Pons *et al. Journal of Big Data (2024) 11:152*
https://doi.org/10.1186/s40537-024-01013-5

Journal of Big Data

RESEARCH

Open Access

SMT efficiency in supervised ML methods: a throughput and interference analysis

Lucia Pons[1*], Marta Navarro[1], Salvador Petit[1], Julio Pons[1], María E. Gómez[1] and Julio Sahuquillo[1]

*Correspondence:
lupones@disca.upv.es

[1] Universitat Politècnica de València, Valencia, Spain

Abstract

The microarchitecture of general-purpose processors is continuously evolving to adapt to the new computation and memory demands of incoming workloads. In this regard, new circuitry is added to execute specific instructions like vector multiplication or string operations. These enhancements and the support of multiple threads per core make simultaneous multithreading (SMT) processors dominate the market for data center processors. Regarding emerging workloads, machine learning is taking an important role in many research domains like biomedicine, economics, and social sciences. This paper analyzes the efficiency of machine learning workloads running in SMT mode (two threads per core) versus running them in ST mode (single-threaded) with twice the number of cores. Experimental results in an Intel Xeon Skylake-X processor show an SMT efficiency falling between 80% and 100% across the studied workloads. These results prove two main findings: i) last-generation SMT processors are excellent candidates to execute ML workloads as they achieve a high SMT efficiency, and ii) if the performance of two major resources (i.e., FP double operator and core's caches) was boosted, all the workloads would achieve an almost perfect SMT efficiency. Moreover, results show that there is still room to support more threads without adding extra hardware. The discussed findings are aimed at providing insights to design future processors for ML workloads.

Introduction

General-purpose processors are continuously evolving to adapt their microarchitecture to software advances. Currently, simultaneous multithreading (SMT) processors are prevalent among the server platforms installed at data centers worldwide. SMT processors improve the utilization of the costly functional units present at the core by allowing instructions of multiple threads to share the issue ports. Moreover, SMT is the only multithreading paradigm [1] capable of issuing instructions from multiple threads simultaneously.

Because of their throughput nature and performance over single-threaded processors, commercial vendors dominating the microprocessor market, like Intel, IBM, ARM, or AMD, have opted to include SMT processors among their commercial products. As multiple threads are supported in each core, these processors introduce *inter-thread* interference within the core. Consequently, the performance of an

Springer Open

© The Author(s) 2024, corrected publication 2024. **Open Access** This article is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License, which permits any non-commercial use, sharing, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if you modified the licensed material. You do not have permission under this licence to share adapted material derived from this article or parts of it. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit http://creativecommons.org/licenses/by-nc-nd/4.0/.

individual thread can be severely affected when co-running with another thread in the core with respect to its individual execution. The extent to which the individual thread performance is affected depends on the co-runner and the microarchitecture.

From a certain point of view, SMT processors can be seen as heterogeneous processors as the performance of a given thread (or single-threaded application) is enhanced when running the thread alone in the core in single-threaded (ST) mode. In contrast, the threads can be run in SMT mode, allowing them to compete for core resources with other co-running threads. Recent work [2, 3] has been done in this direction, focusing on identifying symbiotic pairs of single-threaded applications to be run on the same core.

One of the current trends regarding the evolution of SMT processors is the type and amount of functional units deployed to adapt to the workload evolution. Among emerging workloads, machine learning (ML) workloads have increased in popularity in recent years in the scientific community [4, 5]. ML workloads have become key workloads in current data centers that deal with large and heterogeneous workloads [6]. These workloads allow performing an exhaustive list of tasks: image and speech recognition, data clustering, pattern identification, classification, prediction, etc. Thus, ML plays an important role in the context of data analysis in many areas, including biomedical engineering, economics, and social sciences.

The high number of threads these applications spawn make accelerators or GPUs [7–12] excellent computing devices to run them. Therefore, many ML frameworks [13–17] provide support to be executed on GPUs. However, due to the throughput nature of SMT cores and advances in their execution units, which add more functionalities to accelerate the execution of ML workloads in each processor generation, it makes sense to study the way SMT processors perform when running ML workloads and identify potential performance bottlenecks that should be faced to improve performance.

This paper studies the efficiency of SMT processors when executing machine learning (ML) benchmarks. The study uses an Intel Skylake Server (Skylake-X) processor implementing Hyper-Threading, where each core can support the execution of up to two threads. To focus the research, we evaluate four standard ML-supervised classification methods with ten publicly available datasets. We compare the efficiency of the core when running in SMT mode over running in ST with twice the number of cores. Experimental results show that SMT efficiency is over 90% in two learning methods and around 80% on average in the other two studied learning methods.

This paper makes two main contributions:

- We show that last-generation simultaneous multithreading processors are fit to execute supervised ML classification methods. This claim is supported by the fact that SMT efficiency is especially significant in ML workloads, which makes these processors excellent candidates for such tasks.
- We found that two main reasons prevent ML workloads from achieving a high SMT efficiency: the performance of the floating-point (FP) scalar double operator and the storage capacity of the core's caches (L1 and L2), which are shared among the threads running on the same core. These findings are aimed at providing insights to guide future processors' design to boost ML workloads' performance.

Following the taxonomy proposed in [18] to classify innovation ideas, the work proposed in this paper mainly falls under the class *Implantation*.

Background

This section presents some background on simultaneous multithreading and ML benchmarking concepts to ease the reading and make this paper self-contained. Authors with solid expertise on these topics can skip this section.

Simultaneous multithreading

Understanding the essence of SMT processors and to what extent they can improve workload performance is challenging due to the high complexity of the microarchitecture. Obtaining a sound knowledge of SMT processors is only achievable through guided learning of commercial processors, as many implementation issues need to be considered. Moreover, it is not easy to foresee i) to what extent this paradigm is capable of improving the processor throughput and ii) how it can affect (damage) the performance of individual workloads. The reason is that the answers to these questions require a strong grounding of many architectural concepts and depend both on the specific microarchitecture implementation and on the particular characteristics of the workload to be executed.

In an SMT processor's core, most of the core resources, like the register file, the arithmetic operators, or the ROB, are shared among the co-running threads. Resource sharing requires both extra logic and additional structures (e.g., auxiliary queues) to implement the policy that controls which share of the resource each thread has available. Because important resources are shared among the threads, the performance of a given thread strongly depends on the co-runner, making the execution time unpredictable. In [19], it is shown that, in the context of compute-intensive workloads, the execution time can more than double. In other words, if we want to boost the performance of a single-threaded application, then we should execute it alone in a single core in ST mode. This kind of policy can be applied by an OS that is aware of the threads running and schedules a specific thread to be executed alone in a given core. In this way, all the core resources are made available to the running thread.

To help understand the way in which inter-thread interference affects the performance, we first introduce some basic notions of these processors focusing on the issue ports since, as results will show, they define the performance. Let us illustrate these concepts through an example. Assume a hypothetical SMT processor implementing four issue ports as depicted in Fig. 1. There are two ports devoted to arithmetic operations

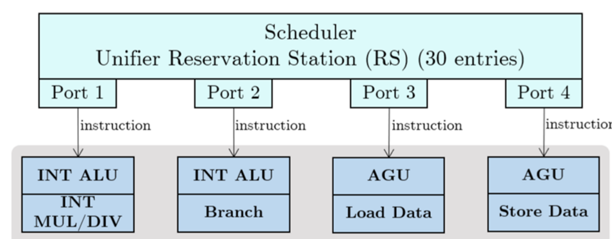


Fig. 1 Example of the distribution of issue ports in a hypothetical SMT processor

and two ports devoted to read/write access data in the (cache) memory. The arithmetic ports (1 and 2) carry out the most common basic arithmetic/logic integer operations (INT ALU). While port 1 is attached to the integer multiplier/divisor, port 2 is linked to the branch unit. Regarding the data access ports (3 and 4), they both include an address generation unit (AGU), but port 3 handles load operations and port 4 manages store operations. Notice that this design example is *balanced* in the sense that half of the ports are devoted to arithmetic operations and the other half to data accesses.

Figure 1 also shows the reservation station, which is unified and shared among all the threads. This is implemented as an out-of-order queue, and instructions from any thread are scheduled to an issue port provided that the corresponding operator is available and the source operands are ready. Thus, all the ports are available to a single thread if the core runs in ST mode and to all the threads if it runs in SMT mode. In the latter case, a thread may wait for a given port while other threads are using it. That is, the execution of the thread can be delayed with respect to executing alone in the core.

Machine learning workloads

Machine learning provides systems the ability to learn without human intervention. Among the types of learning algorithms, supervised learning is one of the most commonly used since it allows to make future predictions based on a series of known input–output pairs and a learning model. For instance, traffic prediction, face detection, and image classification.

ML workloads comprise two main components: the ML method and the dataset. The ML method refers to the ML algorithm being studied, which is implemented as a multithreaded application. Thus, with the term workload, we refer to a given learning method with a specific dataset. Unlike traditional HPC workloads [20–23], ML workloads' performance highly depends on the dataset. A wide set of datasets focusing on specific problems are publicly available and can be used to assess ML methods' performance.

Benchmarking can be applied to machine learning (ML) [4] from different perspectives. For instance, it can be applied to assess the ability of the studied method to learn patterns from 'standard' datasets [24, 25] or identify strengths and weaknesses of existing learning methods [26, 27]. Many problems and the corresponding datasets have been published in the open literature, and the conclusion was that there is no global ML method that performs the best across all the datasets, but performance is highly dependent on the problem being addressed.

Problem definition

This section introduces the problem addressed in this paper. Understanding the wide variety of SMT concepts and their impact on performance is challenging for academic instructors as it is not easy to map theoretical concepts to implementation aspects. In this paper, we address this problem from a new and more practical perspective: exploring the suitability of general-purpose processors to run ML workloads.

To do so, we focus on the SMT efficiency of high-performance processors typically installed in data centers. We define the *SMT Efficiency* metric as the ratio of the execution time an ML workload takes running in single-threaded (ST) mode to the time taken by the same application to execute in SMT mode (see Eq. 1). The execution time of a

given workload is defined as the time used to complete the evaluation of an ML method on an individual dataset.

$$\text{SMT Efficiency} = \frac{\text{Execution Time in ST mode}}{\text{Execution Time in SMT mode}} \quad (1)$$

In ST mode, each core runs a single thread from the workload, and in SMT mode, as many as the number of supported threads (e.g., two threads per core with Intel Hyper-Threading). Thus, in SMT mode, much fewer cores are needed. For instance, with Intel Hyper-Threading, just half the number of cores is required. Nevertheless, overall *Processor Throughput* improves as expressed in Eq. 2.

$$\text{Processor Throughput} = \text{SMT Efficiency} \times \#\text{Threads per Core} \quad (2)$$

For example, as stated in Eq. 1, if the SMT efficiency is 0.8, it means that the performance is $\frac{1}{0.8} = 1.25$ higher in ST mode than in SMT mode. However, the processor throughput (i.e., the performance per core) is 0.8×2 *threads/core* $= 1.6$ times higher in SMT mode.

Existing solutions

There are few works [28–33] that analyze the impact of CPU microarchitecture on the performance of machine learning workloads.

One of the first works [28] analyzes the performance of a machine learning workload parallelized with OpenMP running on an Intel Pentium 4 processor supporting Hyper-Threading. This paper focuses on understanding the performance of genetic-related machine learning workloads (SNP or single-nucleotide polymorphisms) and the underlying reasons behind that performance. However, the architecture of the processor used in this work is outdated, and many improvements have been included in new processor generations.

In [29], the authors present a comprehensive performance analysis of machine learning algorithms on the Apache Spark platform. Some of the algorithms used, Logistic Regression and Decision Tree, have also been used in this work. They identify bottlenecks at the microarchitectural level following Intel's Top-Down performance analysis [34] and propose solutions to address the discovered performance issues. Unlike this work, the focus is software-oriented, as the authors identify the functions and high-level functions responsible for producing such bottlenecks. In addition, they study the effectiveness of Hyper-Threading (Intel's SMT implementation), concluding that it should be enabled when running ML workloads.

Prieto et al. [30] perform an architectural analysis of how effective a set of DNN models use the underlying hardware. The analysis is performed following Intel's Top-Down performance analysis. The authors conclude that bottlenecks are found both in the core and the memory, highlighting that adding more FP units is essential to improve the performance of some DNN applications. Unlike our work, they do not study the effect of Hyper-Threading.

Focusing on parallelism, in [31], authors analyze the performance impact of key settings of a popular machine learning framework, Tensorflow, and quantify the role of parallelism. They observe that SMT or Hyper-Threading does not improve performance significantly, especially on those workloads bottlenecked by the fused

multiply-accumulate (FMA) units, which are shared between threads running in the same core. They propose a set of guidelines to achieve higher training and inference speedups, outperforming the performance improvements obtained with the settings recommendations from Intel and TensorFlow for the studied framework. Among these, they propose placing threads from the ML applications and from the framework operator to share the same core. However, as experimental results will prove in this work, some ML worker threads can share the core without barely experiencing performance degradation.

In [32], a survey of techniques for optimizing deep learning applications on CPUs is presented. The survey identifies CPUs' strengths and weaknesses in the field of deep learning. Regarding Hyper-Threading, it performs a small analysis, identifying that it increases the latency in some of the layers of the evaluated neural networks, especially when SIMD hardware needs to be time-multiplexed between the running threads. It summarizes several optimization techniques proposed for inference and training in mobile, desktop/server, and distributed systems.

Finally, some studies have considered energy consumption when running machine learning workloads. In [33], a comprehensive study is conducted on the power behavior and energy efficiency of CNNs and training frameworks on CPUs and GPUs. The authors provide a detailed workload characterization to facilitate the design of energy-efficient deep learning solutions. Among the studies features, they study the effect of Hyper-Threading, concluding that it does not significantly affect power consumption. However, in some applications, it increases the execution time and, thus, the energy consumption.

Proposed solution

This paper provides sound knowledge that helps understand why SMT efficiency is almost perfect in some ML workloads and why it drops in others. This is done by following a refinement approach that covers a wide range of key architectural concepts.

The analysis starts from a high-level perspective to go step-by-step into the core microarchitecture. This is done in a simple way that helps the reader to slowly introduce to complex architectural mechanisms.

Firstly, classical metrics such as the IPC in SMT mode (quantified both per core and per thread) and in ST mode are studied. Comparing both IPCs offers an overall overview of SMT efficiency. These throughput metrics are strongly connected to low-level architectural metrics, such as the issue port utilization. For instance, an IPC of 3 instructions per cycle means that, on average, at least,[1] three issue ports are used per cycle. However, by looking at the issue ports, one can know which type of functional units require the workload and if some of them are being overused or underused by these workloads.

This way of proceeding helps the reader to get introduced step by step into the microarchitecture. However, the study does not solve the problem yet. After that, the approach goes deeper into the hardware to identify the major backend core components that prevent twice the number of instructions in SMT mode from progressing in order to

[1] Since some instructions are mispeculated.

achieve a high SMT efficiency. In this regard, the approach classifies instructions according to the functional units they use (e.g., integer ALU, FP scalar double, etc.). After that, it is investigated if there is a relationship between the type of instructions each workload executes and a drop in the SMT efficiency. We found that there is a strong relationship with the FP scalar double instructions. This observation led us to an important finding: the number of issue ports having attached FP scalar double functional units should be increased to boost both the SMT efficiency and the performance of some ML workloads.

Nevertheless, this finding does not explain all the workloads presenting low SMT efficiency. We then looked at the core's caches (L1 and L2) as they are shared by the two threads running in the same core. We found that, indeed, the cache performance dropped in some workloads when running two threads due to the inter-thread interference.

To the best of our knowledge, this is the first time that ML workloads are studied in a commercial processor through a rigorous measurement methodology that only makes use of the performance events available in the machine, providing insights on the design of future processors to boost the performance of these workloads.

Elaboration

This section evaluates the SMT efficiency of machine learning workloads running two threads per core with respect to ST execution using twice as many cores. To understand the efficiency obtained by different workloads, we perform two main studies:

- A throughput-oriented analysis from two perspectives: committed instructions per cycle (IPC) and issue ports utilization.
- An interference analysis to identify potential performance bottlenecks in SMT mode, both in the functional units and processor caches.

Before presenting the results, a detailed description of the ML workloads used in this work, as well as the experimental setup, is provided.

ML Workloads

There exists a wide set of ML algorithms that are typically evaluated using different datasets. The algorithm used depends on many factors, like the function to be performed (e.g., classification or regression) or the size of the dataset. On the other hand, there are many publicly available benchmark datasets [35–38], but their widely different organizations and formats imply that significant pre-processing efforts are required before they can be used with ML algorithms [39]. Furthermore, many of the published benchmarks present similar features, which can make it difficult to represent a sufficiently diverse range of data science problems. In addition, some repositories like Kaggle [36], and OpenML [37] are not designed specifically for comprehensive ML benchmarking but rather focus on collaboration to solve data science problems.

In this work, we evaluate a subset of popular machine learning algorithms following the methodology to evaluate the PMLB benchmarks described in [40]. To carry out the

experiments, we have used the code[2] from scikit-learn benchmarks [41] which provides the implementation of ML supervised methods to evaluate their performance using Python Scikit-learn library [42]. The original code has been adapted to suit the mentioned methodology (further details can be found in Appendix A.2).

Below, a detailed description of the algorithms and datasets used is provided.

Classification algorithms

We evaluate the performance of four standard statistical ML supervised classification methods [43, 44], which present different characteristics and are used in the real world for different purposes:

Decision Tree Classifier The decision tree classification algorithm classifies the values of a target variable based on a set of input features. It creates a model in the form of a tree structure, with internal nodes representing features, branches representing decision rules, and leaf nodes representing outputs (classes). The algorithm works by recursively partitioning the data into smaller subsets based on the values of the attributes.

K-Nearest Neighbors (K-Neighbors) Classifier The k-nearest neighbors classification algorithm works by storing all available cases and classifying new cases based on a similarity metric (usually distance functions). A case is assigned to the most common class among its K-nearest neighbors. K-Neighbors is particularly useful for cases where the decision boundary is not clearly defined. One limitation of this learning method is that it can be computationally expensive since it requires storing all the data and calculating distances between the new case and all stored cases.

Linear Support Vector Classifier (SVC) The linear support vector classification algorithm finds the hyperplane in N-dimensional space (N is the number of features) that differentiates the data points of one class from the other. SVC is computationally intensive and does not scale well with large datasets.

Logistic Regression The logistic regression algorithm provides a linear model that predicts the likelihood of a given classification with a set of input features. A prediction is made using a function that maps the output of the linear model to a value between 0 and 1. This value is used to classify the input value into one of the predefined classes. The goal is to find the best coefficients for the input features that minimize the difference between the predicted classification class and the real one. Compared to the previous classification algorithms, it is relatively simple and fast to train.

Datasets

In this work, we use several datasets from the Penn Machine Learning Benchmark (PMLB) suite [40] to evaluate the machine learning algorithms. PMLB is a publicly available dataset suite hosted on GitHub that pursues to overcome the aforementioned shortcomings. PMLB includes classification and regression datasets from several well-known ML benchmark suites, comprising over 200 datasets, and it seeks to help identify the strengths and weaknesses of different ML algorithms.

[2] Available at https://github.com/rhiever/sklearn-benchmarks

Table 1 Characteristics and description of the datasets from PMLB used in this work to assess the ML algorithms

Dataset	Instances	Features	Classes	Description
krkopt	28056	6	18	Chess Endgame Database for White King and Rook against Black King
Letter	20000	16	26	Identification of images as capital letters
Magic	19020	10	2	Particle shower measurements used to classify types of radiation
Nursery	12958	8	4	Applications ranking for nursery schools
pendi-gits	10992	16	10	Pen-based recognition of handwritten digits
coil2000	9822	85	2	Insurance company benchmark (COIL 2000) information about customers
Agaricus lepiota	8145	22	2	Features of mushrooms in agaricus-lepiota family
Mush-room	8124	22	2	Mushroom records drawn from The Audubon Society Field Guide to North American Mushrooms
Ring	7400	20	2	Piston ring diameter data
ann thyroid	7400	21	3	Patient information concerning Hyperthyroidism

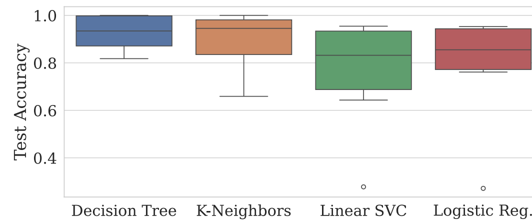


Fig. 2 Accuracy distribution for the studied ML methods and datasets

Table 1 shows the details of the ten datasets used in this work, sorted by descending order of instances or values to classify. The computational time of the evaluated ML algorithms is directly related to the number of instances, which represents the values of the dataset. Features and classes denote the data attributes and the possible values, respectively. The number of features and classes depends on the problem being addressed. For instance, for the `krkopt` dataset, the features represent possible positions (White King file, White King rank, White Rook file, White Rook rank, Black King file, and Black King rank) and the classes represent the game-theoretical values (draw, zero, and one to sixteen).

To compare the performance of the ML methods, we have computed the accuracy [45] (obtained from the `score()` method from Python Scikit-learn library) when running with each dataset. Figure 2 shows the results. In general, all the applications achieve high accuracy, with Decision Tree and K-Neighbors being the methods that have higher accuracy. The outlier points in Linear SVC and Logistic Regression correspond to `krkopt` dataset, as these linear models are better suited for binary classification (two classes) rather than multiclass datasets (see column *Classes* in Table 1).

Experimental setup

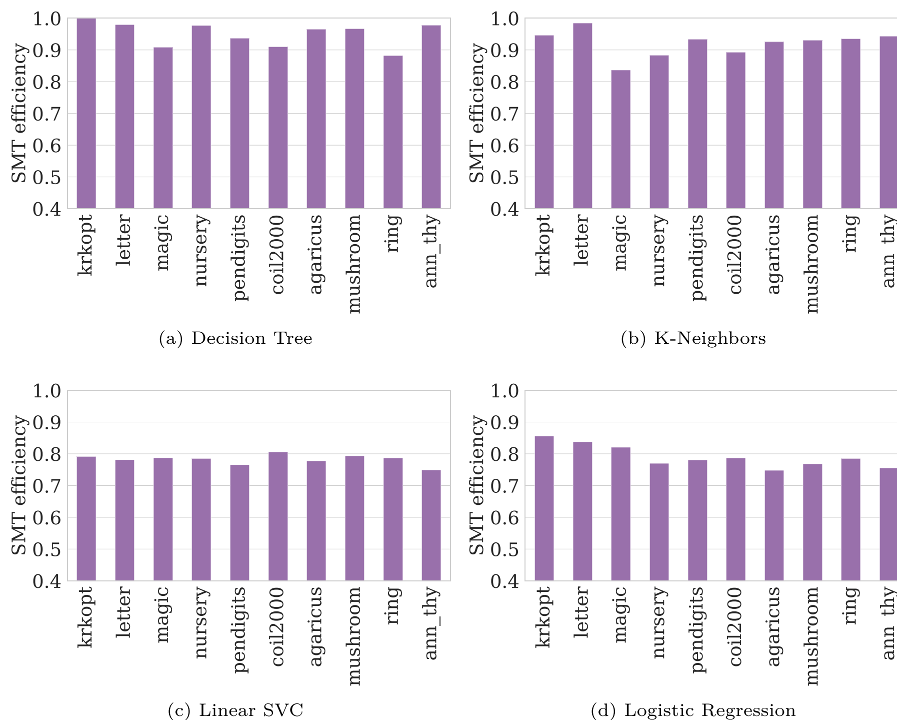
The machine used to evaluate the ML algorithms is equipped with an Intel Xeon Silver 4116 processor (Skylake-X microarchitecture), which features 12 Hyper-Threading cores

(i.e., two execution threads per physical core). Regarding the memory hierarchy, each core has private 64-KB L1 (32-KB instruction and 32-KB data) caches, a private 1-MB L2 cache, and all cores share a 16.5-MB L3 cache. The main memory consists of 4 channels, each one equipped with 1 DIMM DDR4 2400 MT/s with a total storage capacity of 64GB. The OS installed is Ubuntu 18.04 LTS with kernel version 5.4.

SMT efficiency results

We first evaluate the SMT efficiency of the studied ML workloads to quantify the performance of executing ML workloads in SMT cores compared to ST execution, which uses twice as many cores. Figure 3 presents the results for the four studied learning methods. It can be appreciated that Decision Tree and K-Neighbors achieve an SMT efficiency above 90% on average and almost 100% in some datasets like `krkopt` and `letter`. This is surprising since the SMT executions achieve these values with half the number of cores than the ST execution. A lower SMT efficiency is achieved in Linear SVC and Logistic Regression, but the average is still around 80%. The SMT efficiency in Linear SVC presents minor changes regardless of the studied datasets. In contrast, the SMT efficiency achieved by Logistic Regression presents significant deviations, ranging from 75% to 86%, depending on the studied dataset.

Next, we delve into providing insights regarding the distinct workload behaviors from a microarchitectural point of view. Multiple options have been explored. We first study basic throughput metrics that corroborate that an individual core is not able to sustain the per-thread performance when running two threads.



(a) Decision Tree (b) K-Neighbors

(c) Linear SVC (d) Logistic Regression

Fig. 3 SMT efficiency results for the studied ML methods and datasets

Throughput analysis

SMT processors are throughput-oriented processors, meaning that they allow increasing the number of executed instructions per cycle over single-threaded processors. In this section, throughput is analyzed from two main perspectives: committed instructions per cycle and issue port utilization.

IPC evaluation

The processor throughput is commonly quantified in terms of committed instructions per cycle or IPC. SMT processors increase throughput over ST processors since they execute instructions from multiple threads in the same cycle.

From a performance perspective, *an SMT processor can be seen to behave as ideal for a given workload when its SMT efficiency is 1 or close to 1 for such a workload*. For this situation to happen, the core throughput in SMT mode should double the throughput of a core when running in ST mode. In other words, the core should double the IPC as it is able to execute twice as many instructions when running in SMT mode than in ST mode for the same amount of time.

Figure 4 shows the IPC calculated as the quotient of the following performance events:

```
inst_retired.any:
    Instructions retired from execution (per thread)
cpu_clk_unhalted.thread:
    Core cycles when the thread is not in halt state
```

The graphs plot the IPC obtained by each workload in ST mode (IPC-ST) and SMT mode. Two IPCs, per thread (IPC-thread-SMT) and per core (IPC-core-SMT), are presented for SMT mode. For a single core, the IPC per thread corresponds to the average

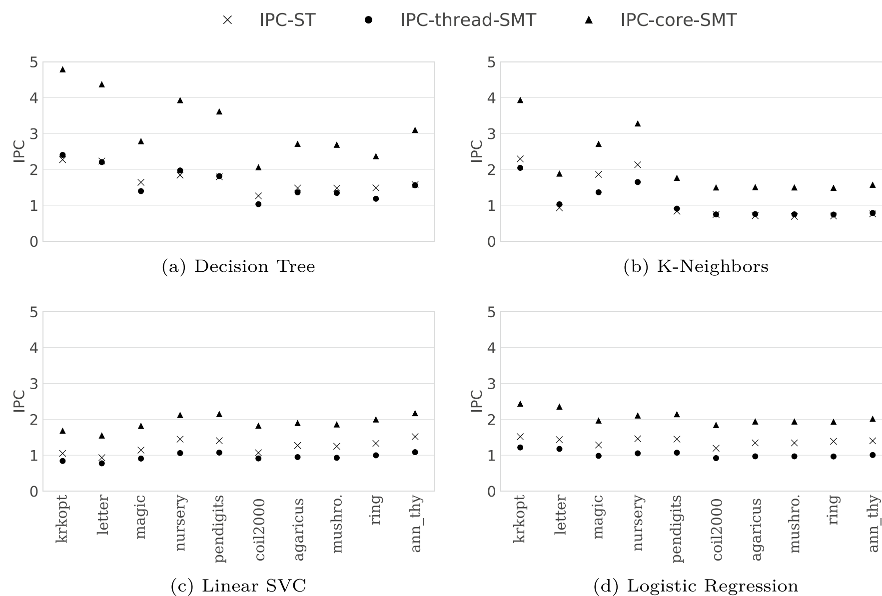


Fig. 4 Average IPC results for the studied ML methods and datasets. Results in SMT are provided both per logical core (*thread*) and physical core (*core*)

IPC for both threads running on the core, and the IPC per core to the sum of the IPC of both threads. Comparing Figs. 3 and 4, it can be observed that in those workloads where IPC-thread-SMT matches IPC-ST, the SMT efficiency is close to ideal (1). This was expected as it means that the SMT architecture behaves ideally with respect to ST mode since it is able to double the IPC over ST mode. This happens, for instance, in Decision Tree with `krkopt` dataset.

In contrast, it can be appreciated that in the aforementioned workloads where the SMT efficiency is around 80%, the IPC-thread-SMT drops around 20% over IPC-ST. For instance, K-Neighbors with the dataset `magic` or nearly all the datasets of Logistic Regression present this behavior.

Architectural analysis at port level

The previous discussion can be seen as a high-level analysis; however, a lower-level analysis is required to understand why SMT efficiency is over 95% across most of the studied datasets in Decision Tree and K-Neighbors. This section analyzes the utilization of the issue ports to identify if any ports are preventing the processor from achieving a high SMT efficiency. Before presenting the analysis, the architectural issue ports of our experimental platform are described.

Architectural Ports Description

Figure 5 presents a block diagram with the eight issue ports and the attached functional units of our experimental Intel Skylake-X processor: four ports are devoted to arithmetic-logic operations (ports 0, 1, 5, and 6), and the other four to the load/store unit. Arithmetic ports have attached multiple functional units, especially ports 0 and 1. Regarding the load/store unit, two ports are used for load operations (including the address generation unit (AGU) and load data), and the other two are used for store operations (AGU and store data). In this way, the design balances arithmetic operations with memory accesses, replicating the most critical functional units (e.g., integer and load) used by typical workloads to avoid these ports becoming potential performance bottlenecks.

Effective Issue Width Utilization

In order to double the IPC per core in those workloads exhibiting high SMT efficiency, the issue ports need to be used twice as long. On the contrary, port utilization would be

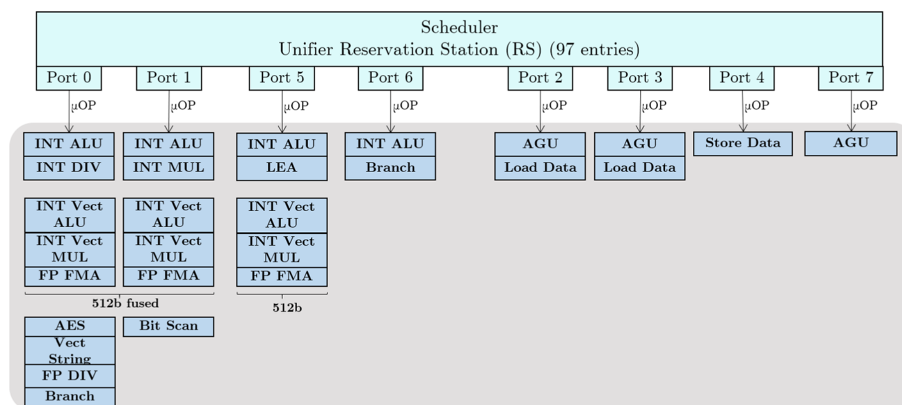


Fig. 5 Issue ports of the Intel Skylake-X microarchitecture

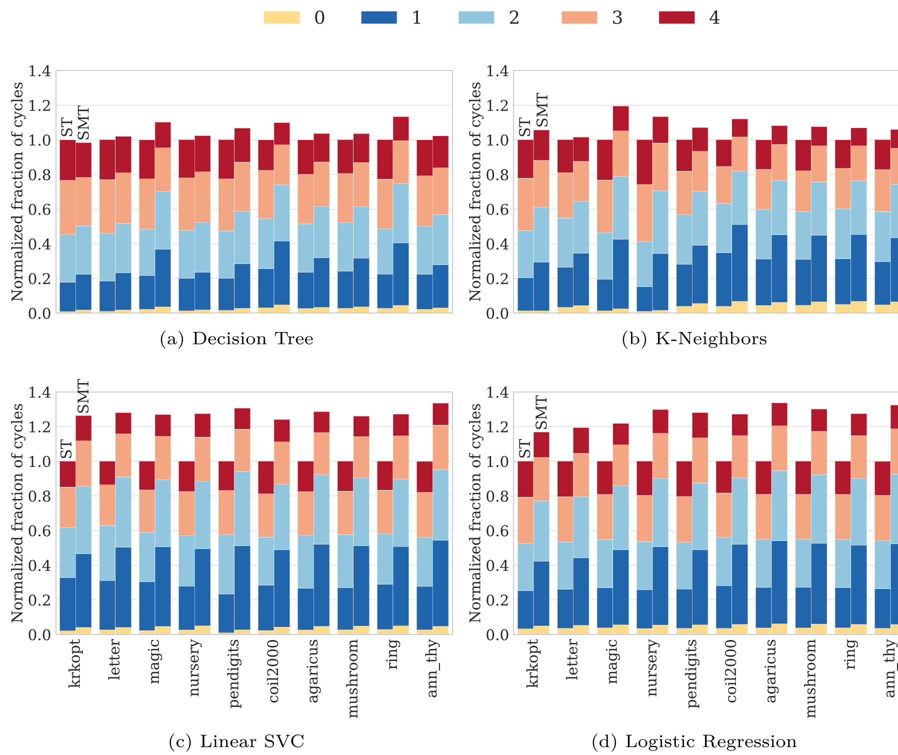


Fig. 6 Average fraction of cycles (normalized to the cycles obtained in ST mode) per logical core where 0, 1, 2, 3, and 4 μops are executed in all ports for ST (left sidebar) and SMT (right sidebar) modes

less for low-efficiency workloads. This fact can be studied from two main perspectives: the distribution of the effectively used issue ports and the port utilization. The focus of this section is on the first perspective.

The studied processor can issue up to four instructions (issue width) per cycle. So, the distribution function quantifies the fraction of time 0, 1, 2, 3, and 4 instructions are issued. To this end, we looked into the available performance counters and found a set of interesting performance events in this regard. Below, we list the hardware events and the publicly available description, where `i` is a value ranging from 1 to 4:

```
exe_activity.exe_bound_0_ports:
      Cycles where no uops were executed, the Reservation
      Station was not empty, the Store Buffer was full and
      there was no outstanding load
exe_activity.i_ports_util:
      Cycles total of i uops is executed on all ports and
      Reservation Station was not empty
```

Figure 6 presents the results[3] for ST and SMT modes normalized with respect to the cycles in ST. For each workload, two columns are shown; the first corresponds to the results obtained in ST mode, and the second to those obtained in SMT mode.

[3] Results only refer to the time the issue queue is not empty.

Results for SMT are averaged and presented on a per-thread level so they must be multiplied by two to obtain the equivalent value per core. For instance, if 20% of the time, four instructions are issued per thread (upper slice of the bar), then 40% of the time, the core issues four instructions as it hosts two threads. As results are normalized over ST cycles, the higher the SMT bar, the lower the SMT efficiency. In the case of an ideal SMT efficiency, the height of the SMT bar is equal to 1. It can be observed that in high SMT efficient workloads, not only are the bar heights similar, but also the heights of the slices, especially for a high number of instructions (e.g., 4). Notice that this means that the processor in SMT mode is able to work at its maximum issue rate for twice as long as in ST mode, which is averaged per thread. In contrast, in low SMT efficient workloads, slices 1 and 2 grow while slice 4 significantly decreases.

Individual ports utilization This section explores whether there is any issue port limiting the processor from achieving a high SMT efficiency in those workloads with lower SMT efficiency. For this purpose, we analyzed the utilization of the eight issue ports of each core. Figure 7 presents the results for both ST and SMT modes for the studied workloads in a stacked manner. In other words, each bar plots the sum of the eight issue ports' utilization; thus, eight is the maximum value. For a given port, the utilization has been computed as the fraction of cycles the port has been used with the performance counters listed below, where X is the port number that ranges from 0 to 7.

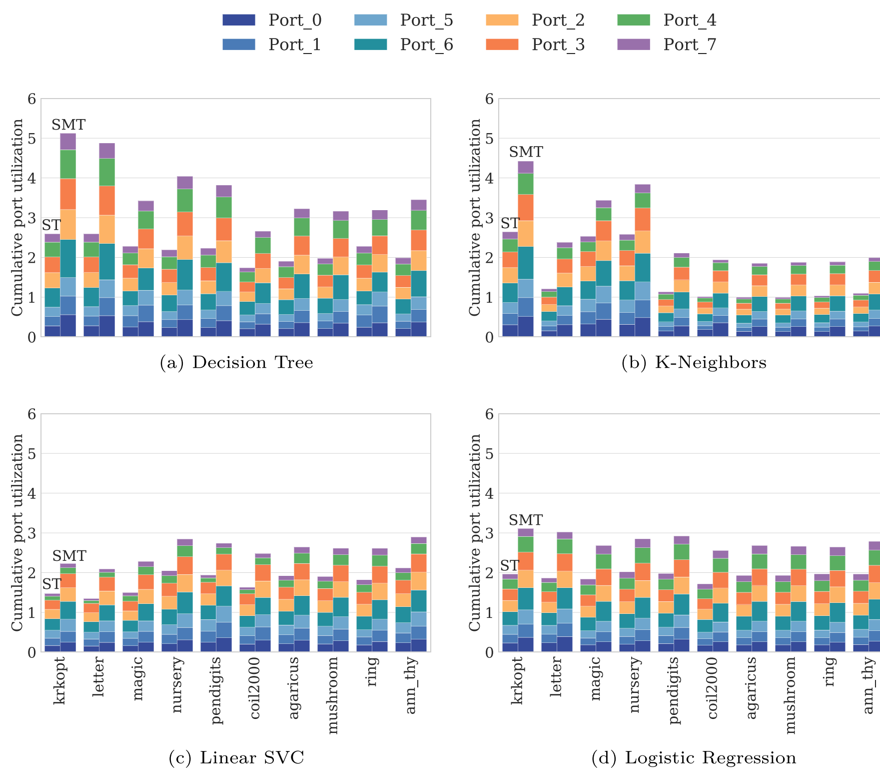


Fig. 7 Cumulative average port utilization per physical core for ST and SMT modes (one bar for each one)

```
uops_dispatched_port.port_X:
   Cycles per thread when uops are executed in port X
cpu_clk_unhalted.thread:
   Core cycles when the thread is not in halt state
```

In the case of SMT mode, the utilization is the sum of the utilization for both threads. Notice that to reach an ideal SMT efficiency, the execution time (i.e., `cpu_clk_unhalted.thread` performance counter) should be similar to that obtained in ST mode. Since two threads are executed in the same SMT core, the port utilization in SMT mode should be twice as large as in ST mode, and so the height of the bar. Three main observations can be drawn:

1. The port utilization grows according to the SMT efficiency. For instance, in those workloads having an SMT efficiency of 0.8, the port utilization rises in a 1.25 ($\frac{1}{0.8}$) factor in SMT over ST mode.
2. A relatively high value of the port utilization in ST mode (e.g., over 0.4) does not necessarily incur a low SMT efficiency. In fact, all datasets executed with the Decision Tree method and some workloads of K-Neighbors present relatively high values in some ports while achieving a high SMT efficiency. This claim can be better observed in Fig. 8, which plots, for illustrative purposes, the utilization of port #6 (INT ALU Branch) as it is one of the most used in both ST and SMT modes. As it can be seen, the workloads with the highest SMT efficiency (the plots in the top row) are close to doubling the utilization of this port (e.g., Decision Tree with dataset `krkopt`), whereas the ones with poorer efficiencies obtain a more modest port-6 increase with respect to ST execution.

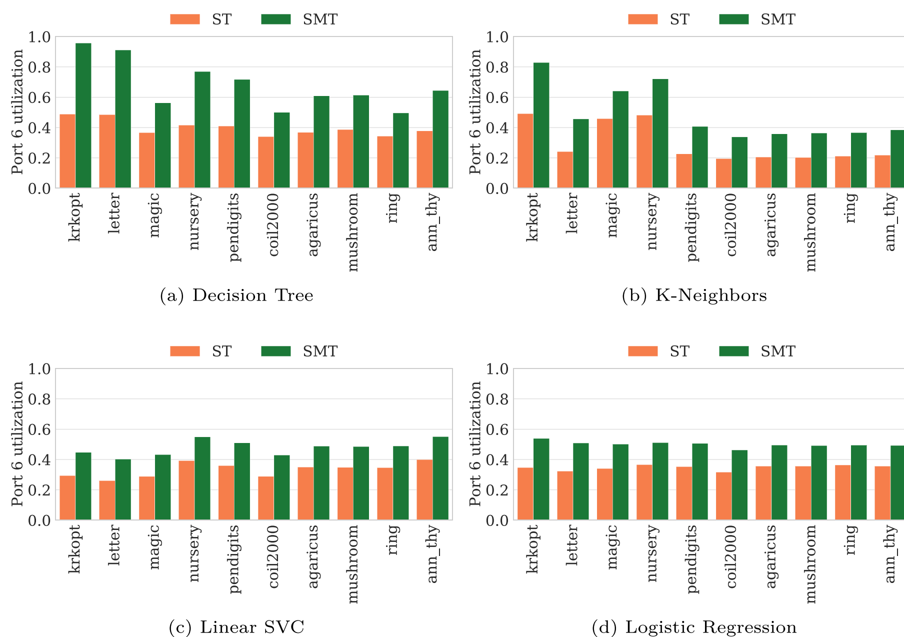


Fig. 8 Port with the highest utilization for ST and SMT, which corresponds in all cases to port 6

3. There are some workloads, like K-Neighbors, where the six rightmost datasets achieve high efficiency with a very low port utilization (none of them exceed 0.25) in SMT mode, which means there is still room in the processor for improving performance in the core. In other words, if the processor had supported four threads, a good SMT efficiency would likely have been achieved.

Takeaway i) the utilization of the different ports grows at the same rate when executing in SMT mode, ii) a relatively high port utilization (e.g., over 0.35) in ST execution does not prevent the system from achieving a high SMT efficiency, and iii) there is still room for the core to boost the performance if it supported more threads as the utilization of the most-used issue port is below 2/3 in most workloads.

Interference analysis

As concluded in the throughput analysis, a high utilization of the issue ports does not necessarily imply a low SMT efficiency. In fact, in some cases, a utilization over 0.9 led to an SMT efficiency of over 0.95. This makes sense, as a port is used in only one cycle for each executed instruction. Thus, performance losses do not appear because of port contention. This observation led us to move behind the issue ports and focus the research on identifying the major components of the processor backend that are preventing the core from reaching a high SMT efficiency: functional units contention and core caches (i.e., L1 and L2).

Functional units

This section looks into the available functional units to identify contention due to possible structural hazards. To this end, we first classified the retired instructions into four main categories according to the functional unit they use (floating point, load and store, branches, and integer). After that, we refined each group as much as possible, considering the available performance counters. Below, we list these groups and the performance counters that are used.

- Floating-Point Operations Floating-point (FP) operations are used to define the performance of computers (e.g., FLOPS) since they are computationally expensive. FP numbers are represented in 32-bit single-precision and 64-bit double-precision IEEE-754 formats. FP operations instructions are also executed using vector processing, and the vector size ranges from a 32-bit single scalar operand to 512 bits (e.g., 16 32-bit single precision operands). In this regard, Intel provides the following performance counters:

```
fp_arith_inst_retired.scalar_double:
    Number of SSE/AVX computational scalar double-precision
    FP instructions retired
fp_arith_inst_retired.scalar_single:
    Number of SSE/AVX computational scalar single-precision
    FP instructions retired
fp_arith_inst_retired.128b_packed_double:
    Number of SSE/AVX computational 128-bit packed double-
    precision FP instructions retired
fp_arith_inst_retired.128b_packed_single:
    Number of SSE/AVX computational 128-bit packed single-
    precision FP instructions retired
fp_arith_inst_retired.256b_packed_double:
    Number of SSE/AVX computational 256-bit packed double
    -precision FP instructions retired
fp_arith_inst_retired.256b_packed_single:
    Number of SSE/AVX computational 256-bit packed single
    -precision FP instructions retired
fp_arith_inst_retired.256b_packed_double:
    Number of packed double-precision FP arithmetic
    instructions
fp_arith_inst_retired.256b_packed_single:
    Number of packed single-precision FP arithmetic
    instructions
```

- Load and Store Operations. These operations quantify the read and write accesses performed to the cache memory, which can be computed with the following performance events:

```
mem_inst_retired.all_loads
    All retired load instructions
mem_inst_retired.all_stores
    All retired store instructions
```

- Branch Operations. Branches, depending on the branch predictor accuracy, can impact the processor throughput due to speculative execution. The number of branch instructions retired can be measured with the following performance event:

```
br_inst_retired.all_branches
    All (macro) branch instructions retired
```

- Integer Operations. These instructions perform mathematical operations with integer values. Unlike FP operations, Intel does not provide specific performance events to quantify these instructions. However, having quantified all the other types of instructions, we assume that the integer operations constitute the remaining instructions that do not fall under one of the previous categories.

Figure 9 shows the fraction of instructions corresponding to each of the above categories for the studied workload. To have further insights, the FP operations have been further divided into scalar (vector size equal to one) or vector (vector size greater than one), and in turn, into single and double precision. For each workload, two bars

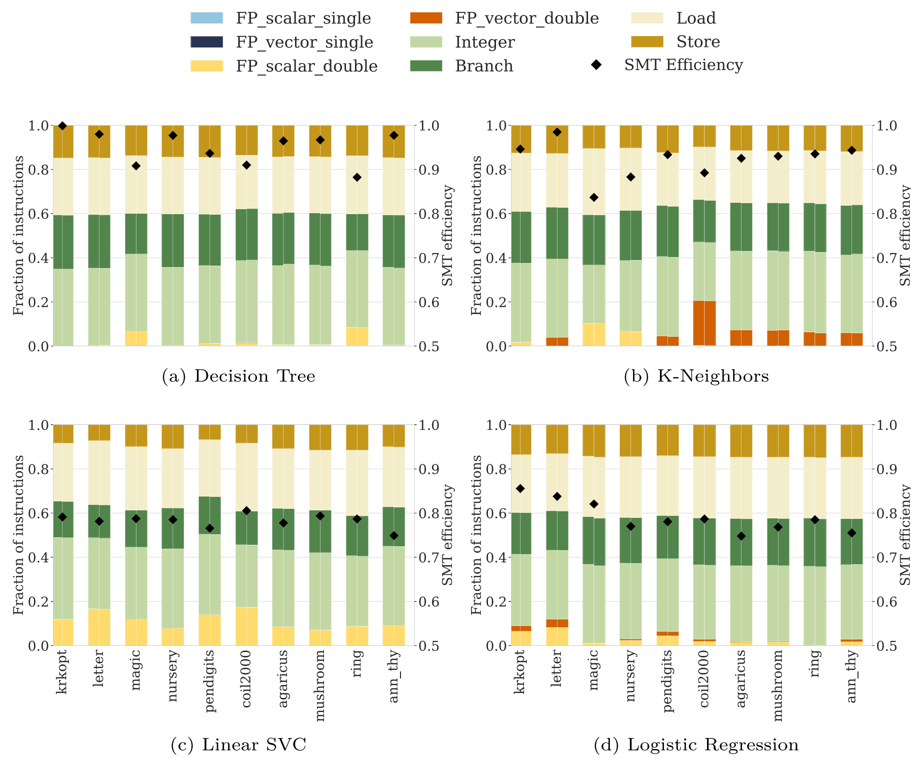


Fig. 9 Breakdown of retired instructions into categories according to the functional unit they use, for ST (left bar) and SMT modes (right bar)

are shown: the first corresponds to the execution in ST mode, and the second to SMT mode. To ease the analysis, the SMT efficiency is plotted on the Y-right axis.

Several observations can be made. Firstly, none of the studied workloads presented single-precision FP instructions (i.e., FP_scalar_single and FP_vector_single events count zero); thus, they are not shown in any of the graphs.

Some workloads present a significant fraction of double-precision FP instructions, either scalar or vector. In particular, those workloads experiencing low SMT efficiency (below 0.8) present a significant fraction of FP_scalar_double instructions. This happens in all the datasets in Linear SVC, `magic` and `ring` in Decision Tree, `magic` and `nursery` in K-Neighbors, and `krkopt` and `letter` in Logistic Regression. This means that structural hazards happen in these functional units, probably because these FP operations (especially multiplication and division operations) are time-consuming and are unlikely to be fully pipelined. Regarding the remaining instruction types (Integer, Branch, Load, and Store), all the studied workloads present a similar magnitude. On average, 35% of the instructions retired perform integer operations, 25% perform branch operations, 25% perform load operations, and 10–15% perform store operations.

Finally, notice that FP_scalar_double instructions do not explain the low SMT efficiency in all the workloads. For instance, in Logistic Regression, most data sets with just a few exceptions (e.g., `krkopt` and `letter`) barely present any FP_scalar_double instructions. Despite this fact, they achieve a low SMT efficiency, which means

that there is another functional unit, other than the FP unit, that is hindering the performance of these datasets.

Memory resources

From the core resources (functional units) analysis, we have concluded that structural hazards due to FP operations can significantly lower SMT efficiency. However, some workloads executed with Decision Tree or Logistic Regression experience a drop in SMT efficiency despite barely having FP instructions. Therefore, we looked into the memory resources to see if they bottleneck the performance. More precisely, as both threads share the L1 cache within the core, it is likely that the relatively small size of this cache affects the miss ratio of the co-running threads and, therefore, their performance. In addition, the L2 cache was also investigated, as this cache is also shared by the threads running in the same core.

Figure 10 shows the average misses per kilo instruction (MPKI) for these caches computed with the performance counters listed below.

```
mem_load_retired.l1_miss
    Retired load instructions missed L1 cache as data sources
mem_load_retired.l2_miss
    Retired load instructions missed L2 cache as data sources
inst_retired.any
    Number of instructions retired from execution
```

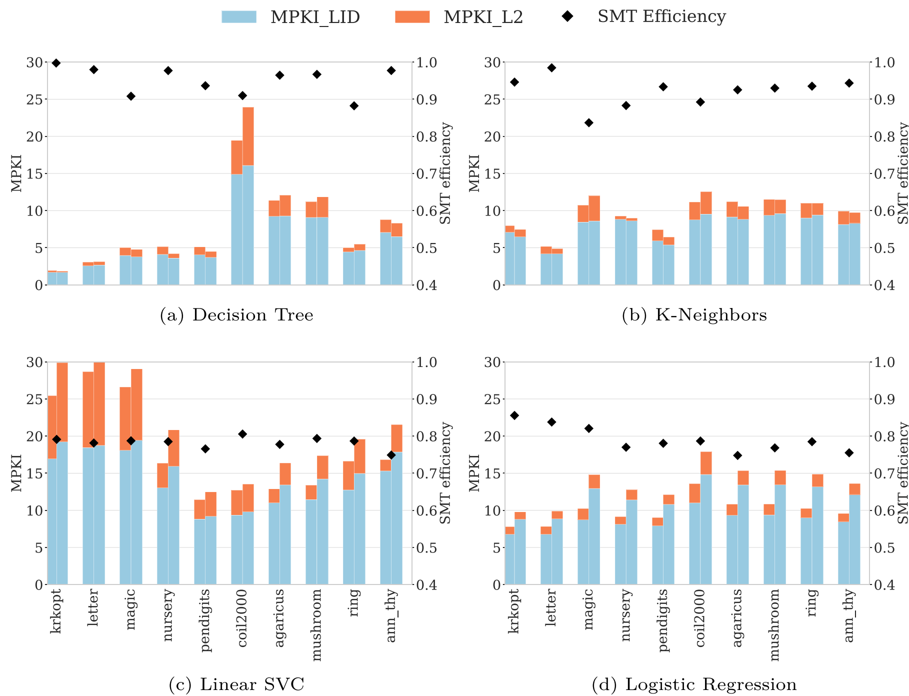


Fig. 10 Average Misses per kilo instruction (MPKI) of the L1 data cache and L2 cache per logical core for ST and SMT modes (one bar for each one)

For each workload, results are shown for ST mode (left bar) and SMT mode (right bar). The SMT efficiency is also plotted (Y-right axis) to ease the analysis. It can be appreciated that those workloads that achieve very high SMT efficiency (over 0.9) present low MPKI at both cache levels (sum less than 10) and barely increase the number of misses when running in SMT mode compared to ST mode. In some workloads, like in `nursery` with Decision Tree or `krkopt` with K-Neighbors, the MPKI per thread barely decreases in SMT mode, possibly because both threads in the same core share the part of the dataset and the lack of cache space is not an issue as very few misses are observed.

Those workloads that experience a negligible number of FP instructions but a relatively low SMT efficiency show a high difference in the height of the bars between ST and SMT. Examples are `coil2000` with Decision Tree and most datasets executed with Logistic Regression. In the former case, the MPKI of L1 and L2 caches increases over 5% and 70%, respectively. In Logistic Regression, the average increase is 40% for the L1 cache and over 25% for the L2 cache.

Finally, workloads like Linear SVC with most of the datasets experience both a significant fraction of FP instructions and a noticeable MPKI difference (in percentage) between the ST and SMT bars.

Takeaway High pressure on the core's caches (L1 and L2), the FP units' structural hazards, or both can significantly impact the SMT efficiency.

Performance of 4th generation intel xeon processors

So far a 1st generation of the Intel Xeon Scalable processor has been used to obtain the results. However, processors evolve in each microprocessor generation, including new or enhanced microarchitectural features. For comparison purposes, this section evaluates the ML applications in an Intel Xeon Gold 6438Y+, that is, a 4th generation (launched in Q1, 2023) Intel Xeon Scalable processor. This processor presents important microarchitectural advances like increased ROB size, more issue ports, and larger cache sizes.

To illustrate how last-generation processors perform on ML workloads, the results for Linear SVC with the same datasets are shown. This ML method has been chosen as it presents, on average, the lowest SMT efficiency. Thus, the objective is to analyze if this new processor allows for the achievement of higher SMT efficiency results.

To analyze the impact of the microarchitectural evolution, the applications have been executed under the same conditions regarding the number of processor cores (12) and CPU frequency (2.10 GHz). We would like to mention that this is the base frequency of the 1st generation platform, which ranges from 800 MHz to 3.0 GHz, while the fourth-generation platform ranges from 800 MHz to 4.0 GHz. Figure 11a shows the speedup results in terms of the execution time obtained in the 4th generation processor (4th gen) compared to that obtained in the 1st generation processor (1st gen). Counterintuitively, only six out of the ten tested datasets improve their performance when executed in the 4th-gen processor. In general, higher improvements are obtained in these applications when running in ST mode (12 cores). Per contra, SMT manages to obtain a higher speedup in applications with worse performance.

To analyze how the SMT performs in the 4th-gen processor, Fig. 11b shows the comparison of the SMT efficiency obtained for Linear SVC in 1st-gen and 4th-gen

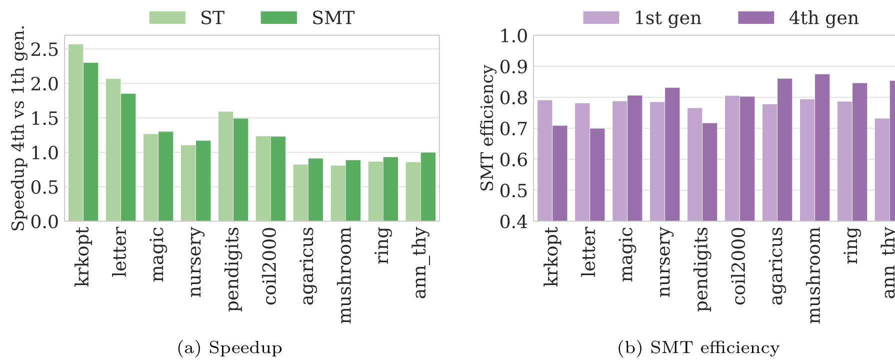


Fig. 11 Comparison of the performance obtained when running Linear SVC in 1st and 4th Generation Intel Xeon Scalable processors with 12 cores. Notice that only the microarchitecture is being compared as the frequency has been fixed to the base of the 1st generation. The 4th Generation always outperforms if technological aspects (frequency) are also considered

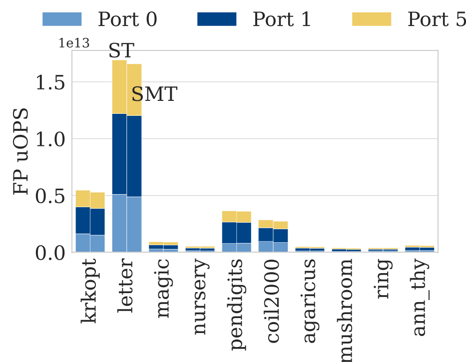


Fig. 12 FP μOPs executed when running Linear SVC in the 4th gen. processor

processors. Those applications that obtained the highest speedup (`krkopt`, `letter`, and `pendigits`) obtain worse SMT efficiency. However, the applications that did not improve their performance in the 4th-gen processor (`agaricus`, `mushroom`, `ring` and `ann_thyroid`) significantly improve their SMT efficiency.

To provide further insights into why some applications improve while others do not, we have analyzed the utilization of the different functional units. The 4th-gen processor includes new performance events that help with this task. For instance, the number of FP μops executed in each of the issue ports that are attached to FP units:

```
fp_arith_dispatched.port_0
      [FP_ARITH_DISPATCHED.PORT_0]
 fp_arith_dispatched.port_1
      [FP_ARITH_DISPATCHED.PORT_1]
 fp_arith_dispatched.port_5
      [FP_ARITH_DISPATCHED.PORT_5]
```

Figure 12 shows the values obtained for the three above performance events. Those applications that dispatch a higher number of FP operations (`krkopt`, `letter`, `pendigits` and `coil2000`) correspond to those that 1) obtained better performance

(speedup) and 2) showed a lower SMT efficiency in the 4th-gen processor. Notice that this newer processor has a total of 32 cores, and the results show the values corresponding to experiments where only 12 cores (and 6 in SMT mode) were used, the same number of cores as the 1st-gen processor.

Therefore, to obtain better performance, more cores should be assigned to those applications that make use of the FP units. To test this claim, we run the applications with 32 cores (ST mode) and 16 cores (SMT mode). Figure 13 shows the results of the SMT efficiency obtained, which is almost perfect in all the cases, proving the high computational power of this new processor.

Takeaway New-generation processors provide architectural enhancements (higher core count, more FP units) that are able to provide an almost perfect SMT efficiency of ML applications.

Conclusions

This paper has evaluated multi-threaded machine learning workloads in a recent Intel server processor with Hyper-Threading technology. Keeping the number of threads constant, we have analyzed the SMT efficiency achieved with only half of the number of processor cores. Two main findings have been found, elaborated next.

First, last-generation SMT processors are excellent candidates for executing ML workloads as they achieve high SMT efficiency. Effectively, the designed experiments reveal that the performance per core (IPC and port utilization) significantly grows over 80% in some workloads (e.g., Linear SVC and Logistic regression) or almost double in others (e.g., Decision Tree). This is achieved because, counter-intuitively, the issue port utilization is not a problem in most workloads. None of the ports presents a utilization above 0.35 when executing in SMT mode.

Second, looking into the functional unit utilization, we found that two major processor resources can potentially act as the major performance bottleneck in some ML workloads in last-generation processors when running in SMT execution mode: i) the FP scalar double functional unit, and ii) the L1 and L2 processor caches.

Finally, we claim that because the issue port utilization is low in high SMT-efficient workloads, there is still room to increase the multithreading level to support more threads in current processors. As comparative experimental results illustrate, this

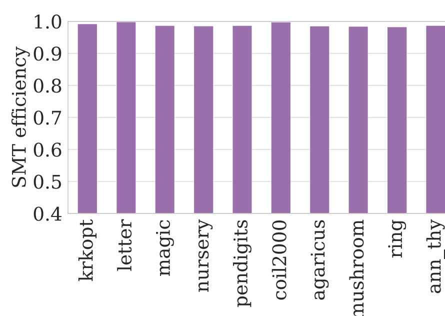


Fig. 13 SMT efficiency when running Linear SVC with 32 cores (all the available cores) in the 4th gen. processor

finding has been correctly addressed in the recent Q1–2023 Intel Xeon Golden Cove processor, which deploys three FP double-issue ports.

To the best of our knowledge, this is the first work identifying the critical issue ports when running ML workloads that should be improved (e.g., replicated) for performance boosting from an SMT-efficiency perspective and validating the claim through experimental results. The proposed analysis can be replicated in a different processor if the corresponding performance events are available. For instance, all Intel Xeon processors have similar performance events. Regarding processors from other architectures, AMD EPYC processors provide similar performance events (e.g., number of μops dispatched to the different execution units) that could be used analogously.

In summary, the elaborated findings are aimed at helping processor designers boost the performance of ML workloads in next-generation processors. In this regard, the processor caches (especially the L1 DCache) should be made larger, and the FP double-functional unit's performance should be boosted by adding more powerful and/or more FP double-functional units.

Experimental methodology: detailed steps

This appendix details the steps followed to set up and run the ML methods and datasets used in this work to ease experimental replicability. In addition, a shell script is provided to run the experiments and obtain the results shown in Sect. 6.

PMLB Installation

One major strength of PMLB is its easy installation and its user-friendly interface for datasets. Additionally, unlike other benchmark suites [46, 47], it does not rely on the usage of any external framework, which eases the experimental setup.

To install and use PMLB,[4]. the user first needs to download the PMLB Python wrapper using the *pip* package installer:

```
$ pip install pmlb
```

Once installed, to access the datasets, the user must use the `fetch_data` function from `pmlb`. Below, we show an example of a small Python program that reads all the available classification datasets and prints the description of each one of them:

```python
from pmlb import fetch_data, classification_dataset_names

for classification_dataset in classification_dataset_names:
    data = fetch_data(classification_dataset)
    print(data.describe())
```

Obtaining datasets in this way avoids downloading and storing the datasets locally, which occupy tens of GBs.

[4] Full instructions available at https://github.com/EpistasisLab/pmlb.

ML workloads evaluation

In this work, we have followed the methodology to evaluate the PMLB benchmarks described in [40]. To carry out the experiments, we have used the code[5] from scikit-learn benchmarks [41] which provides the implementation of ML supervised methods to evaluate their performance using Python Scikit-learn library [42]. The original code has been adapted to suit the mentioned methodology.

Firstly, data is preprocessed as datasets provided in PMLB are not scaled nor normalized, and some ML methods like K-Neighbors require data to be scaled. To this end, the features are scaled by subtracting the mean and then scaling them to the unit variance:

```
# 1) Import dataset
features, labels = fetch_data(dataset, return_X_y=True)

# 2) Apply pre-processing
scaler = StandardScaler()
scaler.fit(features)
features_scaled = scaler.transform(features)
```

Once datasets are scaled, each ML method is evaluated by performing a 10-fold cross-validation [48]. Since in this work, we aim to study multithreading efficiency, we have modified the original code so that the cross-validation is performed by launching as many processes as available cores by adding `n_jobs=-1` parameter:

```
cv_predictions = cross_val_predict(n_jobs=-1, estimator=clf,
X=features_scaled, y=labels, cv=StratifiedKFold(n_splits=10,
shuffle=True, random_state=90483257))
```

After the single-threaded preprocessing is complete, a file named STARTED is created to signal the start of the parallel computation. Similarly, a file named FINISHED is created to indicate that the ML method evaluation has ended:

```
try:
    f = open("STARTED", "x")
except:
    print("Error creating STARTED file!")

#### [...] #####
##### PARALLEL COMPUTATION TO EVALUATE ML METHOD #####

try:
    f = open("FINISHED", "x")
except:
    print("Error creating FINISHED file!")
```

Running the experiments

Given the large number of experiments to perform, it is of paramount importance to automate the execution, monitoring, and data collection of experimental results.

[5] Available at https://github.com/rhiever/sklearn-benchmarks.

```
#!/bin/bash
##### Variables and Functions #####
function join_by {local IFS="$1"; shift; echo "$*";}
DATASETS=$1; INI_REP=0; MAX_REP=3; QUANTUM=200
EVENTS="inst_retired.any,cpu_clk_unhalted.thread"
CORES="0,1,2,3,4,5,6,7,8,9,10,11"

##### Prepare the Environment #####
mkdir -p data; mkdir -p run
echo performance | tee /sys/devices/system/cpu/cpu*/cpufreq/
    scaling_governor > /dev/null
echo 2100000 | tee /sys/devices/system/cpu/cpufreq/policy*/
    scaling_min_freq > /dev/null
echo 2100000 | tee /sys/devices/system/cpu/cpufreq/policy*/
    scaling_max_freq > /dev/null

##### Experiments Main Loop #####
for ((REP=${INI_REP};REP<${MAX_REP};REP++)); do
    while read DS; do
        echo $DS $((REP+1))/${MAX_REP}
        DS=$(echo $DS | tr '\-[],' " ")
        ID=$(join_by - ${DS[@]})
        OUT=data/${ID}_${REP}.csv
        FIN_OUT=data/${ID}_${REP}_fin
        ## Start running workload
        mkdir -p run/${ID}_${REP}
        cd run/${ID}_${REP}
        taskset -c ${CORES} PATH/LogisticRegression.py ${DS} >
            out 2> err &
        FILE=STARTED
        while [ ! -f "$FILE" ]; do echo "$WL not started!"; done
        ## Start monitoring workload
        start=$(date +%s.%N)
        echo "${DS} has started!"
        perf stat -e ${EVENTS} -C ${CORES} -I ${QUANTUM} -x ',' -
            A -o ../../${OUT} &
        ## workload is running
        FILE=FINISHED
        while [ ! -f "$FILE" ]; do sleep 0.2; done
        ## Record execution time
        end=$(date +%s.%N)
        runtime=$(echo "$end - $start" | bc)
        echo $runtime > ../../${FIN_OUT}
        ## Stop monitoring Perf
        echo "$DS has finished!"
        killall perf
        cd ../..
    done < $DATASETS
done
```

Fig. 14 Example of the shell script used to launch the experiments for the Logistic Regression algorithm

For this purpose, we have created a shell script that allows running a given ML method with a list of datasets and monitoring the performance. Figure 14 presents this script's code for the specific Logistic Regression case. Note that this script can be used for any other algorithm by changing the name of the file to be executed[6] (in this case `LogisticRegression.py`) in line 25. The different parts of the script are described in the following subsections.

Variables and Functions Declaration The top part of the script in Fig. 14 defines the variables to be used:

- `DATASETS`. List of datasets to be processed by the ML method. Notice that the `DATASETS` variable is defined by the first command-line argument passed to the

[6] `PATH` should be replaced by real the file path.

```
1  - [krkopt]
2  - [letter]
3  - [magic]
4  - [nursery]
5  - [pendigits]
6  - [coil2000]
7  - [agaricus_lepiota]
8  - [mushroom]
9  - [ring]
10 - [ann_thyroid]
```

Fig. 15 Example of the YAML file with the list of datasets

script. We use the YAML[7] language to detail the list of datasets. Figure 15 shows an example of the YAML file format with the list of the ten datasets analyzed in this work.

- `INI_REP` and `MAX_REP`. Number of repetitions of the experiments to be performed. Experiments have been repeated a minimum of 3 times until the results present minor variations (less than 5%) among three consecutive repetitions.
- `EVENTS`. List of performance events to be monitored. For illustrative purposes, Fig. 14 includes the names of the events corresponding to the number of instructions retired (`inst_retired.any`) and the number of execution cycles when the thread is not halted (`cpu_clk_unhalted. thread`). The names of all the available performance events can be obtained by executing the `perf list` command or in the manuals of the processor's microarchitecture. Section 6 details the performance events that need to be monitored to calculate the analyzed metrics.
- `QUANTUM`. Length of quantum in milliseconds.
- `CORES`. List of logical cores where the workload will be executed. For ST experiments, all 12 cores in our experimental platform (from 0 to 11) have been utilized. For SMT, half of the physical cores have been used so that the same amount of logical cores (i.e., threads) are available to the workload. The file `/proc/cpuinfo` provides logical to physical core mapping information.

Finally, the top section of the script also defines the function `join_by`, which extracts the names of the datasets from the lines of the YAML file by removing special characters and spaces.

Environment Preparation The next step performed by the script is preparing the execution environment. To this end, it creates two folders named `data` and `run` to store the data collected from the performance counters and the output from each experiment, respectively.

Then, it sets the processor frequency to its base frequency (lines 11 and 12), which is 2.10GHz in our experimental platform. Although the system allows running in higher frequencies, the base frequency has been selected to ensure the machine does not overheat. Otherwise, if a higher frequency is set and the processor overheats, the OS automatically throttles the clock frequency until the temperature drops and is within the allowed threshold, after which the frequency will be raised again to reach the configured value. These

[7] http://yaml.org/.

fluctuations in the working frequency cause significant variations in the results; thus, we avoid them by setting the frequency to the base value.

Execution Main Loop Once the environment has been prepared, the experiments are executed. Two nested loops are used to iterate over all the datasets in the YAML file (line 15) for the number of repetitions defined (line 14). In each iteration, the first step is to run the ML algorithm (i.e., Logistic Regression in the figure) with the dataset `${DS}`. The command `taskset` is used to set the workload affinity to the specified cores (`${CORES}`).

As we want to evaluate the parallel computation of ML workloads, we do not start monitoring the workload from the start of its execution but after the STARTED file has been created (line 27) because the creation of this file signals the start of the parallel computation. To monitor the hardware performance counters, we use `perf stat` command (line 31). Since an ML algorithm spawns many processes during its execution to perform the parallel computation, and the creation of these processes cannot be easily monitored, we opt to perform monitoring at a logical core level (`-C ${CORES}`). Note that the monitoring is performed every quantum period (`-I ${QUANTUM}`).

Performance counters are sampled until the shell script detects a file named FINISHED has been created (line 34). When this occurs, the execution time is recorded in a file inside the `data` folder (lines 36–38), and the Perf process is killed to stop monitoring the performance counters (line 41).

It is important to note that root privileges are needed to execute this script since actions like reading the performance counters or fixing the processor frequency cannot be performed with user-level privileges.

Acknowledgements
Not applicable.

Author contributions
All authors contributed equally to the work's conception and design. Experimental evaluation, software development, and workload installation were performed by L.P., M.N., and J.P. L.P. performed data collection and representation. J.S. was responsible for the project administration, supervision, and data validation. J.S., M.E.G., and S.P. were responsible for the funding acquisition and administration. All the authors contributed to the writing and reviewing of the manuscript.

Funding
This work has been supported by the Spanish Ministerio de Ciencia e Innovación and European ERDF under grants PID2021–123627OB-C51 and TED2021–130233B-C32. Marta Navarro is supported by Subvenciones para la contratación de personal investigador predoctoral by CIACIF/2021/413.

Availability of data and materials
Datasets from PMLB are publicly available at https://github.com/EpistasisLab/pmlb, as well as the code for classification algorithms used https://github.com/rhiever/sklearn-benchmarks. Appendix A provides a detailed explanation to help the reproducibility of the experiments conducted in this work.

Declarations

Ethics approval and consent to participate
Not applicable.

Consent for publication
Not applicable.

Competing interests
Not applicable.

Received: 1 May 2024 Accepted: 13 October 2024
Published: 29 October 2024

References

1. Eggers SJ, Emer JS, Levy HM, Lo JL, Stamm RL, Tullsen DM. simultaneous multithreading: a platform for next-generation processors. IEEE Micro. 1997;17(5):12–9.
2. Feliu J, Eyerman S, Sahuquillo J, Petit S, Eeckhout L. Improving IBM POWER8 performance through symbiotic job scheduling. IEEE Trans Parallel Dis Syst. 2017;28(10):2838–51.
3. Feliu J, Sahuquillo J, Petit S, Eeckhout L. Thread isolation to improve symbiotic scheduling on SMT multicore processors. IEEE Trans Parallel Dis Syst. 2020;31(2):359–73.
4. Thiyagalingam J, Shankar M, Fox G, Hey T. Scientific machine learning benchmarks. Nat Rev Phys. 2022;4(6):413–20.
5. Hey T, Butler K, Jackson S, Thiyagalingam J. Machine learning and big scientific data. Philos Trans Royal Soc A. 2020;378(2166):20190054.
6. Umer A, Mian AN, Rana O. Predicting machine behavior from Google cluster workload traces. Concurrency and Computation: Practice and Experience, 2022;7559
7. Coleman C, Narayanan D, Kang D, Zhao T, Zhang J, Nardi L, Bailis P, Olukotun K, Ré C, Zaharia M. Dawnbench: An end-to-end deep learning benchmark and competition. Training. 2017;100(101):102.
8. James S, Ma Z, Rovick Arrojo D, Davison AJ. RLBench: The robot learning benchmark & learning environment. IEEE Robotics and Automation Letters;2020.
9. Thiyagalingam J, Leng K, Jackson S, Papay J, Shankar M, Fox G, Hey T. SciMLBench: A Benchmarking Suite for AI for Science 2021; https://github.com/stfc-sciml/sciml-bench
10. RedisAI: aibench 2020; https://github.com/RedisAI/aibench
11. Jiang Z, Gao W, Wang L, Xiong X, Zhang Y, Wen X, Luo C, Ye H, Lu X, Zhang Y. HPC AI500: a benchmark suite for HPC AI systems. In: Proceedings of Bench, 2018;pp. 10–22. Springer
12. Mattson P, Cheng C, Diamos G, Coleman C, Micikevicius P, Patterson D, Tang H, Wei G-Y, Bailis P, Bittorf V. Mlperf training benchmark. Proc Mach Learn Syst. 2020;2:336–49.
13. Ketkar N, Moolayil J. Introduction to pytorch. In: deep learning with python, 2021;pp. 27–91. Springer.
14. Abadi M, Barham P, Chen J, Chen Z, Davis A, Dean J, Devin M, Ghemawat S, Irving G, Isard M. {TensorFlow}: a system for {Large-Scale} machine learning. In: Proceedings of OSDI, 2016;pp. 265–283
15. Bradbury J, Frostig R, Hawkins P, Johnson MJ, Leary C, Maclaurin D, Necula G, Paszke A, VanderPlas J, Wanderman-Milne S, Zhang Q. JAX: composable transformations of Python+NumPy programs. 2018; http://github.com/google/jax
16. Chen T, Li M, Li Y, Lin M, Wang N, Wang M, Xiao T, Xu B, Zhang C, Zhang Z. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. 2015; arXiv preprint arXiv:1512.01274
17. Ma Y, Yu D, Wu T, Wang H. PaddlePaddle: an open-source deep learning platform from industrial practice. Front Data Comput. 2019;1(1):105–15.
18. Blagojević V, Bojić D, Bojović M, Cvetanović M, Đorđević J, Đurđević Đ, Furlan B, Gajin S, Jovanović Z, Milićev D. A systematic approach to generation of new ideas for PhD research in computing. In: Advances in Computers 2017;vol. 104, pp. 1–31. Elsevier.
19. Feliu J, Sahuquillo J, Petit S, Duato J. Addressing Fairness in SMT Multicores with a Progress-Aware Scheduler. In: Proceedings of IPDS, 2015; pp. 187–196
20. Singh JP, Weber W-D, Gupta A. SPLASH: stanford parallel applications for shared-memory. ACM SIGARCH Comput Architecture News. 1992;20(1):5–44.
21. Bailey D, Harris T, Saphir W, Van Der Wijngaart R, Woo A, Yarrow M. The NAS parallel benchmarks 2.0. Technical report, Technical Report NAS-95-020, NASA Ames Research Center 1995.
22. Dongarra JJ, Luszczek P, Petitet A. The LINPACK benchmark: past, present and future. Concurr Comput. 2003;15(9):803–20.
23. Henning JL. SPEC CPU2006 benchmark descriptions. ACM SIGARCH Comput Architecture News. 2006;34(4):1–17.
24. Segal MR. Machine learning benchmarks and random forest regression 2004.
25. Stallkamp J, Schlipsing M, Salmen J, Igel C. Man vs. computer: benchmarking machine learning algorithms for traffic sign recognition. Neural Netw. 2012;32, 323–332
26. Singh A, Thakur N, Sharma A. A review of supervised machine learning algorithms. In: Proceedings of INDIACom, 2016;pp. 1310–1315
27. Maseer ZK, Yusof R, Bahaman N, Mostafa SA, Foozy CFM. Benchmarking of machine learning for anomaly based intrusion detection systems in the CICIDS2017 dataset. IEEE Access. 2021;9:22351–70.
28. Ge S, Song J, Lai C, Li E, Hu W, Tian X. Performance evaluation of SNPs machine-learning workload on Intel pentium 4 hyper-threading architectures. In: Proceedings of PDCS 2004.
29. Cepillo JAM. A Microarchitectural Analysis of Machine Learning Algorithms on Spark. PhD thesis, University of Toronto (Canada) 2017.
30. Prieto P, Abad P, Gregorio JA, Puente V. Performance characterization of popular DNN models on out-of-order CPUs. In: Proceedings of PACT, 2023;pp. 199–210.
31. Wang YE, Wu C-J, Wang X, Hazelwood K, Brooks D. Exploiting parallelism opportunities with deep learning frameworks. ACM Trans Archit Code Optim. 2021;18(1):1.
32. Mittal S, Rajput P, Subramoney S. A survey of deep learning on CPUs: opportunities and co-optimizations. IEEE Tras Neural Netw Learning Syst. 2022;33(10):5095–115.
33. Li D, Chen X, Becchi M, Zong Z. Evaluating the energy efficiency of deep convolutional neural networks on CPUs and GPUs. In: Proceedings of BDCloud-SocialCom-SustainCom, 2016;pp. 477–484

34. Yasin A. A top-down method for performance analysis and counters architecture. In: Proceedings of ISPASS, 2014;pp. 35–44. IEEE
35. Asuncion A, Newman D. UCI machine learning repository. Irvine, CA, USA (2007). http://archive.ics.uci.edu/ml Accessed 1 Jan 2023.
36. Goldbloom A, Hamner B, Moser J, et al. Kaggle: your machine learning and data science community 2010; https://www.kaggle.com/ Accessed 1 Jan 2023.
37. Vanschoren J, Rijn JN, Bischl B, Torgo L. OpenML: networked science in machine learning. SIGKDD Explor. 2013;15:49–60.
38. Derrac J, Garcia S, Sanchez L, Herrera F. Keel data-mining software tool: Data set repository, integration of algorithms and experimental analysis framework. J Mult Valued Logic Soft Comput. 2015;17:255–87.
39. Macia N, Bernadó-Mansilla E. Towards UCI+: a mindful repository design. Inf Sci. 2014;261:237–62.
40. Olson RS, La Cava W, Orzechowski P, Urbanowicz RJ, Moore JH. PMLB: a large benchmark suite for machine learning evaluation and comparison. BioData Mining. 2017;10(1):1–13.
41. Olson RS, La Cava W, Mustahsan Z, Varik A, Moore JH. Data-driven Advice for Applying Machine Learning to Bioinformatics Problems. arXiv e-print. https://arxiv.org/abs/1708.05070 2017.
42. Pedregosa F, Varoquaux G, Gramfort A, Michel V, Thirion B, Grisel O, Blondel M, Prettenhofer P, Weiss R, Dubourg V, Vanderplas J, Passos A, Cournapeau D, Brucher M, Perrot M, Duchesnay E. Scikit-learn Mach Learn Python. J Mach Learning Res. 2011;12:2825–30.
43. Kotsiantis SB, Zaharakis I, Pintelas P. Supervised machine learning: a review of classification techniques. Emerging artificial intelligence applications in computer engineering. 2007;160(1):3–24.
44. Hastie T, Tibshirani R, Friedman JH, Friedman JH. The elements of statistical learning: data mining, Inference, and Prediction vol. 2. Springer, 2009.
45. Demšar J. Statistical comparisons of classifiers over multiple data sets. J Mach Learning Res. 2006;7:1–30.
46. Ben-Nun T, Besta M, Huber S, Ziogas AN, Peter D, Hoefler T. A modular benchmarking infrastructure for high-performance and reproducible deep learning. In: Proceedings of IPDPS, 2019; pp. 66–77. IEEE
47. Braun S. LSTM benchmarks for deep learning frameworks. arXiv preprint arXiv:1806.01818 2018.
48. Beheshti N. Cross Validation and Grid Search 2022; https://towardsdatascience.com/cross-validation-and-grid-search-efa64b127c1b

Publisher's Note
Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.