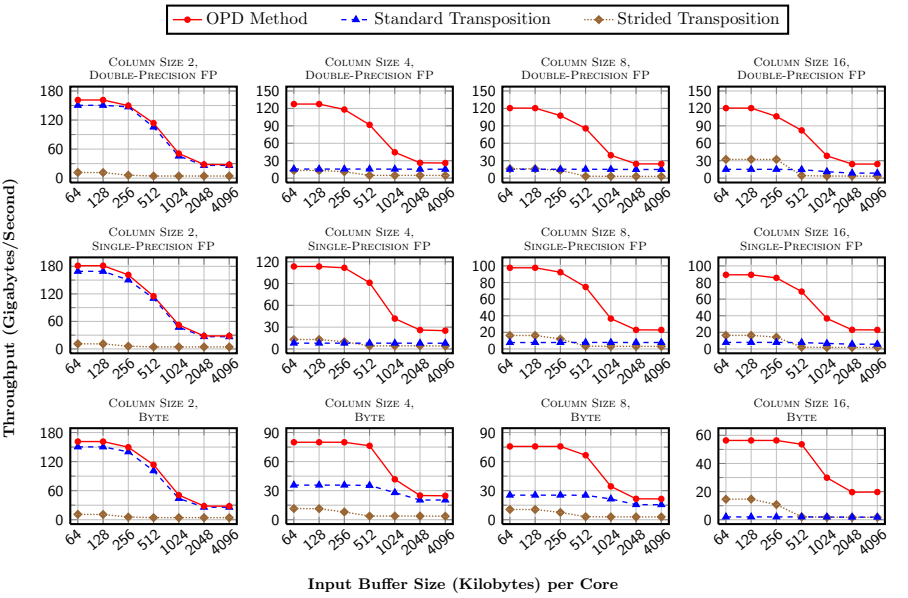


Fig. 5. Throughput performance applying STREAM micro-benchmarks when deinterleaving single-precision, double-precision floating-point (FP), and byte variables on the AMD Opteron system utilizing all 16 cores

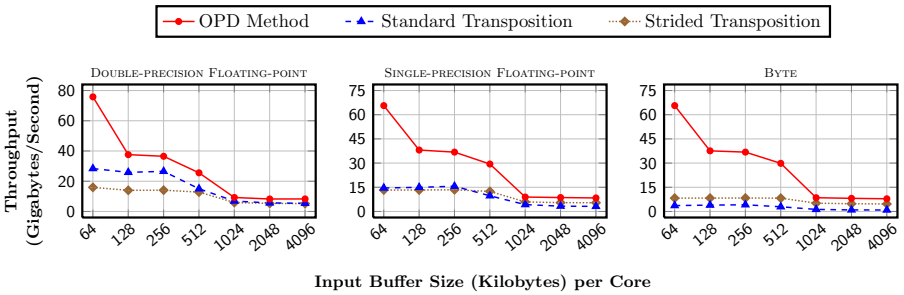


Fig. 6. Throughput performance applying STREAM micro-benchmarks when deinterleaving single-precision, double-precision floating-point, and byte variables with 16-variable interleaved data on the Intel i7 system utilizing all cores

Table 1. Instruction set architecture for deinterleaving methods

Data Type	Method	Column Size			
		2	4	8	16
Double	Standard	SSE2	x86_64	x86_64	x86_64
	Strided	x86_64	x86_64	x86_64	x86_64
Float	Standard	SSE	SSE	SSE	SSE
	Strided	SSE	x86_64	x86_64	x86_64
Byte	Standard	SSE2	SSE2	SSE2	SSE2
	Strided	x86_64	x86_64	x86_64	x86_64

4.3 Deinterleaving Energy Performance

The energy performance measurements were performed on a dedicated Intel server connected to a Watts Up Pro meter, which provides a recording of power measurements (watts) per second during the collection of throughput metrics. The power was measured for each deinterleaving method on 21 micro-benchmarks of 16-variable interleaved data of varying input sizes and data types. Energy performance normalization was done for the deinterleaving methods by calculating gigabytes per joule (throughput/watt) for each test case.

In all cases tested, our deinterleaving had better energy utilization than the other methods, with throughput/watt improvements up to 7.8x, when compared to the next best method. The results of our energy experiments can be seen in Figure 7. The improved energy performance of our method is attributed to the increased throughput (Figure 6), the effective cache utilization similar to the standard transposition method, and the optimized cache line writes like the strided transposition method.

5 Related Work

Out-of-place matrix transpositions have been studied extensively in the past. Majority of these transposition algorithms, initially proposed decades ago, focus on methodologies for optimizing use of secondary storage (tapes, disks, etc.). Although these algorithms are not well suited for modern computer systems due

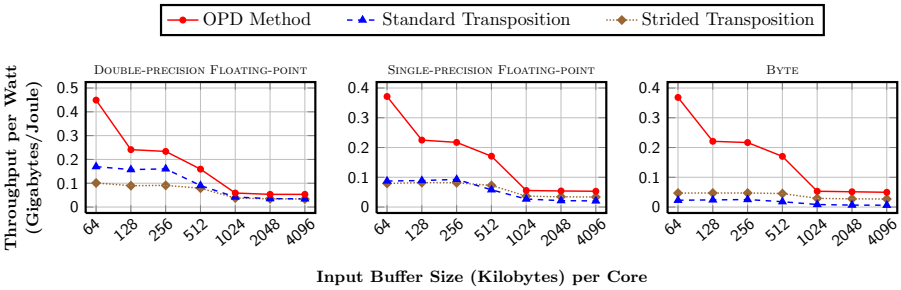


Fig. 7. Normalized energy performance measurements (throughput/watt) collected with power meter during STREAM throughput benchmarks on Intel system (Figure 6)

to processor cache inefficiency, we still use these techniques for references since secondary storage of the past is analogous to RAM in modern systems. A fast matrix transposing method was given in [20] where the algorithm was specifically designed for $2^n \times 2^n$ square matrices and it is compared with many other matrix transposition algorithms. Another algorithm called single radix algorithm was proposed in [21], and shows better performance in disk seeks and accesses. For transposing a large arbitrary matrix, PRIM was introduced in [22].

In-place matrix transpositions can be used as an alternative to out-of-place methods; however, in-place methods are often complex and can be performance inefficient for simulations requiring interleaved variables to continue with the calculation phase. Furthermore, in-place methods commonly have constraints on row and column sizes making them unusable as a generic method for deinterleaving scientific data. Six algorithms are investigated in [23] for transposing a large square matrix in-place. They use 32-bit single-precision floating-point numbers and have the length of both the row and column equal to 2^n. In their experiments, the non-linear array layout algorithm outperforms other algorithms as it uses "Morton ordering" [24]. This algorithm also uses recursion to divide the problem into smaller subproblems, as in [12], but terminates at an architecture-specific tile size. Even by using a "blocking" and "tiling" technique, a higher cache efficiency might not be achieved as claimed in [16]; instead, they proposed a buffer must be used in order to be cache efficient.

Although much attention has been paid to matrix transposition, very few of the studies focus on the utilization of cache in a specific domain requiring deinterleaving of variables. Our method applies to any data type and utilizes full cache line writes to be throughput and energy efficient when deinterleaving data. Blocking, shuffling, and compression library, Blosc, was introduced in [25], which uses a high-performance byte deinterleaving technique to reduce activity on the memory bus. Our approach differs from this technique in that we support not just byte-level but float- and double-level as well. Moreover, Blosc currently utilizes 16-byte SSE2 register writes instead of full cache line writes compared to our deinterleaving method.

6 Conclusion

We proposed a deinterleaving method that is high performance, energy efficient, and generic to any data type. Our method has increased throughput and energy performance by utilizing the system architecture in three ways: 1) improving data cache prefetching, 2) reducing memory accesses, and 3) optimizing the use of full cache line writes.

Our method results in better throughput and energy performance when compared against two common deinterleaving methods during 105 STREAM standard micro-benchmarks evaluations, which includes 84 throughput and 21 energy performance test cases. When compared to the next best case, our method improved throughput up to 26.2x and throughput/watt up to 7.8x.

Acknowledgements. We like to thank Argonne National Laboratory (ANL) and Oak Ridge National Laboratory (ORNL) for use of resources at their Leadership Computing Facilities, ALCF and OLCF respectively. Also, thanks goes to Flash Center for Computational Science at the University of Chicago for providing scientific datasets. This research and ALCF resources at ANL have been funded and supported in part by the U.S. Department of Energy, Office of Science, Advanced Scientific Computing Research, under Contract DE-AC02-06CH11357. ORNL is managed by UT-Battelle for the LLC U.S. D.O.E. under Contract DE-AC05-00OR22725. Also, partial support for this work was provided through Scientific Discovery through Advanced Computing (SciDAC) program funded by U.S. Department of Energy, Office of Science, Advanced Scientific Computing Research and the U.S. National Science Foundation (Expeditions in Computing and EAGER programs).

References

1. Ma, K.: In situ visualization at extreme scale: Challenges and opportunities. IEEE Computer Graphics and Applications 29(6), 14–19 (2009)
2. Laurenzano, M.A., Meswani, M., Carrington, L., Snavely, A., Tikir, M.M., Poole, S.: Reducing energy usage with memory and computation-aware dynamic frequency scaling. In: Jeannot, E., Namyst, R., Roman, J. (eds.) Euro-Par 2011, Part I. LNCS, vol. 6852, pp. 79–90. Springer, Heidelberg (2011)
3. Stevens, R., White, A., Dosanjh, S., Geist, A., Gorda, B., Yelick, K., Morrison, J., Simon, H., Shalf, J., Nichols, J., Seager, M.: Scientific grand challenges: Architectures and technologies for extreme scale computing. Tech. Rep., DOE (2009)
4. Ge, R., Feng, X., Sun, X.: SERA-IO: Integrating energy consciousness into parallel I/O middleware. In: Proceedings of the 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, pp. 204–211 (2012)
5. Latham, R., Daley, C., Liao, W., Gao, K., Ross, R., Dubey, A., Choudhary, A.: A case study for scientific I/O: Improving the FLASH astrophysics code. Computational Science & Discovery 5(1), 015001 (2012)
6. Fryxell, B., Olson, K., Ricker, P., Timmes, F., Zingale, M., Lamb, D., MacNeice, P., Rosner, R., Truran, J., Tufo, H.: FLASH: An adaptive mesh hydrodynamics code for modeling astrophysical thermonuclear flashes. The Astrophysical Journal Supplement Series 131(1), 273 (2008)

7. Chen, J., Choudhary, A., De Supinski, B., DeVries, M., Hawkes, E., Klasky, S., Liao, W., Ma, K., Mellor-Crummey, J., Podhorszki, N., Sankaran, R., Shende, S., Yooothers, C.S.: Terascale direct numerical simulations of turbulent combustion using S3D. Computational Science & Discovery 2(1), 015001 (2009)
8. Fischer, P.F., Lottes, J.W., Kerkemeier, S.G.: (2008), http://nek5000.mcs.anl.gov/
9. Schendel, E.R., Jin, Y., Shah, N., Chen, J., Chang, C., Ku, S.H., Ethier, S., Klasky, S., Latham, R., Ross, R., Samatova, N.F.: ISOBAR preconditioner for effective and high-throughput lossless data compression. In: Proceedings of the 28th International Conference on Data Engineering (ICDE), pp. 138–149. IEEE (2012)
10. Schendel, E.R., Pendse, S., Jenkins, J., Boyuka II, D.A., Gong, Z., Lakshminarasimhan, S., Liu, Q., Kolla, H., Chen, J., Klasky, S., Ross, R., Samatova, N.F.: ISOBAR hybrid compression-I/O interleaving for large-scale parallel I/O optimization. In: Proceedings of the 21st International Symposium on High-Performance Parallel and Distributed Computing (HPDC), pp. 61–72. ACM (2012)
11. Jenkins, J., Schendel, E.R., Lakshminarasimhan, S., Boyuka II, D.A., Rogers, T., Ethier, S., Ross, R., Klasky, S., Samatova, N.F.: Byte-precision level of detail processing for variable precision analysis. In: ACM/IEEE International Conference for High Performance Computing, Networking, Storage, and Analysis, p. 48 (2012)
12. Frigo, M., Leiserson, C., Prokop, H., Ramachandran, S.: Cache-oblivious algorithms. In: Symposium on Foundations of Computer Science, pp. 285–297. IEEE (1999)
13. Gamoudi, O., Drach, N., Heydemann, K.: Using runtime activity to dynamically filter out inefficient data prefetches. In: Jeannot, E., Namyst, R., Roman, J. (eds.) Euro-Par 2011, Part I. LNCS, vol. 6852, pp. 338–350. Springer, Heidelberg (2011)
14. Conway, P., Kalyanasundharam, N., Donley, G., Lepak, K., Hughes, B.: Cache hierarchy and memory subsystem of the AMD Opteron processor. IEEE Micro 30(2), 16–29 (2010)
15. Dow, M.: Transposing a matrix on a vector computer. Parallel Computing 21(12), 1997–2005 (1995)
16. Gatlin, K., Carter, L.: Memory hierarchy considerations for fast transpose and bit-reversals. In: Proceedings of Fifth International Symposium High-Performance on Computer Architecture, pp. 33–42. IEEE (1999)
17. Drepper, U.: What every programmer should know about memory. Tech. Rep., Red Hat, Inc. (2007)
18. McCalpin, J.D.: STREAM: Sustainable memory bandwidth in high performance computers (2000), http://www.cs.virginia.edu/stream/
19. Gschwandtner, P., Fahringer, T., Prodan, R.: Performance analysis and benchmarking of the Intel SCC. In: Proceedings of International Conference on Cluster Computing, pp. 139–149. IEEE (2011)
20. Eklundh, J.: Efficient matrix transposition. In: Two-Dimensional Digital Signal Prcessing II, pp. 9–35 (1981)
21. Kaushik, S., Huang, C., Johnson, J., Johnson, R., Sadayappan, P.: Efficient transposition algorithms for large matrices. In: Proceedings of Supercomputing 1993, pp. 656–665. IEEE (1993)
22. Goldbogen, G.: PRIM: A fast matrix transpose method. IEEE Transactions on Software Engineering (2), 255–257 (1981)
23. Chatterjee, S., Sen, S.: Cache-efficient matrix transposition. In: Sixth International Symposium on High-Performance Computer Architecture, pp. 195–205. IEEE (2000)
24. Morton, G.M.: A Computer Oriented Geodetic Database and a New Technique in File Sequencing. IBM, Ltd. (1966)
25. Alted, F.: Why modern CPUs are starving and what can be done about it. Computing in Science & Engineering 12(2), 68–71 (2010)