Cache Complexity of Cache-Oblivious Approaches: A Review and Extension

Inas Abuqaddom, Sami Serhan, Basel A. Mahafzah
Department of Computer Science
King Abdullah II School of
Information Technology
The University of Jordan
Amman, Jordan

***Abstract*—The latest direction in cache-aware/cache-efficient algorithms is to use cache-oblivious algorithms based on the cache-oblivious model, which is an improvement of the external-memory model. The cache-oblivious model utilizes memory hierarchies without knowing memories' parameters in advance since algorithms of this model are automatically tuned according to the actual memory parameters. As a result, cache-oblivious algorithms are particularly applied to multi-level caches with changing parameters and to environments in which the amount of available memory for an algorithm can fluctuate. This paper shows the state of the art in cache-oblivious algorithms and data structures; each with its complexity concerning cache misses, which is called cache complexity. Additionally, this paper introduces an extension to minimize the cache complexity of neural networks by applying an appropriate cache-oblivious approach to neural networks.**

***Keywords*—Cache complexity; cache-oblivious algorithm; memory hierarchy; neural network**

I. Introduction

The processor speed is much faster than the main memory speed. The impact of this hole can be decreased by utilizing a hierarchy of multi-level caches in an effective way between the processor and the main memory [1]. Thus, modern computers have a memory hierarchy to speed up accessing memory, such that the speed of accessing a memory level becomes slower as its size becomes larger. Table 1 shows an example of a memory hierarchy. In most processors, the level 1 cache (L1) is on the same chip as the CPU, whereas the level 2 cache (L2) is on a separate chip [2], [3], [4].

TABLE I. Memory Hierarchy

Memory level	Size	Response time
CPU registers	around 100B	around 0.5ns
L1 Cache	around 64KB	around 1ns
L2 Cache	around 1MB	around 10ns
Main memory	around 2GB	around 150ns
Hard disk	around 1TB	around 10ms

The response time increases as the memory size increases, as shown in Table I, consequently, there is an inverse relationship between memory size and its speed; as the memory level becomes larger, its speed becomes slower. Generally, each memory level communicates directly with the slower directly connected memory level. Additionally, data is transferred in blocks to reduce the effects of slow access [5]. For example, to read one element of an array, the main memory will additionally transmit a "block" of consecutive words. Thus, accessing the other words of the transferred block is free in terms of memory transfers. The design of a multi-level memory hierarchy is more complex compared with single-level memory; such as extra design decisions are needed for each level of the memory hierarchy [6].

One performance metric of an algorithm is called memory performance, which measures the utilization of the memory hierarchy in an algorithm. Thus, algorithms have to know the memory hierarchy, memory size, and block size to achieve high memory performance; these algorithms are called cache-aware algorithms or cache-efficient algorithms. Nevertheless, cache-aware algorithms reduce running time up to 50% compared with other algorithms. On the other hand, algorithms, that worry about memory performance, have to be tuned according to the underlying cache size. The big question here is: "Can we design algorithms to efficiently utilize memory hierarchy, without knowing the underlying cache size?" The answer is yes by using cache-oblivious algorithms [7].

Neural networks greedily consume the memory hierarchy, hence there is a strong need to design neural network algorithms efficiently in terms of memory performance [8], [9]. In this paper, we show the recent direction of the memory model, which is the cache-oblivious model, and the state of the art in cache-oblivious algorithms and data structures. Furthermore, we applied the appropriate cache-oblivious approach to neural networks to improve their memory performances.

The rest of this paper is organized as follows: Sections II and III explain the cache-oblivious model and cache complexity, respectively. Section IV discusses a set of cache-oblivious algorithms, while Section V introduces the extension, which is cache-oblivious neural networks. Finally, we provide a summary of the best lately known cache-oblivious algorithms in Section VI.

II. Cache-Oblivious Model

The literature discusses two memory-hierarchy models; the external-memory model and the cache-oblivious model. The memory hierarchy of the external-memory model consists of two levels; cache and disk. Such that, cache refers to the near memory to the CPU as disk refers to the far memory from the CPU. The basic unit of transferring data between two memory levels is a block, so the memory is divided into blocks of B

words (a certain number of bytes). At most, the memory of size M can store $\frac{M}{B}$ blocks from data of size N words [10], [11].

To reduce the number of transferring blocks, cache-aware algorithms are designed based on the external-memory model; such algorithms worry about the existence of two levels of memory and their parameters. Cache-aware algorithms perform efficiently with two levels of memory, in terms of memory transfers. But in reality, memory hierarchy is more than just two levels of memory. To cope with this fact, the cache-oblivious model is introduced as an extension to the external-memory model [12], [11].

The cache-oblivious model utilizes all cache levels efficiently without tuning, which means algorithms can utilize the memory without knowing its size M and block size B. Cache-oblivious model is the best choice for the environment that provides a fluctuation amount of available memory for an algorithm. Better algorithms for this model can be better for any possible values of M and B. Cache-oblivious algorithms are particularly helpful for multi-level caches and for caches with changing values of M and B. Recently, the direction of designing cache-efficient algorithms is cache-oblivious algorithms. The advantages of cache-oblivious algorithms are as follows [13], [14]:

1) Inclusion: As the cache-oblivious algorithms optimally progress between two adjacent levels of the memory hierarchy, which are cache referring to the near memory to the CPU and disk referring to the far memory from the CPU. Then, these algorithms are automatically adapted between any two adjacent memory levels with different values of M and B, because they are not fixed in cache-oblivious algorithms.
2) Constant optimal factor: Optimality means the minimum number of cache misses for an algorithm. There is no way to reduce the cache misses for an algorithm less than its optimal cache misses. When the number of memory transmissions or cache misses is optimal to a constant c between any two adjacent memory levels, then this optimality is kept within a weighted factor between the other two adjacent memory levels, such that the weighted mixture will be corresponding to the relative speeds of the memory levels. For example, assuming the optimal number of cache misses between two adjacent memory levels to traverse data of size N is $\left\lceil \frac{N}{B} \right\rceil$. Then, this factor is kept between the other two adjacent memory levels and is weighted by their relative speeds. Thus, algorithms in a two-level memory model can be designed and analyzed to gain outcomes for some levels of the memory hierarchy.
3) Self-tuning: Typical cache-aware algorithms need tuning to various cache parameters which are no longer on hand from the manufacturer and are often hard to extract automatically. Parameter tuning makes code portability difficult, while cache-oblivious algorithms perform well on all machines without modifications based on the cache parameters.

III. Cache Complexity

Cache complexity is the number of cache misses that are incurred by an algorithm for a problem of input size N and denoted by $T(N)$. The transfer unit between two adjacent memory levels is a block of size B words to amortize the access time cost. Typically, the main goal of an algorithm is to minimize the cache complexity $T(N)$, which is bounded by N as the upper bound and $\frac{N}{B}$ as the lower bound. In other words, the number of memory transfers at most equals the input size when each operation incurs a cache miss, whereas storing related elements in the same memory block B, which is called locality, reduces the number of cache misses into $\frac{N}{B}$ as a lower bound. We are concerned about complexity for large problem N when it is greater than B or even greater than M [15].

Typical cache complexity is a function of N and B because the minimum unit of transferred data is B with one cost unit. Also, M is relevant in cache complexity especially for algorithms with recursion when data fits in the cache and has been loaded in it, then the accessing cost will be zero. Generally, to compute the cache complexity of divide and conquer algorithms, we have to enlarge the base case to fit either B or M sizes [16], [17].

IV. Cache-Oblivious Algorithms

Cache-oblivious algorithms are mainly concerned with the efficiency of fetching large data into memory, which needs many memory transfers. This Section shows various cache-oblivious algorithms and their lower bound of memory transfers. Generally, cache-oblivious algorithms provide the lower bound of memory transfers utilizing the ideal cache, which is based on the following assumptions [18], [19], [20]:

- Optimal page replacement, such as evicting the least-recently-used block (LRU) or evicting the oldest-used block (FIFO) [21].
- Full associativity, such that the transferred memory block B can be stored at any available block in the cache.
- Tall cache, which means the number of blocks $\frac{M}{B}$ is greater than the block size B.

A. Scanning Approach

Scanning algorithms access all data items to perform some tasks such as finding maximum element, getting the average of elements, classifying elements, etc. Therefore, scanning algorithms touch all data items once and in the same order as they are stored, consequently, scanning algorithms are not aware of the cache size M. The cache complexity of a scanning algorithm is shown in Equation 1, such that N items lay out in contiguous blocks of memory. The ceiling function indicates one more memory transfer than $\frac{N}{B}$, because either N is not divided by B, or N does not start to lay out items from the beginning of a block. In other words, data items of size B require one memory transfer, but sometimes they require two memory transfers according to the alignment of the data items. For example, if the size of N is 8 and the size of B is 4, then scanning N incurs 2 memory transfers. However, if we did not start to lay out N items from the beginning of a block as shown in Fig. 1, where N items are displayed in gray color. Even though the N size is $2B$, scanning N incurs 3 memory transfers instead of 2 because of the alignment [10].

$$T(N) = O\left(\left\lceil \frac{N}{B} \right\rceil\right) \tag{1}$$

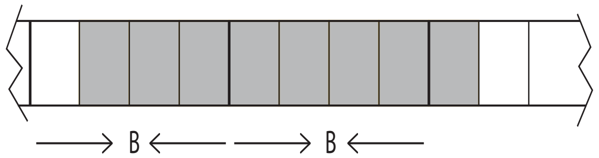

Fig. 1. Two Consecutive Blocks in Memory (the Gray Blocks Represent a Two-Block-Data item).

B. Divide and Conquer Approach

Divide and conquer algorithms recursively split problems into non-overlapping smaller sub-problems, solve them, and combine them. Divide and conquer algorithms recur down to the base case of constant size. In terms of memory transfers, we consider the base case that either fits one block B or fits within cache size $(\leq M)$. Such a size is considered the critical place for memory transfers, where all the cost is. Because once data fits in the cache, the accessing cost will be zero. However, the base case below B is kind of trivial [6].

Nevertheless, the divide and conquer approach is a basic technique for designing cache-oblivious algorithms, which often afford optimal cache complexity within a constant factor among the levels of a memory hierarchy [6].

C. Search Tree Approach

Initially, we will discuss a binary search tree using a divide and conquer approach, where the base case is a sub-tree of size B [6]. Nevertheless, the cache-complexity of the binary search tree on a sorted array is:

$T(N) = O(\lg N - \lg B) = O\left(\lg \frac{N}{B}\right)$

Where $\lg N$ is the height of the binary tree, and $\lg B$ is the leaf height i.e. the base case height. However, B-tree can achieve the optimal cache-complexity i.e. $O\left(\frac{\lg N}{\lg B}\right)$ by making the branching factor value between B and $\frac{B}{2}$. The drawback of B-trees is that the branching factor cannot be tuned easily between any two memory levels. Nevertheless, if the branching factor is known for all levels of the memory hierarchy, then the optimal cache-complexity can be achieved by using a B-tree algorithm [6].

The authors of [22] introduced an efficient search tree algorithm that can be tuned easily between any two memory levels and their algorithm achieves the optimal cache complexity. This algorithm is the so-called cache-oblivious tree (or Van Emde Boas layout), which is a binary search tree, but each recursive sub tree is laid out in a single segment of memory. Accordingly, the tree is recursively split from the middle, so the height of the tree is $\lg N$, see Fig. 2. We keep splitting each half recursively into two almost equal halves. At some point in this recursion, we reach halves of size less than or equal to B [22], [6].

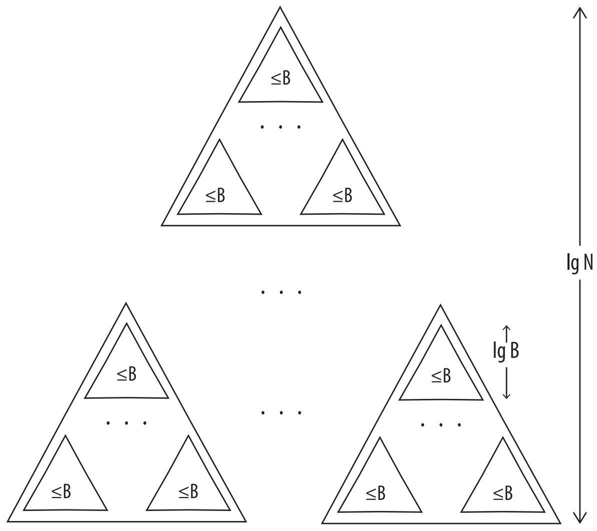

Fig. 2. Layout of the Cache-Oblivious Tree.

For analysis, following a root-to-node path visits some sequence of triangles, where each of them fits in, basically one block. In other words, triangle size is, at most B and at least $\sqrt{B}$, which is generated from splitting $B+1$ sub-tree into two further sub-trees, each of size $\sqrt{B}$. Thus, the height of the smallest triangle i.e. the leaf belongs to $[lg\sqrt{B}, lgB]$. Then, at most the number of visited triangles for a root-to-node path of at most $\lg N$ height is as follows [22]:

$\frac{\lg N}{\lg \sqrt{B}} = \frac{\lg N}{\frac{1}{2}\lg B} = 2\log_B N$

Each block requires one transfer, but sometimes a block requires two transfers because of the alignment of each block, as shown in Figure 1. Therefore, the cache-oblivious tree incurs at most $2 \times (2\log_B N)$ memory transfers. As a result, the cache-oblivious tree performs INSERT, DELETE, and SEARCH operations through an optimal cache complexity, which is shown in Equation 2 [22].

$$T(N) = O(4\log_B N) \tag{2}$$

D. Sorting Algorithms

Sorting algorithms take N elements of some arbitrary order and put them into a sorted order. Nevertheless, sorting is an essential algorithm in computer science, since it can decrease the complexity of a problem, especially in searching and database problems. There are many ways to sort elements using various algorithms, such as bubble, selection, insertion, merge, quick, heap, radix, bitonic, and bucket sort [23], [24], [25], [26].

The obvious and easiest way to sort elements is by doing N inserts into a regular B-tree, which incurs $O(N\log_B N)$. B-trees are efficient for searches but are not efficient for very frequent updates [27]. To get a better result, an efficient sorting algorithm can be used such as merge sort, which uses a divide and conquer approach.

Merge sort divides the problem into two parts, recursively sorts each part, and recursively merges the two parts. Thus, merge sort requires three parallel scans; one for the first portion, another for the second portion, and the third scan for the merged array, i.e. the sorted array. Such that, we compare the first elements of both unsorted portions, output one of them into the merged array, and move the unsorted portion pointer to the next element, then compare, output one of them, and so on till reaching the end of both unsorted portions [28]. Accordingly, the whole block is scanned, kicked out, and then the next one is read, so the three parallel scans can be afforded, as long as, $\frac{M}{B} \geq 3$.

Nevertheless, we always have to be careful with the base case, whose best size is $\leq M$, because when a sub-array of size M is reached, the whole thing is read without incurring any more cost as long as, the sub-array stays within a region of size M. Figure 3 shows the recursion tree of merge sort, which is better than B-trees and incurs $O\left(\frac{N}{B} \lg \frac{N}{M}\right)$. In terms of memory transfers, $\frac{N}{B}$ is the number of sorted elements and $\lg \frac{N}{M}$ is the longest path through the recursion tree to sort an element [29].

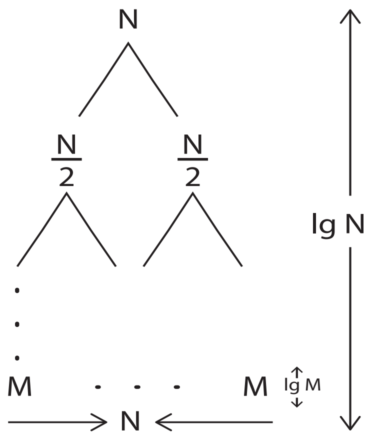

Fig. 3. The Recursion Tree of Binary Merge Sort.

Furthermore, to do a multi-way merge sort such as $\frac{M}{B}$-way merge sort, then we can mimic the binary merge sort but with $\frac{M}{B}$ portions instead of two portions [30]. In other words, $\frac{M}{B}$ parallel scans are needed, so we have to lay out in cache the first blocks of these $\frac{M}{B}$ sub-arrays, whose size for each of them is $\left(N/\frac{M}{B}\right)$. As a result, the same solution as binary merge sort is achieved but with a shorter longest path through the recursion tree as $\left(log_{\frac{M}{B}} \frac{N}{M}+1\right)$, such that:

$$\log_{\frac{M}{B}} \frac{N}{M}+1=\log_{\frac{M}{B}}\left(\frac{N}{B}\frac{B}{M}\right)+1$$
$$=\log_{\frac{M}{B}} \frac{N}{B}+log_{\frac{M}{B}} \frac{B}{M}+1$$
$$=\log_{\frac{M}{B}} \frac{N}{B}-log_{\frac{M}{B}} \frac{M}{B}+1$$
$$=\log_{\frac{M}{B}} \frac{N}{B}-1+1$$
$$=\log_{\frac{M}{B}} \frac{N}{B}$$

Accordingly, Equation 3 shows the cache complexity of sorting N elements using an $\frac{M}{B}$-way merge sort, which is optimal. Also, an $\frac{M}{B}$-way merge sort is the so-called cache-oblivious sorting algorithm [29].

$$T(N)=O\left(\frac{N}{B}\log_{\frac{M}{B}} \frac{N}{B}\right) \tag{3}$$

E. Priority Queue Data Structure

A priority queue is a queue, where every item is associated with a priority. An item with the highest priority is dequeued before any other item. The main operation associated with the priority queue is getting the highest priority item at any given time. Priority queue provides INSERT and DELETE-MIN operations, to add an item to a queue and to dequeue the highest priority item from it, respectively, as the highest priority item corresponding to the minimum number. Literature introduces priority queues supporting a different set of operations. In this section, we discuss the cache-oblivious priority queue supporting INSERT and DELETE-MIN operations [31], [32], [33].

The cache-oblivious priority queue uses a bunch of arrays in a linear order instead of a bunch of B-trees. Because priority queue using arrays is simpler and incurs fewer memory transfers than priority queue using B-trees, which are the best choice for searching operations. The cache-oblivious priority queue using the divide and conquer approach is arranged in levels, as each level is decomposed of two types of buffers; one "up buffer" and a set of "down buffers" such that each buffer is of a certain size. In other words, the cache-oblivious priority queue levels are recursive smaller priority queues, where the highest priority item exists in the smaller priority queue, i.e. the smaller level, in the cache [34].

The cache-oblivious priority queue for N items has $\lg \lg N$ levels, whose sizes are arranged from top to bottom as $N, N^{2/3}, N^{4/9}, ..., X^{9/4}, X^{3/2}, X, X^{2/3}, ..., C$, respectively, as shown in Fig. 4. C is a constant size level, and X is a size between N and C. At the same level, the total size of its "down buffers" at most equals the size of its "up buffer". For example, Level $X^{3/2}$ in Figure 4 has one "up buffer" of size $\Theta(X^{3/2})$ and at most $X^{1/2}$ of "down buffers" each of size $\Theta(X)$. At most, the total size of the "down buffers" at level $X^{3/2}$ is $(X^{1/2} \times X = X^{3/2})$, which matches the size of the "up buffer" at level $X^{3/2}$. For any two consecutive levels, a "down buffer" at the larger level matches the size of the "up buffer" at the smaller level. For example, $X^{3/2}$ and X are consecutive levels as shown in Fig. 4, the size of a "down buffer" at level $X^{3/2}$ (i.e. the larger level) is X, which matches the size of the "up buffer" at level$X$ (i.e. the smaller level) [34].

Generally, minimum and maximum items are corresponding to the highest priority item and the least priority item, respectively. At a level, items of its "up buffer" are disordered and their priorities are less than all items in the "down buffers" at that level. Items of a "down buffer" are disordered and their priorities are greater than items of the next "down buffer" at the same level. However, items of the "down buffers" at the very small levels are ordered and have the minimum item, i.e. the highest priority item. The priority queue algorithm knows the maximum item, which is corresponding to the least priority item, of each "down buffer" at all levels [31], [34], [29].

INSERT operation appends the new item i to the "up buffer" of the smallest level in the cache. Then the algorithm

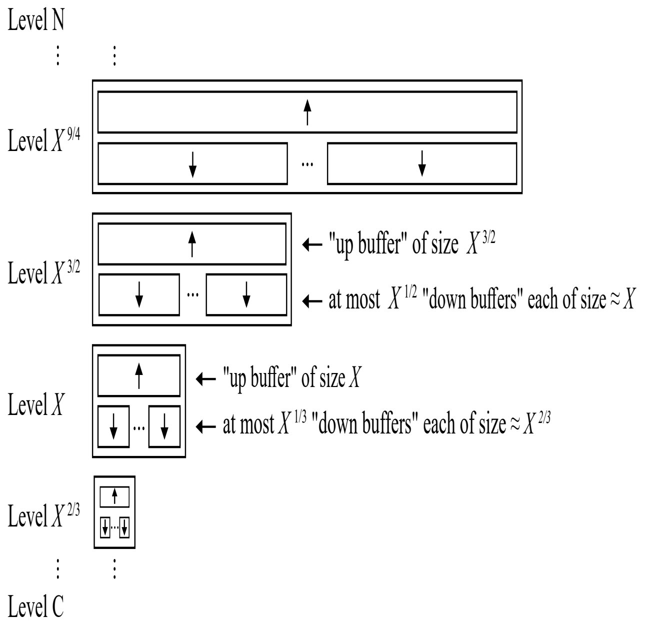
Level N
Level $X^{9/4}$
Level $X^{3/2}$
← "up buffer" of size $X^{3/2}$
← at most $X^{1/2}$ "down buffers" each of size ≈ X
Level X
← "up buffer" of size X
← at most $X^{1/3}$ "down buffers" each of size ≈ $X^{2/3}$
Level $X^{2/3}$
Level C

Fig. 4. Cache-Oblivious Priority Queue Levels.

locates i in an appropriate "down buffer" of the smallest level by comparing i to maximum items of the "down buffers" sequentially. If the selected "down buffer" has space, i is just added to it, otherwise, the "up buffer" swaps i with the maximum item in the selected "down buffer". When the "up buffer" overflows, the algorithm performs the push operation[29].

To describe push operation, assume the "up buffer" of level X overflows, then the algorithm pushes X items of the level X "up buffer" into level $X^{3/2}$ "up buffer". After that, the algorithm sorts the pushed X items and distributes them among the "down buffers" at level $X^{3/2}$ and possibly the "up buffer" at level $X^{3/2}$. Such that to allocate every item in its appropriate buffer, the algorithm scans the X items sequentially and visits "down buffers" at level $X^{3/2}$ in order. When the selected "down buffer" overflows, it is split in half to spawn a new "down buffer" at level $X^{3/2}$. However, if the number of allowed "down buffers" overflows, according to our example, the number of "down buffers" at level $X^{3/2}$ exceeds $X^{1/2}$, then the last "down buffer" at level $X^{3/2}$ is moved into the "up buffer" at level $X^{3/2}$. When this "up buffer" overflows, the algorithm pushes recursively $X^{3/2}$ items of the level $X^{3/2}$ "up buffer" into the level $X^{9/4}$ "up buffer" [29].

DELETE-MIN operation reverses the INSERT operation, therefore the algorithm deletes and pulls instead of inserts and pushes. The pull operation is a sort of reverse distribution step. Usually, "down buffers" of the smallest level are kept sorted in the cache and have the minimum item all the time, so the highest priority item is touched with zero memory transfers. However, there is no need to sort items in "down buffers" of the larger levels, we just need to keep track of the maximum items for the "down buffers" of larger levels. The smallest level in external memory is called the key level, where the algorithm consumes memory transfers to touch any item there. In the cache-oblivious priority queue, most memory transfers are consumed to sort items. Consequently, the cache-oblivious priority queue provides INSERT and DELETE-MIN operations for one element with the cost of sorting, as shown in Equation 4 [29].

$$T(N) = O\left(\frac{1}{B}\log_{\frac{M}{B}}\frac{N}{B}\right) \tag{4}$$

F. Matrix Multiplication Algorithm

Matrix multiplication is the most essential matrix operation since it has significant applications in various fields. Examples are cryptography, wireless communication, computer graphics, computations in linear algebra, solution of linear systems of equations, the transformation of coordinate systems, and computational modeling [35], [36], [37], [38], [39].

Multiplying two matrices of size $N \times N$ using the standard matrix-multiplication algorithm incurs $O\left(\frac{N^3}{B}\right)$ memory transfers. However, the cache-oblivious algorithm uses the divide and conquer approach to solve matrix-multiplication problems. For simplicity, assume that A, B, and C are square matrices of size $N \times N$ for each. The cache-oblivious algorithm recursively partitions these matrices into quadrants, as shown in Fig. 5 [40], [41], [42].

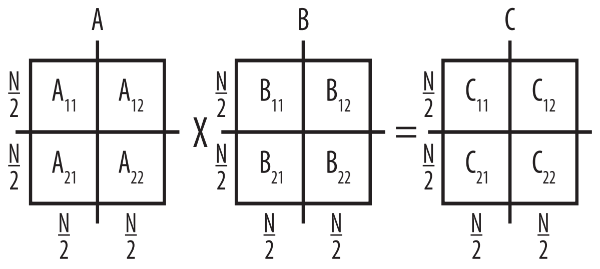
A
B
C
A_{11} A_{12} A_{21} A_{22}
X
B_{11} B_{12} B_{21} B_{22}
=
C_{11} C_{12} C_{21} C_{22}
$\frac{N}{2}$

Fig. 5. Cache-Oblivious Algorithm Recursively Divides Matrices into quadrants.

The algorithm performs eight recursive matrix multiplications to update the four quadrants of C. At some point in this recursion, we get a base case, that the three sub-matrices fit in a certain number of B's, or for the best base case when the three sub-matrices fit in the cache M. For cache complexity analysis, the recursion stops at the best base case, when the three sub-matrices of size $c\sqrt{M}$ x $c\sqrt{M}$, such that, c is a constant due to dividing cache into three sub-matrices. Then, the three sub-matrices fit in the cache, and accessing any element of them is free [43].

Nonetheless, this recursion is dominated by its leaves, so its cache complexity is the total number of leaves times the cache misses per leaf. Accordingly, each leaf of this recursion incurs $O\left(\frac{M}{B}\right)$ memory transfers, as the total number of leaves is:

$$8^{\lg N - \lg c\sqrt{M}} = 8^{\lg \frac{N}{c\sqrt{M}}} = \left(\frac{N}{c\sqrt{M}}\right)^{\lg 8} = \frac{N^3}{cM^{\frac{3}{2}}}$$

So, the total number of memory transfers is $\frac{M}{B} \times \frac{N^3}{cM^{\frac{3}{2}}}$. Consequently, Equation 5 shows the cache complexity of the cache-oblivious matrix-multiplication algorithm using recursive block matrices, which is the optimal cache complexity for matrix-multiplication problems [40], [41], [42].

$$T(N) = O\left(\frac{N^3}{B\sqrt{M}}\right) \tag{5}$$

V. Extension: Cache-Oblivious Neural Networks

Generally, a neural network iteratively learns by datasets, such that a neural network passes the whole dataset several times to learn correctly, as the neural network gets better performance in terms of mean squared error and accuracy [44]. The number of passes through the entire dataset is called epochs. To speed up the learning process, a dataset of size N is divided into mini-batches as a minimum learning unit, where a neural network updates its parameters after passing each mini-batch. Consequently, every epoch consists of several mini-batches, each of them having a size between 1 and N. For brevity, mini-batches are commonly called batches [8], [45], [46].

During the learning process, the neural network must be kept in the cache, but mini-batches are loaded and kicked out as needed. For the cache-oblivious neural network, the dataset must lay out in contiguous segments of memory, in any order, so the neural network scans the dataset in the same order it is stored [6]. The mini-batch size interacts with other hyper-parameters and must be optimized at the end to find the optimal size. However, if the size of a neural network is P, then the mini-batch size equals a multiple of B, as the selected multiple of B is recommended to be $\leq (M-P)$ to avoid incurring a memory transfer within a learning unit. In other words, it is recommended to select the mini-batch size as a power of two, that does not exceed $(M-P)$.

Accordingly, if a neural network of size P, using a dataset of size N, and a number of epochs E, then the cache complexity of this problem is described in Equation 6. As the number of epochs increases to get better results, the cache complexity of a neural network for a large problem (i.e. $P+N > M$) increases by at most the same factor. However, the number of epochs does not incur any memory transfer for a small problem (i.e. $P+N \leq M$).

$$T(N) = \begin{cases} O(\lceil \frac{P}{B} \rceil + \lceil \frac{N}{B} \rceil),\ if\ P+N \leq M \\ O(\lceil \frac{P}{B} \rceil + E \times \lceil \frac{N}{B} \rceil),\ if\ P+N > M \end{cases} \tag{6}$$

Neural networks greedily access memories, consequently, if N does not lay out in contiguous segments of memory, the cache complexity of a neural network will expand by a factor of N. Another extreme case that expands the cache complexity of a neural network, is when the N element is larger than B, so accessing any of the N elements initiates a memory transfer, too.

A. Experimental Results

To examine cache-oblivious neural networks, we implemented a cache simulator based on Intel i7 CPU and a memory hierarchy as described in Table II. Our cache simulator using Python 3.6 is concerned with the L1 cache, whose block size is $8KB$, consequently, $M = 8MB$ and $B = 8KB$. Additionally, the simulator uses a tall full associative cache and the LRU as a page replacement policy.

TABLE II. Experimental Memory Hierarchy

Memory level	Size
L1 Cache	8MB
L2 Cache	1GB
Main memory (RAM)	8GB

Furthermore, a 6-layer-stacked auto-encoder (SAE) model is used to classify the MNIST dataset as being created according to the authors of [8]. The experimental 6-layer SAE of 1139710 parameters occupied a $4.35MB$ of M. We examined two factors, the dataset sizes $\{0.63, 1.59, 3.18, 6.36, 9.69, 13.02, 16.36\}$ in MB and the number of epochs $\{0, 10, 20, ..., 100\}$. Table III shows the experimental results, which are the total cache misses per epoch using different dataset sizes.

The available cache space for a dataset is $3.65MB$ because the experimental SAE occupied $4.35MB$ of M (i.e. $8-4.35$). Thus, three dataset sizes fit the free space of M as the rest dataset sizes are larger than the free space of M. Accordingly, we split the results into two figures; Fig. 6 shows the cache misses of the experimental SAE using small dataset sizes (i.e. $P+N \leq M$), and Fig. 7 shows the cache misses of the experimental SAE using large dataset sizes (i.e. $P+N > M$). Epoch zero represents the initial step when the SAE model is loaded to M without the dataset. Therefore, all experiments using any dataset size have the same number of cache misses at zero epoch, which equals 557 cache misses i.e. $\lceil \frac{modelsize}{B} \rceil = \lceil \frac{4.35MB}{8KB} \rceil$, as illustrated in Table III.

Fig. 6 shows that increasing dataset size increases the cache misses. However, increasing the number of epochs does not perform any additional cache misses. Because the experimental SAE and the dataset fit M. Fig. 7 shows that increasing dataset size increases the cache misses too. Additionally, increasing the number of epochs increases the cache misses by the same factor. Because the experimental SAE and the dataset are larger than M, consequently, every dataset access performs a cache miss. For example, the dataset size of $6.36MB$ in addition to the model size of $4.35MB$ needs $10.71MB$, which is greater than M. Therefore, the cache misses based on Equation 6 are $557 + E \times \lceil \frac{6.36MB}{8KB} \rceil$, as shown in Table III. However, if the dataset size is 3.18, then the total needed space is $(3.18MB + 4.35MB = 7.53MB)$. Thus, the problem fits M, and the cache misses based on Equation 6 are $557 + \lceil \frac{3.18MB}{8KB} \rceil$, which is independent of E, as illustrated in Table III.

VI. Summary

Cache-oblivious algorithms utilize all levels of memory hierarchy efficiently, without knowing their parameters or even the existence of memory hierarchy levels. Thus, cache-oblivious algorithms support portability and better memory performance. Cache complexities of cache-oblivious algorithms are denoted by the cache parameters i.e., the cache size M and the block size B, even though M and B are unknown for cache-oblivious algorithms in reality.

TABLE III. Total Cache Misses per Epoch for the Experimental SAE

epochs / dataset size	0	10	20	30	40	50	60	70	80	90	100
0.63 MB	557	638	638	638	638	638	638	638	638	638	638
1.59 MB	557	761	761	761	761	761	761	761	761	761	761
3.18 MB	557	965	965	965	965	965	965	965	965	965	965
6.36 MB	557	8707	16857	25007	33157	41307	49457	57607	65757	73907	82057
9.69 MB	557	12967	25377	37787	50197	62607	75017	87427	99837	112247	124657
13.02 MB	557	17237	33917	50597	67277	83957	100637	117317	133997	150677	167357
16.36 MB	557	21497	42437	63377	84317	105257	126197	147137	168077	189017	209957

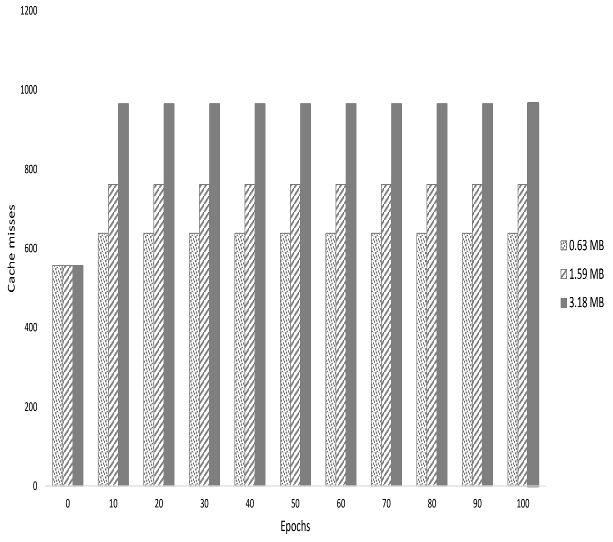

Fig. 6. Cache Misses of SAE using Small Datasets.

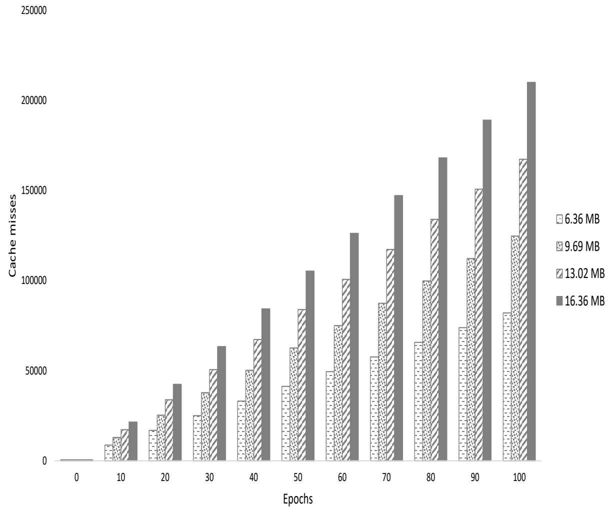

Fig. 7. Cache Misses of SAE using Large Datasets.

This paper discusses the cache complexities of the optimal recently known cache-oblivious algorithms for most essential problems, as summarized in Table IV. Notice that, the same input size N for all algorithms, but each of them incurs a different number of cache misses according to the algorithm target. Generally, cache-oblivious algorithms utilize a divide and conquer approach with a base case based on the algorithm behavior. However, their cache complexities are calculated on different base cases that are proportional to either the block size B or the cache size M.

TABLE IV. Cache-Complexity based on Cache-Oblivious Model

Cache-oblivious algorithm and data structure	**Cache-complexity**
Scanning	$\left\lceil \frac{N}{B} \right\rceil$
Binary search tree	$4 \log_B N$
$\frac{M}{B}$-way merge sort	$\frac{N}{B} \log_{\frac{M}{B}} \frac{N}{B}$
Priority queue	$\frac{1}{B} \log_{\frac{M}{B}} \frac{N}{B}$
Recursive block-matrix multiplication	$\frac{N^3}{B\sqrt{M}}$
Small neural network $(P+N) \leq M$	$\left\lceil \frac{P}{B} \right\rceil + \left\lceil \frac{N}{B} \right\rceil$
Large neural network $(P+N) > M$	$\left\lceil \frac{P}{B} \right\rceil + E \times \left\lceil \frac{N}{B} \right\rceil$

Moreover, we introduce an extension that applies the cache-oblivious scanning approach to neural networks. In other words, to minimize the cache complexity of a neural network, the dataset must lay out in contiguous blocks of memory. When a neural network and its dataset are within cache size, the cache complexity is $\leq \frac{M}{B}$. Otherwise, cache complexity is growing by a factor that at most equals the number of epochs, as shown in Fig. 7. Nevertheless, if the dataset does not lay out in contiguous blocks of memory, then the cache complexity of a neural network expands, consequently, its learning process consumes unreasonable time.

References

[1] S. I. Serhan and H. M. Abdel-Haq, "Improving cache memory utilization," *World academy of science, engineering and technology*, vol. 26, pp. 299–304, 2007.

[2] M. W. Ahmed and M. A. Shah, "Cache memory: An analysis on optimization techniques," *International Journal of Computer and IT*, vol. 4, no. 2, pp. 414–418, 2015.

[3] L. Arge, G. S. Brodal, and R. Fagerberg, "Cache-oblivious data structures," pp. 545–565, 2018.

[4] R. Sawant, B. H. Ramaprasad, S. Govindwar, and N. Mothe, "Memory hierarchies-basic design and optimization techniques," 2010.

[5] N. Rahman, "Algorithms for hardware caches and tlb," pp. 171–192, 2003.

[6] E. D. Demaine, "Cache-oblivious algorithms and data structures," *Lecture Notes from the EEF Summer School on Massive Data Sets*, vol. 8, no. 4, pp. 1–249, 2002.

[7] R. E. Ladner, R. Fortna, and B.-H. Nguyen, "A comparison of cache aware and cache oblivious static search trees using program instrumentation," pp. 78–92, 2002.

[8] I. Abuqaddom, B. A. Mahafzah, and H. Faris, "Oriented stochastic loss descent algorithm to train very deep multi-layer neural networks without vanishing gradients," *Knowledge-Based Systems*, vol. 230, p. 107391, 2021.

[9] N. Ghatasheh, H. Faris, I. AlTaharwa, Y. Harb, and A. Harb, "Business analytics in telemarketing: cost-sensitive analysis of bank campaigns using artificial neural networks," *Applied Sciences*, vol. 10, no. 7, p. 2581, 2020.

[10] E. D. Demaine, "Cache-oblivious algorithms and data structures," *Lecture Notes from the EEF Summer School on Massive Data Sets*, vol. 8, no. 4, pp. 1–249, 2002.

[11] M. A. Bender, R. A. Chowdhury, R. Das, R. Johnson, W. Kuszmaul, A. Lincoln, Q. C. Liu, J. Lynch, and H. Xu, "Closing the gap between cache-oblivious and cache-adaptive analysis," in *Proceedings of the 32nd ACM Symposium on Parallelism in Algorithms and Architectures*, 2020, pp. 63–73.

[12] E. Demaine and A. Schulz, "Mit 6.851 advanced data structures," *Angew. Chem. Int. Edit. Engl*, vol. 6, pp. 53–67, 2010.

[13] A. Lars, M. A. Bender, E. Demaine, C. Leiserson, and K. Mehlhorn, "Cache-oblivious and cache-aware algorithms," in *Cache-Oblivious and Cache-Aware Algorithms (Dagstuhl Seminar 04301)*. Schloss Dagstuhl, 2005.

[14] J.-I. Agulleiro and J.-J. Fernandez, "Tuning the cache memory usage in tomographic reconstruction on standard computers with advanced vector extensions (avx)," *Data in brief*, vol. 3, pp. 16–20, 2015.

[15] R. A. Chowdhury, "Algorithms and data structures for cache-efficient computation: theory and experimental evaluation," Ph.D. dissertation, 2007.

[16] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran, "Cache-oblivious algorithms," in *Foundations of Computer Science, 1999. 40th Annual Symposium on*. IEEE, 1999, pp. 285–297.

[17] Y. Tang and W. Gao, "Processor-aware cache-oblivious algorithms*," in *50th International Conference on Parallel Processing*, 2021, pp. 1–10.

[18] G. S. Brodal, R. Fagerberg, U. Meyer, and N. Zeh, "Cache-oblivious data structures and algorithms for undirected breadth-first search and shortest paths," in *Scandinavian Workshop on Algorithm Theory*. Springer, 2004, pp. 480–492.

[19] A. Aggarwal, J. Vitter *et al.*, "The input/output complexity of sorting and related problems," *Communications of the ACM*, vol. 31, no. 9, pp. 1116–1127, 1988.

[20] T. H. Chan, Y. Guo, W.-K. Lin, and E. Shi, "Cache-oblivious and data-oblivious sorting and applications," in *Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms*. SIAM, 2018, pp. 2201–2220.

[21] M.-K. Lee, P. Michaud, J. S. Sim, and D. Nyang, "A simple proof of optimality for the min cache replacement policy," *Information Processing Letters*, vol. 116, no. 2, pp. 168–170, 2016.

[22] M. A. Bender, E. D. Demaine, and M. Farach-Colton, "Cache-oblivious b-trees," in *Foundations of Computer Science, 2000. Proceedings. 41st Annual Symposium on*. IEEE, 2000, pp. 399–409.

[23] A. Al-Adwan, R. Zaghloul, B. A. Mahafzah, and A. Sharieh, "Parallel quicksort algorithm on otis hyper hexa-cell optoelectronic architecture," *Journal of Parallel and Distributed Computing*, vol. 141, pp. 61–73, 2020.

[24] S. W. A.-H. Baddar and B. A. Mahafzah, "Bitonic sort on a chained-cubic tree interconnection network," *Journal of Parallel and Distributed Computing*, vol. 74, no. 1, pp. 1744–1761, 2014.

[25] B. A. Mahafzah, "Performance assessment of multithreaded quicksort algorithm on simultaneous multithreaded architecture," *The Journal of Supercomputing*, vol. 66, no. 1, pp. 339–363, 2013.

[26] B. Elshqeirat, M. Altarawneh, and A. Aloqaily, "Enhanced insertion sort by threshold swapping," *International Journal of Advanced Computer Science and Applications*, vol. 11, no. 6, 2020.

[27] G. S. Brodal, "Cache-oblivious algorithms and data structures," in *Scandinavian Workshop on Algorithm Theory*. Springer, 2004, pp. 3–13.

[28] G. S. Brodal, R. Fagerberg, and K. Vinther, "Engineering a cache-oblivious sorting algorithm," *Journal of Experimental Algorithmics (JEA)*, vol. 12, pp. 2–2, 2008.

[29] G. S. Brodal, E. D. Demaine, J. T. Fineman, J. Iacono, S. Langerman, and J. I. Munro, "Cache-oblivious dynamic dictionaries with update/query tradeoffs," in *Proceedings of the twenty-first annual ACM-SIAM symposium on Discrete Algorithms*. Society for Industrial and Applied Mathematics, 2010, pp. 1448–1456.

[30] M. Chrobak, M. Golin, J. I. Munro, and N. E. Young, "A simple algorithm for optimal search trees with two-way comparisons," *ACM Transactions on Algorithms (TALG)*, vol. 18, no. 1, pp. 1–11, 2021.

[31] J. Iacono, R. Jacob, and K. Tsakalidis, "Cache-oblivious priority queues with decrease-key and applications to graph algorithms," *arXiv preprint arXiv:1903.03147*, 2019.

[32] E. Shi, "Path oblivious heap: Optimal and practical oblivious priority queue," in *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2020, pp. 842–858.

[33] T. A. Assegie and H. D. Bizuneh, "Improving network performance with an integrated priority queue and weighted fair queue scheduling," *Indonesian Journal of Electrical Engineering and Computer Science*, vol. 19, no. 1, pp. 241–247, 2020.

[34] J. Iacono, R. Jacob, and K. Tsakalidis, "External memory priority queues with decrease-key and applications to graph algorithms," in *27th Annual European Symposium on Algorithms (ESA 2019)*, vol. 144, 2019.

[35] M. H. Qasem, A. A. Sarhan, R. Qaddoura, and B. A. Mahafzah, "Matrix multiplication of big data using mapreduce: a review," in *2017 2nd International Conference on the Applications of Information Technology in Developing Renewable Energy Processes & Systems (IT-DREPS)*. IEEE, 2017, pp. 1–6.

[36] A. Sleit and A. Abusitta, "A visual cryptography based watermark technology for individual and group images," *Systems, Cybernetics and Informatics*, vol. 5, no. 2, pp. 24–32, 2008.

[37] H. Faris, A. A. Heidari, A.-Z. Ala'M, M. Mafarja, I. Aljarah, M. Eshtay, and S. Mirjalili, "Time-varying hierarchical chains of salps with random weight networks for feature selection," *Expert Systems with Applications*, vol. 140, p. 112898, 2020.

[38] J. E. H. Ali, E. Feki, and A. Mami, "Dynamic matrix control dmc using the tuning procedure based on first order plus dead time for infant-incubator," *International Journal of Advanced Computer Science and Applications*, vol. 10, no. 6, 2019.

[39] I. Abuqaddom and A. Hudaib, "Cost-sensitive learner on hybrid smote-ensemble approach to predict software defects," in *Proceedings of the Computational Methods in Systems and Software*. Springer, 2018, pp. 12–21.

[40] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, "Introduction to algorithms," pp. 1–92, 2009.

[41] Y. Tang, "Balanced partitioning of several cache-oblivious algorithms," in *Proceedings of the 32nd ACM Symposium on Parallelism in Algorithms and Architectures*, 2020, pp. 575–577.

[42] M. Dusefante and R. Jacob, "Cache oblivious sparse matrix multiplication," in *Latin American Symposium on Theoretical Informatics*. Springer, 2018, pp. 437–447.

[43] M. De Berg and S. Thite, "Cache-oblivious selection in sorted x+ y matrices," *Information processing letters*, vol. 109, no. 2, pp. 87–92, 2008.

[44] P. M. Radiuk, "Impact of training set batch size on the performance of convolutional neural networks for diverse datasets," *Information Technology and Management Science*, vol. 20, no. 1, pp. 20–24, 2017.

[45] O. A. Alrusaini, "Covid-19 detection from x-ray images using convoluted neural networks: A literature review," *International Journal of Advanced Computer Science and Applications*, vol. 13, no. 3, 2022.

[46] H. Faris, S. Mirjalili, and I. Aljarah, "Automatic selection of hidden neurons and weights in neural networks using grey wolf optimizer based on a hybrid encoding scheme," *International Journal of Machine Learning and Cybernetics*, vol. 10, no. 10, pp. 2901–2920, 2019.