M.S. Thesis

Reducing Disk I/O of Temporary Data in Numerical Matrix Computation

행렬의 수학적 연산에서 생성되는 중간 결과
데이터의 Disk I/O 감축

February 2023

Department of Computer Science and Engineering
College of Engineering
Seoul National University

Sohyun Kim

Reducing Disk I/O of Temporary Data in Numerical Matrix Computation

Advisor Bongki Moon

Submitting a master's thesis of Computer Science and Engineering

February 2023

Department of Computer Science and Engineering
College of Engineering
Seoul National University

Sohyun Kim

Confirming the master's thesis written by Sohyun Kim

February 2023

Chair	Sang-goo Lee	(Seal)
Vice Chair	Bongki Moon	(Seal)
Examiner	U Kang	(Seal)

Abstract

Due to an ongoing development of digital society, large-scale numerical array data has become an essential part of data analytics in scientific domain. Among various data analytics algorithms, some can produce temporary data much larger than the size of main memory during the computation. However, existing systems lack support of efficient out-of-core computation which deal with large temporary data.

TilePACK is a package which enables fast out-of-core computations on large array files. It plans an array task as a directed acyclic graph (DAG) to efficiently express the computational pattern of iterative procedures of machine learning algorithms. To utilize a DAG task plan ideally, all temporary data should be kept on memory for reuse. However, this causes unnecessary disk writes and inefficient use of memory space because temporary data which are no longer needed will still be loaded on memory. To address this problem, we propose a method of discarding temporary data from memory as soon as they are no longer needed.

Our proposed scheme has two main contributions. Discarding temporary data helps avoid unnecessary disk writes of temporary data. At the same time, this allows more space of memory to be maintained for other data leading to the reduction of unnecessary disk reads. Through various experiments we have concluded that our scheme takes effect when the size of temporary data produced is bigger than memory capacity. In addition, the effect of our scheme is maximized as the size of input data is smaller than memory space and as the size of temporary data is larger than input data.

Keyword: array data, out-of-core computation, discarding temporary data, query plan in DAG, disk I/O

Student Number: 2021-25515

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1. Background

Due to an ongoing development of digital society, the volume of various data in different formats is growing rapidly. Among various data types, numerical array data analysis in scientific domains is becoming more and more active. For example, in the study of medical field, a large-scale biomedical database UK Biobank [1] provides diverse genetic datasets which are in more than hundreds of gigabytes in size. These datasets are generally expressed as arrays to be used in many applications such as whole genome sequencing or human joint analysis. Furthermore, huge number of customers and products data are produced in near-real time in the e-commerce field. Websites like Amazon or eBay involve daily increase of data such as customer reviews and rating scores and many others. These retail datasets are important sources for building good recommendation models, which commonly utilize matrix factorization receiving matrices and vectors as inputs.

Consequently, numerical computation of large-scale arrays has become an essential part of many data analytic strategies such as machine learning (ML) and statistics algorithms. Many of these algorithms entail iterations of same calculations. Therefore intermediate, or temporary results, are produced for making the final outcomes. In some cases, the amount of generated temporary results is much bigger than the amount of initial input. Moreover, large-scale arrays are now much larger than the size of main memory, making out-of-core computation inevitable. Handling large temporary data in out-of-core computation causes performance degradation in terms of disk I/O. Therefore, how can we efficiently execute on-disk numerical matrix computations which produce a relatively large amount of temporary data?

1.2. Contributions

For most out-of-core computations, time consumed in disk reads and writes is dominant among total elapsed time. Therefore, reducing the amount of disk I/O as much as possible has long been, and still is, a critical homework. In this thesis, we suggest a method to reduce disk I/O of temporary data produced during numerical matrix computation. The method deletes certain temporary data as soon as it is confirmed to be no longer needed.

This scheme expects two contributions. First, discarding temporary data avoids unnecessary disk writes when evicted from fully occupied memory space. At the same time, this method allows larger memory space to be maintained for other data, especially the input data which are repeatedly read many during iterative procedures of a computation, leading to the reduction of unnecessary disk reads. We elaborate more on problem definition and scheme explanation in the next chapters.

1.3. Thesis Roadmap

The rest of the thesis is outlined as follows. We first introduce TilePACK, a package for out-of-core numerical matrix computation as one of the related works in Chapter 2. Chapter 3 elaborates on our problem statement, describes the main points of our proposed new scheme with detailed implementation of how it works. In Chapter 4, we analyze and discuss the experimental results by providing comparison with the existing system. Finally, Chapter 6 concludes this thesis and states some future works.

Chapter 2

Related Works

2.1. Existing Systems

There exist many systems which support numerical array computations. Several user-friendly packages including TensorFlow [2], PyTorch [3], and NumPy [4] support fast linear algebra (LA) operations of arrays which let users design and execute statistical or ML algorithms. However, such LA packages mostly do not support out-of-core computation. In detail, Tensorflow and PyTorch throws out-of-memory error when attempted to compute large matrices that exceed memory capacity. Although NumPy provides a function which enables the mapping of large binary files to the virtual memory space, it purely depends on buffer-caching of the operating system for reading and writing files and therefore causes an enormous disk I/O overhead as the available memory size is low.

Another widely used system SciDB [5] is an array-native database intended primarily for use in application domains that involve very large-scale array data. It also supports many linear algebra functions through their query interface. It adopts some efficient techniques for processing large arrays such as managing arrays in small pieces of chunks for locality and executing queries by pipelining the chunks. However, the system is composed of clusters of servers, which presents significant barriers in implementation and setup for a domain scientist.

2.2. TilePACK

As previously shown, the support for fast out-of-core computations is insufficient among existing systems. In this section, we introduce a numerical array package which efficiently performs numerical calculation of large-scale matrices.

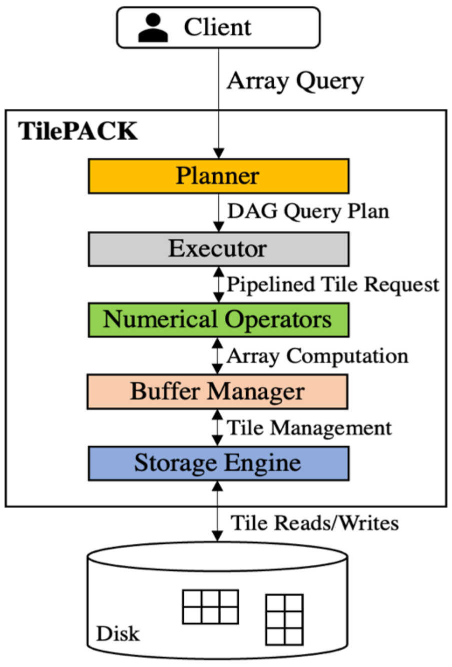

Figure 2.1: Architectural Outline of TilePACK

TilePACK is a package which enables out-of-core computations on large array files. The package has its own storage engine and a buffer manager which manages an array by splitting it into the same size of several data *tiles*. A *tile* is the logical unit of disk I/O of this system which is much smaller than the size of buffer pool. These tiles are loaded on the buffer pool to be used for pipelined numerical computation.

The system is consisted of multiple layers as shown in Figure 2.1. There lies a storage engine at the very bottom which stores array data into many pieces of tiles. On top of the storage engine, a buffer manager is in charge of loading and evicting data tiles during computations. Above the buffer manager, there is the numerical operator module that performs mathematical operations on matrices. Some of the provided operations include basic element-wise arithmetic operations, dot product, transposition, map, etc. Map operation is very similar to the `map()` method in Python, and returns a result tile after applying a user-defined lambda function to each cell of an input tile. Next, the top two layers participate in executing numerical computation tasks queried from a client. To be specific, when

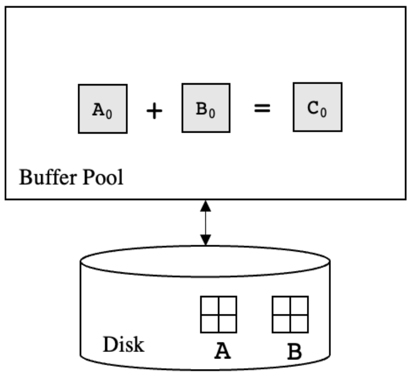

Figure 2.2: Example of Adding Two Matrices in Materialization Method

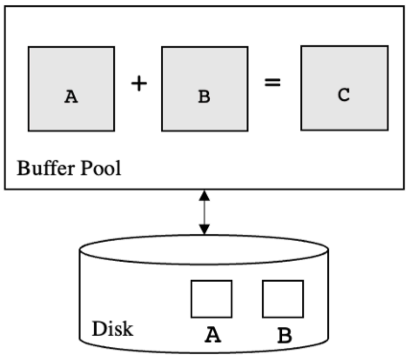

Figure 2.3: Example of Adding Two Matrices in Pipelined Method

a client queries a task, the planner interprets it by creating a directed acyclic graph (DAG) as the task plan [6]. The DAG plan is then passed down to the executor layer to begin the actual pipelined computation. The executor determines the result tiles to be calculated by specifying their coordinates. That is, the tiles become the computational unit of the pipelined execution as well as the unit of disk reads and disk writes.

Some techniques of array processing in TilePACK are referred from some existing array systems such as SciDB and TileDB [7]. Managing large-scale matrices with smaller tiles or chunks for a pipelined query execution is one of them. Consider a simple example of a binary arithmetic operation `A+B=C`, where matrices `A`, `B` and `C` are all equal in data size. Let us define the size value as 4*s for each array. If the operation adopts materialization model [8], or also called blocking model, for adding `A` and `B` as shown in Figure 2.2, total 12*s size of data should be loaded on buffer pool to write the added results on `C`. With materialization method applied, each operator processes and emits bulk input and output matrices all at once. On the other hand, since TilePACK stores matrices in tiles, it only calculates one result tile step by step like the way of volcano model [9]. For the same example task, let us say the matrices are each consisted of four tiles as illustrated in Figure 2.3. Once the executor layer requests the result tile C_0, it is

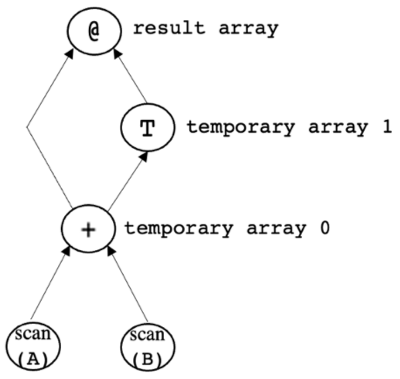

Figure 2.4: An Example of a DAG of Task `(A+B)@(A+B)`^{T}

generated by loading only the necessary tiles A_0, B_0, and C_0 on memory space, which occupies only the size of total 3*s. Obtaining the result array `C` becomes relatively inefficient with blocking method if the main memory space is smaller than 12*s, whereas pipelining method makes it possible by adding two tiles which only occupies 3*s of memory repetitively for four times. To summarize an important point, pipelining execution itself can reduce considerable amount of disk I/O in environments with limited memory capacity.

Lastly, TilePACK applies the buffer eviction policy similar to that of SciDB. It is obvious that systems having their own buffer pools should be able to evict data efficiently when there is no more space available. In TilePACK, the buffer manager first searches for a least recently used (LRU) [10] unpinned clean tiles through a list of tiles in buffer pool from the head and evicts them. This will require no disk writes. If there are no clean tiles left on memory space, the buffer manager loops through unpinned dirty tiles and flushes them. This action will entail disk I/O overhead. In many iterative algorithms of matrix computation, clean tiles are mostly the initial input data and dirty tiles are mostly the intermediate data.

There exists a project named FLAME [11] which utilizes some similar techniques of TilePACK in implementing dense matrix computations of linear algebra. It designs the computation of out-of-core Cholesky factorization [12] as a motivating example by using a system called Supermatrix [13]. Supermatrix also expresses a task plan as a DAG and runs the array task with blocking matrix algorithms. However, FLAME uses a multi-threaded implementation to reduce the cost of disk I/O. In detail, FLAME generates a future list of tasks by looking at the code before execution. Then a "scout" thread inspects the pending block from the list and fetch it into the main memory and a "worker" thread proceeds the actual computation of linear algebra operations by blocks. Data eviction from memory is done with LRU buffer policy. As the multiple threads work concurrently, they state that the overlapping of computation and disk I/O leads to reduction of time. ultimately aiming for efficient parallel execution, which TilePACK currently does not focuses on. That is, TilePACK uses a single thread for running an entire task.

2.3. Query Representation as a Directed Acyclic Graph

One noticeable technique of TilePACK is the adoption of directed acyclic graph structure for expressing a query plan. Each node of a DAG represents a temporary array object formed by a numerical operator which keeps the operator information with it. To help better understanding, the symbol of the operator is written inside each of the node as shown in Figure 2.4 above. Unlike other operators, `scan` operator is an operator implemented in TilePACK for simply opening an existing file, therefore does not create a separate temporary array. A directed edge in a DAG distinguishes the input node and output node. Figure 2.4 illustrates a DAG of the task $(A+B)@(A+B)^T$ where `A` and `B` are two-dimensional arrays. The symbols + stands for element-wise addition, `T` for transposing a matrix, and `@` represents dot product. The two parent nodes at the very bottom, `scan(A)` and `scan(B)` represent the initial input matrices to be opened and loaded on memory. Looking at each outgoing edge from these nodes, we can easily figure out that matrices `A` and `B` are the inputs of element-wise addition. In the same context, the child node named `temporary array 0` which is the result matrix of `A+B,` becomes an input node twice for both `@` and `T` operations.

It is typical in many relational databases to create a query plan into a tree where a child node should only have a single parent. Why does TilePACK adopt a DAG instead of a tree? Many ML algorithms are iterative procedures, involving repetitive utilization of intermediate results as inputs. If we use a tree to express

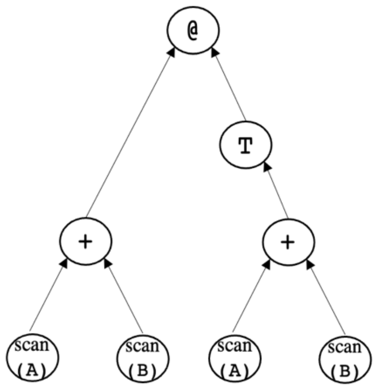

Figure 2.5: An Example of a Tree of Task (A+B)@(A+B)^T

this access pattern, the structure will have many duplicated subtrees. Figure 2.5 is the same task interpreted into a tree plan having an additional subtree of adding A and B on the left bottom side. This causes the executor to compute temporary array 0 twice. It might seem trivial looking at this example task, so let us consider logistic regression algorithm (LR) [14] presented in Algorithm 3. LR calculates a result vector w_i ($i \geq 0$) in ith iteration by using w_{i-1} as input twice which is produced from the previous iteration. This implies that the size of the tree almost doubles as the iteration increments. If the algorithm is iterated for 10 times, the executor must calculate w_1 256 times with the initial input w_0. In short, adopting tree structure to represent an iterative numerical matrix computation causes the planner to create a redundantly bigger graph with many duplicate subtrees. When this task plan is passed down to the executor, identical results will be repetitively produced.

Chapter 3

Method

We now clarify the problem in detail that this thesis is trying to solve and propose a method for solving the clarified problem. First, we introduce a simple logic named *Consumer Count Algorithm* for counting the number of times that each node in DAG is needed as inputs. Next, we elaborate on how this algorithm engages in improving a buffer policy. We use a constant value and a variable for implementing the method. Overall, the scheme enables data tiles to be discarded from the buffer pool as soon as they are no longer required as inputs.

3.1. Problem Statement

TilePACK avoids redundant calculations of temporary data by adopting DAG structure to express task plans. However, a node of DAG can have two or more children, implying that the corresponding matrix can be required for several different numerical operations many times. Therefore, buffer manager should hold the temporary results as long as possible, because the executor has no information of how many child nodes each node has. This causes two problems. First, it may incur unnecessary disk I/O if dirty tiles of temporary matrices are evicted from fully occupied buffer pool during the computation. Second, buffer pool will gradually be filled with data tiles that are no longer needed, causing unnecessary evictions of other data tiles. To avoid these risks, we propose a scheme that can discard a data tile after it is reused as many as it is needed.

3.2. Consumer Count Algorithm

When a client queries TilePACK a numerical computational task, the planner first creates an execution plan of the task as a DAG expression internally optimized with some simple rewriting rules to reduce computational costs. Next the

Algorithm 1: Consumer Count Algorithm

```
  CalculateConsumerCnt (Node : c)
1     if c.visited then
2         return
      end
3     $c.visited \leftarrow true$
4     foreach $p \in ParentNodes(c)$ do
5         $howmany \leftarrow ConsumerCntForOP(c, p, OPtype)$
6         $p.ConsumerCnt \leftarrow p.ConsumerCnt + howmany$
7         CalculateConsumerCnt(p)
      end
```

finalized task plan is passed down to the executor layer where the actual task calculation begins. It is at this point where *Consumer Count Algorithm* is run, before the executor accesses data tiles.

The algorithm is fast and simple. Each array node of a DAG contains a number named `consumerCnt.` Starting from the final output node of a given task graph, we traverse through each node of the graph with depth-first search (DFS) approach and set `consumerCnt` value to the total counting of times each array will be consumed as inputs. Once this value is set, it is never modified.

Referring to Algorithm 1, the counting is done as follows. Shown in lines 1 to 2, the algorithm cleverly calculates `consumerCnt` of a node by ignoring the counting if that node has been visited before, because the result tile can be obtained from a temporary tile without calculating the same tile again through recursive computations with parent tiles. If the node is not visited yet, it is first marked as visited as in line 3 before the actual counting starts. The algorithm calculates `howmany` as the return value of `ConsumerCntForOP` function and adds it to the value of `consumerCnt` for each parent node *p* which is described in lines 4 to 6. The `ConsumerCntForOP(`*c, p, OPtype*`)` method calculates the number of times which node *c* requires node *p* as input. Therefore, the calculation depends on the operation type written inside the node *c*. Pipeline operations such as element-wise arithmetic operations or map operation simply determine `howmany` as 1. This implies that the parent tile is used only once and is no longer needed for producing the result data tile of child node. If the operation is matrix multiplication and the node *p* is the left-hand side parent node, `howmany` is determined as the number of data tiles that makes up a whole column of the right-hand side array node. After incrementing the `consumerCnt` value of *p* as much as the value of `howmany`, `consumerCnt` of *p* is calculated recursively. Here we assume that the `consumerCnt` value is set per array, not data tile, since the tiles belonging to the same array generally become the same inputs during computation.

3.3. Improved Buffer Policy

When the *Consumer Count Algorithm* is run, the calculated `consumedCnt` of each array is passed down to the buffer manager for every data tile. The buffer manager maintains a variable named `consumedCnt` for every tile, which expresses the actual number of how many times it has been consumed. The `consumedCnt` value of a tile is set to 0 in default when the tile is loaded from disk. The buffer manager increments the `consumedCnt` value of the tile by one when the tile used for numerical computation is unpinned. The information of `consumedCnt` is not stored to disk when a tile is evicted. The buffer manager resets the variable to zero when the tile is read again. The buffer manager employs an improved buffer policy by discarding a tile when the `consumedCnt` value becomes the same as the `consumerCnt` value of the array.

Chapter 4

Experiments

4.1. Settings

We have performed various experiments to measure the impact of our proposed scheme on disk I/O. We executed two types of numerical matrix computations on three different systems. The first baseline is the system of NumPy run with `memmap()` method. This function maps large data files to the virtual memory space by internally using `mmap()` function of Linux OS [15]. The other systems are TilePACK implemented with two different buffer policies. We call the second system as *TilePACK Baseline* where the pure LRU buffer policy is used, and the third system as *TilePACK Consumer Cnt* where the improved buffer policy of discarding unnecessary data tiles is applied. As a reminder, pure buffer policy always evicts clean tiles first. To better observe the difference of two buffer policies, TilePACK systems were carried out with direct I/O [16] which bypass the OS buffer cache. Note that NumPy with `memmap()` inevitably utilizes OS caching effect unlike direct I/O. The first system was implemented in Python, and the others were implemented in C.

The two matrix computations are common machine learning algorithms involving iterative procedures. The first task is non-negative matrix factorization (NMF) primarily used for feature extraction [17] and the other is logistic regression (LR) using gradient for binary classification. These algorithms well include basic numerical operations such as element-wise addition, matrix multiplication and transposition. Linear algebra scripts for the algorithms are shown in Algorithm 2 and Algorithm 3 below. We have used the scripts provided from SLAB [18], an open-source array linear algebra benchmark which gives a unified comparative evaluation of the efficiency and effectiveness of various existing systems.

The main goal of our experiments was to evaluate the performance of disk

```
Algorithm 2: NMF
  Inputs: X, factorized W and H, I: number of iterations, r: rank
1 for $i = 1$ to $I$ do
2     $\mathbf{W} \leftarrow \mathbf{W} \odot ((\mathbf{X}\mathbf{H}^T)/(\mathbf{W}\mathbf{H}\mathbf{H}^T))$
3     $\mathbf{H} \leftarrow \mathbf{H} \odot ((\mathbf{W}^T\mathbf{X})/(\mathbf{W}^T\mathbf{W}\mathbf{H}))$
4 return (W,H)
```

```
Algorithm 3: LR with Batched Gradient Descent
  Inputs: X, y, I: number of iterations, $\alpha$: step size
1 for $i = 1$ to $I$ do
2     $\mathbf{w} \leftarrow \mathbf{w} - \alpha \cdot \mathbf{X}^T(\frac{1}{1+e^{-\mathbf{X}\mathbf{w}}} - \mathbf{y})$
3 return w
```

I/O by analyzing the time difference consumed in disk reads and writes of three systems. Time measurement is as follows. We used Linux time command for NumPy with `memmap()` function and obtained disk I/O time by subtracting 'user' time from 'real' time. For TilePACK, we used `gettimeofday()` function for measuring wall clock time and disk I/O time. We summed up all measured values of time consumed in buffer management for disk I/O time. The obtained disk I/O time shall also include CPU time required for system calls, but this is ignorable because time consumed in disk reads and disk writes is dominant.

All experiments were performed on a single computer with CPU (Intel Core i7-9700K) 3.60 GHz, 32 GB RAM and 1 TB of SSD. The machine was operated by Ubuntu Linux 18.04 64-bit with 5.4.0-77-generic kernel version.

4.2. Datasets

We used synthetic array datasets for our experiments. The SLAB benchmark provides an array data generator, so we partially modified this generator script to create raw datasets of synthetic dense matrices and vectors in csv files. The generated files are then converted to npy files for NumPy and to the format that the storage engine of TilePACK supports. Matrix shapes are tall and skinny with far more rows than columns. We have created dense matrices each in size of

N*100 (N by 100), N*10, and 10*100, and dense vectors of N*1 and 100*1 of number of rows by columns. The value of N varies with 10M (million), 20M, 40M and 80M. All arrays are filled with random double numbers.

4.3. Results

For both algorithms where input parameters are fixed, two kinds of experiments were performed. The first type was conducted with a fixed size of memory and different cases of initial data size. Four different sizes of input data are 8 GB, 16 GB, 32 GB and 64 GB. The memory space is fixed as 32 GB for NumPy, and the buffer pool capacity is fixed as 28 GB for TilePACK systems. The other type of experiment was done on the two TilePACK systems, by varying the size of buffer pool with a fixed size of initial dataset.

4.3.1. Non-negative Matrix Factorization (NMF)

NMF needs three matrices X, W and H as the initial inputs where the shapes are N*100, N*10, and 10*100. We have set the number of iterations I to 4, and rank r to 10 as fixed parameters. There is a distinct feature of the intermediate data produced by the presented algorithm. For a single iteration, the size of temporary data exceeds that of initial input data due to matrix multiplication of W and H. This implies that the total amount of temporary arrays during 4 iterations is much larger than the input data size and buffer pool capacity. As you can see in Table 4.1, the total size of intermediate data grows in a large extent as the initial input data size grows.

We first observe the results of experiments performed on varied input dataset sizes. Figure 4.1 shows the total elapsed time and Figure 4.2 shows disk I/O time consumed in NMF computation for the three systems. Both bar charts show identical tendency in the difference of spent time which implies that disk I/O time is the dominant part of out-of-core computation. Therefore, we focus on analyzing disk I/O time. First of all, NumPy with memmap() function takes the longest time in disk I/O. Not only does this system process the calculation in a materialization

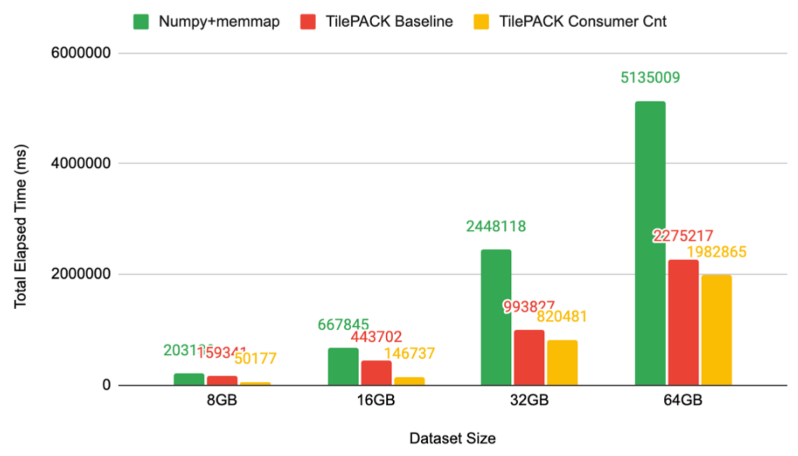

Figure 4.1: Total Elapsed Time of NMF for Varied Size of Inputs

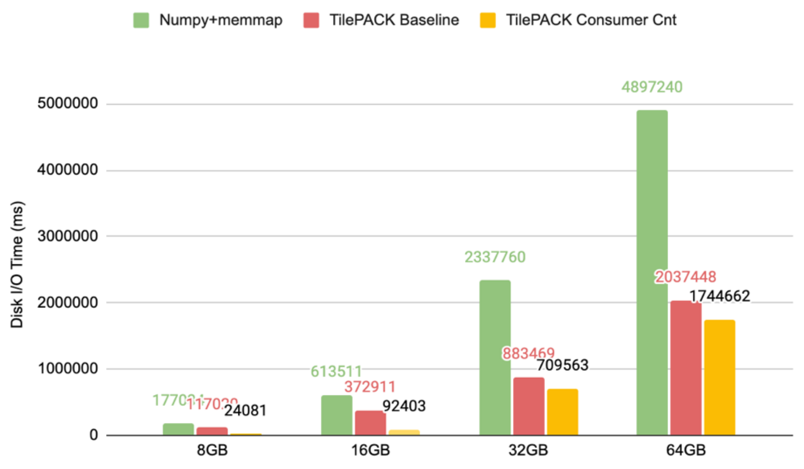

Figure 4.2: Disk I/O Time of NMF for Varied Size of Inputs

method, but also using `memmap()` function on large files causes consistent disk reads and writes for data which exceed the memory capacity. On the other hand, the systems of TilePACK are both much faster in overall than NumPy due to its pipelined computation and efficient buffer management. Next, we should look at the red and yellow bars colored in the stated figures. From the comparison of disk I/O time in TilePACK Baseline and TilePACK Consumer Cnt, we can discover at once that the latter system which discards no more needed temporary data during

(GB)			(%)
Input Data	**Temporary Data**	**Discarded Data**	**Discarded Rate**
8	45	42	93%
16	90	83	92%
32	180	115	64%
64	361	148	41%

Table 4.1: Discarded Temporary Data in TilePACK Consumer Cnt (NMF)

(GB)	**TOTAL READS**	
Input Data	**TilePACK**	**TilePACK Consumer Cnt**
8	43	8
16	130	46
32	323	322
64	770	742

Table 4.2: Total Amount of Disk Reads in TilePACK Systems (NMF)

(GB)	**TOTAL WRITES**	
Input Data	**TilePACK**	**TilePACK Consumer Cnt**
8	15	0
16	66	0
32	150	60
64	331	203

Table 4.3: Total Amount of Disk Writes in TilePACK Systems (NMF)

the computation is faster in all four cases. In the experiments of the first two sizes of varied input datasets, the improvement is remarkable. With the two result graphs, we can calculate the improvement rate by using a formula of $(t_1-t_2)/t_2$, where t_1 is the disk I/O time in TilePACK Baseline and t_2 is the disk I/O time in TilePACK Consumer Cnt. The improvement rate in speed is obtained as follows: 386 %, 303 %, 25 %, and 17 % in each 8 GB, 16 GB, 32 GB, and 32 GB input dataset sizes. That is, the improved buffer policy is about 5 times faster, and about 3.6 times faster than pure LRU policy in the aspect of Disk I/O. Referring to Table 4.1 again, as the size of initial input dataset increases, the proportion of the data discarded compared to temporary data gradually decreases. But still, note that the absolute size of discarded data increases. The higher the discarded rate, the higher the improvement rate. What are the reasons? In a pipelined out-of-core computation where temporary data is consistently produced, the buffer manager using pure LRU policy will start to evict data tiles from the moment the buffer pool is fully occupied. Before we continue explaining why the improved buffer policy is faster, let us introduce two keywords. While the buffer capacity is fixed to a static size and the initial input size grows in each of the four cases, larger amounts of temporary results are be produced and this will cause more frequent evictions of data tiles. For clean tiles including data of initial input matrices or temporary data which are flushed to disk, consistent eviction and loading on the buffer pool will repeatedly occur throughout the iterations to reuse them when required as inputs. We call this phenomenon *read amplification*. The other keyword is simple and straightforward. During the computation, tens of gigabytes of temporary results are discarded in TilePACK Consumer Cnt system. This action likely avoids unnecessary disk writes of temporary data tiles that are no longer needed to stay in buffer space. We name this effect *write avoidance*.

In Table 4.2, from the fact that the total amount of disk reads has decreased in all four cases by applying our scheme, we can derive a point that write avoidance helps to not only reduce disk writes, but also reduce repetitive disk reads of clean data tiles. To be specific, most of these data are the initial input data. Furthermore, we can observe from Table 4.3 that all temporary results are discarded in the first two cases, and that the proportion of the temporary data written on disk increases. All things considered, we can briefly state that read amplification is relatively not heavy and the impact of write avoidance is maximized in the first two cases, leading to a successful reduction of the overall disk I/O. For the next two cases of 32 GB and 64 GB input sizes, read amplification counterbalances the write avoidance effect.

Now we look at the second experiment performed on TilePACK systems. For 8 GB of initial input dataset, we have varied the buffer pool size in 7 GB, 14 GB, and 28 GB. This setting makes a scenario where the total data size is fixed while the buffer pool capacity increases. In line with what we have analyzed so far, the eviction of data tiles occurs less frequently as the size of buffer pool becomes bigger. In other words, read amplification decreases and at the same time, write avoidance increases, finally resulting in larger amount of reduction of disk I/O. The improvement rates obtained other than the last case (the result overlaps with the first experiment) are as follows: 25 % and 236 % in each 7 GB and 14 GB size of buffer pool. To sum up the key point from the experiments of NMF algorithm, the

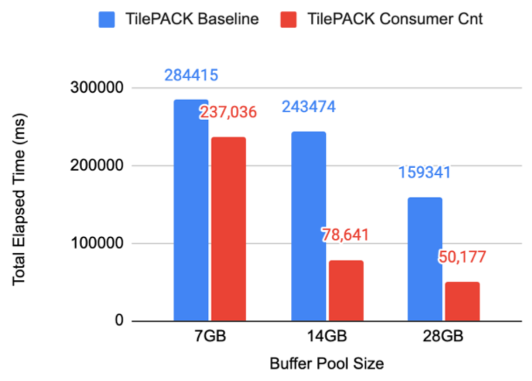

Figure 4.3: Total Elapsed Time of NMF for Varied Size of Buffer Pool

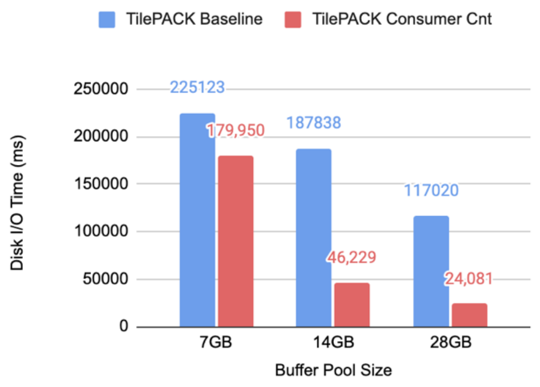

Figure 4.4: Disk I/O Time of NMF for Varied Size of Buffer Pool

effect of the improved buffer policy is maximized under two conditions. First, the size of initial input data should be smaller than the buffer pool capacity. Next, the size of temporary result data should be much larger than the initial input data.

4.3.2. Logistic Regression (LR)

LR receives a matrix `X` and two vectors `y` and `w` as the initial inputs where the shapes are `N*100, N*1,` and `100*1`. We have set the number of

(GB)			(%)
Input Data	**Temporary Data**	**Discarded Data**	**Discarded Rate**
8	2.5	2.5	100%
16	5	5	100%
32	10	10	100%
64	20	20	100%

Table 4.4: Discarded Temporary Data in TilePACK Consumer Cnt (LR)

iterations I as 10 and step size α as 0.000001 for the parameters. The distinct feature of the experiment on this algorithm is as follows. For a single iteration, the size of temporary data produced is relatively very small compared to that of initial input data due to dot product. Therefore, the total size of temporary arrays during 10 iterations is much smaller than both the input data size and the buffer pool capacity. In Table 4.4, the size of temporary data doubles as the initial input sizes doubles, and the amounts are very small in all four cases. This makes the first two cases in-memory computation, and the last two cases out-of-core computation.

Let us observe the results of experiments performed on varied input dataset sizes. In fact, there is not much to analyze from the results obtained compared to the previous algorithm, but we do have some important discoveries to state. Figure 4.5 shows the total elapsed time and Figure 4.6 shows disk I/O time consumed in LR computation for the three systems. Again, both bar charts show identical tendency in the difference of spent time, so we focus on disk I/O time. In the first two in-memory computations, the three systems show relatively similar duration which is somewhat an obvious result. In-memory computation will only involve the disk reads of the initial inputs once, and disk writes for nothing but final results. In addition, two other systems of TilePACK are both faster than NumPy due to the same reason stated for the previous algorithm. Next, from the comparison of disk I/O time in TilePACK Consumer Cnt and TilePACK Baseline, we can easily notice that there exists almost no difference in both total elapsed time and disk I/O time. Here we have made an important discovery. In the previous chapters, we have explained and emphasized how the pure LRU buffer policy is

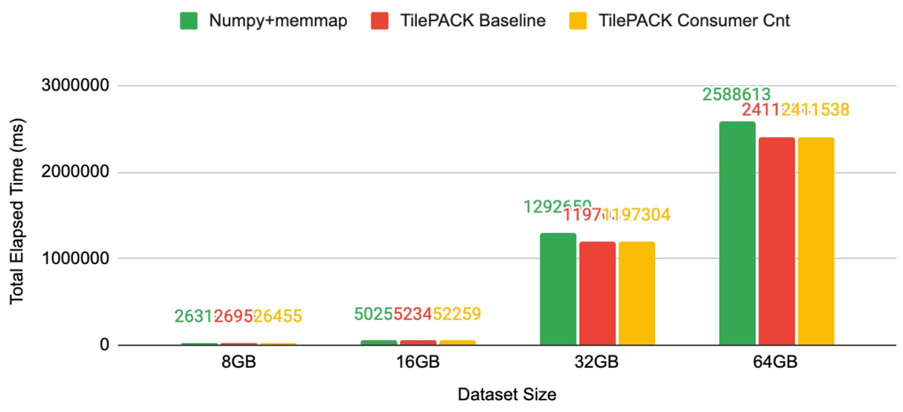

Figure 4.5: Total Elapsed Time of LR for Varied Size of Inputs

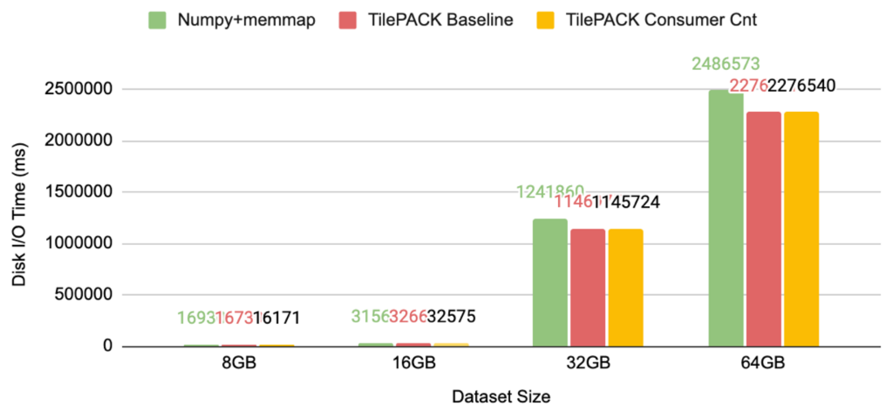

Figure 4.6: Disk I/O Time of LR for Varied Size of Inputs

implemented in TilePACK. The buffer manager will always evict clean tiles first. When total amount of temporary result data is smaller than the buffer pool, there always are clean data tiles of the initial input matrices or vectors loaded on the pool. In this scenario, due to the eviction pattern of pure buffer policy, only the clean tiles will be evicted from and reloaded on the memory space because the buffer manager will always skip the dirty tiles of temporary results as long as any clean tiles exist. Referring to Table 4.5 and Table 4.6, we can observe the fact that in all four cases the effect of write avoidance is 100%. The reason that total amount of disk reads grows rapidly in the last two out-of-core cases is because the algorithm

(GB)	TOTAL READS	
Input Data	TilePACK	TilePACK Consumer Cnt
8	8	8
16	16	16
32	640	640
64	1280	1280

Table 4.5: Total Amount of Disk Reads in TilePACK Systems (LR)

(GB)	TOTAL WRITES	
Input Data	TilePACK	TilePACK Consumer Cnt
8	0	0
16	0	0
32	0	0
64	0	0

Table 4.6: Total Amount of Disk Writes in TilePACK Systems (LR)

iterates for 10 times, requiring the input matrix X to be repeatedly read many times. The only difference in the two systems of TilePACK is whether the buffer pool has the remaining temporary results before evicting after the computation is finished. In other words, after the 10th iteration is over, there remains the exact size of temporary data on the buffer pool in TilePACK Baseline system. In TilePACK Consumer Cnt system, there are no remaining dirty tiles because all the temporary results have been discarded throughout the computation. Because the eviction of dirty tiles after the computation is done does not entail disk writes, there is no time delay as well. For example, for the case where the initial input data size is 32 GB, the time consumed for evicting the remaining 10 GB of temporary tiles with no disk writes at the end will have no impact on disk I/O time. We can also derive another key point in the analysis of experiment results on LR algorithm. The reduction of disk I/O of the improved buffer policy takes effect only when the size of temporary result data is bigger than the buffer pool capacity.

Based on the analyses of the experiment results on NMF and LR algorithms, we can summarize the effect of our task processing scheme as follows. In an environment where there is a limited size of buffer pool and a buffer policy which always evicts clean data first, the reduction of disk I/O is maximized as the size of temporary data is much larger than that of the initial input data and as the size of initial input data is smaller than the buffer capacity, under the condition that the size of temporary data produced during computations is bigger than the buffer capacity.

We have done additional experiments by varying the environment. We have performed the computations also on a 4 TB of hard disk drive (5400 RPM). As the throughputs of the device differ from those of SSD, we were able to obtain better improvement rates. We have also tried increasing the parameter `I` (number of iterations), which has the effect of producing larger amount of temporary data with the same size of input data. As expected, we could achieve higher improvement on the time spent on disk I/O and therefore we were able to further strengthen our key points. We attached the results of these experiments in the Appendix section.

Chapter 5

Conclusion

In this thesis, we deal with enhancing the performance of disk I/O in out-of-core computations of numerical array data. Since existing systems generally do not support efficient large-scale array computations, we first introduced an array package which enables fast out-of-core array computations and solved a problem incurred from handling array task plans as directed acyclic graphs.

Given a DAG task plan, we proposed a scheme that calculates the numbers of times each array will be used as inputs before the execution of the actual numerical operations. The scheme therefore enables a query planner to represent a task plan as a simple DAG instead of a tree that has many duplicate subtrees. This leads to the avoidance of inefficiently calculating a same result data more than once. In addition, the scheme is applied to an existing buffer policy, which helps the reduction of unnecessary disk reads and disk writes.

Through various experiment results, we have found an important discovery of the effect of our scheme. In an environment where a buffer policy which always evicts unpinned clean data first, the effect of reducing disk I/O occurs when the size of temporary data produced during computations is bigger than the buffer pool capacity. The system which our scheme is applied is faster in various test cases in the aspect of disk I/O compared to the original system.

The significance of our proposed scheme is very clear. Optimizations for processing too large data on a single machine have their limitations. In addition, dealing with too small data can be a trivial problem. In this context, our method is stemmed from a simple idea and yet has explicit and considerable improvement in out-of-core computations of moderately large datasets. We expect that our scheme is generally appliable on any system which has a buffer manager and expresses a query or task plan as a DAG. It is because our main idea is to avoid unnecessary disk writes that can occur from a traditional buffer management where temporary data produced during a computation will be marked dirty and unconditionally be written to disk when evicted later. Although the FLAME project mentioned earlier as one of the related works computes arrays in materialization model and therefore our proposed method is not likely to incur dramatic improvement, it can be a valid example of the appliable system.

We can imply that our scheme can yield different patterns of result if we use different existing buffer policies. Therefore, we can further try using LRU-k [19] or MRU buffer policies to discover different results. Furthermore, the Co*nsumer Count Algorithm* can be improved as a future work in the part where it calculates `consumerCnt` value per array. In other words, complicated matrix operations may need a more sophisticated counting algorithm. For example, different value of `consumerCnt` for each data tile belonging to a single array is needed for planning LU decomposition [20] as a pipelined method, which is generally used for solving linear equations for matrices.

Chapter A

Appendix

To confirm our discoveries, we attach the results of additional experiments in this chapter. First, we show the experiments done with modified parameter of I (number of iterations). The next section briefly shows results performed on a 4 TB of hard disk drive (5400 RPM) instead of SSD.

A.1. Experiments with Modified Parameters

The purpose of this experiment is to observe the results when a bigger size of temporary data is produced with a fixed size if initial input dataset. One simple way to achieve this is to increase the number of iterations. For our second algorithm, we have tried increasing the parameter `I` from 4 to 7. The size of temporary data in this experiment is shown in Table A.1, as well as total amounts of disk reads and writes in Tables A.2 and A.3. All the other settings are the same, and the difference of total elapsed time and Disk I/O time shows the equal tendency. We therefore attach the result of Disk I/O time difference among the three systems as Figure A.1.

(GB)			(%)
Input Data	**Temporary Data**	**Discarded Data**	**Discarded Rate**
8	79	73	92%
16	158	146	92%
32	316	199	63%
64	630	258	41%

Table A.1: Discarded Temporary Data in TilePACK Consumer Cnt with Increased Iterations (NMF)

(GB)	TOTAL READS	
Input Data	**TilePACK**	**TilePACK Consumer Cnt**
8	90	8
16	227	88
32	566	565
64	1347	1300

Table A.2: Total Amount of Disk Reads in TilePACK Systems (NMF, 7 Iterations)

(GB)	TOTAL WRITES	
Input Data	**TilePACK**	**TilePACK Consumer Cnt**
8	49	0
16	129	0
32	285	110
64	600	363

Table A.3: Total Amount of Disk Writes in TilePACK Systems (NMF, 7 Iterations)

The improvement rate in speed is obtained as follows: 993 %, 302 %, 27 %, and 18 % in each 8 GB, 16 GB, 32 GB, and 64 GB input dataset sizes. That is, our improved buffer policy enhances the performance by enabling the disk I/O time to be 10 times faster when the size of initial input data is 8 GB. Here we have

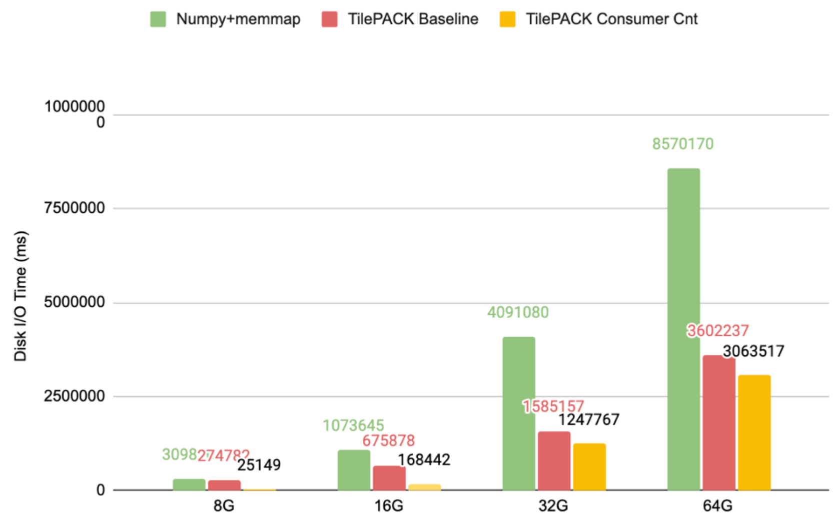

Figure A.1: Disk I/O Time of NMF with Increased Iterations for Varied Size of Inputs

observed an important fact that the improvement rate generally becomes higher when the input dataset is relatively smaller than the buffer pool size and the temporary data is produced in a much larger amount. In addition, although the improvement rate has not increased in the experiments of 16 GB, 32 GB, and 64 GB input data, the rate is steadily maintained. We therefore can conclude that our proposed method takes effect in all four cases, but the effect is significantly maximized in the first case.

To even further concretize the main points that, we attach the experiment results of varied buffer pool sizes for the 8GB input case. The analysis is not much different from the results shown in Chapter 4. The improvement rates obtained other than the last case (the result overlaps with the first experiment) are as follows: 27 % and 322 % in each 7 GB and 14 GB size of the buffer pool. Note that the improvement rate has slightly increased in the first case where the size of input data (8 GB) and the buffer pool size (7 GB) has not big difference. On the other hand, the improvement rate has increased conspicuously in the second case where the buffer pool (14 GB) is bigger than the input data.

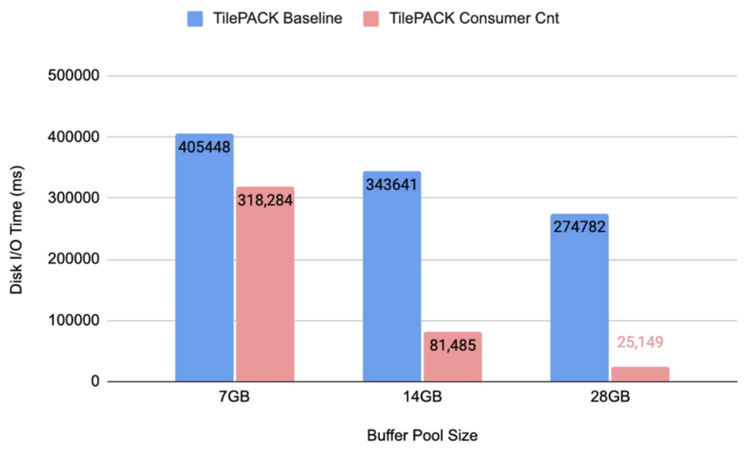

Figure A.2: Disk I/O Time of NMF with Increased Iterations for Varied Size of Buffer Pool

A.2. Experiments on Different Storage Device

For sequential reads and writes, the SSD used in our experiments is known to have the throughput of around 550 Mbps and 520 Mbps (Samsung 860 Pro 1TB). The HDD which we have tested is known to have read throughput of 160Mbps and write throughput of 120 Mbps (Western Digital Blue 4TB). Performing the identical experiments on HDD instead of SSD was due to the expectation that as the speed of disk reads is much faster than the speed of disk writes, the improvement rate will increase [21]. Our expectations were correct, and therefore we attach the performance results of NMF algorithm in this section. Again, we have excluded the results of total elapsed time results.

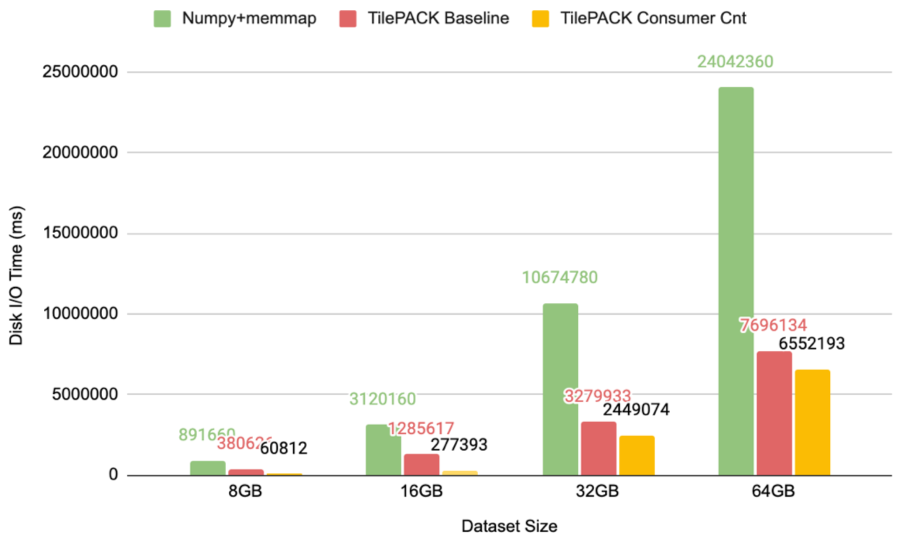

Figure A.3: Disk I/O Time of NMF for
Varied Size of Inputs (4 Iterations, HDD Device)

The improved rate of disk I/O time from TilePACK to TilePACK Consumer Cnt system shown in Figure A.3 is as follows: 526 %, 363 %, 34 %, 17 % in each 8 GB, 16 GB, 32 GB, and 64 GB input dataset sizes. The improvement rate from the

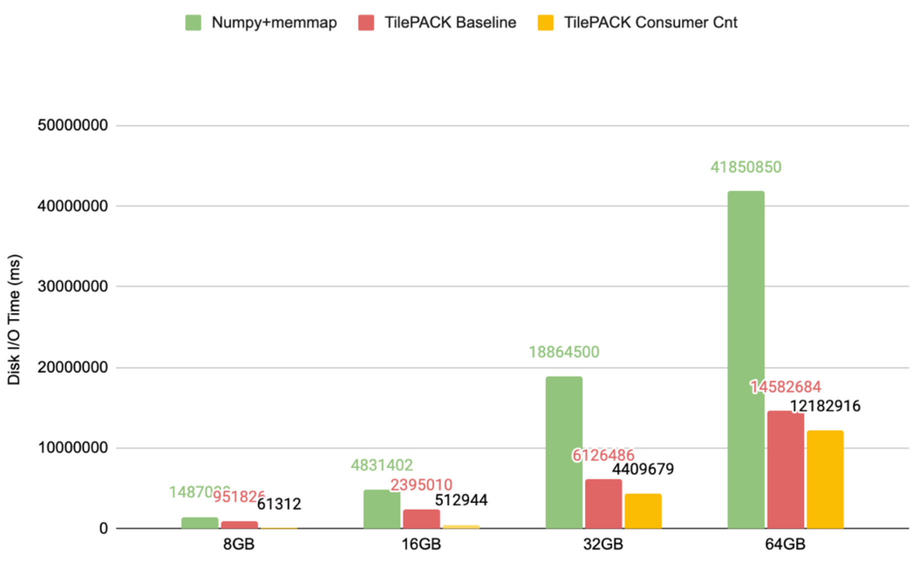

Figure A.4: Disk I/O Time of NMF for

Varied Size of Inputs (7 Iterations, HDD Device)

experiment above is as follows: 1452 %, 367 %, 39 %, 20 % in each 8 GB, 16 GB, 32 GB, and 64 GB input dataset sizes. Regarding all the attached results, we can conclude that our proposed scheme can show better results when the algorithms are performed on storage devices where read throughput is much faster than the write throughput.

Bibliography

[1] UK Biobank: a large-scale biomedical database and research resource. https://www.ukbiobank.ac.uk/

[2] End-to-end machine learning platform. https://www.tensorflow.org/

[3] An open-source machine learning framework. https://pytorch.org/

[4] A fundamental package for scientific computing. https://numpy.org/

[5] Michael Stonebraker, Paul Brown, Alex Poliakov, and Suchi Raman. 2011. The Architecture of SciDB. *In Scientific and Statistical Database Management*, 1–16.

[6] Thomas Neumann and Guido Moerkotte. 2009. Generating Optimal DAG-structured Query Evaluation Plans. *Computer Science - Research and Development* 24, (2009), 103–117.

[7] Stavros Papadopoulos, Kushal Datta, Samuel Madden, and Timothy Mattson. 2016. The TileDB Array Data Storage Manager. *Proceedings of the VLDB Endowment* 10, 4 (November 2016), 349–360.

[8] Thomas Neumann. 2011. Efficiently Compiling Efficient Query Plans for Modern Hardware. *Proc. VLDB Endow.* 4, 9 (2011), 539–550.

[9] Goetz Graefe. 1994. Volcano-An Extensible and Parallel Query Evaluation System. *IEEE Transactions on Knowledge and Data Engineering* 6, 1 (1994), 120–135.

[10] Marek Chrobak and John Noga. 1999. LRU is Better than FIFO. Algorithmica 23, 2 (1999), 180–185.

[11] Mercedes Marqués, Gregorio Quintana-Orti, Enrique S Quintana-Orti, and Robert A van de Geijn. 2009. Solving “Large” Dense Matrix Problems on Multi-Core Processors. *In 2009 IEEE International Symposium on Parallel & Distributed Processing*, 1–8.

[12] Nicholas J Higham. Cholesky factorization. Wiley interdisciplinary reviews: computational statistics 1, 2 (2009), 251–254.

[13] Ernie Chan, Field G. Van Zee, Paolo Bientinesi, Enrique S. Quintana-Orti, Gregorio Quintana-Orti, and Robert van de Geijn. 2008. SuperMatrix: A Multithreaded Runtime Scheduling System for Algorithms-by-Blocks. *In Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming,* 123–132.

[14] Jorge Nocedal and Stephen J Wright. 1999. Numerical Optimization (Second ed.). Springer.

[15] mmap(2) - Linux manual page. https://man7.org/linux/man-pages/man2/mmap.2.html

[16] Peter Wai Yee Wong, Ric Hendrickson, Haider Rizvi, and Steve Pratt. 2006. Performance Evaluation of Linux File Systems for Data Warehousing Workloads. *In Proceedings of the 1st International Conference on Scalable Information Systems*, 43–50.

[17] Nicolas Gillis. 2017. Introduction to Nonnegative Matrix Factorization. arXiv:1703.00663 (2017).

[18] Anthony Thomas and Arun Kumar. 2018. A Comparative Evaluation of Systems for Scalable Linear Algebra-based Analytics. *Proceedings of the VLDB Endowment* 11, 13 (2018), 2168–2182.

[19] Elizabeth J. O'Neil, Patrick E. O'Neil, and Gerhard Weikum. 1993. The LRU-K Page Replacement Algorithm for Database Disk Buffering. *In Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, 297–306.

[20] Saeid Abbasbandy, Reza Ezzati, and Ahmad Jafarian. 2006. LU decomposition method for solving fuzzy system of linear equations. Applied Mathematics and computation 172, 1 (2006), 633–643.

[21] Sanam Shahla Rizvi and Tae-Sun Chung. 2010. Flash SSD vs HDD: High Performance Oriented Modern Embedded and Multimedia Storage Systems. *In 2010 2nd International Conference on Computer Engineering and Technology* 7, 297–299

초록

정보 사회의 지속적인 발전으로 인해 점점 더 큰 규모의 수치 데이터를 배열 형태로 활용하는 데이터 분석이 활발해 지고 있다. 일부 반복 학습을 진행하는 머신 러닝 알고리즘은 연산 과정에서 중간 결과, 즉 임시 데이터가 많이 생성될 수 있다. 그렇기에, 메모리 크기를 넘어가는 매우 큰 배열 데이터의 아웃 오브 코어 (out-of-core)의 특징을 띠는 수학적 연산이 많이 필요해 지는데, 현재 이를 지원하는 시스템은 부족한 실정이다.

TilePACK은 위에서 언급한 배열 연산을 가능하게 하는 패키지이다. 이 시스템은 여러 번 재사용이 될 같은 데이터를 여러 번 계산하는 비효율성을 막기 위해 태스크 (task) 플랜을 트리가 아닌 유향 비순환 그래프 (DAG) 의 구조로 표현한다. 그러나 DAG를 활용하게 되면 한번 계산된 중간 결과를 재사용하기 위해 해당 데이터가 메모리 상에서 유지되는데, 더 이상 필요하지 않은 경우에도 계속 남아있기 때문에 불필요한 디스크 쓰기와 메모리 공간의 비효율적인 사용으로 이어진다. 이 문제를 해결하기 위해 임시 데이터가 더 이상 필요하지 않음이 확인되는 즉시 폐기하는 메소드를 제안한다.

우리가 제안한 계획은 크게 두 가지 기여를 한다. 필요 없는 임시 데이터를 삭제하면 해당 데이터의 불필요한 디스크 쓰기를 피할 수 있다. 동시에, 다른 데이터를 위한 더 많은 메모리 공간을 확보할 수 있게 한다. 이는 결과적으로 모두 연산에 소요되는 시간을 줄이는 효과를 보인다. 다양한 실험을 통해 우리가 제안한 메소드는 임시 데이터의 크기가 메모리 용량보다 클 때 효과가 나타나고, 입력 데이터의 크기가 메모리 용량보다 작고 임시 데이터가 입력 데이터보다 크기가 클수록 효과는 최대화 됨을 확인하였다.