Performance Characterization of Popular DNN Models on Out-of-Order CPUs

***Abstract*—DNN popularity, which is driving advances in a growing number of fields, has increased the amount of computing resources running this kind of applications at an unprecedent rate. Specialized hardware, such as GPUs or ASIC-based accelerators, has been the preferred platform to run these applications. However, the ubiquity of DNN models is rapidly extending the presence of this software to general-purpose CPUs. For this reason, there is a pressing need to gain understanding of the main features of state-of-the-art DNN models to adapt CPU microarchitecture accordingly. In this paper we investigated a representative set of DNN models and, based on data collected from real hardware, we evaluated how efficiently they utilize the underlying system. We analyzed overall system performance, as well as the amount of vectorization provided by CPU-optimized frameworks. We quantified the performance loss caused by processor backend, and the contribution of memory hierarchy and functional units to it. We compared the backend utilization of DNN applications to popular benchmarks such as SPEC CPU2017 and found a lower balance in the use of the elements that make up the processor microarchitecture. Although many workloads seem to be constrained by functional unit availability, in a significant group of applications we found a non-negligible impact of memory hierarchy on performance.**

Keywords—Deep Neural Networks, Metrics, CPU

I. INTRODUCTION

The DNN revolution has created an unprecedent growth in fields where these applications are driving current advances. Since the relatively recent success of Convolutional Neural Networks for image recognition [1], DNN models have rapidly acquired relevance in such heterogeneous areas as language processing [2][3], gaming [4]or medical diagnosis [5]. The large amount of computational resources required to train many of these models has made specialized hardware such as GPUs or FPGAs the preferred computing platform. However, in such a heterogeneous scenario, the utilization of general-purpose CPUs is rapidly growing [6]. Aspects such as the flexibility and high availability of CPUs are some of the advantages of CPUs for DNN computing [7][8] and multiple training and inference processes are currently performed in this kind of hardware [9][10].

The main CPU manufacturers have recently started to include specific hardware features to accelerate DNN execution in some of their commercial products. IBM's POWER10 processor introduced Matrix-Multiply Assist (MMA) instructions and functional units that perform two-dimensional matrix operations. The third generation of Intel Xeon Scalable processors include AVX-512 ISA extensions [10] implementing both deep learning inference (Fused Multiply-Add instruction for 8-bit multiplies and 32-bit accumulates) and training-oriented instructions (multiple instructions to operate with bfloat16 datatype [11]). More recently, Intel announced the inclusion of Advanced Matrix Extension to their ISA [12], a new programming paradigm consisting of a set of 2-dimensional registers and an accelerator able to operate on them. Similarly, contemporary M1 and M2 processors from Apple seem to include a set of instructions intended to execute matrix multiplication operations on a special acceleration unit inside the CPU. All these features clearly target Artificial Intelligence workloads, more specifically Deep Neural Networks. This trend seems to confirm the relevance of CPUs as a hardware platform for these applications.

However, the definition of DNN models is usually done by means of high-level programming languages such as python. The API of most popular machine learning platforms such as TensorFlow [13] or PyTorch [14] is defined as a python library. Internally, these frameworks are implemented making use of highly optimized algebra libraries written in C/C++ such as Eigen or Intel's MKL. The way these libraries are compiled determines the low-level code generated and is critical to obtain as high a performance as possible. This "distance" between high-level definition of a DNN model and actual ISA instructions makes it difficult to understand how these applications are suited to running on contemporary hardware systems. In-depth understanding of the hardware-software interactions is a key aspect to make the correct design decisions in next generation microarchitectures and is the main goal of this work. To do so, we undertake a detailed evaluation of a representative group of DNN applications (belonging to Intel's Model Zoo) to discover their performance characteristics on modern general-purpose processors. Based on data collected from real hardware, our evaluation aims to provide meaningful insights not only for computer architects to design targeted hardware with improved compatibility with DNN models, but also for applications designers to understand the performance of their software and tune it to better exploit underlying systems.

In the light of the ever-increasing diversity and number of DNN models, we evaluate a wide range of applications and seek to find out how they compare against each other. Based on our evaluation, we show that the variability in terms of both performance and inherent characteristics between the different models is marginal. We also compare the use that DNN applications make of microarchitectural elements to that made by other applications, regularly present in general-purpose hardware environments, such as desktop-computing (SPEC CPU2017). The objective of this comparison is to obtain a reference point with applications that, most likely, have served as benchmarks during processor design phases.

Our evaluation identifies a strong regular behavior of the tested workloads in terms of specific performance characteristics (e.g., cache behavior), and significant differences when compared to regular general-purpose applications. We have not been able to find a publication covering all the features presented in this work, making it a robust reference on fundamental aspects which can sometimes be treated in a vague and insufficiently generic way. Our work can also

aid performance analysts by choosing a smaller subset (7) of benchmarks representative of all of them (32). This will simplify performance evaluation in early stages of design analysis and/or using time-accurate hardware simulators. Based on our evaluation of the performance of DNN models, we seek to answer the following questions:

- How effectively do modern DNN models utilize the underlying computer system? For instance, is cache hierarchy suitable for these applications?
- Which are the main sources of performance inefficiencies? What microarchitectural elements should be added/improved to increase performance?
- Is the performance of DNN applications noticeably different from other application classes? If so, what are the most unique features defining the behavior of these applications?
- Do DNN applications display very heterogeneous behavior, or can they be grouped into a small number of clusters?

The rest of this paper is organized as follows: In Section II, we present an overview of the DNN models and our methodology. In section III, we describe our test infrastructure. Section IV presents our results and analysis. Finally, we discuss previous work and summarize our key findings in sections V and VI respectively.

II. METHODOLOGY

In this section, we present a brief background of the tested DNN models and their datasets, as well as our experimental methodology.

A. Models

The group of applications evaluated in this work correspond to the "Model Zoo for Intel Architecture" repository. This repository contains a representative group of popular open-source deep learning models **optimized** to run on intel Xeon processors. Available models are implemented through both TensorFlow and PyTorch frameworks. In most cases, models are deployed to run inference processes, but some of them also include training. Additionally, some models can be deployed with alternative data types, from "classic" 32-bit floats to smaller 16-bit bfloat format or discretized 8-bit integer weights.

The Model Zoo is organized in categories, corresponding to the field where each model is employed. In the following subsections we will describe the models in each category, as well as the datasets employed for training and inference. To make the results more concise, we have eliminated those models whose similarity of results leads to very similar behaviors. TABLE I. summarizes all the models employed in this study, with their main features and the labels that represent them in the graphs of the rest of the document. In the rest of the subsection we summarize the functionality of each model.

1) Image Recognition

Image recognition (or image classification) is the task of predicting the type or class of an object in an image and categorizing it in one of several predefined distinct classes. The input in these models corresponds to an image (usually a photograph) with a single or multiple objects, and the expected prediction is a list of class labels (e.g., one or more integers that are mapped to class labels).

The models in this category correspond mainly to alternative implementations of convolutional neural networks (CNNs). DenseNet [15] is a DNN architecture with an extremely dense connectivity pattern. All layers with matching feature-map sizes are directly connected to each other to maximize information flow. This preserves the feed-forward nature, each layer obtains additional inputs from all preceding layers and passes on its own feature-maps to all subsequent layers. Inception-based [16][17][18] network models are focused on the optimization of computational cost to perform well even under strict memory and computational constraints. Expensive convolutions with large spatial filters are replaced by multi-layer networks with much smaller filters [19], drastically reducing the number of parameters in the model. Similarly, MobileNet [20] models are light weight CNNs implemented making use of a special form of factorized convolutions. A standard convolution is factorized into a depthwise convolution and a 1×1 convolution to combine the outputs. This factorization has the effect of drastically reducing computation and model size. ResNet models [21] address the accuracy degradation problems derived from increasing network depth. Skip connections are used to jump over some layers performing identity mapping, and their outputs are added to the outputs of the stacked layers.

TABLE I. SUMMARY OF DNN MODELS EVALUATED

Model	*Dataset*	*Framework*	*Category*	*Label*
DenseNet169	ImageNet	TensorFlow	Img Recognition	TF-DN169
Inception V4	ImageNet	TensorFlow	Img. Recognition	TF-INCV4
MobileNet V1	Imagenet	TensorFlow	Img. Recognition	TF-MOBV1
Inception V3	Imagenet	PyTorch	Img. Recognition	PY-INCV3
ResNet 50	Imagenet	PyTorch	Img. Recognition	PY-RN50
ResNet 101	Imagenet	PyTorch	Img. Recognition	PY-RN101
ResNet 152	Imagenet	PyTorch	Img. Recognition	PY-RN152
VGG-11	Imagenet	PyTorch	Img. Recognition	PY-VG11
GoogLeNet	Imagenet	PyTorch	Img. Recognition	PY-GOOG
MaskRCNN	MS COCO	Tensorflow	Img. Segmentation	TF-MASKR
UNet	Synthetic	TensorFlow	Img. Segmentation	TF-UNET
BERT	SQuAD	TensorFlow	Lang. Modeling	TF-BERTM
BERT base	SQuAD/MRPC	PyTorch	Lang. Modeling	PY-BERTB
BERT large	SQuAD	PyTorch	Lang. Modeling	PY-BERTL
DistilBERT	SQuAD	PyTorch	Lang. Modeling	PY-DISTB
RNN-T	RNN-T	PyTorch	Lang. Modeling	PY-RNNT
RoBERTa	SQuAD	PyTorch	Lang. Modeling	PY-ROBR
T5	T5-Large	PyTorch	Lang. Modeling	PY-T5
BERT-Trans	MRPC	TensorFlow	Lang. Translation	TF-BERTT
Faster R-CNN	COCO	TensorFlow Pytorch	Obj. Detection	TF-FASTR PY-FASTR
R-FCN	COCO	TensorFlow	Obj. Detection	TF-RFCN
SSD-MobileNet	COCO	TensorFlow	Obj. Detection	TF-SSDM
SSD-ResNet34	COCO	TensorFlow Pytorch	Obj. Detection	TF-SSDR PY-SSDR
MaskR-CNN	COCO	Pytorch	Obj. Detection	PY-MASK
MaskR-CNN ResNet50	COCO	Pytorch	Obj. Detection	PY-MKRN
RetinaNet	COCO	Pytorch	Obj. Detection	PY-RETN
NCF	Movielens	TensorFlow	Recommendation	TF-NCF
Wide & Deep	Display Advertise	TensorFlow	Recommendation	TF-WANDD
WaveNet	Synthetic	TensorFlow	Text to Speech	TF-WAVE
TransNetV2	Synthetic	Pytorch	Shot Boundary Detect.	PY-TRANS

All these convolutional models are trained and evaluated making use of the ImageNet dataset [22]. ImageNet is a visual database with more than 14 million images. These images are hand-annotated indicating what objects are pictured. ImageNet contains more than 20,000 categories; each one typically contains several hundred images.

2) Object Detection and Image Segmentation

These processes extend the classification task, predicting the presence of objects in an image and indicating their location. Images (or video frames) are partitioned into multiple segments or objects that belong to the same object class. The expected prediction in the Object Detection process is a list of bounding boxes (e.g., defined by a point, width, and height) locating each object in the input image. Segmentation carries out a more fine-grained detection process, trying to fit segments to object shape. The expected prediction in this case is a pixel-level labeling with a set of object categories (for example, people, trees, sky, cars) for all image pixels.

In this field CNN models are also predominant. Region-based convolutional network [23] is one of the methods achieving excellent accuracy in object detection. Its architecture relies on a two-stage approach, each implemented with a CNN. The first stage generates candidate boxes and the second one implements the classification process. Alternative models such as Faster R-CNN [24], Mask R-CNN [25] or R-FCN [26] employ the same baseline structure, with alternative implementations of some modules. Some alternative methods such as the SSD approach [27] or Retinanet [28] do not make use of the two-stage structure. SSD architectures are based on a single feed-forward convolutional network that produces a fixed-size collection of bounding boxes and scores for the presence of object class instances in those boxes, followed by a non-maximum suppression step to produce the final detections. The early network layers can be based on alternative standard architectures used for image classification, such as ResNet or MobileNet. RetinaNet is composed of a main CNN in charge of computing a convolutional feature map over an entire input image and two subnetworks (also CNNs) to perform object classification and bounding box regression.

Detection and Segmentation models make use of specific datasets, such as Microsoft COCO (Common Objects in Context) [29]. The dataset contains 91 common object categories with 82 of them having more than 5,000 labeled instances. In total the dataset has 2,500,000 labeled instances in 328,000 images. For each image, every instance of every object category is labeled and fully segmented.

3) Language Modeling and Translation

DNNs have also demonstrated their relevance in several fields related to natural language processing such as question answering, text classification, language modeling, translation, etc. Focusing on processing sequential data and time series, Recurrent Neural Networks are naturally fitted for these tasks. Sentences are treated sequentially, capturing relations among words. More recently, transformers have demonstrated improved results in many NLP tasks, making use of attention mechanisms. Attention avoids the use of RNNs or CNNs, improving the representation task (Encoding) and allowing the parallel processing of input tokens.

BERT Model [30] architecture is a multi-layer bidirectional Transformer encoder based on the original transformed implementation [2]. Its main target consists of pretraining language representations (modeling), which can be employed in multiple NLP tasks such as translation, question answering or next sentence prediction. Alternative models such as RoBERTa [31] or DistilBERT [32] are optimized implementations of BERT targeting computational effort (DistilBERT reduces model size by 40% while retaining 97% of language understanding capabilities) or accuracy (RoBERTa retrains BERT model to match performance of or exceed performance of previous alternatives). RNN-T [33] is an end-to-end model that makes use of recurrent neural networks to perform automatic speech recognition. This model is made up of two components, prediction and transcription networks, each of them implemented as an RNN with multiple LSTM hidden layers.

There are multiple datasets related to NLP. The models in this work make use of two of the most relevant ones. The Stanford Question Answering dataset (SQuAD) [34] is a reading comprehension dataset, consisting of 100,000 questions on a set of more than 500 Wikipedia articles, where the answer to every question is a segment of text from the corresponding reading passage. The Microsoft Research Paraphrase Corpus (MRPC) [35] contains 5801 pairs of sentences extracted from web news sources, along with hand-labeled annotations indicating whether each pair constitutes a paraphrase.

4) Recommendation

Multiple works have already shown the power of DNN for user modeling tasks such as recommendation. Deep learning-based recommendation systems is widely used throughout industry to predict rankings for news feed posts or entertainment content. Content-based filtering systems try to infer the behavior of users given their positive reactions to an item's features. Collaborative filtering systems evaluate which items a particular user likes and the items that users with similar behavior like, to infer recommendations to that user.

Neural Collaborative Filtering (NCF) [36] proposes a neural network-based collaborative learning framework that will use Multi perceptron layers to learn user-item interaction function. The Wide & Deep learning framework [37] achieves both memorization (learning frequent co-occurrence of items and exploit correlations) and generalization (exploring unseen item combinations) in a single model, by jointly training a linear model and a multi-layer perceptron DNN. Finally, DLRM models [38] implement a two-stage recommendation system where interactions are also processed at each stage through a multi-layer perceptron.

Recommendation datasets are quite heterogeneous, and the applications from the Model Zoo evaluated make use of three different datasets. NCF employs MovieLens [39], a benchmark dataset with 1 million ratings from 6000 users on 4000 movies. Large & Deep and DLRM use display advertising datasets from Criterio [40], which estimates the probability of clicking on a given ad given a user and the page he is visiting.

B. Profiling CPUs with Top-Down

Reasoning about low-level behavior in out-of-order superscalar microarchitectures is difficult for applications with complex software stacks. This is a key task for both software and hardware optimization, and for this reason new processor generations provide a growing amount of hardware runtime information through their Performance

Monitoring Units (PMUs). These units are configured to count and collect multiple hardware events simultaneously, writing results to special registers accessible by the programmer.

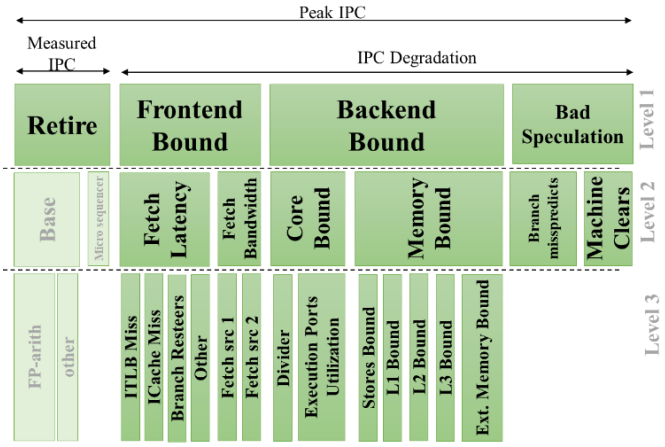

Fig. 1. Top-Down Hierarchy

Organizing the large number of metrics available to identify performance bottlenecks in the microarchitecture is far from straightforward. Fortunately, methodologies such as Top-Down [41] can help with this issue. This methodology describes the processor pipeline through a multi-level hierarchy of components, using and organizing PMU events to identify which fraction of pipeline stalls are caused by each component. The use of this methodology is regular for tasks such as software optimization [42][43][44], characterization [45][46][47][48] and hardware evaluation [49][50][51].The complete hierarchy, described in Fig. 1, defines the first-level components of the hierarchy as follows:

- Frontend Bound: processor frontend groups the pipeline stages in charge of instruction fetching and decoding. Frontend stalls are related to cache instruction misses, (Frontend Latency Bound) or decoding inefficiencies (Frontend Bandwidth Bound).
- Bad Speculation: this component identifies performance stalls caused by the CPU executing a speculative instruction that is not retired later and the cycles devoted to restoring the processor state before prediction miss.
- Backend Bound: The backend includes the stages in charge of performing the operations defined by instructions. The stalls in this category are related to the inability to complete an operation, caused by the lack of any operand (Memory Bound) or the lack of available functional units (Core Bound).
- Retiring: this category represents correctly retired instructions. It does not represent stalls, but real performance, corresponding to the IPC performance metric.

As the hierarchy level increases, each part of the pipeline is described in more detail, splitting the microarchitecture into more specific components and determining their contribution to performance loss.

III. Experimental Framework

For our analysis, we use a cluster of nodes with 64-bit Intel® Xeon® Silver processors (Cascade Lake) at 2.10GHz, each with two sockets of 16 out-of-order cores with SMT. Each core has private L1 caches (32KB instruction and 32KB data cache) and 1MB L2 cache, and each socket share 22MB of LLC cache. Each node has 256GB of main memory. Each node runs Debian 11 with Linux kernel 4.19.

To run a representative set of deep learning models, we make use of Model Zoo for Intel® Architecture. We use the github repository [52] version r2.8 (commit b63c719), and execute the benchmark without modifications, following the recommended setup. Models are implemented using two different frameworks, PyTorch and TensorFlow, making use of the corresponding Intel® Extensions [53], [54] for an extra performance boost. We use PyTorch version 1.13, and TensorFlow version 2.5 and 1.15 when recommended, both running on Python 3. Our evaluation process assumes as a starting point that the software does not impose any unnecessary performance penalties. Model Zoo applications are fine-tuned to maximize performance when running on Intel CPUs.

We use appropriate mechanisms to measure the models during the execution of the region of interest. We define the region of interest as the region when the model is processing batches at inference, after data processing, model loading, and model warm-up. For the measures, we use the hardware event counters available in the PMU in the system under test, using Linux profiling command, *perf* [55], and Intel pmu-tools [56].

To provide a reference point, we evaluate, under the same conditions, conventional applications included in the benchmark suite SPEC CPU2017 in rate mode. The suite is compiled and executed with two different configurations:

- A standard configuration, with default options and GNU compilers gcc, g++ and gfortran version 10.2.1.
- A Xeon Silver specific configuration, based on previously presented reports, using Intel OneAPI compilers (icx, icpx and ifort), version 2022.2.0.

IV. Evaluation and Analysis

In this section, we carried out a performance evaluation of the described system by running both DNNs and SPEC 2017 applications. In each figure, an average value for the different suites will be represented. “AVG-DNN” is the average value across all the models of the Intel Model Zoo evaluated. “SPEC-INT” and “SPEC-FP” correspond to the average values for SPEC CPU integer and floating-point benchmarks respectively, while “AVG-SPEC” is the average of all the SPEC CPU applications.

A. Overall Micro-architectural Performance Efficiency

We will start our analysis by measuring the overall performance of DNN applications compared to SPEC CPU 2017 in rate mode. We are especially interested in the overall performance efficiency, and the performance related to floating-point execution. To do so, we will make use of well-known metrics such as instructions per cycle (IPC), and floating-point operations per cycle (FLOP/cycle), which, when executed in the same system, is equivalent to FLOPS without the interference caused by CPU frequency scaling mechanisms.

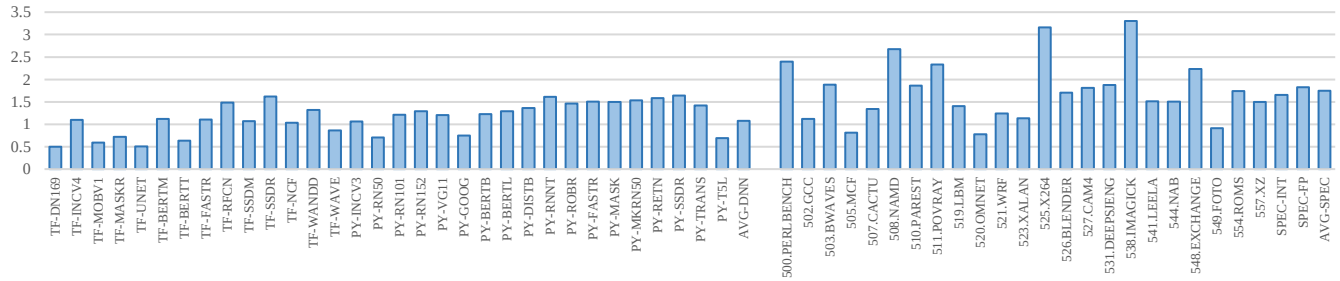

Fig. 2. Instructions per cycle (IPC) for all evaluated workloads

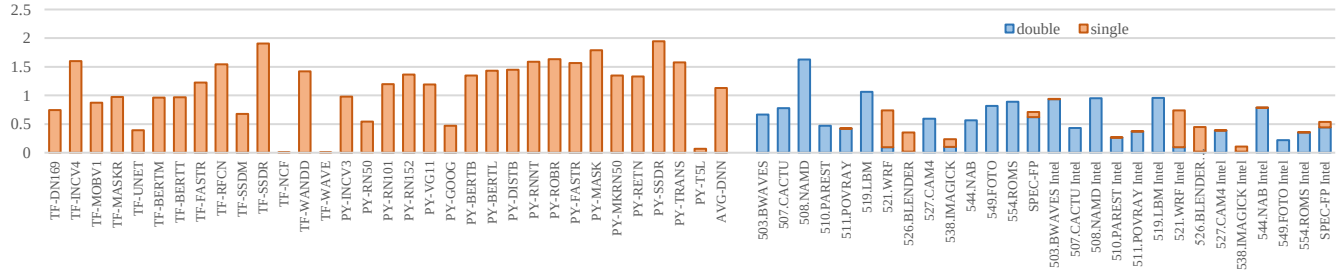

Fig. 3. Floating Points Instructions per Cycle (FLIN/cycle).

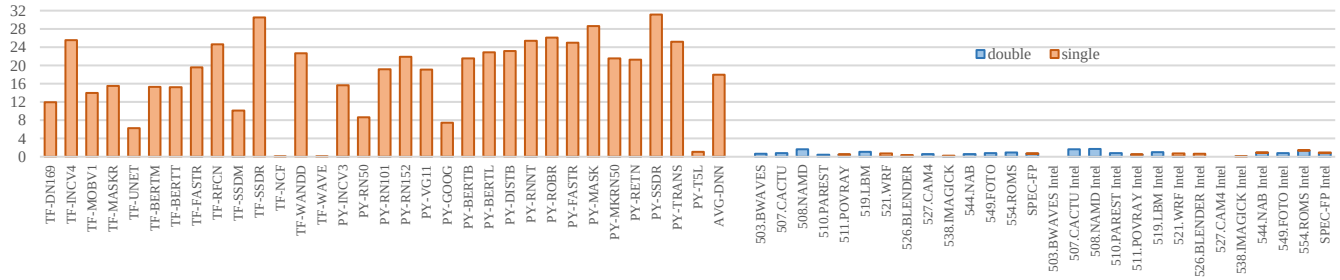

Fig. 4. Floating Point Operations per Cycle (FLOP/cycle).

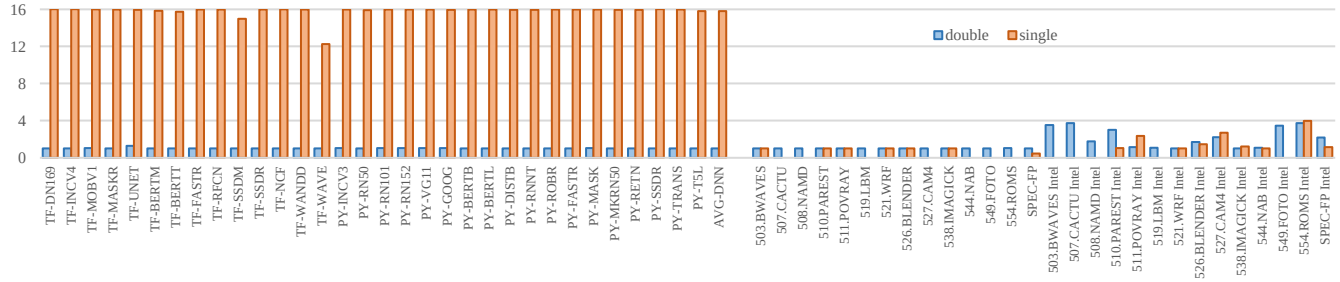

Fig. 5. Floating Point Operations per Floating Point Instruction.

Fig. 2 shows the IPC obtained for the different applications, considering that the maximum IPC achievable on the test system is 4. As can be seen, SPEC CPU 2017 applications obtain higher IPC on average, with six applications reaching over 2 instructions per cycle. On the other hand, none of the DNN applications can reach those values, remaining slightly below 1.1 instructions per cycle on average.

However, IPC can be a misleading metric, because it does not show what is really happening underneath. As can be seen in Fig. 3, DNN models generally have a greater number of floating-point instructions per cycle (FLIN/cycle), which is greater still when considering Floating Point Operations per cycle (FLOP/cycle) as seen in Fig. 4 (Note that some instructions, e.g. FMA, count as two instructions). This is because the amount of vectorization obtained by the DNN models is substantially higher than SPEC CPU 2017 Floating Point applications. As can be seen in Fig. 5, compiler auto-vectorization cannot achieve over 256b instructions in SPEC applications. Even though, when using Intel compiler tools with an Intel specific configuration vectorization increases, it is not comparable to DNN's. Most of the floating-point operations in DNN models are AVX-512, which, being mostly single precision, implies 16 operations per instruction. In contrast, SPEC floating-point applications barely obtains 4 operations per instruction at best, using Intel compiler tools.

B. Performance Breakdown (Level 1 Top-Down)

In an attempt to understand in depth the reasons for the performance differences, and the potential impact of the different

components of the processor pipeline we extended the analysis by applying Top-Down methodology. To facilitate the readability of the results, the components of each Top-Down level of the hierarchy will be normalized, as can be seen in the schematic diagram in Fig. 6.

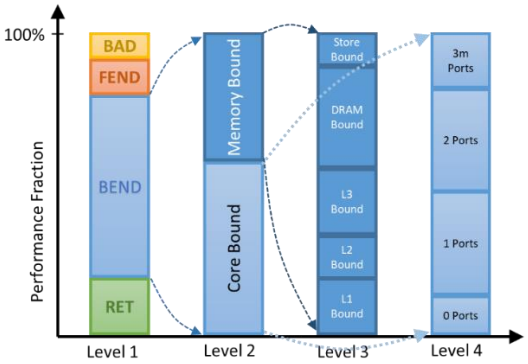

Fig. 6. Top-Down analysis breakdown for the different levels of the hierarchy in the following figures.

Beginning with the first level of the Top-Down hierarchy we obtain the data shown in Fig. 7, where the impact of frontend and backend in the overall performance can be observed, as well as the effect of miss speculation. As can be clearly seen, SPEC CPU 2017 applications have better overall performance (represented by the Retired portion of the Top-Down analysis, which matches the IPC metric mentioned above (Fig. 6)). Although they are not homogeneous, in the case of SPEC applications performance loss is distributed, on average, between 60% of stalls caused by the backend and 40% distributed between both the frontend and bad speculation stalls. However, in the case of DNN applications, more than 95% of the performance loss is due solely to stalls in the backend. Only snippets of the effect of the frontend appear in some of the models, related to Object Detection and Recommendation, without ever outweighing the impact of the backend.

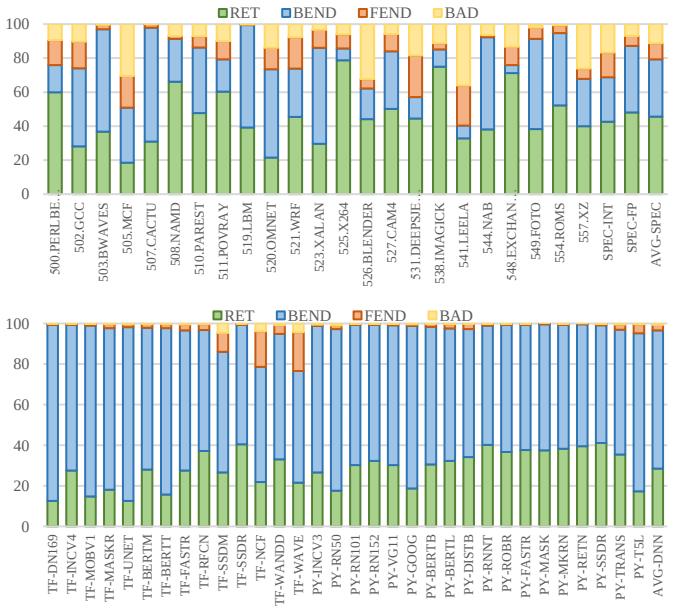

Fig. 7. Top-Down level 1 analysis of SPEC CPU 2017 applications (up) and DNN models from Intel Model Zoo (down)

C. *Backend Issues*

Taking these results into account, we extended the level of detail of the Top-Down hierarchy in the Backend, in order to find out where the bottlenecks are located.

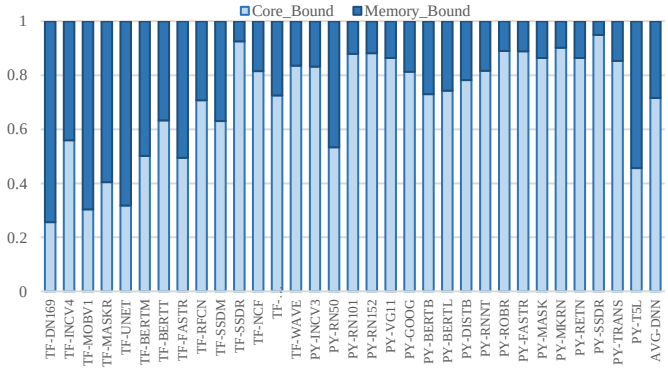

Fig. 8. Top-Down level 2 analysis of the Backend of DNN models from Intel Model Zoo.

Fig. 8 shows the second level of the Top-Down hierarchy related to the Backend, which is divided into Core Bound and Memory Bound. As can be seen, on average DNN applications seem to suffer mainly in the core portion of the backend, which could be related to the heavy use of vectorized floating point operations. However, the impact of the memory hierarchy is dominant in the applications corresponding to the image processing models in TensorFlow. This is important, because it indicates that, in about one third of the applications, core enhancement might have a reduced impact on performance.

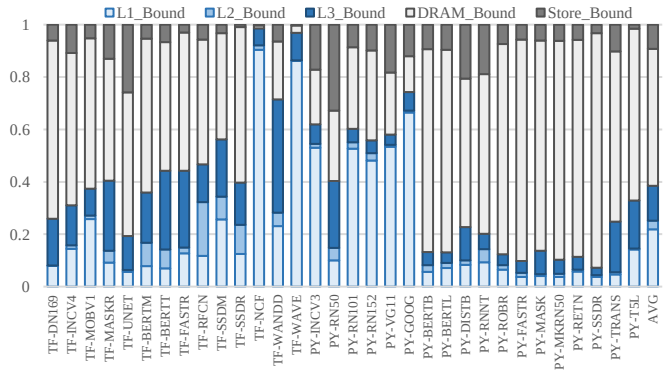

Fig. 9. Top-Down analysis level 3 of the Memory Bound portion of DNN models from Intel Model Zoo.

1) *Memory Hierarchy*

Continuing to explore the causes provoking the loss of performance, the next step is to carry out a level 3 Top-Down analysis focusing on memory. As can be seen in Fig. 9, main memory (DRAM) seems to be the main culprit of the memory hierarchy in the impact on performance in this type of applications. However, there are rare cases when a significant impact of the L1 cache can be observed. Specifically, in two models, NCF and Wavenet, belonging to the Recommendation and Text-to-Speech domains and most of the Image Recognition models from Pytorch. Other components of the memory hierarchy, such as L2 and L3 caches, have a smaller impact, which seems to indicate that, beyond the dataset that we can store in L1, the volume of the data managed and its reutilization, make it difficult to take advantage of spatial or temporal locality. That is, there is a strong

imbalance in the hierarchy and the L2 and L3 levels are unable to alleviate the long access times to the last DRAM level.

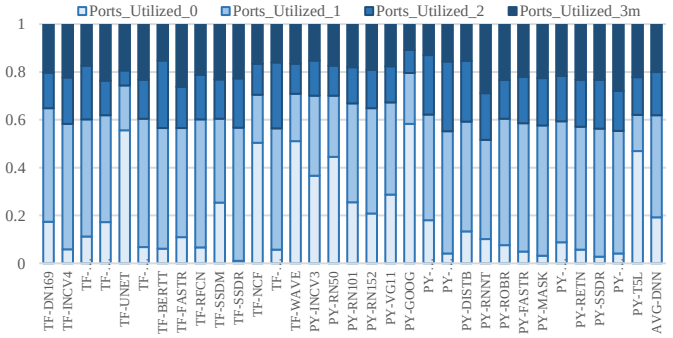

Fig. 10. Top-Down level 4 analysis of the Core Bound portion of DNN models from Intel Model Zoo

2) *Functional Units*

Focusing on the Core, the last levels of the Top-Down hierarchy provide no information other than the number of ports used on average per cycle. As can be seen in Fig. 10, on average, DNN applications use one port per cycle, with some few cases where most of the time the core is waiting. However, the information is not complete enough, and it is necessary to go further into the reasons why so few ports are being used at any given moment.

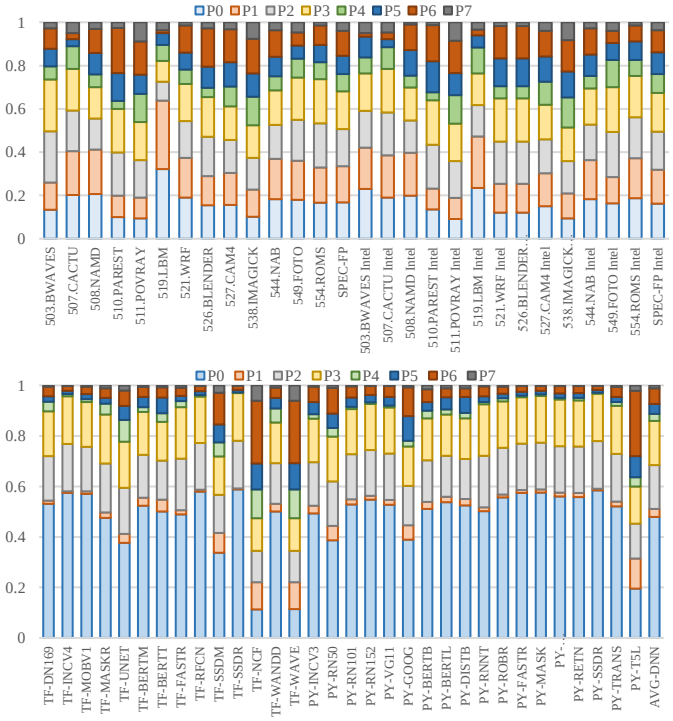

Fig. 11. Normalized cycles per core port utilization of SPEC CPU 2017 FP applications (up) and DNN models from Intel Model Zoo (down).

To further elucidate this information, we performed an analysis with the *perf* evaluation tool to find out the number of cycles during which each one of the eight ports of the core is being used. In Fig. 11, we represent this normalized information, so that the total port usage adds up to 100%. In this case we show results from running both DNN and SPEC CPU 2017 applications in order to compare behaviors. As can be seen, the use of execution units in SPEC CPU 2017 floating-point applications is quite homogeneous, which corresponds to what you want to see in the use of a CPU as a designer. In contrast, DNN models have a high usage of Port 0 in general. This port corresponds to various units, including FMA, Branch Execution and Floating-Point division, according to Intel documentation [57]. Clearly, DNN models have a need for vector execution units, specifically FMA, which confirms what we previously observed in Figure 4 and Figure 5. Load Units and Address Generation Units (located both on Port 2 and Port 3) also have a significant impact. In contrast, most DNN models make negligible usage of ports like 6 and 7, corresponding to Branch Execution Unit and Address Generation Unit.

There are some exceptions to this unbalanced behavior in port usage: mainly NCF, Wavenet and T5. At first glance, it might seem that these applications make more efficient use of the core, as in the case of SPEC CPU 2017 however their IPC is below 1 (Fig. 2). What seems to happen is that the number of floating-point instructions is low (Fig. 4). In some cases, specially T5, they have a significant memory bottleneck, while others, especially in WaveNet and NCF, the impact of the frontend is significant (Fig. 7).

D. *Second FMA Unit effect on the backend*

The Functional Unit configuration is a microarchitecture and processor specific feature. Including additional functional units seems to be a reasonable step to improve the performance of DNN applications on CPUs. In this section, we explore the impact of additional FMA units on backend behavior. The Cascade Lake microarchitecture has an AVX512 FMA unit in the Xeon® Silver family at its Port 0. In contrast, the Xeon® Gold processor family, adds a second AVX512 FMA unit at Port 5 among other features (mainly targeting operation frequency), presumably targeting applications similar to those analyzed in this work. We have performed a similar set of experiments for a 64-bit Intel® Xeon® Gold processor to those already performed for the Xeon Silver processor. The only change in this new set of experiments is the processor, the rest of the hardware components as well as the complete software stack (OS, PyTorch and TensorFlow frameworks) remain unchanged.

Figures 12 through 15 summarize the results obtained for the new processor model. Each graph includes he previous (Silver) results to facilitate the comparison process. We begin our analysis with Fig. 12, which shows the results for the first level of the Top-Down hierarchy. Two relevant aspects of the Silver vs. Gold results should be highlighted. First, the contribution of both Frontend and Speculation to performance loss remains marginal and extremely similar between the two families. This result is consistent with the core description, as these pipeline elements of the microarchitecture are the same in both processor families. Second, applications running on the Gold processor achieve higher IPC, improving from 1.12 to 1.77 instructions per cycle on average. This performance improvement seems to come directly from a proportional reduction in the backend performance penalty. The second FMA Unit seems to have a direct performance effect that should be analyzed in detail, to determine what fraction of the improvement is related to this enhancement.

Fig. 13 provides some insight into the source of the better backend performance by showing the results for the second level of the Top-Down hierarchy related to the Backend, divided into Core Bound and Memory Bound. As can be seen, the differences between Silver and

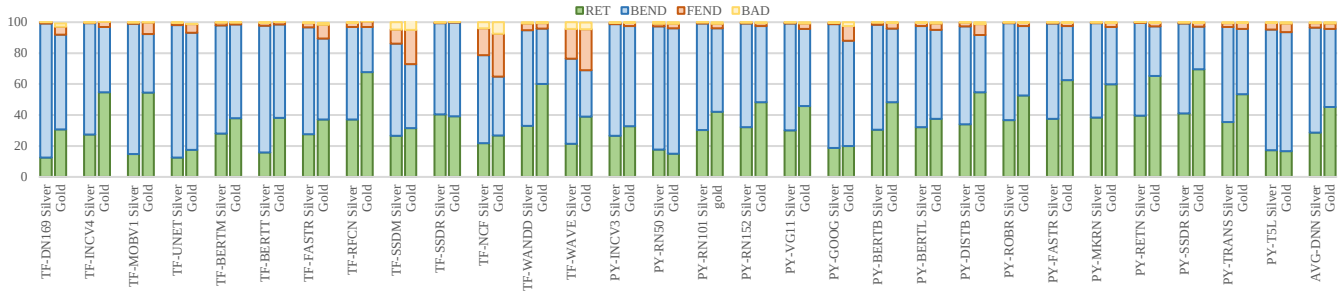

Fig. 12. Top-Down level 1 analysis of DNN models from Intel Model Zoo, Silver vs. Gold.

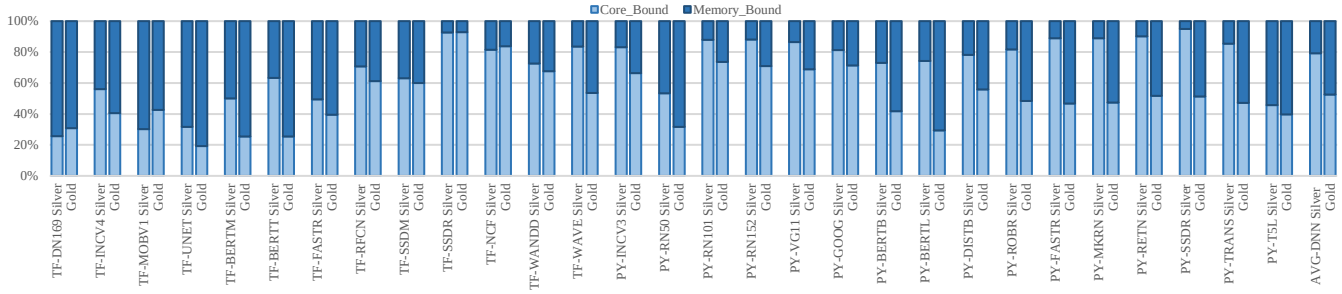

Fig. 13. Top-Down level 2 analysis of the Backend of DNN models from Intel Model Zoo, Silver vs. Gold.

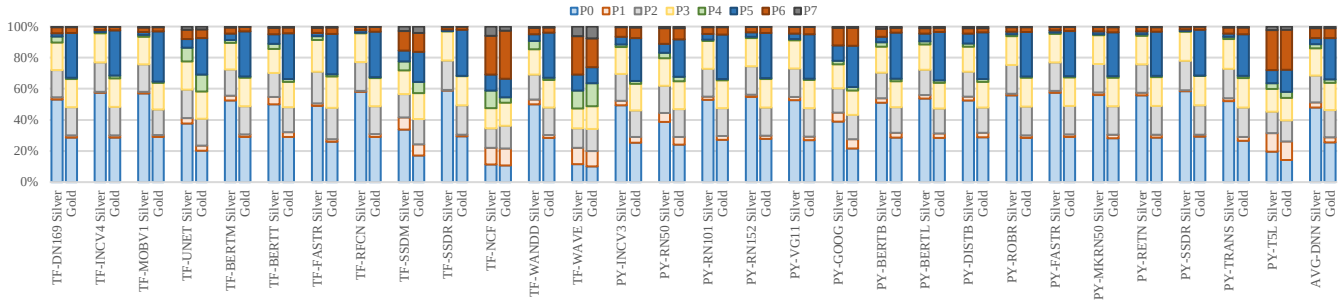

Fig. 14. Normalized cycles per core port utilization of DNN models from Intel Model Zoo, Silver vs. Gold.

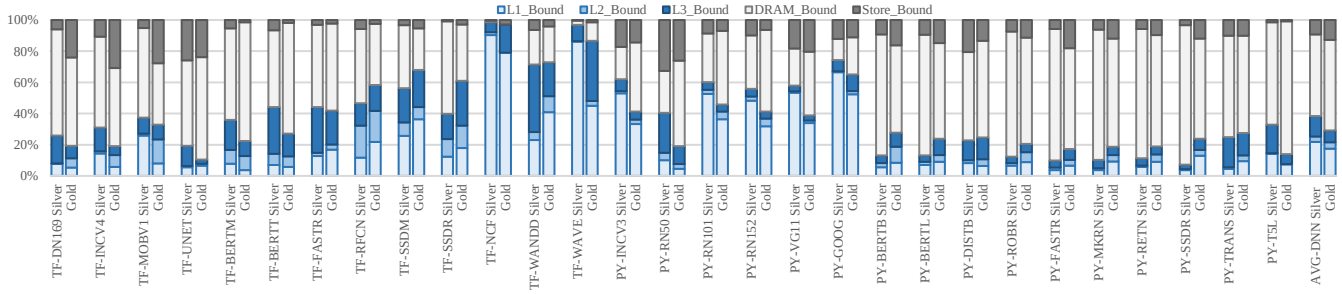

Fig. 15. Top-Down analysis level 3 of the Memory Bound portion of DNN models from Intel Model Zoo, Silver vs. Gold.

Gold are significant in this case, and the microarchitectural improvements of the Gold processor have a direct impact on how each part of the backend penalizes performance. In general, the Gold processor reduces the penalty caused by the Functional Units, balancing the contribution of both backend components. In most cases where the core was clearly the dominant performance bottleneck (TF-RFCN, TF-BERTT, PY-SSDR, etc.), this benefit is more relevant. On the other hand, most of the models where the Memory Hierarchy limited the performance with an FMA (TF-UNET, PY-RN50, PY-T5, etc.) no significant performance improvement can be achieved. It can be observed that, some applications are able to obtain great benefits from a second FMA, but become memory bound, provoking a swing from being core bound (TF-BERTT, PY-BERTL, PY-RN50, etc.).

We extend our experiments to the third Top-Down level for both the Core and the Memory Hierarchy. Starting with the Functional Unit utilization, Fig. 14 shows the utilization distribution of the seven execution ports present in both cores. As can be seen, there is a notable difference between the backend behavior of one and two FMAs. The fraction of instructions executed on port 0 (FMA unit in both architectures) decreases significantly, while the fraction of instructions at port 5 (where the second FMA is located in the Xeon Gold

processor) follows the opposite trend. This is a clear effect of the presence of a second Functional Unit for FMA operations, and DNN applications improve data parallelism by performing two simultaneous FMA operations. However, it should be noted that despite the doubling of FMA capacity and utilization, the IPC is far from doubling, which is mainly limited by the memory hierarchy penalty. Once a balance is reached between Core and Memory bound in the backend, continuing to add FMA Functional Units without improving the memory hierarchy will progressively reduce the performance benefits gained from including more FMA operations in parallel. For this reason, it is important to understand where the main bottlenecks are along the whole storage hierarchy. To this end, Fig. 15 shows the performance penalty contribution of each level for both Silver and Golden processors. Both architectures share an equivalent on-chip memory hierarchy, but certain differences can be observed between the two processors. The changes in the Functional Units are one of the factors behind this different behavior, as the presence of a second FMA Unit certainly changes the timing of Load/Store operations. Despite the differences, both processors have the main bottleneck at the memory controller, and the DRAM is not able to serve the requests arriving at the controller in time.

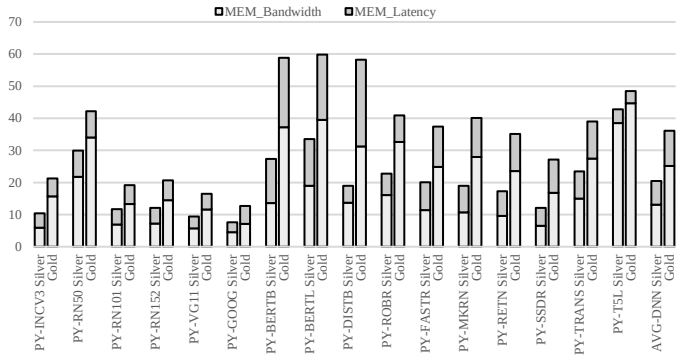

Fig. 16. Top-Down analysis level 4 of the DRAM portion of DNN models from Intel Model Zoo, Silver vs. Gold.

Extending the analysis to the fourth Top-Down level for DRAM behavior, Fig. 16 shows that the main component of the memory bottleneck is the bandwidth (as opposed to Latency) for both processor families. However, as can be observed, the addition of an extra FMA unit increases the pressure on the memory controller, resulting in even more bandwidth contention.

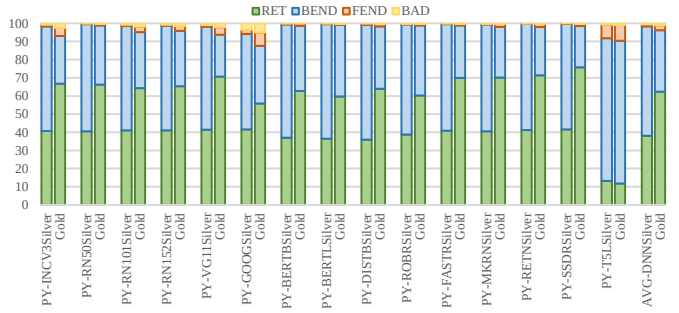

Fig. 17. Top-Down analysis level 1 of single-thread execution, Silver vs. Gold.

To better characterize the impact of the memory hierarchy, we repeated the evaluation on the same systems, but executing each model with a single thread running on a single processor, to emulate a situation with more than enough bandwidth for the CPU needs. Fig. 17 shows that when the bandwidth penalty is significantly reduced, the Xeon Gold processor is able to achieve better performance gains thanks to the second FMA unit, with some applications, such as PY-SSDR, almost doubling its IPC.

E. Similarity Analysis

Finally, we study the similarities between the different Intel Model Zoo models. We used a Principal Component Analysis (PCA) to reduce the number of dimensions, from all the components of the Top-Down hierarchy shown in this work, to two principal components (PCA1 and PCA2). Next, we applied the K-Means clustering algorithm to separate the models into K clusters based on similarity. We use Scikit-Learn's implementation of K-means, using the elbow and silhouette methods to select K, which is estimated to be 7 clusters. Fig. 18 represents all the Intel Model Zoo models according to their Principal Components, and each application is represented in the distribution colored according to the cluster it belongs to. The centroids of the different clusters are represented by red crosses.

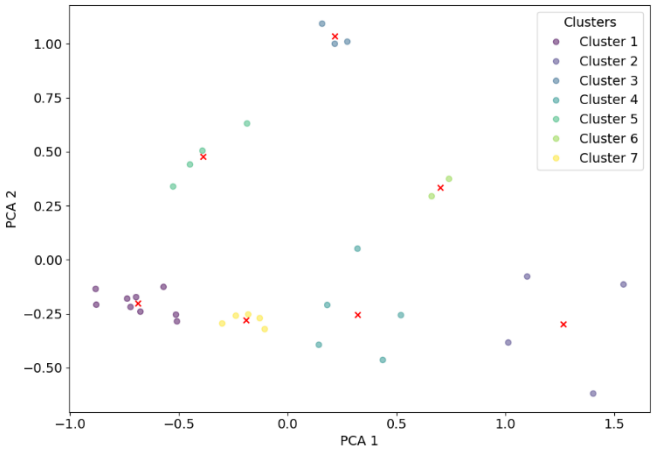

Fig. 18. Model distribution and clustering.

TABLE II. DNN MODELS DISTRIBUTION IN CLUSTERS

Cluster 1	*Cluster 2*	*Cluster 3*	*Cluster 4*	*Cluster 5*	*Cluster 6*	*Cluster 7*
TF-SSDR	TF-DN169	TF-NCF	TF-INCV4	PY-INCV3	PY-RN50	TF-RFCN
PY-RNNT	TF-MOBV1	TF-WAVE	TF-BERTM	PY-RN101	PY-T5L	TF-WANDD
PY-ROBR	TF-MASKR	PY-GOOG	TF-BERTT	PY-RN152		PY-BERTB
PY-FASTR	TF-UNET		TF-FASTR	PY-VG11		PY-BERTL
PY-MASK			TF-SSDM			PY-DISTB
PY-MKRN						
PY-RETN						
PY-SSDR						
PY-TRANS						

TABLE II. shows the distribution of the Intel Model Zoo models in the clusters obtained. As can be seen in Fig. 18, no single model is able to represent the entire spectrum. However, as seen in TABLE II, most of the models fit into Cluster 1, with fewer models in the remaining Clusters. It seems reasonable to select only a few models as representative, namely those closest to the centroids, and to select more models from the most crowded cluster. This could be important for benchmarking purposes, such as architectural simulation or future

benchmarking suites. The highlighted cells in TABLE II summarize our proposal for the selected representative models.

V. Related Work

The work presented in this paper extensively analyzes the microarchitectural performance of a representative group of DNN models on modern computer systems. Despite the existence of alternative ML benchmarking suites [57][58], Intel's Model Zoo is the most suitable for the characterization process. It provides a much larger number of models than alternatives such as MLPerf or CloudSuite and frameworks are optimized for the underlying hardware.

There are multiple research works evaluating DNN-application performance. The authors in [59] limit evaluation to two CNN models, AlexNet and ResNet50. Their target consists of comparing total and per-layer training time in CPU and GPU architectures. Similarly, the authors in [60][61] analyze thread and batch size effect on performance (time), also comparing GPU and CPU hardware. DNN characterization is a different process to performance evaluation, pursuing other goals. In this work we do not analyze raw execution time values, instead we try to understand the hardware-software interactions and their effects on performance.

Some recent works have carried out a similar characterization process. The authors in [62] compare MKL, MKL-DNN, and Eigen with GEMM (general matrix multiplication) microbenchmarks to highlight architectural bottlenecks. However, they limit the Top-Down analysis to single-threaded GEMM kernels with a variety of matrix sizes. Similarly, the authors in [63] make use of Top-Down methodology to perform a memory performance characterization of DNN workloads. In this case, their study is limited to a very specific type of DNNs, namely Graph Neural Networks.

VI. Conclusions

Using a representative group of DNN applications, belonging to Intel's Model Zoo, we have analyzed their behavior on a real system (64-bit Intel® Xeon® Silver 4216 and Gold 6242) using the hardware counters of the cores under analysis. It has been found that there is a large group of DNN applications whose behavior is quite similar. This is of interest in the initial design phases or for analysis with detailed simulators where the size of the significant set of applications should be reduced as much as possible.

In the absence of a second FMA Unit, the average IPC of this set is relatively low compared to the numerical applications from the SPEC 2017 benchmark. Including the second FMA improves IPC, reaching similar results to conventional benchmark suites. However, for DNN applications the number of floating-point operations per cycle is much higher due to the regularity and massive use of vectorized FP instructions.

Precisely, this regularity is what provokes the existence of bottlenecks. On the one hand, in Silver processors the number of functional units for FP operations is low, as befits a balanced general-purpose system and on the other hand, for Gold processors where this problem is partially corrected the memory hierarchy is not capable of moving data closer to the core at the rate that would be necessary.

Since this type of applications is becoming ubiquitous, this partly explains the tendency of processor manufacturers to introduce new functional units and/or accelerators for this type of applications. It should be noted that the bottlenecks are in both the core and the memory, so if only one is improved, the pressure falls on the other. This suggests that memory hierarchy optimizations might be necessary.

Finally, it should also be noted that the results show that there are some DNN applications whose behavior deviates significantly from the rest. This should be considered carefully as their execution will only be slightly accelerated through the improvements introduced to speed up the execution of others.

References

[1] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet Classification with Deep Convolutional Neural Networks," *Advances In Neural Information Processing Systems*, pp. 1–9, 2012, doi: http://dx.doi.org/10.1016/j.protcy.2014.09.007.

[2] A. Vaswani *et al.*, "Attention is all you need," in *Advances in Neural Information Processing Systems*, 2017, pp. 6000–6010.

[3] T. B. Brown *et al.*, "Language Models are Few-Shot Learners," May 2020, doi: 10.48550/arxiv.2005.14165.

[4] D. Silver *et al.*, "Mastering the game of Go without human knowledge," *Nature*, vol. 550, no. 7676, 2017, doi: 10.1038/nature24270.

[5] A. Y. Hannun *et al.*, "Cardiologist-level arrhythmia detection and classification in ambulatory electrocardiograms using a deep neural network," *Nature Medicine*, vol. 25, no. 1, 2019, doi: 10.1038/s41591-018-0268-3.

[6] S. Mittal, P. Rajput, and S. Subramoney, "A Survey of Deep Learning on CPUs: Opportunities and Co-Optimizations," *IEEE Transactions on Neural Networks and Learning Systems*, 2021, doi: 10.1109/TNNLS.2021.3071762.

[7] Z. Gong, H. Ji, C. W. Fletcher, C. J. Hughes, S. Baghsorkhi, and J. Torrellas, "Save: Sparsity-aware vector engine for accelerating DNN training and inference on CPUs," in *Proceedings of the Annual International Symposium on Microarchitecture, MICRO*, 2020. doi: 10.1109/MICRO50266.2020.00070.

[8] K. Hazelwood *et al.*, "Applied Machine Learning at Facebook: A Datacenter Infrastructure Perspective," in *Proceedings - International Symposium on High-Performance Computer Architecture*, 2018, pp. 620–629. doi: 10.1109/HPCA.2018.00059.

[9] M. Hamanaka *et al.*, "CGBVS-DNN: Prediction of Compound-protein Interactions Based on Deep Learning," *Molecular Informatics*, vol. 36, no. 1, 2017, doi: 10.1002/minf.201600045.

[10] A. Rodriguez, E. Segal, E. Meiri, E. Fomenko, Y. J. Kim, and H. Shen, "Lower Numerical Precision Deep Learning Inference and Training," *Intel White Paper*, 2018.

[11] G. Henry, P. T. P. Tang, and A. Heinecke, "Leveraging the bfloat16 Artificial Intelligence Datatype for Higher-Precision Computations," in *Proceedings - Symposium on Computer Arithmetic*, 2019, pp. 69–76. doi: 10.1109/ARITH.2019.00019.

[12] N. Nassif *et al.*, "Sapphire Rapids: The Next-Generation Intel Xeon Scalable Processor," in *Digest of Technical Papers - IEEE International Solid-State Circuits Conference*, 2022, pp. 44–46. doi: 10.1109/ISSCC42614.2022.9731107.

[13] M. Abadi *et al.*, "TensorFlow: A System for Large-Scale Machine Learning," in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI '16)*, 2016, pp. 265–284. doi: 10.1038/nn.3331.

[14] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, and ..., "Automatic differentiation in pytorch," in *31st Conference on Neural Information Processing Systems (NIPS)*, 2017.

[15] M. Rastegari, V. Ordonez, J. Redmon, and A. Farhadi, "XNOR-Net : ImageNet Classification Using Binary," *ECCV*, pp. 1–17, 2016, doi: 10.1007/978-3-319-46493-0_32.

[16] C. Szegedy, W. Liu, Y. Jia, and P. Sermanet, "Going deeper with convolutions," *arXiv preprint arXiv: 1409.4842*, pp. 1–9, 2014, doi: 10.1109/CVPR.2015.7298594.

[17] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna, "Rethinking the Inception Architecture for Computer Vision," in *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, 2016, pp. 2818–2826. doi: 10.1109/CVPR.2016.308.

[18] C. Szegedy, S. Ioffe, V. Vanhoucke, and A. A. Alemi, "Inception-v4, inception-ResNet and the impact of residual connections on learning," in *31st AAAI Conference on Artificial Intelligence, AAAI 2017*, 2017. doi: 10.1609/aaai.v31i1.11231.

[19] K. Simonyan and A. Zisserman, "Very Deep Convolutional Networks for Large-Scale Image Recognition," Sep. 2014, doi: 10.48550/arxiv.1409.1556.

[20] R. Patel and A. Chaware, "MobileNet architecture and its application to computer vision," in *Computer Vision and Recognition Systems Using Machine and Deep Learning Approaches: Fundamentals, technologies and applications*, 2021. doi: 10.1049/pbpc042e_ch11.

[21] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, 2016, pp. 770–778. doi: 10.1109/CVPR.2016.90.

[22] J. Deng, W. Dong, R. Socher, L.-J. Li, Kai Li, and Li Fei-Fei, "ImageNet: A large-scale hierarchical image database," in *2009 IEEE Conference on Computer Vision and Pattern Recognition*, 2009, pp. 248–255. doi: 10.1109/cvpr.2009.5206848.

[23] R. Girshick, J. Donahue, T. Darrell, and J. Malik, "Rich feature hierarchies for accurate object detection and semantic segmentation," in *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, 2014. doi: 10.1109/CVPR.2014.81.

[24] S. Ren, K. He, R. Girshick, and J. Sun, "Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 39, no. 6, 2017, doi: 10.1109/TPAMI.2016.2577031.

[25] K. He, G. Gkioxari, P. Dollár, and R. Girshick, "Mask R-CNN," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 42, no. 2, 2020, doi: 10.1109/TPAMI.2018.2844175.

[26] J. Dai, Y. Li, K. He, and J. Sun, "R-FCN: Object detection via region-based fully convolutional networks," in *Advances in Neural Information Processing Systems*, 2016.

[27] W. Liu *et al.*, "SSD: Single shot multibox detector," in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 2016, vol. 9905 LNCS. doi: 10.1007/978-3-319-46448-0_2.

[28] T. Y. Lin, P. Goyal, R. Girshick, K. He, and P. Dollar, "Focal Loss for Dense Object Detection," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 42, no. 2, 2020, doi: 10.1109/TPAMI.2018.2858826.

[29] T. Y. Lin *et al.*, "Microsoft COCO: Common objects in context," in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 2014, vol. 8693 LNCS, no. PART 5. doi: 10.1007/978-3-319-10602-1_48.

[30] J. Devlin, M. W. Chang, K. Lee, and K. Toutanova, "BERT: Pre-training of deep bidirectional transformers for language understanding," in *NAACL HLT 2019 - 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies - Proceedings of the Conference*, 2019, vol. 1.

[31] Y. Liu *et al.*, "RoBERTa: A Robustly Optimized BERT Pretraining Approach," Jul. 2019, doi: 10.48550/arxiv.1907.11692.

[32] V. Sanh, L. Debut, J. Chaumond, and T. Wolf, "DistilBERT, a distilled version of BERT: smaller, faster, cheaper and lighter," Oct. 2019, doi: 10.48550/arxiv.1910.01108.

[33] J. Li *et al.*, "Developing RNN-T models surpassing high-performance hybrid models with customization capability," in *Proceedings of the Annual Conference of the International Speech Communication Association, INTERSPEECH*, 2020. doi: 10.21437/Interspeech.2020-3016.

[34] P. Rajpurkar, J. Zhang, K. Lopyrev, and P. Liang, "SQuad: 100,000+ questions for machine comprehension of text," in *EMNLP 2016 - Conference on Empirical Methods in Natural Language Processing, Proceedings*, 2016. doi: 10.18653/v1/d16-1264.

[35] W. B. Dolan and C. Brockett, "Automatically Constructing a Corpus of Sentential Paraphrases," *Proceedings of the Third International Workshop on Paraphrasing (IWP2005)*, 2005.

[36] X. He, L. Liao, H. Zhang, L. Nie, X. Hu, and T.-S. Chua, "Neural Collaborative Filtering," Aug. 2017, doi: 10.48550/arxiv.1708.05031.

[37] H. T. Cheng *et al.*, "Wide & deep learning for recommender systems," in *ACM International Conference Proceeding Series*, 2016, pp. 7–10. doi: 10.1145/2988450.2988454.

[38] M. Naumov *et al.*, "Deep Learning Recommendation Model for Personalization and Recommendation Systems," May 2019, doi: 10.48550/arxiv.1906.00091.

[39] F. M. Harper and J. A. Konstan, "The movielens datasets: History and context," *ACM Transactions on Interactive Intelligent Systems*, vol. 5, no. 4, 2015, doi: 10.1145/2827872.

[40] O. Chapelle, "Modeling delayed feedback in display advertising," in *Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2014. doi: 10.1145/2623330.2623634.

[41] A. Yasin, "A Top-Down method for performance analysis and counters architecture," in *ISPASS 2014 - IEEE International Symposium on Performance Analysis of Systems and Software*, 2014, pp. 35–44. doi: 10.1109/ISPASS.2014.6844459.

[42] W. Jung *et al.*, "Accelerating Fully Homomorphic Encryption through Microarchitecture-Aware Analysis and Optimization," in *Proceedings - 2021 IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS 2021*, 2021. doi: 10.1109/ISPASS51385.2021.00045.

[43] J. Kim, P. Shin, M. Kim, and S. Hong, "Memory-Aware Fair-Share Scheduling for Improved Performance Isolation in the Linux Kernel," *IEEE Access*, vol. 8, 2020, doi: 10.1109/ACCESS.2020.2996596.

[44] A. C. Shulyak and L. K. John, "Identifying performance bottlenecks in Hive: Use of processor counters," in *Proceedings - 2016 IEEE International Conference on Big Data, Big Data 2016*, 2016. doi: 10.1109/BigData.2016.7840838.

[45] R. Hebbar and A. Milenkovic, "A Preliminary Scalability Analysis of SPEC CPU2017 Benchmarks," in *Conference Proceedings - IEEE SOUTHEASTCON*, 2021, pp. 1–8. doi: 10.1109/SoutheastCon45413.2021.9401917.

[46] Y. Chen, J. Zhu, T. A. Khan, and B. Kasikci, "CPU Microarchitectural Performance Characterization of Cloud Video Transcoding," in *Proceedings - 2020 IEEE International Symposium on Workload Characterization, IISWC 2020*, 2020. doi: 10.1109/IISWC50251.2020.00016.

[47] R. Panda, S. Song, J. Dean, and L. K. John, "Wait of a Decade: Did SPEC CPU 2017 Broaden the Performance Horizon?," in *Proceedings - International Symposium on High-Performance Computer Architecture*, 2018, pp. 271–282. doi: 10.1109/HPCA.2018.00032.

[48] A. J. Awan, M. Brorsson, V. Vlassov, and E. Ayguade, "Micro-architectural characterization of Apache Spark on batch and stream processing workloads," in *Proceedings - 2016 IEEE International Conferences on Big Data and Cloud Computing, BDCloud 2016, Social Computing and Networking, SocialCom 2016 and Sustainable Computing and Communications, SustainCom 2016*, 2016. doi: 10.1109/BDCloud-SocialCom-SustainCom.2016.20.

[49] P. Prieto, P. Abad, J. A. Gregorio, and V. Puente, "Fast, Accurate Processor Evaluation through Heterogeneous, Sample-based Benchmarking," *IEEE Transactions on Parallel and Distributed Systems*, 2021, doi: 10.1109/TPDS.2021.3080702.

[50] Y. Huerta and D. Lilja, "Analysis of a ThunderX2 System Using Top-Down and Purchasing Power Parity Methods," in *ACM International Conference Proceeding Series*, 2021. doi: 10.1145/3437359.3467027.

[51] E. Rotem *et al.*, "Intel Alder Lake CPU Architectures," *IEEE Micro*, vol. 42, no. 3, pp. 13–19, 2022.

[52] "Model Zoo for Intel® Architecture." https://github.com/IntelAI/models

[53] "Intel® Extension for PyTorch." https://github.com/intel/intel-extension-for-pytorch

[54] “Intel® Extension for TensorFlow.” https://github.com/intel/intel-extension-for-tensorflow

[55] “perf: linux profiling with performance counters.” https://perf.wiki.kernel.org/

[56] “Intel PMU Profiling Tools Source Code and Documentation.” https://github.com/andikleen/pmu-tools

[57] V. J. Reddi *et al.*, “MLPerf Inference Benchmark,” in *Proceedings - International Symposium on Computer Architecture*, 2020. doi: 10.1109/ISCA45697.2020.00045.

[58] M. Ferdman and E. Al., “Clearing the Clouds: A Study of Emerging Scale-out Workloads on Modern Hardware,” in *ASPLOS’12*, 2012, vol. 40, pp. 37–48. doi: 10.1145/2189750.2150982.

[59] A. A. Awan, H. Subramoni, and D. K. Panda, “An In-depth Performance Characterization of CPU- and GPU-based DNN Training on Modern Architectures,” in *Proceedings of MLHPC 2017: Machine Learning in HPC Environments - Held in conjunction with SC 2017: The International Conference for High Performance Computing, Networking, Storage and Analysis*, 2017. doi: 10.1145/3146347.3146356.

[60] A. Jain, A. A. Awan, Q. Anthony, H. Subramoni, and D. K. D. K. Panda, “Performance Characterization of DNN Training using TensorFlow and PyTorch on Modern Clusters,” in *Proceedings - IEEE International Conference on Cluster Computing, ICCC*, 2019. doi: 10.1109/CLUSTER.2019.8891042.

[61] K. Jabłońska and P. Czarnul, “Benchmarking Deep Neural Network Training Using Multi- and Many-Core Processors,” in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 2020, vol. 12133 LNCS. doi: 10.1007/978-3-030-47679-3_20.

[62] Y. E. Wang, C. J. Wu, X. Wang, K. Hazelwood, and D. Brooks, “Exploiting Parallelism Opportunities with Deep Learning Frameworks,” *ACM Transactions on Architecture and Code Optimization*, vol. 18, no. 1, 2021, doi: 10.1145/3431388.

[63] Z. Gong, H. Ji, Y. Yao, C. W. Fletcher, C. J. Hughes, and J. Torrellas, “Graphite: optimizing graph neural networks on CPUs through cooperative software-hardware techniques.,” in *49th Annual International Symposium on Computer Architecture*, 2022, pp. 916–931.