Architecture-Centric Bottleneck Analysis for Deep Neural Network Applications

Jihyun Ryoo[1], Mengran Fan[1], Xulong Tang[1], Huaipan Jiang[1],
Meena Arunachalam[2], Sharada Naveen[2], Mahmut T. Kandemir[1]
[1]Pennsylvania State University, [2]Intel
[1]{jihyun, mxf97, xzt102, hzj5142, kandemir}@cse.psu.edu, [2]{meena.arunachalam, sharada.naveen}@intel.com

***Abstract*—The ever-growing complexity and popularity of machine learning and deep learning applications have motivated an urgent need of effective and efficient support for these applications on contemporary computing systems. In this paper, we thoroughly analyze the various DNN algorithms on three widely used architectures (CPU, GPU, and Xeon Phi). The DNN algorithms we choose for evaluation include i) Unet – for biomedical image segmentation, based on Convolutional Neural Network (CNN), ii) NMT – for neural machine translation based on Recurrent Neural Network (RNN), iii) ResNet-50, and iv) DenseNet – both for image processing based on CNNs. The ultimate goal of this paper is to answer four fundamental questions: i) whether the different DNN networks exhibit similar behavior on a given execution platform? ii) whether, across different platforms, a given DNN network exhibits different behaviors? iii) for the same execution platform and the same DNN network, whether different execution phases have different behaviors? and iv) are the current major general-purpose platforms tuned sufficiently well for different DNN algorithms? Motivated by these questions, we conduct an in-depth investigation of running DNN applications on modern systems. Specifically, we first identify the most time-consuming functions (hotspot functions) across different networks and platforms. Next, we characterize performance bottlenecks and discuss them in detail. Finally, we port selected hotspot functions to a cycle-accurate simulator, and use the results to direct architectural optimizations to better support DNN applications.**

***Index Terms*—DNN, CPU, GPU, Xeon Phi, Characterization**

I. Introduction

With the capability to provide unprecedented accuracy, Deep Neural Networks (DNNs) have been widely adopted by various application domains such as image classification [17, 25], natural language processing [14], real-time text translation [31], self-driving/autonomous vehicles [9], and cancer diagnosis [23]. While the employed high-level training and inference procedures are similar across these different deep learning (DL) based applications, the low-level algorithms, and neural network topological structures are tailored for different application domains. For instance, Convolution Neural Networks (CNNs) are mostly used for image classification [20], whereas Recurrent Neural Networks (RNNs) are usually employed for natural language processing [16]. Such differences in network structures bring multiple challenges to the underlying system, including i) different demands for computation resources, ii) different data access patterns, and iii) different degrees of data reuse, which collectively restrict the execution efficiency of DNN applications.

To cater to these challenges, there have been substantial efforts from both the software side [12, 19, 22] and the hardware side [10, 24, 27–30], focusing on improving the performance and accuracy of DNN applications. While prior efforts focus on improving individual DNN applications, they fail to i) cross-compare different DNN applications on the same platform, ii) cross-compare same DNN application on different platforms, iii) conduct detailed analysis to reveal the reasons behind the observed performance differences, and iv) explore general system optimizations for DNN applications.

Motivated by the aforementioned reasons, in this paper, we conduct a thorough investigation of a set of DNN applications from the architectural perspective. Specifically, we seek to answer the following four questions: **Question 1:** *whether the different DNN networks exhibit similar behavior on a given execution platform?* **Question 2:** *whether, across different platforms, a given DNN network exhibits different behaviors?* **Question 3:** *for the same execution platform and same DNN network, whether different execution phases have different behaviors?* and finally, **Question 4** *are the current major general-purpose platforms tuned sufficiently well for different DNN algorithms?* To this end, we first use four representative DL applications (Unet [23], NMT [7], ResNet [17], and DenseNet [18]) and characterize them on three representative systems (Intel Xeon, NVIDIA GPU, and Intel Xeon Phi). Second, we pick up the most time-consuming kernels from our applications that exhibit different types of bottlenecks and use a simulator platform, built upon gem5 [8], to further study their scalability. This paper makes the following **contributions**:

- It identifies the top 5 most execution time-consuming hotspot functions in DNN applications using the TensorFlow [6] framework running on three different compute platforms.
- It conducts an in-depth analysis of the hotspot functions from an architectural viewpoint and determines where the primary bottlenecks exist in major hardware components (core, L1 cache, L2 cache, L3 cache, DRAM).
- It answers the aforementioned four important questions both *qualitatively* and *quantitatively*. Specifically, our experimental results indicate that the DNN application behaviors vary from application to application, as well as from system to system (**Question 1 and Question 2**). In addition, a given application exhibits varied runtime behaviors in different execution phases (**Question 3**). Finally, our results reveal that current systems can be optimized to better support DNN

applications (**Question 4**).

- Using a cycle-accurate simulation infrastructure, it further performs a limit study to show how current systems can be optimized for DNN applications. The limit study indicates that the performance of DNN applications can be significantly improved with moderate architectural enhancements. Note that the insights obtained from these simulation results can be useful in developing future architecture enhancements targeting DNN applications.

The remainder of this paper is structured as follows. Section II introduces our targeted DNN applications and Section III provides an in-depth hotspot analysis. We discuss our limit study in Section IV, and summarize our key observations in Section V. Finally, Section VI discusses the related work, and Section VII concludes the paper.

II. Background

A. Deep Learning Neural Networks

Unet: Unet [23] is a widely-used deep neural network for biomedical image processing. It consists of several convolution layers interleaved with several pooling layers. The input biomedical images first pass through the convolution layers and then go to the deconvolution layer which is transposed convolution layer. In the deconvolution layer, the images are concatenated with the images from the corresponding max-pooling layers. As a result, Unet creates an "U-shaped".

Neural Machine Translation (NMT): NMT [7] is a deep learning network designed for language-to-language translation. It consists of several steps. First, it encodes a sentence into a vector in which each element represents a word in the sentence. Second, it decodes the number vector into another sentence in a different language. While NMT has many different implementations, those based on Recurrent Neural Network (RNN) and Long Short-Term Memory (LSTM) are the most common ones. Generally, LSTM is preferred compared to RNN, because the translation needs to remember the "status" vector and find appropriate translation for the context.

ResNet-50: ResNet [17] is based on "residual learning". Specifically, as the network gets deeper, the accuracy becomes worse due to the vanishing gradient problem. ResNet solves this problem by introducing training on the residual through function $H(x) - x$ where $H(x)$ is the output of the original CNN, and x is its input. The residual learning detects the small fluctuations of input which improves accuracy.

DenseNet: DenseNet [18] has a different layer connection compared to ResNet. It employs several "Dense Blocks" that have "Dense Connections" between layers. "Dense Connection" connects the lower layers to all the upper layers in the same "Dense Block". Thus, DenseNet has $L(L+1)/2$ connections, which is more than the general CNN algorithms (L is the number of layers), and this can make a very narrow structure, adding only a small set of feature maps.

B. Deep Learning Framework

We use TensorFlow [6] as the framework for the aforementioned four deep neural networks tested in this study. TensorFlow formulates a given neural network into a data flow graph (DFG) where nodes represent computations and edges capture the data flow across the computations. Data are transferred between nodes as a multi-dimensional array which is called "tensor". TensorFlow supports various commonly-used computations in DL algorithms, like multiplication and summation (convolution), pooling, Relu (regulation), and Sigmoid. Using its data expression and set of operations, TensorFlow can easily support various DL algorithms. It also supports multiple computational architectures, multi-core CPUs, general-purpose GPUs (GPGPUs), Intel Xeon Phi, Tensor Processing Units (TPUs), and various ASICs.

TABLE I: Target architectures

	CPU	**GPU**	**KNL**
Version	Skylake @2.4 GHz	TESLA P100	Knights Landing @1.40 GHz
compute configuration	40 cores, 2 SMT	60 SMs, 3480 cores, 64 warps/SM Dual warp scheduler	68 cores, 4 SMT
Cache configuration	32 KB L1 Private 1 MB L2 Private	64 KB L1 Private 4 MB L2 Uniform	32 KB L1 Private 1 MB L2 shared between 2 cores
Socket number	2	N/A	1
Vector Processing Unites	80	N/A	136

TABLE II: Software Tools

Tools	MKL	MPI	Python	TensorFlow	Nvprof	Vtune
version	2018.0.0	2017 update2	2.7	1.6 for SKL and KNL 1.4 for GPU	Nvidia 8.0	2018u3

III. Characterization

A. Experimental setup

We conduct an in-depth characterization of the four DNN networks (Unet, NMT, ResNet-50, and DenseNet) on three different compute platforms (Intel Xeon CPU, Nvidia GPU, and Intel Xeon Phi Knight's Landing co-processor). The major configuration parameters of the platforms used are summarized in Table I. We employ multiple tools to conduct our application characterization. Table II provides the toolsets we used on different systems. Specifically, we use Intel® Vtune™ [1] on Xeon CPU and Xeon KNL, and Nvprof on Pascal GPU. The input dataset of Unet is PhC-C2DH-U373 [2] – a biomedical image of Glioblastoma-astrocytoma U373 cells on a polyacrylamide substrate. For NMT, we use the WMT'14 English - German data [3], and for ResNet-50 an DenseNet we use the cifar-10 dataset [4].

B. Detailed Architecture Explanation with Metrics

1) Xeon and Xeon Phi: In both Xeon CPU and Xeon Phi, at a high level, we consider four categories of different states in the micro-operation execution pipeline: front-end bottleneck, back-end bottleneck, bad speculation, and retiring. These four categories are formed by answering the following three questions: i) whether the pipeline is stalled or allocated, ii) if the pipeline is empty, then which one – front-end or back-end – cause the stalls, and finally, iii) if the pipeline is allocated, whether the μOp is retired or flushed (bad speculation).

Figure 1, 2 depicts the Xeon and Xeon Phi (KNL) architectures. In both Xeon and Xeon Phi, the execution pipeline is divided into two parts: front-end and back-end. Front-end is responsible for fetching the instructions, decoding the instructions into micro-operations (μOps), and allocating pipeline execution slots for μOps. After the front-end execution, back-end executes in-pipeline μOps when all the operands of an

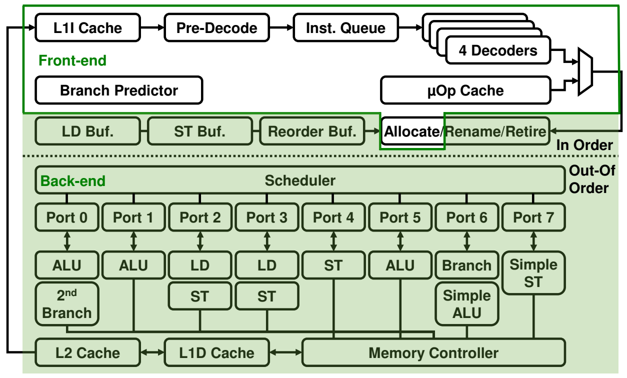


Fig. 1: Front-end and back-end of Xeon architecture [1].

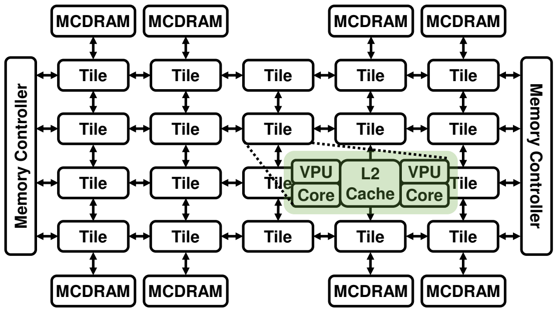


Fig. 2: Intel Xeon Phi (KNL) architecture [26].

allocated μOp are ready. When μOps finish execution, the head-of-ROB (reorder buffer) μOps are *retired*. The *retirement* of instruction is mostly about updating the architecture states, such as updating registers and writing back to cache or memory. However, some of the μOps might not be able to retire due to miss prediction in the branch predictor. These μOps are flushed in the pipeline because of bad speculation.

The pipeline width of Xeon CPU is four μOps per cycle, indicating that there are four available pipeline slots every cycle to allocate four μOps in that cycle. The allocation process on the front-end assigns μOps to pipeline slots. During this process, if there is no instruction to fill the pipeline, a pipeline stall occurs. It is to be noted that there can be multiple different reasons behind a pipeline stall. Front-end bound can be caused by two reasons: front-end latency and bandwidth. For instance, handling instruction cache misses (Icache misses), instruction TLB misses (ITLB overhead), branch resteers (Branch Resteers), switches between Decoded Stream Buffer and MITE (DSB switches), length changing prefixes (Length changing prefixes), and switches of μop delivery to the Microcode Sequencer (MS switches) all cause front-end latency stalls. A more detailed discussion of front-end latency breakdown can be found elsewhere [1].

Pipeline stalls can also occur due to back-end execution; in this case, the executing application is called back-end bound. Being back-end bound is typically a result of the lack of architecture-level resources. Different components can cause back-end bottlenecks – in this study, the components of interest are core, L1 cache, L2 cache, L3 cache, and DRAM. In most cases, the term ”back-end bound” is used to refer to a core if

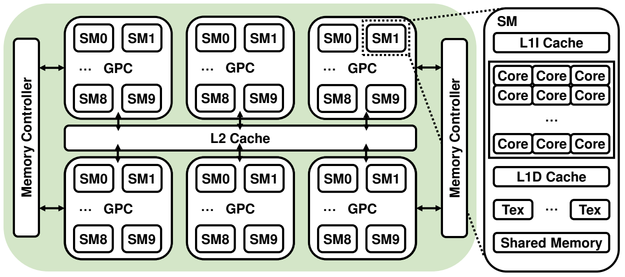


Fig. 3: Nvidia Pascal P100 architecture [21].

the thread it executes is stalled by the shortage of computation resources or due to the instruction dependencies. L1 cache bottleneck shows the stalls caused by the L1 data cache (note however that the stalls caused by L1 cache misses are counted as L2 cache bound (since they access the L2 cache), i.e., they not counted for L1 cache bound. Loads blocked by store forwarding, first-level TLB miss (DTLB overhead), lock handling issues (Lock latency), split loads caused by unaligned data (Split loads), how often the memory load is aliased with a 4K address offset by the preceding store (4K aliasing), and how often the fulled L1D Fill buffer causes additional L1D misses (FB full) are among the possible reasons for the L1 bottleneck. Similarly, L2 and L3 cache boundness captures the percentage of stalls on L2 and L3 caches. For example, a data which is written by a thread is read by another thread on a different core (contested accesses), cache coherency issue caused by data sharing for multiple threads (Data sharing), high L3 latency (L3 latency), fulled Super Queue (SQ full) can all cause the stalls on L3. Also, for the main memory, if the stalls were caused by high DRAM latency, the corresponding accesses are categorized as being DRAM latency bound, and if it is caused by the lack of DRAM bandwidth, then it counts as DRAM bandwidth bound. In this paper, we frequently use two terms: ”core bound” and ”memory bound”. The former means that a majority of the stalls are from the core in the back-end, whereas the latter specifies that the stalls come from back-end memory system (L1, L2, L3, DRAM). Note that, if the stall is caused by both front-end and back-end, it is still considered as back-end bound. The back-end issue should be fixed before fixing the front-end issues because solving back-end issues has more impact on overall performance and is more important.

2) GPU: Figure 3 illustrates the micro-architecture of Nvidia Tesla P100 [21]. P100 is featured with a total of 3584 single precision CUDA Cores, 240 texture units and 4096 KB L2 cache. We characterize the bottleneck while running DNN applications on P100 using Nvidia profiling tool. Stall_constant_memory_dependency represents the stalls due to immediate constant cache misses. Stall_memory_dependency and stall_texture represent the stalls caused by limited resources while executing the memory/texture operations. Stall_memory_throttle, together with all the previous memory relevant stall reasons, is used to profile the memory architecture resided in GPU. Compared to the Xeon architecture, P100 has many more compute resources, which makes P100 a suitable platform for data parallel applications. The current P100 consists of 16 GB high-bandwidth global memory. However, we observed from our profiling results that memory access could still be the bottleneck when working with a large number of computing units. Note that the threads in a GPU execute in a lock-step fashion at a "warp" granularity. Typically, a warp consists of 32 threads in modern Nvidia GPUs, including our targeted P100. Warps are further organized into thread blocks and synchronization primitives are provided to synchronize all the warps within a given thread block. To characterize the synchronization overheads, we use the metric stall_sync from Nvprof. To capture the overhead by computation units, we test the stall_pipe_busy metric, which gives us the percentage of the stalls due to the stalled compute operations because of a busy compute pipeline. Also, stall_exec_dependency shows the stalls caused by the application's inherent dependencies, and stall_inst_fetch metric captures the percentage of stalls because of the next assembly instruction which has not been fetched. Stall_not_select is used to reveal the occupancy of the GPU, and stall_other captures all other miscellaneous stalls.

C. *Characterizations on the Xeon architecture*

Table III presents the top 5 functions (listed under the second column) that dominate the execution time (given under the last column) in each application code we tested, and also indicates which component(s) is(are) bottleneck. One can make several interesting observations from these results. First, in Unet, functions `Off_l` and `Pooling_fwd` and, in DenseNet, function `Gebp` clearly dominate the overall execution time, whereas we witness a more balanced execution time distribution among the top 5 functions in NMT and Resnet-50. Second, the most frequently observed bottlenecks in these functions are core and DRAM bandwidth. This is because, some of the functions are compute intensive while others are memory intensive. For example, functions such as `Off_l` perform a lot of computation with small amounts of data, whereas `Binary_product` and `Binary_sum` exhibit an opposite trend. Third, when considering all four applications we have, the top four most time-consuming functions (`Off_l` in Unet, `Gebp` in DenseNet, `GradGenerator` in NMT, and `__stop_notes` in ResNet-50) are clearly core-bound. Therefore, if the system can offer more compute resources to these functions, their execution time can reduce significantly.

1) Unet: We quantify the bounds of Unet and show the results in Figure 4. `Off_l` from the MKLDNN library takes 37.63% of total execution time, and `Poling_fwd`, also from the MKLDNN library, takes 17.27% of the total execution time. Obviously, these two functions are the most interesting functions. Also, as it can be observed from Figure 4, different from the other functions, `Off_l` is *both* front-end bound and back-end bound. During the execution of `Off_l`, about 35.3% of the execution cycles are stall cycles due to front-end side, and 22.4% of the cycles are bounded by back-end side. This means that, the front-end is bounded and could not have enough instructions to feed into the back-end. As a result, there is a large number of stalls in the pipeline because of the front-

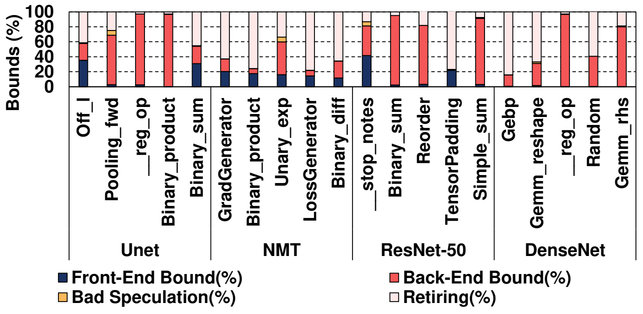


Fig. 4: Percentage of the bottlenecks (front-end, back-end, bad speculation) and retiring on Xeon.

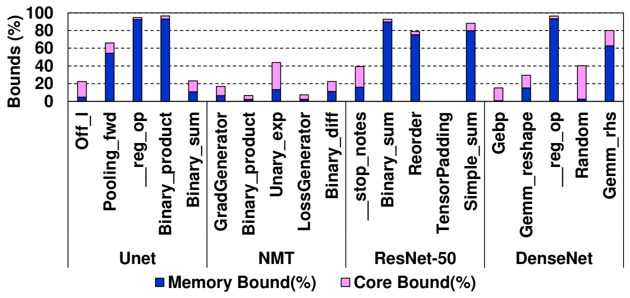


Fig. 5: Percentage of core and memory bounds in back-end (Xeon).

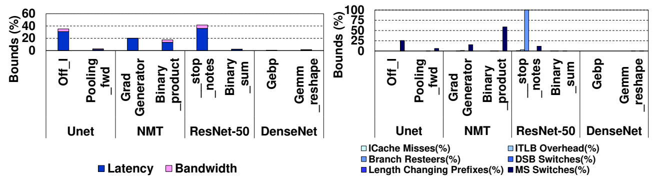
end. Specifically, `Off_l` is bounded by front-end latency (as shown in figure 6) caused by "MS Switches". As explained before, MS Switches metric represents what percentage of CPU cycles are stalled by the switches of μOp delivery to the "Microcode Sequencer (MS)". MS is used to deliver long μOp flow for CISC instructions.

However, as explained earlier, if a function is both front-end bound and back-end bound, the back-end bottlenecks play a more significant role in shaping the overall application performance. It can be seen from Figure 5 that `Off_l` is mostly core bound. `Off_l` is an offset computation function; it returns the corresponding physical offset given a logical offset. Thus, it requires several addition operations, one remainder operation (%), and one division. The input data of `Off_l` use the offset between the two data elements. Though the sizes of the offsets are smaller compared to the original data elements, the small data size does not necessarily reduce the computation cost of `Off_l` function, due to the long latency of division operation. We can confirm this in Figure 7 where a large portion of bounds are from divider. Also, the other half of bounds come from "port utilization", which represents the fraction of cycles stalled by the core non-divider-related issues. The vector capacity usage metric shows the number of cycles the vectorization units are busy on floating-point computations. `Off_l` accesses Port 0, Port 1 for ALU and branches, and Port 2 for memory accesses (figure 1).

Next function is `Pooling_fwd`, which takes about 17.27% of the total execution time. This function is a forward pooling function and is mostly bounded by the back-end, especially memory. Obviously, pooling generates plenty of memory accesses and it is natural to be memory bound. We can confirm

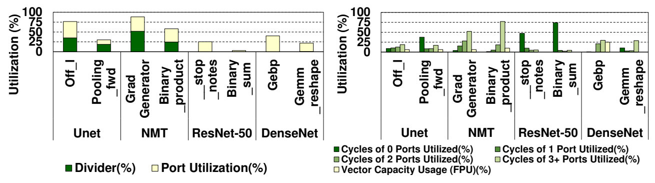
TABLE III: Top 5 hotspot functions of Unet, NMT, ResNet-50, DenseNet run on **Xeon** and their main bounds (FL: Front-end Latency, FB: Front-end Bandwidth, C: Core, L1: L1 cache, L2: L2 cache, L3: L3 cache, DL: DRAM Latency, DB: DRAM Bandwidth, S: Store)

			Front-end		Back-end							
Algorithm	Function (Addr)	Explanation	FL	FB	C	L1	L2	L3	DL	DB	S	%
Unet	mkldnn::off_l (Off_l)	Offset conversion	O		O							37.63%
	mkldnn::cpu::ref_pooling_fwd_t (Pooling_fwd)	Forward pooling						O	O			17.27%
	__reg_op (__reg_op)	Bitmap allocation, release									O	5.99%
	TensorCwiseBinaryOp::scalar_product_op (Binary_product)	Binary product								O		3.42%
	TensorCwiseBinaryOp::scalar_sum_op (Binary_sum)	Binary sum	O		O		O					3.00%
NMT	Eigen::TensorGeneratorOp::SparseXentGradGenerator (GradGenerator)	Calculate backward gradient			O							16.06%
	CwiseBinaryOp::scalar_product_op (Binary_product)	Vector product	O									13.23%
	CwiseUnaryOp::scalar_exp_op (Unary_exp)	Vector Unary exponential			O							11.04%
	TensorReductionOp::SparseXentLossGenerator (LossGenerator)	Calculate loss	O									10.83%
	CwiseBinaryOp::scalar_difference_op (Binary_diff)	Binary difference			O	O	O					10.66%
ResNet-50	__stop_notes (__stop_notes)	Stores the content information	O		O							14.11%
	CwiseBinaryOp::scalar_sum_op (Binary_sum)	Binary sum								O		7.21%
	mkldnn::simple_reorder_impl (Reorder)	Reorder				O			O	O		7.17%
	Eigen::TensorPaddingOp (TensorPadding)	TensorPadding	O									5.77%
	mkldnn::simple_sum_t (Simple_sum)	Simple sum								O		4.35%
DenseNet	gebp_kernel (Gebp)	Kernel for GEMM			O							59.01%
	gemm_pack_rhs; Copy matrix mul data with reshape (Gemm_reshape)	Matrix multiplication with reshape			O			O				7.88%
	__reg_op (__reg_op)	Bitmap allocation, release									O	5.26%
	FillPhiloxRandom (Random)	Random			O							3.19%
	gemm_pack_rhs; Copy matrix mul data (Gemm_rhs)	Matrix multiplication								O		2.31%



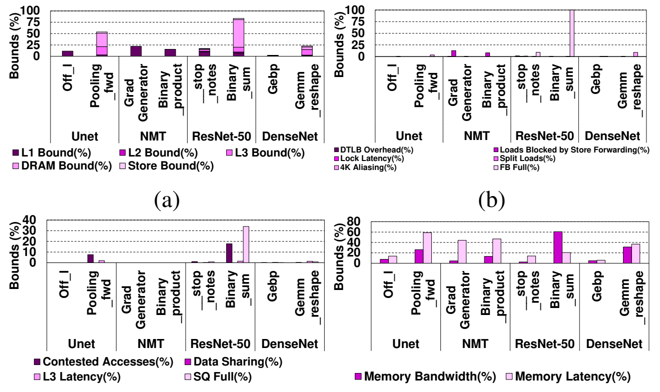
(a) (b)

Fig. 6: Breakdown of the Xeon (a) Front-end (b) Front-end latency.



(a) (b)

Fig. 7: Breakdown of the Xeon (a) Back-end core (b) Port utilization.



(a) (b) (c) (d)

Fig. 8: Breakdown of Xeon back-end in (a) Total memory (b) L1 (c) L3 (d) DRAM.

from Figure 8 that `Pooling_fwd` is bounded by the L3 cache and DRAM. In particular, it can be seen from Figure 8c that `Pooling_fwd` experiences some contested accesses on L3 cache, and also experiences DRAM stalls due to the main memory latency. These both imply that `Pooling_fwd` is stalled due to cache misses and long memory access latencies.

From the profiling results of Unet, we can give our answer to **Question 3** raised earlier: for the same platform and the same DNN application, different kernels can have different behaviors, which leads to bottlenecks for various reasons (core bound, cache bound, etc).

2) NMT: For NMT (shown in figure 4), `GradGenerator` from Eigen library takes 16.1% of total execution time. `Binary_product` takes 13.2%, and other functions like `Unary_exp`, `LossGenerator`, `Binary_diff` together take around 11%. All these functions together take around 61.8% of the total execution time. `GradGenerator` is used for calculating the gradient during back-propagation, and is used to update weights. It includes lots of exponential, addition and division operations, which make it time-consuming. `Binary_product` function does, as from the name, scalar product operation on two vectors.

As shown in figure 4 `GradGenerator` function is bounded by both front-end and back-end. Front-end takes 20.2% and back-end takes 16.9%. The front-end bound also exists in `Binary_product` function, `LossGenerator` function, where each of them spent 17.4% and 14.3% of their total execution time. These functions are front-end bound because of front-end latency (as shown in figure 6a). Similar to Unet, the front-end latency comes from MS Switches.

The 16.9% back-end bounded stalls of `GradGenerator` consist of 10.5% of stalls which are caused by core and another 6.4% of stalls which are caused by memory (as one can observe from figure 5). To have a deeper understanding of the core bound and memory bound, we show a detailed breakdown in figure 7a. As one can see, divider consumes most execution time, which means the DIV units in Xeon architecture are occupied for long execution time. This is intuitively reasonable because the `GradGenerator` function calls division operation many times. Although stalls caused by port utilization are fewer than the stalls caused by the divider, it still takes 36.3%. This is because the stalls from the ports mainly come from port 5 which is connected to ALU. That is further because of the complex calculation in the function.

By comparing the NMT profiling results with Unet, we start to answer **Question 1** – whether the different DNN networks experience similar behaviors on the same platform? From table III together with the figures and our discussion, it is clear that the Unet and NMT have different runtime execution behaviors. They have different hotspot functions and different bottlenecks on the same platform. This is due to the fact that CNN and RNN have different behaviors.

3) ResNet-50: The top five functions of ResNet-50 take around 38.6% of execution time. Unlike the other networks, ResNet-50 does not have any single function that dominates the total execution time.

Figure 4 shows the front-end/back-end breakdown of each function in ResNet-50. `__stop_notes` is bounded by both front-end (41.6%) and back-end (39.5%), whereas, other three functions are mainly bounded by back-end (78.9% - 92.7%). This is because `__stop_notes` is both compute intensive and memory intensive whereas other three functions are mainly memory intensive.

The `__stop_notes` function is the most time-consuming function of ResNet-50 which stores the content information for the kernels. In the back-end, 23.5% of the stalls are from the core, whereas 16% of the stalls are from memory. `__stop_notes` function only uses ALU, and does not have division operation. Therefore, from figure 7, all of the bounds are caused by port utilization, not by divider utilization. For the memory, most of the bounds are from the L1 (9.6% shown in figure 8a), and they are all caused by FB full (figure 8b). As previously explained, FB full metric shows the percentage of stalls due to the unavailability of L1D fill buffer.

`Binary_sum` function adds tensors. Unlike the unary sum, the binary sum takes two operands as input. From figure 4, we can observe that this function is definitely bounded by the back-end (92.7%) due to the memory (figure 5). Additionally, from figure 8a, we can observe that the significant stalls are because of the DRAM in the memory hierarchy. Moreover, DRAM bandwidth (figure 8d) constitute the significant portion of DRAM stalls. The reason behind this is that ResNet-50 performs addition operations on large sized tensors, which generates a large number of DRAM reads and writes.

Comparing ResNet-50 with Unet using the hotspot function analysis and our in-depth characterization, we can say that the runtime behaviors of different DNN applications on the same platform are quite different from each other, even they are both CNN-based (**Question 1**).

4) DenseNet: From table III, we can observe that `Gebp` occupies the majority of the execution time (59.01%). `Gebp` function from the Eigen library is basically matrix multiplication, and is used for GEMM (General Matrix Multiply). About 84.2% of `Gebp` is retiring, which means there is only a small portion of the pipeline is stalled. Thus, it shows that the `Gebp` function is already optimized.

The second most time-consuming function is `Gemm_reshape` which also performs matrix multiplication

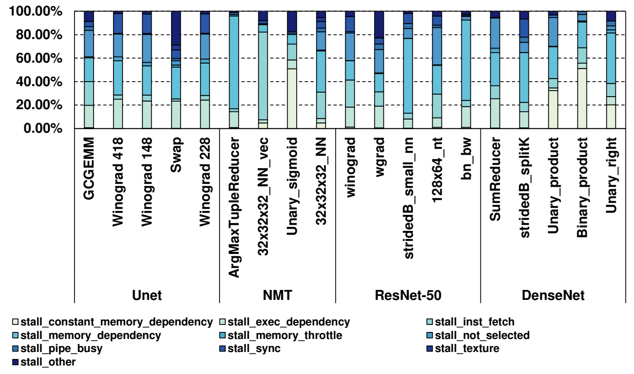
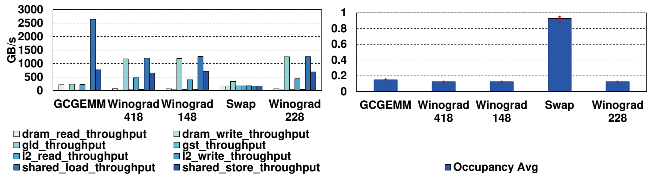


Fig. 9: The stall reasons for GPU.



(a) (b)

Fig. 10: Unet on GPU (a) Memory throughput (b) Occupancy.

(GEMM) with data reshaping. It is also bounded by back-end, and the stalls come from both core and memory. For core, `Gemm_reshape` is bounded by ports which is similar as `Gebp`. Since `Gemm_reshape` requires a lot of computation, it heavily exploits Port 0 (connected to ALU). Moreover, `Gemm_reshape` experiences stalls caused by the memory, especially the L3 and the DRAM (figure 8). This is because `Gemm_reshape` generates a large amount of data reads and writes needed for matrix multiplication.

D. Characterization on GPU Architecture

Table IV shows the top 5 GPU hotspot functions of each DNN application in its CUDA version. We ignore the execution on the host CPU side and only characterize the kernels executed on GPU. From the table, it is clear that the most time-consuming functions are GEMM or GEMM related functions (like Winograd) from cuDNN library [12]. Additionally, the GPU kernels are mostly delayed because of the application intrinsic data dependencies – *execution dependency* and *memory dependency*. Execution dependency occurs when the input of the instruction is not available, and likewise, memory dependency happens when the instructions cannot be executed due to the waiting of required data. For the stalls due to the execution dependency, it can be hidden by increasing instruction-level parallelism (ILP). For the stalls caused by memory dependency, increasing memory-level parallelism (MLP) can help. Moreover, keeping the data inside the SMs' shared memory is also helpful. From table IV, one can observe that 20% of the top 5 hotspot functions for the four algorithms have both execution dependency and memory dependency issue, 42.5% of the top 5 hotspot functions experience at least one of these two issues. There can be other issues causing the degradation of GPU performance such as instruction fetch problem and warp scheduling problem.

TABLE IV: Top 5 hotspot functions of Unet, NMT, ResNet-50, DenseNet run on **GPU** (CM: Constant Memory, E: Execution dependency, IF: Assembly Instruction not Fetch, MD: Memory Dependency, MT: Memory Throttle, N: Warp-Not-selected, P: Pipe busy, S: Sync, T: Texture, O: Other)

Algorithm	Function (Addr)	Explanation	CM	E	IF	MD	MT	N	P	S	T	O	%
Unet	Batched GCGEMM (GCGEMM)	Batched GEMM		O	O	O		O					24.99%
	CuDNN Winograd 418 tile (Winograd 418)	Winograd		O		O		O					5.66%
	CuDNN Winograd 148 tile (Winograd 148)	Winograd		O		O		O					5.17%
	Tensorflow Swap Dim (Swap)	Swap		O		O						O	3.97%
	CuDNN Winograd 228 tile (Winograd 228)	Winograd		O		O		O					3.89%
NMT	ArgMaxTupleReducer	Argmax Tuple				O							74.06%
	CUDA memcpy DtoH	Memory copy Device to Host											8.95%
	32x32x32_NN_vec	SGEMM			O								2.21%
	Unary_sigmoid	Sigmoid	O										1.91%
	32x32x32_NN	SGEMM			O	O							1.75%
ResNet-50	Winograd	Winograd		O	O	O		O					12.29%
	Wgrad	Winograd		O	O	O		O				O	12.05%
	stridedB_small_nn	Strided access				O							5.07%
	128x64_nt	SGEMM			O	O		O					4.76%
	bn_bw	CUDA kernel call				O							4.32%
DenseNet	SumReducer	Sum reduce		O		O		O					12.65%
	stridedB_splitK	Strided access				O							8.74%
	Unary_product	Unary product	O			O		O					8.51%
	Binary_product	Binary product	O										7.65%
	Unary_right	Unary right				O							5.50%

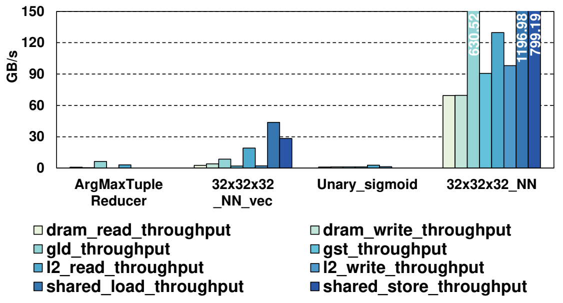
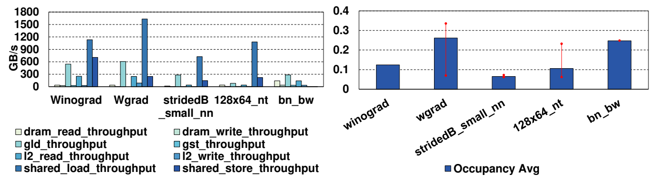


Fig. 11: GPU memory throughput for NMT.

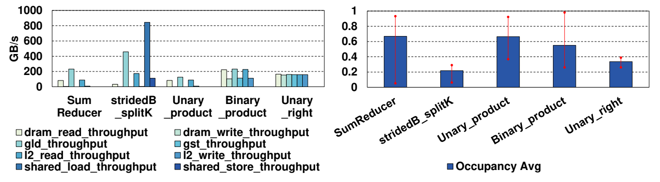


(a) (b)

Fig. 12: ResNet-50 on GPU (a) Memory throughput (b) Occupancy.

Instruction fetch shows the percentage of stalls caused by the failure of fetching the next assembly instruction. We use warp-not-selected to represent the stalls caused by warp scheduling problem. Specifically, warp-not-selected expresses the percentage of stalls because the warp is ready but not issued by the warp scheduler.

1) Unet: For Unet, `GCGEMM` takes 25% of the total execution time. `GCGEMM` is part of cuBLAS library which runs general matrix multiplication (GEMM) on GPU. From Figure 9, where the reasons of the stalls for the top 5 hotspot functions are presented, we can find out that `GCGEMM` is almost equally stalled because of execution dependency, failure in fetching assembly instructions, memory dependency, and warp not being selected. Also, from figure 10b, occupancy of `GCGEMM` is low. Specifically, only 12% of warps are active on average. This is because `GCGEMM` consists of many execution dependencies and memory dependencies which make the GPU resources underutilized.



(a) (b)

Fig. 13: DenseNet on GPU (a) Memory throughput (b) Occupancy.

By comparing the profiling results of Unet running on CPU against the results of Unet running on GPU, we can partially answer **Question 2**. As one can observe, the Unet hotspot functions on CPU and GPU are different. That is, between CPU and GPU, the same DNN network behaves differently.

2) NMT: For NMT, `ArgMaxTupleReducer` is the major function which takes 74% of total execution time. `ArgMaxTupleReducer` is the function that reduces the tuple with the maxima. From figure 9, 79% of stalls in `ArgMaxTupleReducer` are caused by memory dependency. From figure 11, `ArgMaxTupleReducer` only accesses global memory and L2 cache for read, and the average throughputs are 6.30 GB/s and 3.09 GB/s, respectively. Input data of NMT is text so that its memory throughput is naturally much smaller than other applications that use images as an input. Considering that the memory bandwidth of GPU is much higher than NMT memory throughput, the stalls caused by memory dependency are not because of the limited memory bandwidth, but it is likely happened due to waiting for the data to be ready. Next, CUDA memcpy DtoH (memory copy from device to host) takes 9% of the execution time.

The runtime behavior between NMT and Unet on GPU is different (**Question 1**). In addition, the GPU runtime behavior of both Unet and NMT is different from their corresponding execution on CPU (**Question 2**).

3) ResNet-50: For ResNet-50, `Winograd` from scuDNN and `Wgrad` cuDNN are the hotspots. They together constitute 24.34% of total execution time (12.29% from `Winograd` and 12.05% from `Wgrad`). Both of the functions are stalled by execution dependency, instruction fetch, memory dependency, and warp-not-selected, which are similar to the GCGEMM function in Unet. These are the reasons why both functions have low GPU occupancy (12.46% and 26.13% in figure 12b).

4) DenseNet: For DenseNet, SumReducer is the most time-consuming function, which takes 12.65% of total execution time. The stall reasons for the hotspot function SumReducer are execution dependency, memory dependency, and warp-not-selected. As explained before, stalls due to execution dependency and memory dependency are caused by unready input and on-flight memory accesses. The average occupancy of SumReducer is comparably high where 66.86% of warps are active. However, the standard deviation of GPU occupancy is also high where the lowest occupancy is only 5.18%. That means, the kernel cannot occupy the GPU in some execution phased due to the aforementioned reasons.

E. Characterizations on the Xeon Phi architecture

Table V shows top 5 hotspot functions and their bottlenecks. Like Xeon, we also used Vtune to characterize the algorithms. By characterizing the algorithms run on Xeon Phi, we could find out this knowledge. First, the hotspot functions from Xeon Phi are similar to the ones from Xeon. It is because 1) the kernels running on Xeon and Xeon Phi are similar 2) the kernels are already well parallelized on Xeon so even though it runs on Xeon Phi it does not change that much. Second, applications run on Xeon Phi experience less memory bounds compared to the Xeon run. It is because Xeon Phi has a lot of physical cores, thus the data size needed for one core is reduced. Moreover, the total cache and memory size is much larger than the ones on Xeon. From this, the functions could have a high L1 and L2 hit ratio which in turn makes the functions relatively more computation bound.

1) Unet: Like Xeon, Off_l is the most time-consuming function, for Xeon Phi Off_l takes 29.45% of the total execution time. Figure 14 shows the bottlenecks of the functions on Unet. Off_l is back-end bound, especially computation bound. The L1 cache hit ratio of Off_l is 100%, which fully utilizes the L1 cache. However, about 14% of Off_l experience poor CPU utilization, meaning its CPU resource usage is less than or equal to 50% while 14% of CPU runtime. Off_l is also computation bound on Xeon, since this function requires a lot of computations by the algorithm itself, which in turn makes the function as computation bound naturally.

Comparing Unet run on Xeon Phi with Unet run on CPU, it can be found that their behavior is comparably similar (**Question 2**). Unlike GPU, the architecture of Xeon and Xeon Phi is relatively similar, moreover, they can share the same libraries. Thus, for Xeon Phi, the same DNN network experience similar behaviors with the Xeon.

2) NMT: GradGenerator takes 21.97% of total execution time on NMT, which makes GradGenerator function most time-consuming as same as Xeon. From figure 14, we can check that GradGenerator is back-end bound, especially the bottleneck is computation. For L1 cache, the hit ratio is about 99.9%, which is very high like the Off_l from Unet. Oppositely, GradGenerator CPU utilization is very low, 100% of its runtime is poor which utilizes less than or equal to 50% of CPU resource. Same as the run on Xeon, GradGenerator is also core bound since it employs a divider unit for gradient calculation.

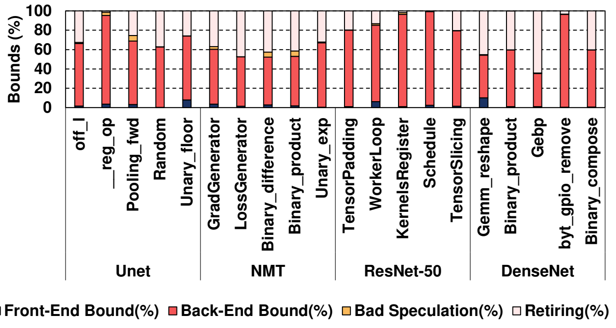


Fig. 14: Percentage of the bottlenecks (front-end, back-end, bad speculation) and retiring on Xeon Phi.

Comparing the Unet and NMT run on Xeon Phi, it is clear that their behavior is different like Xeon and GPU which can fully answer the **Question 1**.

3) ResNet-50: TensorPadding takes 15.51% of total execution time on ResNet-50. On Xeon, TensorPadding is not the most time-consuming function, it was the fourth. From figure 14, we can check that TensorPadding is back-end bound, and also core is the bottleneck. L1 cache hit ratio is about 100%, which is also very high. On the other hand, CPU utilization of TensorPadding is low, during the 100% of its runtime, it utilizes less than or equal to 50% of total CPU resource. For the other hospot functions, the difference between ResNet-50 hotspot functions with the other benchmarks run on Xeon Phi is that it has a lot of scheduling functions. WorkerLoop, Schedule are all job scheduling related functions. These functions do not require data accesses a lot, so obviously they are front-end bound or core bound.

4) DenseNet: Gemm_reshape takes 16.65% of the total execution time of DenseNet, which is not the same as Xeon. In Xeon, the Gebp function takes about 59% of the total execution time, but in Xeon Phi, each of the Gemm_reshape, Binary_product, Gebp functions takes 12.47%-16.65% of the total execution time. Thus, for DenseNet on Xeon Phi, we do not have a single function that dominates the overall algorithm. This is because, in Xeon, the functions were well-optimized, and as a result, the percentages of front-end or back-end bound were lower. However, for Xeon Phi, the back-end (especially computation) becomes a more serious bottleneck. For example, the Gemm_reshape function becomes back-end bound (figure 14), specifically from the computation angle. Its L1 cache hit ratio is about 99%, while its CPU utilization is quite low (in fact, it utilizes less than 50% of the total CPU resources).

IV. Simulating Larger Cores/Memory Capacities

In this section, our goal is to measure whether increasing some of the compute and memory resources can help us alleviate the bottlenecks identified and quantified in the previous section. For these purposes, we target the CPU platform

TABLE V: Top 5 hotspot functions of Unet, NMT, ResNet-50, DenseNet run on **Xeon Phi** and their main bounds (C: Core, L1: L1 cache, L2: L2 cache)

			Front-end	Back-end			Bad speculation	Retiring	
Algorithm	Function (Addr)	Explanation		C	L1	L2			%
Unet	Off_l	Offset conversion		O				O	29.45%
	__reg_op	Bitmap allocation, release		O					17.94%
	ref_pooling_fwd_t (Pooling_fwd)	Forward pooling				O			9.12%
	PhiloxRandom (Random)	Randomize		O				O	2.99%
	Unary_floor	Unary floor		O					2.74%
NMT	GradGenerator	Calculate backward gradient		O		O		O	21.97%
	LossGenerator	Calculate loss		O				O	13.83%
	Binary_difference	Binary difference		O				O	13.74%
	Binary_product	Binary product		O				O	9.38%
	Unary_exp	Unary exponential		O		O		O	5.42%
ResNet-50	TensorPadding	Padding		O					15.51%
	WorkerLoop	Loop		O		O			8.95%
	KernelsRegisteredFor (KernelsRegister)	Kernels Registered		O		O			8.04%
	Schedule	Scheduler		O					4.39%
	TensorSlicing	Slicing		O					4.28%
DenseNet	Gemm_reshape	Copy matrix mul data with reshape		O				O	16.65%
	Binary_product	Binary product		O				O	15.41%
	Gebp	Kernel for GEMM				O		O	12.47%
	byt_gpio_remove	Byt gpio remove		O					10.12%
	Binary_compose	Binary compose		O				O	8.78%

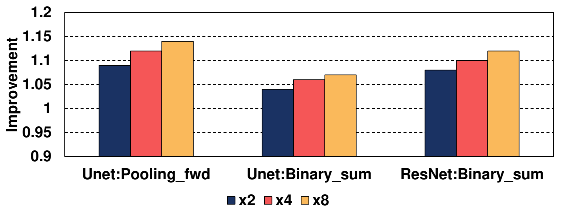


Fig. 15: Simulation results.

and use Gem5 simulator [8]. We choose three different kernels from different applications, which are `Pooling_fwd`, `Binary_sum` from Unet, `Binary_sum` from ResNet-50. Recall that, each of these kernels has a different bottleneck, as explained earlier in Section III. From this, we find out that `Pooling_fwd` (top 2 function on Unet) is L3 bound, `Binary_sum` (top 5 function on Unet) is L2 bound, `Binary_sum` (top 2 function on ResNet-50) is DRAM bandwidth bound. We set the base configuration as L1 cache = 64 KB, L2 cache = 2 MB, and L3 cache = 16 MB, and then increase their sizes. If the kernel is L2 bound, we increase L1 cache to reduce L1 misses, and likewise for L3 bound, we increase the L2 cache capacity. Also, we mimic increasing DRAM bandwidth by adding banks.

Figure 15 shows the improvements achieved over the original configuration. The bottleneck of `Pooling_fwd` is L3; so, we increase the L2 size by $\times 2$ (4 MB), $\times 4$ (8 MB), and $\times 8$ (16 MB). From this, we can improve `Pooling_fwd` by 14% ($\times 8$), which means that we can improve the Unet application's performance by 2.17% since Pooling_fwd takes 17.27% of the total execution time of Unet. Likewise, we improve `Binary_sum` kernel from ResNet-50 12% by increasing the DRAM bandwidth. Note that, `Binary_sum` takes 7.21% of ResNet-50, and as a result, we can improve the overall performance of ResNet-50 by 2.23%.

These results clearly show that, by increasing the select hardware resources, 1) the bottleneck kernels of these applications can be significantly improved, and 2) this, in turn, can lead to decent savings in the overall application performance.

V. Discussion

We now summarize our characterization and give answers to the four questions in Section I.

A. Whether the different DNN networks exhibit similar behavior on a given execution platform?

The answer is *No*. From the characterization study presented in Section III, it is clear that each DNN algorithm has different hotspot functions and these different hotspot functions experience, in general, different architectural bottlenecks. Naturally, the hotspot functions in NMT are different from the hotspot functions in other CNN based networks; however, even within the CNN-based networks, the most time-consuming function can be different for different applications. Even for some similar functions across different DNN algorithms, for instance, `Binary_product` in both Unet and NMT, the underlying architectural bottlenecks are different due to the variance in input data.

B. Whether, across different platforms, a given DNN network exhibits different behaviors?

The answer is *yes or no*. It depends on the architecture design. Clearly, the executions in Xeon/Xeon Phi and GPU are totally different. This is primarily because of the significant architectural differences. However, between Xeon and Xeon Phi, the works they perform are similar to some extent. It can be seen from our experiments that, out of 4 DNN applications, two of them have the same top 1 hotspot function, and the reason for the performance bottleneck is similar between the functions. This is because the architectural difference between Xeon and Xeon Phi is relatively small compared to GPU.

C. For the same execution platform and same DNN network, whether different execution phases have different behaviors?

This is about comparing different kernels in one DNN application running on a given compute platform. And, the answer is *Yes*. From Tables III, IV, and V, it is clear that each function has different behaviors (compared to others) running on the same platform. This is because each function of an application is bounded by different reasons.

D. Are the current major general-purpose platforms tuned sufficiently well for different DNN algorithms?

The answer is a *partial yes*. Across all the compute platforms tested, functions from libraries (like MKL BLAS, cuDNN, etc.) are well-optimized for the targeted architectures. However, some functions still experience high computational or memory bottlenecks, which provide architectural optimization potentials to better support DNN applications. For example, on Xeon, there are several functions bounded by DRAM bandwidth; the performance of these kernels can be improved by increasing DRAM bandwidth. Likewise, the performance of many GPU kernels can be improved by minimizing the dependency issue and, for Xeon Phi, improving core utilization would be the most beneficial option.

VI. Related Work

Software DNN acceleration: There exist many software optimizations aiming at improving DNN execution on commercial compute platforms [5, 12]. NVIDIA released cuDNN [12], which is a GPU-accelerated library of primitives for DNNs. It provides highly optimized implementations for compute-intensive kernels (e.g., convolution and normalization) in DNN applications. Intel(R) proposed Math Kernel Library for Deep Neural Networks (Intel(R) MKL-DNN) [5], which accelerates DNN applications running on Intel architectures.

Specific hardware DNN accelerators: Multiple specific hardware accelerator designs for DNN applications have been proposed recently [13]. Chen et al. [11] proposed the Eyerris DNN accelerator to minimize the data movement energy cost by row stationery (RS) processing dataflow on a spatial architecture with 168 processing elements. Fowers et al. [15] proposed a configurable architecture that can be used to scale DNN processing, targeting low latency, high throughput, and high efficiency. Compared to those specific accelerators, we focus on general-purpose computing platforms.

VII. Conclusion

In this paper, we analyzed four well-known DNN algorithms – Unet, NMT, ResNet-50, and DenseNet – on three widely-used compute platforms, namely, Xeon, GPU, and Xeon Phi. Our results reveal that i) the different DNN networks exhibit different behaviors on the same platform; ii) across different platforms, the same application can have different behaviors, depending on the architectural differences; iii) for the same platform and same application, different execution phases have different behaviors; and, iv) current major general-purpose platforms perform fairly well when working with the available DNN libraries (like MKLBLAS, CuDNN, etc). Still, current platforms have further scope to improve – for example, by increasing the DRAM bandwidth in Xeon, improving the core utilization in Xeon Phi, and minimizing the dependency issues in GPUs.

VIII. Acknowledgement

This work is supported in part by NSF grants 1526750, 1763681, 1439057, 1629129, 1629915, 1931531, 1908793, 1822923 and two grants from Intel.

References

[1] https://software.intel.com/en-us/vtune.
[2] http://celltrackingchallenge.net/2d-datasets/.
[3] https://nlp.stanford.edu/projects/nmt/.
[4] https://www.cs.toronto.edu/ kriz/cifar.html.
[5] https://software.intel.com/en-us/mkl-developer-reference-fortran.
[6] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. Tensorflow: A system for large-scale machine learning. In *OSDI*, 2016.
[7] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. *arXiv*, 2014.
[8] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. The gem5 simulator. *Comput. Archit. News*, 2011.
[9] Mariusz Bojarski, Davide Del Testa, Daniel Dworakowski, Bernhard Firner, Beat Flepp, Prasoon Goyal, Lawrence D Jackel, Mathew Monfort, Urs Muller, Jiakai Zhang, et al. End to end learning for self-driving cars. *arXiv*, 2016.
[10] Tianshi Chen, Zidong Du, Ninghui Sun, Jia Wang, Chengyong Wu, Yunji Chen, and Olivier Temam. Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning. In *ACM Sigplan Notices*, 2014.
[11] Y. Chen, T. Krishna, J. S. Emer, and V. Sze. Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks. *IEEE Journal of Solid-State Circuits*, 2017.
[12] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. cuDNN: Efficient primitives for deep learning. *arXiv*, 2014.
[13] Woong Choi, Kwanghyo Jeong, Kyungrak Choi, Kyeongho Lee, and Jongsun Park. Content addressable memory based binarized neural network accelerator using time-domain signal processing. In *DAC*, 2018.
[14] Ronan Collobert, Jason Weston, Léon Bottou, Michael Karlen, Koray Kavukcuoglu, and Pavel Kuksa. Natural language processing (almost) from scratch. *Journal of machine learning research*, 2011.
[15] Jeremy Fowers, Kalin Ovtcharov, Michael Papamichael, Todd Massengill, Ming Liu, Daniel Lo, Shlomi Alkalay, Michael Haselman, Logan Adams, Mahdi Ghandi, Stephen Heil, Prerak Patel, Adam Sapek, Gabriel Weisz, Lisa Woods, Sitaram Lanka, Steven K. Reinhardt, Adrian M. Caulfield, Eric S. Chung, and Doug Burger. A configurable cloud-scale dnn processor for real-time ai. In *ISCA*, 2018.
[16] Klaus Greff, Rupesh K Srivastava, Jan Koutník, Bas R Steunebrink, and Jürgen Schmidhuber. Lstm: A search space odyssey. *Transactions on neural networks and learning systems*, 2016.
[17] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *CVPR*, 2016.
[18] Gao Huang, Zhuang Liu, Laurens Van Der Maaten, and Kilian Q Weinberger. Densely connected convolutional networks. In *CVPR*, 2017.
[19] Huaipan Jiang, Anup Sarma, Jihyun Ryoo, Jagadish B Kotra, Meena Arunachalam, Chita R Das, and Mahmut T Kandemir. A learning-guided hierarchical approach for biomedical image segmentation. In *SOCC*, pages 227–232, 2018.
[20] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *NIPS*, pages 1097–1105, 2012.
[21] Tesla NVIDIA. P100 white paper. *NVIDIA Corporation*, 2016.
[22] Minsoo Rhu, Natalia Gimelshein, Jason Clemons, Arslan Zulfiqar, and Stephen W Keckler. vDNN: Virtualized deep neural networks for scalable, memory-efficient neural network design. In *Micro*, 2016.
[23] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. U-net: Convolutional networks for biomedical image segmentation. In *International Conference on Medical Image Computing and Computer-Assisted Intervention*, 2015.
[24] Jihyun Ryoo, Orhan Kislal, Xulong Tang, and Mahmut Taylan Kandemir. Quantifying and Optimizing Data Access Parallelism on Manycores. In *MASCOTS*, 2018.
[25] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv*, 2014.
[26] Avinash Sodani, Roger Gramunt, Jesus Corbal, Ho-Seop Kim, Krishna Vinod, Sundaram Chinthamani, Steven Hutsell, Rajat Agarwal, and Yen-Chen Liu. Knights landing: Second-generation intel xeon phi product. *IEEE Micro*, 36(2):34–46, 2016.
[27] Xulong Tang, Mahmut Kandemir, Praveen Yedlapalli, and Jagadish Kotra. Improving Bank-Level Parallelism for Irregular Applications. In *MICRO*, 2016.
[28] Xulong Tang, Mahmut Taylan Kandemir, Hui Zhao, Myoungsoo Jung, and Mustafa Karakoy. Computing with near data. In *SIGMETRICS*, 2019.
[29] Xulong Tang, Orhan Kislal, Mahmut Kandemir, and Mustafa Karakoy. Data movement aware computation partitioning. In *MICRO*, 2017.
[30] Xulong Tang, Ashutosh Pattnaik, Huaipan Jiang, Onur Kayiran, Adwait Jog, Sreepathi Pai, Mohamed Ibrahim, Mahmut Kandemir, and Chita Das. Controlled Kernel Launch for Dynamic Parallelism in GPUs. In *HPCA*, 2017.
[31] John R Trudeau. Language translation for real-time text-based conversations, November 16 1999. US Patent 5,987,401.