WET: Write Efficient Loop Tiling for Non-Volatile Main Memory

Mohammad Alshboul
North Carolina State University
Raleigh, NC
maalshbo@ncsu.edu

James Tuck
North Carolina State University
Raleigh, NC
jtuck@ncsu.edu

Yan Solihin
University of Central Florida
Orlando, FL
yan.solihin@ucf.edu

Future systems are expected to increasingly include a Non-Volatile Main Memory (NVMM). However, due to the limited NVMM write endurance, the number of writes must be reduced. While new architectures and algorithms have been proposed to reduce writes to NVMM, few or no studies have looked at the effect of compiler optimizations on writes.

In this paper, we investigate the impact of one popular compiler optimization (loop tiling) on a very important computation kernel (matrix multiplication). Our novel observation includes that tiling on matrix multiplication causes a $25\times$ write amplification. Furthermore, we investigate techniques to make tilling more NVMM friendly, through choosing the right tile size and employing hierarchical tiling. Our method *Write-Efficient Tiling (WET)* adds a new outer tile designed for fitting the write working set to the Last Level Cache (LLC) to reduce the number of writes to NVMM. Our experiments reduce writes by 81% while simultaneously improve performance.

I. Introduction

Emerging Non-Volatile Memories (NVMs), such as Intel Optane DC Persistent Memory, have recently been integrated into computer systems as main memory [1] or as storage [2]. As main memory, it provides non-volatility, byte addressability, higher density and better scaling potential than DRAM, and has low idle power. On the flip side, NVMs have critical drawbacks when it comes to write operations. Writes in NVMs are slow and power/energy hungry. Even more serious is that NVMs have limited write endurance. This means that the number of writes to NVMM is a critical factor that determines the lifetime of memory and the computer system [3]–[5].

The limited write endurance of non-volatile main memory (NVMM) encouraged many researchers to investigate techniques to reduce the number of writes and spread writes more uniformly across memory cells, including new hardware techniques [6]–[8] as well as new algorithms [9]–[11]. However, few studies, if any, have investigated the impact of compiler optimizations in translating algorithms to code and how they affect the number of writes to NVMM.

In this paper, we investigate the impact of one popular compiler optimization (loop tiling) on a very important computation kernel (matrix multiplication). Loop tiling is a heavily used optimization for loop-based applications because it is very effective in improving cache locality [12]–[17] that leads to substantially improved performance. Tiled Matrix Multiplication is heavily used in different fields; for example, it contributes to about 80% of the execution time in the deep learning neural network workload [18], [19]. Tiling divides the entire matrix into several sub-matrices that fit in the L1 cache, reducing the number of L1 cache capacity misses. Each sub-matrix produces an intermediate result of the original matrix multiplication problem.

In this work, we make a novel observation that there can be a fundamental tension between making code cache friendly vs. making it NVMM write friendly. For loop tiling optimization, the tension is very high: tiling reduces execution time by 89% (i.e. $8.5\times$ speedup) but increases the number of writes by $25\times$ for matrix multiplication.

We investigate techniques to improve the write efficiency of tiling. In particular, we consider two techniques. The first technique tinkers with the tile size used in tiling, and chooses the tile size that minimizes writing to NVMM. Changing the tile size directly affects the total number of sub-matrices, and hence affects the number of times intermediate results are written to NVMM. However, as the tile size is designed to fit in the fast L1 cache, increasing it may degrade performance as the L1 cache miss rate may increase. Hence, the tension between cache friendliness and NVMM write friendliness persists, and one must be careful in selecting the tile size in this trade-off space that lies along the Pareto frontier. A second technique we consider is hierarchical tiling [15]–[17], [20], where two (or more) tile sizes are chosen. We found that having two tile sizes are advantageous. The inner tile can chosen to be small and designed to fit the inner working set in the L1 cache, but the outer tile can be chosen to fit the outer working set in the last level cache (LLC), which effectively makes the LLC a write buffer for the NVMM. We found that hierarchical tiling to be much more effective than tuning the tile size of traditional tiling, allowing us to break the Pareto frontier. Compared to the best single level tiling, hierarchical tiling reduces NVMM writes by 80% while simultaneously increases performance by 25%.

To summarize, the contributions of this paper are as follows:

1) We made a novel observation that loop tiling, while effective in improving cache locality, also increases NVMM writes substantially. The observation highlights the importance of including compiler transformations in evaluating NVMM write endurance.
2) We evaluate the effectiveness of tuning the tile size. Varying the tile size, we found Pareto frontier in the trade-off space of cache friendliness and NVMM write friendliness.
3) We propose write efficient tiling (WET) that relies on hierarchical tiling, with an outer tile tuned to keep most writes at the LLC. We evaluate WET and found that it effectively reduces NVMM writes substantially without penalizing performance (it usually improves performance).

II. Background

Matrix multiplication is one of the most heavily used kernels in scientific computing as well as machine learning [18], [19]. In the paper, we include two approaches in matrix multiplication: traditional and divide and conquer. They are described below.

978-1-7281-1085-1/20/$31.00 ©2020 IEEE

A. Tiled Matrix Multiplication

Figure 1 shows the code for the standard 6-loop implementation of Tiled Matrix Multiplication (TMM).

```
1  // moves horizontally on matrix A, and vertically on matrix B
2  for (k2=0; k2<n; k2+=tsize)
3      for (i2=0; i2<n; i2+=tsize) // vertical on matrix A and C
4          for (j2=0; j2<n; j2+=tsize) // horizontal on matrix B and C
5              for (i=i2; i<(i2+tsize); i++)
6                  for(j=j2; j<(j2+tsize); j++) {
7                      sum = R[i][j];
8                      for(k=k2; k<(k2+tsize); k++)
9                          sum += A[i][k]*B[k][j];
10                     R[i][j] = sum;
11                 }
```

Fig. 1: Tiled Matrix Multiplication code.

In *tmm*, the matrix multiplication code is divided into two parts. The first part consists of the outermost 3 loops (i.e. *k2*, *i2*, and *j2*). These loops are used to define the boundaries of the current tile, as illustrated lines (2-4) in Figure 1. The second part of *tmm* consists of the remaining 3 loops (i.e. *i*, *j*, and *k*) which are shown in lines (5-8) in Figure 1. These loops are similar to the regular matrix multiplication code, but it is only performed within the boundaries of the current tile.

Keep in mind that each *k2* iteration will pass over the entire *R* matrix and write to all its elements. This is the basis for our observation that tiling needs to be revisited with NVM in mind due to the increase in the number of writes. This increase equals to $\frac{n}{tsize}$ times per element, as opposed to only one write per element in the regular non-tiled matrix multiplication implementation.

B. Matrix Multiplication using Divide and Conquer

Another well-known method for performing matrix multiplication is using the Divide-And-Conquer (DAC) recursive algorithm [21]–[23].

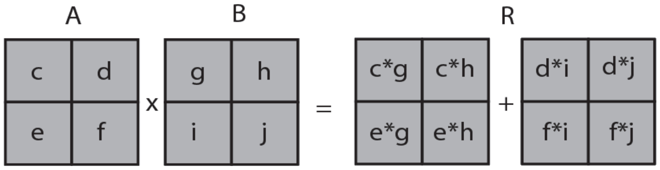


Fig. 2: Illustration of the Tiled Matrix Multiplication process using Divide and Conquer cache oblivious algorithm (DAC).

This version of Matrix Multiplication is very popular due to being *cache oblivious* [22]. This means that achieving high performance using this scheme can be done without requiring the programmer to tune the algorithm according to the cache specification in the underlying hardware system.

As shown in Figure 2, each of the input matrices will be divided into 4 quarters, and each of these quarters will be treated as an element in the matrix. This will make the original matrix multiplication into a 2 × 2 matrix multiplication. As in any 2 × 2 matrix multiplication, the process will result in 8 different multiplications. Then each corresponding two of these 8 multiplications will be added to produce the final multiplication output. This process will recursively divide the matrices until reaching a threshold matrix size where the actual multiplication will take place. Most implementations suggest using a threshold size designed to make matrices fit in the L1 cache, which guarantees performance, but not take number of writes to NVMM into consideration.

III. Related Work

Works related to NVM. Due to their attractive features, Non-Volatile Memories (NVM) have been an increasingly hot research area [24]–[29]. Many of them proposed solutions for improving NVM write endurance, using hardware techniques [3]–[5], as well as software techniques [9], [11]. To the best of our knowledge, no prior studies have investigated loop tiling in the context of NVMM. A method for reducing the number of writes for DAC matrix multiplication was suggested in [30], which was used as the base case we compared WET against.

Tiling Optimization. Tiling is one of the most well-known loop transformation techniques [31]. It achieves higher locality at the memory hierarchy by creating blocked algorithms [15]–[17], [32]. Moreover, some works suggested using tiling at multiple levels in the memory hierarchy [15]–[17], [32]. These multi-level tiling techniques have only been considered for performance; our work is the first to investigate and demonstrate their effectiveness for reducing NVMM writes. Consequently, while multi-level tiling can be applied to any cache hierarchy, prior multi-level tiling studies typically focused on adding a *new inner tile* for the register file (in addition to the L1 cache) [13], [32]–[34]. In contrast, since our goal is to reduce NVMM writes, WET adds a *new outer tile* for the LLC to allow coalescing of writes at the LLC.

IV. Write Efficient Tiling

In this section, we will discuss our exploration of techniques for reducing NVMM writes on matrix multiplication.

A. Tuning the Tile Size

The first technique we consider is to tune the tile size. The tile size directly affects the total number of sub-matrices, and hence affects the number of times intermediate results are written to NVMM. If the tile size is small, the cache locality is high for the L1 cache but not for the LLC. Here, we question whether increasing the tile size will lead to better result.

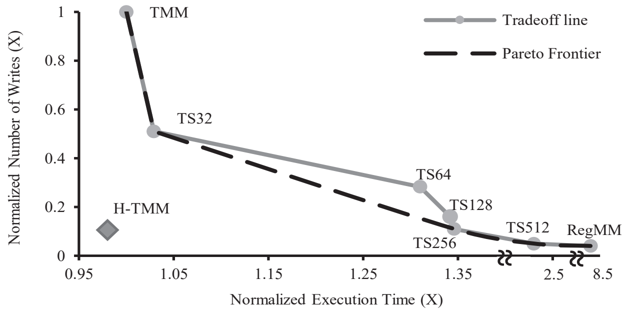


Fig. 3: Trade-off between normalized execution Time (x-axis) and normalized number of writes to memory (y-axis) when varying the innerTile size. Both of the axes are normalized to TMM with innerTile of size 16.

Figure 3 shows the impact of changing the tile size on the execution time (x-axes) and number of NVMM writes (y-axes), normalized to a naive tiled matrix multiplication (TMM). At one extreme, TMM uses the smallest possible tile size, resulting in the best performance but 25× write amplification vs. regular matrix multiplication (RegMM) with no tiling. At another extreme, RegMM (bottom right corner) has the lowest number of writes but is the slowest (8.5× slower compared to TMM). Other tile sizes fare somewhere in between. We can make several observations. First, the figure shows a Pareto frontier that includes some tile sizes but not others. For example, size 128 and 256 achieve nearly identical execution time but size 256 incurs fewer writes (31.2% fewer), which indicates that size 256 is Pareto efficient but size 128 is not. Size 512 achieves nearly the same number of writes as RegMM but size 512 is nearly three times faster than RegMM, indicating that RegMM is also not Pareto efficient.

Overall, our Pareto frontier analysis helps programmers and tuners to ignore tile sizes that are not Pareto efficient. However, ultimately the tension between cache friendliness and NVMM write friendliness still persists.

B. Hierarchical Tiling

With a single tile size, the tension between cache friendliness and NVMM write friendliness persists, so in the second approach we hypothesize that the tension can be relieved by relying on two tile sizes simultaneously. This requires a multi-level tiling approach which adds another hierarchy level to the tiling optimization. Hierarchical tiling was proposed in the past, primarily for improving locality at the register level (inner tile) and the L1 cache (outer tile). Here, we advocate *write-efficient tiling* (WET), which exploits the outer tile to improve write locality at the last level cache (LLC), for the purpose of reducing the number of NVMM writes. This distinguishes WET even from the techniques that looked into LLC tiling for performance [17], [20]. Our introduced tiling level *outerTile*, chosen correctly, can serve to coalesce writes generated by the tiling algorithm, reducing the number of writes going out to the NVMM. The inner tile, on the other hand, can still be chosen to fit L1 cache.

There are subtle differences between WET and prior hierarchical tiling [15]–[17]. They use hierarchical tiling focuses on fitting the read/write working set on a particular cache level. In contrast, for WET, what matters is only the write working set, and only at the LLC, which is the last cache before NVMM.

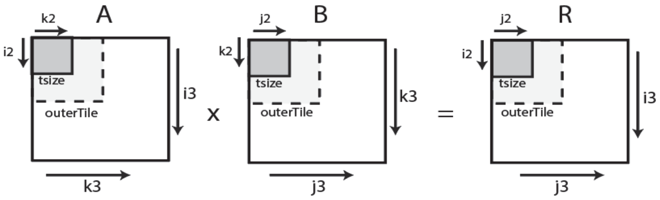
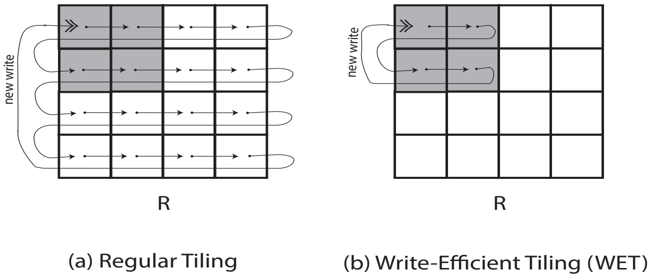


Fig. 4: Illustration of an overview of the matrix multiplication using Write-Efficient Tiling (WET).

Figure 4 illustrates an overview of matrix multiplication using WET. The program is divided into several outerTiles with size *outersize* (shown in light grey), each of these outerTiles is subsequently divided into several innerTiles with size *tsize*. This outerTile level will restrict the movement of the innerTile to only be within the outerTile boundaries. Thus, as the outerTile is designed to fit into the LLC, this restriction increases the likelihood that all the data needed for the multiplication will be contained in the LLC. This significantly reduces the evictions from LLC to NVMM, and thus lowering the number of writes.

To better understand the impact of having this extra level of tiling, Figure 5 illustrates the scheduling of the written innerTiles in the result matrix *R*, for both regular tiling and WET. The figure focuses on the innerTiles visited between writing two consecutive intermediate results to the same output innerTile. As can be observed for the scheduling arrows in Figure 5(a), regular tiling visits and writes to all the output innerTiles in the entire *R* matrix before returning to write on the first output innerTile again. This increases the likelihood that the first innerTile would have already been evicted and written to the NVMM when the innerTile is visited the next time. This is especially likely since the LLC size is much smaller than the entire matrix size (as illustrated in the 4 highlighted grey tiles). On the other hand, WET in Figure 5(b) quickly returns to write again to the first innerTile when it reaches the boundaries of the outerTile. This increases the likelihood that the first tile will be found in the LLC when visited again for the next write, and thus coalesce these multiple writes at the LLC without the need to write them back to the NVMM. After finishing the writes corresponding to the outerTile, the multiplication will move to operate on the next outerTile.



(a) Regular Tiling

(b) Write-Efficient Tiling (WET)

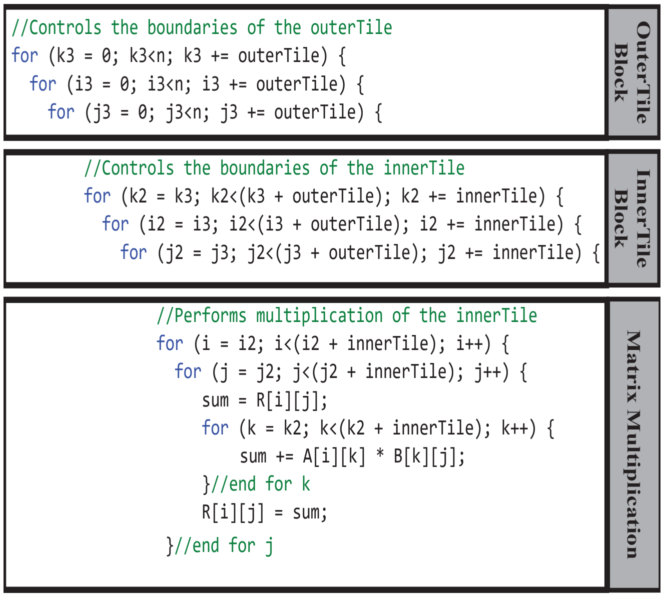
Fig. 5: The scheduling order of the innerTiles in Regular Tiling (a), as opposed to the proposed Write-Efficient Tiling (WET) (b). The grey highlighted innerTiles represent the capacity of the Last Level Cache (i.e. 4 tiles).

Matrix multiplication with WET only affects the scheduling of innerTile operations. Thus, matrix multiplication with WET will have the same number of innerTile operations as in the case with normal Tiled Matrix Multiplication (TMM). Furthermore, the total number of intermediate results that the program writes is identical in both TMM and WET. This guarantees that WET does not lose the performance gains brought by TMM, because it operates on the same innerTile size to achieve similar high locality at the L1 cache. However, exploiting the locality at the Last Level Cache (LLC) using the outerTile helps to coalesce most of the intermediate results generated by innerTile at the LLC before reaching the NVMM. This is fine because the LLC itself is implemented in SRAM or DRAM, hence it does not have a write endurance limitation.

C. Implementation

The code shown in Figure 6 represents an implementation of WET. The most important part of this code is the first part (i.e. outerTile block). The loops in this part (i.e. *k3*, *i3*, and *j3* loops) are responsible for defining the boundaries of the outerTile, which will be used to guide the progress of the rest of the multiplication process. The way these loops work is similar to the one discussed for normal Tiled Matrix Multiplication (Section II). The *k3* loop moves horizontally on the *A* matrix and vertically on the *B* matrix. The *i3* loop moves vertically in matrix *A* and *R*. Finally, the *j3* loop divides the *B* matrix into square tiles of *outersize* dimensions.

The rest of the code in Figure 6 is similar to the code already discussed in Figure 1. However, as shown in the second code block (i.e. InnerTile block), the limiting condition for all the loops is modified to limit their iterations to be only within the boundaries of



```
//Controls the boundaries of the outerTile
for (k3 = 0; k3<n; k3 += outerTile) {
  for (i3 = 0; i3<n; i3 += outerTile) {
    for (j3 = 0; j3<n; j3 += outerTile) {

        //Controls the boundaries of the innerTile
        for (k2 = k3; k2<(k3 + outerTile); k2 += innerTile) {
          for (i2 = i3; i2<(i3 + outerTile); i2 += innerTile) {
            for (j2 = j3; j2<(j3 + outerTile); j2 += innerTile) {

                //Performs multiplication of the innerTile
                for (i = i2; i<(i2 + innerTile); i++) {
                  for (j = j2; j<(j2 + innerTile); j++) {
                     sum = R[i][j];
                     for (k = k2; k<(k2 + innerTile); k++) {
                         sum += A[i][k] * B[k][j];
                     }//end for k
                     R[i][j] = sum;
                 }//end for j
```

Fig. 6: An illustrative code for matrix multiplication using Write-Efficient Tiling (WET).

the current outerTile. For simplicity, the code in Figure 6 is a single-threaded version. Our implementation is multi-threaded, where each thread works on different *j3* outerTiles.

D. Design Decisions when using WET

Although WET can be implemented entirely in software and does not require hardware modifications, achieving good results with WET and exploiting all of its potential requires careful analysis and architecture-level understanding. This is because hierarchical tiling in WET requires tuning the tile sizes based on the number of parallel threads and cache sizes to achieve high locality at different levels of caches.

Assuming a reasonable outerTile size (*outerSize*), increasing the outerTile size reduces the total number of writes to NVMM. This is because almost all the writes within the same outerTile boundaries get coalesced at the LLC and do not reach the NVMM. However, writes reach the NVMM when switching between outerTiles. In other words, the total number of writes to the NVMM is proportional to the ratio of $\frac{N}{outerTile}$, where N is the size of the matrix in each dimension. Thus, the larger the outerTile, the fewer the writes to NVMM. However, there is a limit. If the outerTile size is too large, writes cannot be contained in the LLC and will spill to the NVMM. Hence, we need to find the largest outerTile that does not overflow the LLC.

Such a selection of the outerTile must also consider the number of threads that are running simultaneously sharing the LLC. When dealing with a multi-threaded program, each thread will be working on a different outerTile, which increases the total number of outerTiles that must fit the LLC. To keep the total size of the outerTile within the capacity of the LLC, the outerTile size per thread needs to be reduced. Hence, there is a trade-off between the performance achieved from thread parallelism and the number of writes to the NVMM that is affected by the outerTile size. Section VI investigates these trade-offs in detail.

We suggest the following plan to tune these parameters using the following steps:

1) Choose InnerTile to fit in the L1 cache.
2) Choose the desired number of threads (P) for performance and scalability.
3) Choose *outerSize* such that: $outerSize < \frac{LLCsize}{P}$

If the number of threads is statically determined apriori, independently of desired performance and scalability, the compiler can generate code that corresponds to a specific *outerSize* satisfying the above steps. On the other hand, if the number of threads needs to be selected to get the best trade-off point, these steps may need to be repeated several times in conjunction with recompilation and profiling because the number of threads and *outerSize* interact in determining both scalability and write amplification. In our experiments, two iterations were usually enough to find efficient parameters. Finally, if the code dynamically auto tunes the thread count, the compiler needs to generate multiple outerSize and thread count combinations that will be profiled and selected at run time.

V. Methodology

Our evaluation methodology consists of experimenting on a real machine, which is the default setup. We also evaluated on our simulator that was built on top of *gem5*.

Real-Machine Setup. This part used a machine running 40 Intel Xeon E5-2650v3 Cores, oparating at 2.30 GHz. Further, each core is assigned a private L1 instruction and data cache of 32kB each and a private L2 cache of 256kB; the L3 cache (LLC in this case) is a single, shared write-back cache of 25MB. Finally, a 32 GB DRAM is used as main memory due to NVMM-based hardware systems not being widely available yet for commercial use. Statistics were collected using Intel performance counter monitor.

Simulation Configuration. We extended *gem5* simulator to build an NVM-based system. This system consists of 8 out-of-order cores operating on 2GHz. Further, each core is assigned a private L1 instruction and data cache of 64kB each and a shared L2 cache (LLC in this case) of 512kB. The main memory is NVM-based, with latencies 60ns and 150ns for read and write operations, respectively.

A. Benchmarks

For our evaluation, we study several Matrix Multiplication-based benchmarks, shown in Table I.

TABLE I: Summary of all the benchmarks we evaluated.

Benchmark	Description
TMM	8k-square matrix multiplication using tiling
DAC	2k-squre Divide-And-Conquer multiplication
SNN	32k-input Single-Layer Neural-Networks

In the real machine evaluation, each of these benchmarks was executed from the beginning to the end. For the simulations, it was ensured that the simulation window represents identical progress in all the compared schemes. The simulated window is about $\frac{1}{16}$ of the total benchmark execution. On average, this simulated window reports the timing of 500 million instructions for all threads. In addition, the simulation window is preceded by an average of 300 million instruction for warm-up. The default number of threads for all our experiments is 8.

Our evaluation compares two schemes for performing matrix multiplication. These schemes are: (1) The base multiplication with regular tiling (representing base in most figures), and (2) The proposed write-efficient hierarchical tiling (WET). We also implement non-tiled matrix multiplication (RegMM).

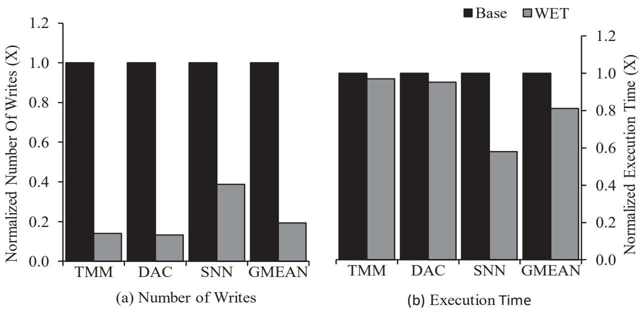
In the evaluated schemes, *Base* represents regular tiled matrix multiplication optimized for performance. For DAC, *Base* represents an optimized recursive divide-and-conquer matrix multiplication. The optimizations include a threshold value that fit in the L1 cache, scheduling of multiplications that ensure good cache locality, and a write optimization via a larger dividing factor [30]. On the other hand, our proposed DAC keeps a dividing factor of two, but with a threshold value that fits in the LLC.

In all schemes, the default values were chosen to provide the best configuration on the experimented setup. The default configuration for the innerTile dimension is 16 elements. The default outerTile is 256 elements for the real-machine and 64 for the simulation evaluation (with TMM dimension of 1k).

VI. Evaluation

We implemented several workloads for performing matrix multiplication, and focus on comparing the execution time and the number of writes using the methodology described in Section V.

Figure 7 compares the number of writes to memory (a) and execution time (b) between our scheme WET (grey) normalized to the Base (black) which uses regular tiling. WET reduces the number of writes in all the studied benchmarks to an average of only 19% compared to Base (ranging from 13.2% to 39%). Interestingly, WET also reduces the execution time to an average of 81% vs. Base (ranging from 58% to 97%) due to better locality at the LLC leading to fewer LLC misses.



(a) Number of Writes (b) Execution Time

Fig. 7: Comparison for the Number of writes to memory (a) and Execution Time (b) for all the studied workloads.

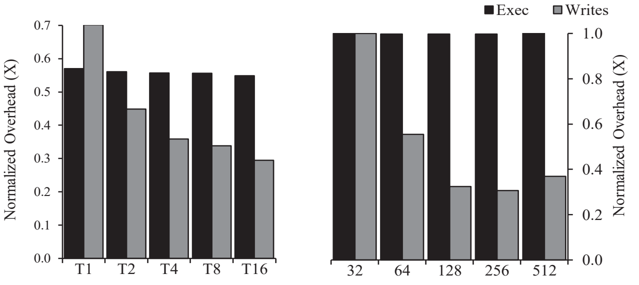
To provide more insights on the shown results, we provide a detailed analysis focusing on the TMM benchmark throughout the rest of this section. Due to adding a tile specific to LLC, We found that using WET improved LLC Miss Rate to be 0.4%, compared to 1.1% in BaseTMM. Which explains the improvement in performance. On the other hand, the instructions needed for the new tiling caused only a 1% increase in total instruction count, which can be easily hidden in modern processors without harming the performance.

A. Sensitivity Study

In this section, we report results for WET's sensitivity to multiple configuration parameters, focusing on the TMM benchmark.

Figure 8(a) shows the overhead on both execution time and the number of writes to WET, normalized to BaseTMM with a corresponding number of threads. This emphasizes the difference in scalability between the two schemes when varying the number of threads. As shown in the figure, introducing another level of tiling didn't add any negative side-effects on scalability, and thus having similar ratio with BaseTMM. Furthermore, WET does better than BaseTMM in number of writes when increasing the number of threads, since choosing reasonable outerTile size for WET mitigates the thread competition on the LLC, which is not protected in BaseTMM. This explains the increase in the difference in the number of writes between Base and WET when increasing the number of threads, as shown in Figure 8(a).

Another factor that has an important impact on WET is the outerTile size. As discussed in Section IV-D, increasing the outerTile size usually comes with a reduction in the number of writes to the NVMM. This is because of exploiting more locality at the LLC, and also helps to divide the entire matrix into a smaller number of outerTile blocks. However, the benefits of increasing the outerTile size are only applicable while the total size of the outerTile blocks by all the threads do not exceed the capacity of LLC. Figure 8(b) illustrates the sensitivity study for WET when varying the outerTile size, normalized to an outerTile size of 32 (leftmost column). As shown in the figure, increasing the outerTile size comes with a reduction in the number of writes (as in outerTile size 32 to 256). However, with outerTile size of 512, the total size of the outerTile blocks from the threads exceed the LLC capacity. Thus, this increased the contention on the LLC and caused more writes to the NVMM. On the other hand, the execution time is not very sensitive to the outerTile size, because all of these setups already have two levels of tiling.

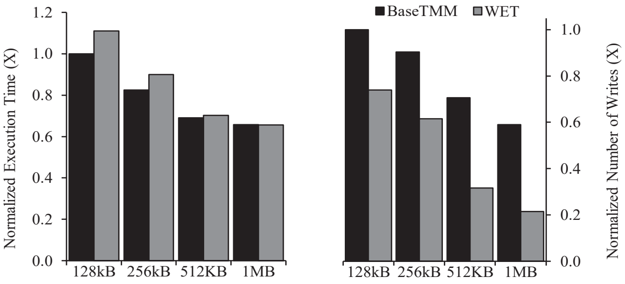


(a) Number of Threads (b) Outer Tile Size

Fig. 8: Sensitivity study of the execution time and the number of writes for WET when varying the number of threads (a) and the outerTile size (b). Each column in (a) is normalized to BaseTMM with a corresponding number of threads. Due to running on lower number of threads on (a), we had to use a different setup (TMM dimension 1k, outerTile 64). All the columns in (b) are normalized to outerTile size 32 (leftmost group).

B. Simulation-Based Evaluation

This section evaluates WET on the simulation-based system described in Section V. Figure 9(a) illustrates the impact on execution time when varying the LLC size on WET compared to BaseTMM. As shown in the figure, increasing the LLC size improves the performance for both of the schemes. Furthermore, WET benefits more from increasing the LLC size due to its better utilization of LLC locality. This can be observed by the reducing gap between the performance of WET and BaseTMM when increasing the LLC size. Similarly, as shown in Figure 9(b) the number of writes for both schemes also decreases when increasing the LLC size. This is because more data can be kept at the LLC without being evicted and written back to the NVMM. Similar to the execution time comparison, the number of writes for WET also benefits more from increasing the LLC size compared to BaseTMM, for the same reason. Moreover, This variation in LLC size explains the difference in the achieved gains for WET in the simulation-based evaluation compared to real-machine evaluation. Due to the long simulation time, we had to simulate a smaller kernel, running on a smaller system (LLC size: 512kB vs. 25MB), as described in Section V.



(a) Execution Time

(b) Number of writes

Fig. 9: Sensitivity study of the execution time (a) and the number of writes (b) when varying the size of the Last-Level-Cache (LLC). All columns are normalized to leftmost column.

Acknowledgement

Solihin was supported in part by NSF grant 1914717 and 1908079, and by UCF. Alshboul and Tuck were supported in part by NSF grant 1717486 and by NC State.

VII. Conclusion

In this work, we made a novel observation that loop tiling, a well-known and effective performance optimization technique, significantly increases the number of writes to NVMM (about 25× for matrix multiplication compared to no tiling). We investigated the resulting tension between performance efficiency vs. NVMM write efficiency. We then advocated *Write-Efficient Tiling (WET)*, a multi-level tiling that adds a new outer tile for the Last Level Cache (LLC) to reduce the number of writes to the NVMM. We evaluated WET across several matrix multiplication benchmarks on both a real machine and the gem5 simulator. Our results show that WET reduces the number of writes on average to only 19% compared to regular tiling, while simultaneously adds 1.23× speedup due to better LLC hit rates.

References

[1] Intel. apache pass. [Online]. Available: https://ark.intel.com/content/www/us/en/ark/products/codename/67560/apache-pass.html

[2] Intel *et al.*, “Intel and micron produce breakthrough memory technology,” 2015.

[3] J. S. Meena *et al.*, “Overview of emerging nonvolatile memory technologies,” *Nanoscale Research Letters*, Sep 2014.

[4] A. Akel *et al.*, “Onyx: A protoype phase change memory storage array,” *HotStorage*, 2011.

[5] X. Dong *et al.*, “Pcramsim: System-level performance, energy, and area modeling for phase-change ram,” in *2009 IEEE/ACM International Conference on Computer-Aided Design-Digest of Technical Papers*, 2009.

[6] B. C. Lee *et al.*, “Architecting phase change memory as a scalable dram alternative,” in *Proceedings of the 36th Annual International Symposium on Computer Architecture*, 2009.

[7] M. K. Qureshi *et al.*, “Improving read performance of phase change memories via write cancellation and write pausing,” in *HPCA - 16 2010 The Sixteenth International Symposium on High-Performance Computer Architecture*, 2010.

[8] M. K. Qureshi, “Pay-as-you-go: Low-overhead hard-error correction for phase change memories,” in *2011 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2011.

[9] G. E. Blelloch *et al.*, “Parallel write-efficient algorithms and data structures for computational geometry,” in *Proceedings of the 30th on Symposium on Parallelism in Algorithms and Architectures*, ser. SPAA ’18. New York, NY, USA: ACM, 2018, pp. 235–246.

[10] G. Blelloch *et al.*, “Efficient algorithms with asymmetric read and write costs,” *arXiv preprint arXiv:1511.01038*, 2015.

[11] S. D. Viglas, “Write-limited sorts and joins for persistent memory,” *Proc. VLDB Endow.*, vol. 7, no. 5, pp. 413–424, Jan. 2014.

[12] A. Buttari *et al.*, “A class of parallel tiled linear algebra algorithms for multicore architectures,” *Parallel Comput.*, vol. 35, no. 1, pp. 38–53, Jan. 2009. [Online]. Available: http://dx.doi.org/10.1016/j.parco.2008.10.002

[13] M. Wolf *et al.*, “A data locality optimizing algorithm,” in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, June 1991.

[14] I. E. Venetis *et al.*, “Mapping the lu decomposition on a many-core architecture: Challenges and solutions,” in *Proceedings of the 6th ACM Conference on Computing Frontiers*, ser. CF ’09, 2009.

[15] L. Renganarayana *et al.*, “Towards optimal multi-level tiling for stencil computations,” in *2007 IEEE International Parallel and Distributed Processing Symposium*, 2007.

[16] D. Kim *et al.*, “Multi-level tiling: M for the price of one,” in *SC ’07: Proceedings of the 2007 ACM/IEEE Conference on Supercomputing*, 2007.

[17] C. Yount *et al.*, “Multi-level spatial and temporal tiling for efficient hpc stencil computation on many-core processors with large shared caches,” *Future Generation Computer Systems*, vol. 92, pp. 903 – 919, 2019.

[18] Y. Jia, “Learning semantic image representations at a large scale,” Ph.D. dissertation, University of California at Berkeley, 2014.

[19] K. Osawa *et al.*, “Accelerating matrix multiplication in deep learning by using low-rank approximation,” in *2017 International Conference on High Performance Computing Simulation (HPCS)*, 2017.

[20] K. Goto *et al.*, “Anatomy of high-performance matrix multiplication,” *ACM Trans. Math. Softw.*, vol. 34, 2008.

[21] R. D. Blumofe *et al.*, “An analysis of dag-consistent distributed shared-memory algorithms,” in *Proceedings of the Eighth Annual ACM Symposium on Parallel Algorithms and Architectures*, ser. SPAA, 1996.

[22] M. Frigo *et al.*, “Cache-oblivious algorithms,” in *Proceedings of the 40th Annual Symposium on Foundations of Computer Science*, ser. FOCS, 1999.

[23] R. D. Blumofe *et al.*, “Dag-consistent distributed shared memory,” in *Proceedings of the 10th International Parallel Processing Symposium*, ser. IPPS, 1996.

[24] M. Alshboul *et al.*, “Lazy persistency: A high-performing and write-efficient software persistency technique,” in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, June 2018, pp. 439–451.

[25] M. Alshboul *et al.*, “Efficient checkpointing with recompute scheme for non-volatile main memory,” *ACM Trans. Archit. Code Optim.*, vol. 16, no. 2, 2019.

[26] H. Elnawawy *et al.*, “Efficient checkpointing of loop-based codes for non-volatile main memory,” in *2017 26th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2017.

[27] S. Shin *et al.*, “Proteus: A flexible and fast software supported hardware logging approach for nvm,” in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, 2017.

[28] S. Pelley *et al.*, “Memory Persistency,” in *Proceedings of International Symposium on Computer Architecture (ISCA)*, 2014.

[29] D. R. Chakrabarti *et al.*, “Atlas: Leveraging locks for non-volatile memory consistency,” in *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, ser. OOPSLA ’14, 2014.

[30] G. E. Blelloch *et al.*, “Sorting with asymmetric read and write costs,” in *Proceedings of the 27th ACM Symposium on Parallelism in Algorithms and Architectures*, ser. SPAA, 2015.

[31] Y. Solihin, *Fundamentals of Parallel Multicore Architecture*. Chapman & Hall, CRC Computational Science, 2015, iSBN-13 978-1482211184.

[32] M. Jimenez *et al.*, “A cost-effective implementation of multilevel tiling,” *IEEE Transactions on Parallel and Distributed Systems*, 2003.

[33] S. Carr, “Memory-hierarchy management,” Rice University, Tech. Rep., 1992.

[34] I. Kodukula *et al.*, “Data-centric multi-level blocking,” in *Proceedings of the ACM SIGPLAN 1997 Conference on Programming Language Design and Implementation*, ser. PLDI ’97, 1997.