2018 51st Annual IEEE/ACM International Symposium on Microarchitecture

In-Register Parameter Caching for Dynamic Neural Nets with Virtual Persistent Processor Specialization

Farzad Khorasani†
Tesla, Inc.
Palo Alto, CA, USA
fkhorasani@tesla.com

Hodjat Asghari Esfeden
Department of Computer Science
University of California Riverside
Riverside, CA, USA
hasgh001@ucr.edu

Nael Abu-Ghazaleh
Department of Computer Science
University of California Riverside
Riverside, CA, USA
naelag@ucr.edu

Vivek Sarkar
School of Computer Science
Georgia Institute of Technology
Atlanta, GA, USA
vsarkar@gatech.edu

***Abstract*—Dynamic neural networks enable higher representation flexibility compared to networks with a fixed architecture and are extensively deployed in problems dealing with varying input-induced network structure, such as those in Natural Language Processing. One of the optimizations used in training networks is persistency of recurrent weights on the chip. In dynamic nets, a possibly-inhomogeneous computation graph for every input prevents caching recurrent weights in GPU registers. Therefore, existing solutions suffer from excessive recurring off-chip memory loads as well as compounded kernel launch overheads and underutilization of GPU SMs.**

In this paper, we present a software system that enables persistency of weight matrices during the training of dynamic neural networks on the GPU. Before the training begins, our approach named Virtual Persistent Processor Specialization (VPPS) *specializes* a forward-backward propagation kernel that contains in-register caching and operation routines. VPPS *virtualizes* persistent kernel CTAs as CISC-like vector processors that can be guided to execute supplied instructions. VPPS greatly reduces the overall amount of off-chip loads by caching weight matrices on the chip, while simultaneously, provides maximum portability as it does not make any assumptions about the shape of the given computation graphs hence fulfilling dynamic net requirements. We implemented our solution on DyNet and abstracted away its design complexities by providing simple function calls to the user. Our experiments on a Volta micro-architecture shows that, unlike the most competitive solutions, VPPS shows excellent performance even in small batch sizes and delivers up to 6x speedup on training dynamic nets.

***Index Terms*—GPU, Deep Learning, Neural Network, Dynamic Neural Network, Persistent, Specialization, Register**

I. Introduction

With the availability of very large data sets, recent years have seen a revitalization of interest in the use of neural networks for various Machine Learning fields such as computer vision [1], speech recognition [2], and Natural Language Processing [3]. Training deep neural networks using large data sets is a compute-intensive and time-consuming process. Typically GPUs are used to accelerate the training by providing massive compute parallelism. Training neural nets usually includes many iterations of feeding inputs to the network and extracting the gradients using backpropagation to make model parameters converge to the desired answer. The network architecture is commonly represented as a computation graph in which nodes stand for the operations and edges connecting the nodes show the flow of the data (tensors).

An important class of neural networks is dynamic neural nets; as opposed to more traditional static networks, the computation graph in these problems may change for every given input. For example, this change can be due to different input sizes, such as in Long Short-Term Memory (LSTM) [4], or due to variable network structures that different inputs create, such as in tree-structured LSTM [5] networks. The dynamic nature of the neural network provides Machine Learning researchers with model specification freedom.

From a computational perspective, however, dynamic neural networks complicate or even inhibit applying a body of GPU optimizations that are often critical for improving the performance of static networks. One such optimizations is on-chip persistence of recurrent weights. In particular, Persistent RNN [6] exploits caching weight matrices for a Recurrent Neural Net (RNN) on-chip inside the GPU's fast and large register file to eliminate the recurring cost of off-chip loads via persistent threads [7]. However, Persistent RNN assumes a static structure of the computation, including pre-determined placement of inputs as well as computation graph nodes. If the way in which input tensors and parameters mix in the network changes across inputs, Persistent RNN cannot be applied. This is in contrast with the nature of dynamic nets where *every input* may result in a computation graph with a different shape. Moreover, even if the operation set is predictable, Persistent RNN has to be specifically re-crafted by an expert to be applicable for every RNN variation (for example, as in GRU [8]). If a user specifies a custom RNN architecture, this technique would not be readily applicable.

To enable in-register parameter caching (i.e., persistence) for dynamic neural networks, we propose a software system named Virtual Persistent Processor Virtualization (VPPS). Prior to training, our solution Just-In-Time (JIT) compiles a single forward-backward kernel specialized for caching the given model parameters. For every forward-backward pass over multiple possibly-dissimilar computation graphs, VPPS

†This work was done while the author was at Georgia Tech.

978-1-5386-6240-3/18/$31.00 ©2018 IEEE
DOI 10.1109/MICRO.2018.00038

generates a script that guides the execution of Cooperative Thread Arrays (CTAs) within the generated kernel. By allowing the parameters of the given model to stay on chip throughout forward and backward propagation for an input batch, our design eliminates the cost of fetching recurring weight matrices from the off-chip DRAM. When constructing the single kernel that is launched for every forward-backward pass, we utilize the information about model parameters to i) distribute them across register file of SMs, and ii) specialize matrix operations, such as weight matrix multiplication with a vector. This JIT compilation of the kernel happens only once before training loop, and is necessary due to literal indexing of registers within the kernel binary.

Since the computation graph shape can potentially be different for every input, in the kernel every persistent CTA is treated as a virtual CISC-like vector processor capable of executing instructions according to its supplied script. VPPS encodes the operations within batches of input graphs at the host (CPU) side, packages them, and supplies them to the kernel via a host-to-device memory copy. Each virtual processor receives its exclusively assigned set of instructions, interprets them, and executes the operations using its threads accordingly. This scheme not only provides maximum portability as it makes no assumption about the structure of computation graphs in the input batch, but it also eliminates the overhead associated with launching numerous kernels by enabling the execution of all the operations in the forward and backward pass as well as updating the parameters with collected gradients using only one CUDA kernel invocation.

Similar to on-the-fly operation batching [9], multiple dissimilar computation graphs can be fed to the framework together for training. This ability in fact provides virtual processors with more task parallelism to exploit. We implemented our design in DyNet [10] and automated all parts of it as a C++ library. An end-user can essentially benefit from our solution using only two calls. The first call is outside the training loop to pass the information about the model parameters, and results in JIT compilation of the CUDA kernel using the Nvidia Runtime Compiler (NVRTC) behind the scene. The second call is within the training loop, and is performed for every batch of computation graph. Our implementation autonomously carries out script generation and transfer, as well as the kernel launch. As opposed to the state-of-the-art solution [9], our approach delivers excellent performance even with small batches.

This work makes the following contributions.

- We propose Virtual Persistent Processor Virtualization (VPPS), a software technique to enable in-register caching of model's recurring weight matrices in dynamic neural nets and thereby greatly reduce the overall amount of DRAM load requests.
- We discuss efficient implementation of VPPS mechanics, including parameter distribution across GPU SMs, kernel specialization, virtualizing persistent CTAs as CISC-like vector processors, and script generation and execution. We also present a set of optimizations to enhance the concurrency.
- We automated VPPS as a C++ software library integrated with DyNet that abstracts away inner details of our proposal. In comparison with the state-of-the-art solution, VPPS provides up to a 6× performance boost in training throughput.

The rest of the paper is organized as follows. Section II gives a background on dynamic neural nets and persistency of recurrent model parameters and motivates our solution. Section III elaborates VPPS. We present experimental evaluations in Section IV. Section V discusses related work and Section VI concludes the paper.

II. Background and Motivation

Training neural nets is an iterative procedure that includes visiting a training set multiple times. During each visit (*epoch*), one element or a batch of elements in the training set is fed to the network to produce their corresponding predictions. These predictions are contrasted against their respective correct answers using a loss function (*negative softmax log likelihood* for instance). The loss, which indicates the quality of the predictions, is backpropagated through the network to collect the gradients. Updating model parameters with their respective gradients enables the model to gradually enhance the accuracy of its prediction. To systematically perform the training, a specified neural network is usually transformed into a directed acyclic computation graph, where nodes denote operations and edges denote arrays of (typically) floating point values representing the usage of the content generated by the source node as the input for the destination node.

Dynamic neural networks. Dynamic nets are a class of neural networks for which the architecture of the network, and the resulting computation graph depend on the input, and hence, may change from one training input to another. In other words, while the parameters of the model, which are its being-learned pieces, are reused for computation graphs across different input instances, the set of operations tensors have to go through depends upon the input as specified by the user. Figure 1 shows an example of such network, from Tree-Structured Long Short-Term Memory (LSTM) Sentiment Analyzer [5], [10] application, for two inputs unrolled over time. In this example, input word vectors are fed to the LSTM [4] and output vectors produced by the LSTM instances are mixed based on the parse tree of the sentences. Note that depicted LSTMs in Figure 1 share the same parameter set. Clearly, different input sentences have different parse trees, and therefore, different network architectures; consequently, they induce computation graphs with different shapes. The emergence of such dynamic neural nets has given rise to the popularity of frameworks such as Chainer [11], PyTorch [12] and DyNet [10] that can construct the computation graph on-the-fly for every input, which is in contrast with the approach in frameworks such as TensorFlow [13] or Caffe [14] that relies on static definition of the network architecture.

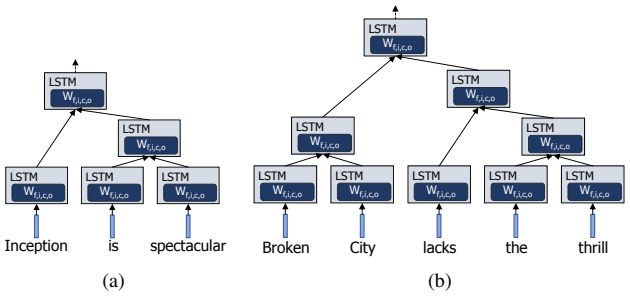


Fig. 1. Two example input sentences creating different network architectures in the Tree-Structured LSTM Sentiment Analyzer application [5], [10], which aims to classify the connotation of the sentence as positive or negative.

On-Chip Parameter Persistency. Within the computation graph induced by one input or a batch of inputs, model parameters in the form of weight matrices are usually used a number of times for multiplying with the input vectors or with the intermediate tensor vectors. In frameworks such as DyNet and PyTorch, upon visiting every multiplication node that uses these matrices, the weights have to be fetched from GPU's off-chip DRAM to carry out the operation. Repeated off-chip loads of these weight matrices, however, account for the majority of global memory loads during training, as shown in Figure 2. The recurrent nature of such operation hints an opportunity to cache the model parameters on the chip in order to avoid excessive memory accesses and to boost the performance; specially when neural net workloads are mostly memory-bound [15].

Diamos *et al.* [6] suggested utilizing GPU register file to cache recurrent weight matrices on chip when training a model using vanilla RNNs. Not only registers have the highest access bandwidth and lowest access latency compared to their most competitive on-chip resource (shared memory), but also register file is the largest memory on SM. Persistent RNN follows persistent threads [7] programming style where enough number of CTAs occupy all the SMs while task decomposition is handled within a single kernel, which, in the case of Persistent RNN, disallows register file content invalidation as a side-effect of kernel relaunch. In Persistent RNN, a weight matrix with a fixed size is fetched and distributed across register file of GPU SMs [1]. The kernel utilizes these cached elements to perform the recurrent multiplication of the matrix with given vectors. Persistent RNN can also be viewed as a special form of kernel fusion cognizant of the model parameter reuse.

The challenges of having both. As we mentioned, the register file content is invalidated upon kernel termination. This makes it necessary for the set of operations in-between weight matrix visits to be executed within the same kernel. As a result, Persistent RNN requires a pre-determined computation procedure. In other words, the shape of the computation graph has to be known ahead of time or repeat a specific pattern. This is in contrast with the nature of dynamic nets where the set of computation graph operations does not necessarily follow a pattern and the graph's shape may change across inputs. Clearly, one cannot afford paying the cost of compiling a special CUDA kernel for every encountered computation graph at runtime due to the relatively long compilation time.

[1]This observation in fact nicely fits in with the existing trend on the collective size of GPU register file. While largest Pascal chip (GP100) had 56 SMs, the currently largest Volta chip (GV100) has 80 SMs, giving a total of 20 MB of on-chip storage capacity. The overall number of registers on GPUs is expected to grow even more with Multi-Chip Module GPUs [16].

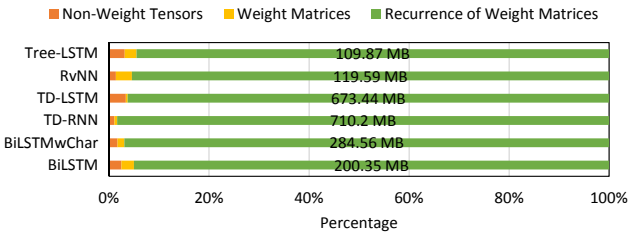


Fig. 2. Averaged distribution of off-chip DRAM loads measured for training different dynamic neural net applications in DyNet [10]. Training settings and benchmarks are the same as those discussed in Section IV.

In addition, caching into an SM register file requires explicit architected register index addressing within the kernel. Essentially the compiler has to be able to realize what specific register in the kernel binary to use for storing a specific element of the weight matrix. Therefore, not only the declaration of register arrays that are supposed to hold parameters has to have a size known at kernel compile time, but also all the references to the content of these arrays have to have literal indices. If any of these two conditions are not met, CUDA compiler defines the array as a thread-private local memory region that, instead of living inside the physical registers, resides inside the off-chip DRAM and may be cached in L1 and L2. This makes every form of neural net with recurrent weights—assuming it is describable as a repeating computation pattern—require an expert developer to design the persistent kernel for.

State-of-the-art work. The focus of the research community for optimizing GPU performance for dynamic nets so far has revolved around enabling mini-batching. While in static networks multiple inputs can be simply packed together to create tensors with higher order and increase data parallelism, this is not trivially achievable for dynamic nets. In a group of existing frameworks such as PyTorch [12] where mini-batching is manual, by default a dynamic neural net may require online learning, and therefore, invoke one kernel per operation node even if the input tensor size for the node is small and the kernel is extremely short-lived. This results in underutilization of available resources leaving a great number of SMs unoccupied. Plus, the kernel preparation overhead for the CPU and the kernel launch overhead for the GPU are comparable to the kernel duration when it is short-lived. These overheads add up to the overall training duration proportional to the frequency of such nodes and degrade the performance.

To overcome this issue, TensorFlow Fold [17] and on-the-fly operation batching [9] implemented in DyNet [10] have enabled dynamic batching of similar operations in concurrent and potentially-dissimilar computation graphs. Although these mini-batching solutions reduce the multiple kernel launch overhead and underutilization of SMs, there is no support

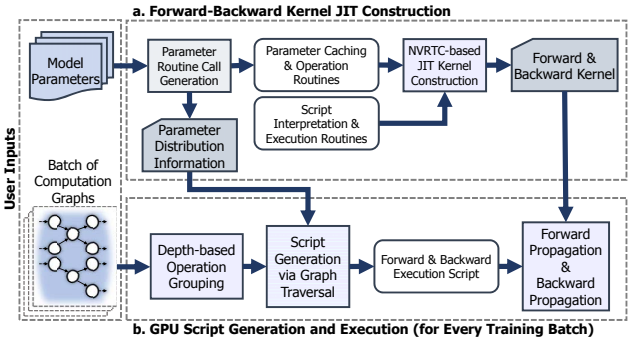


Fig. 3. An overview of our solution: Virtual Persistent Processor Specialization (VPPS).

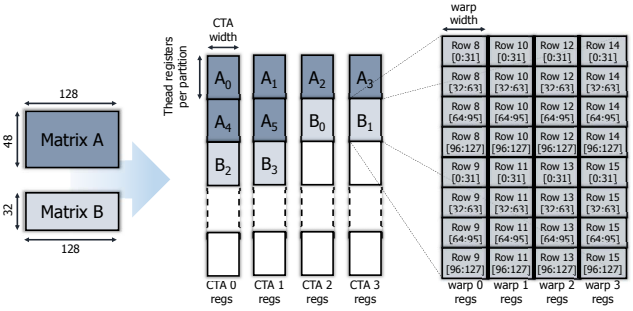


Fig. 4. An example visualizing the round-robin assignment of two weight matrices into register partitions in different CTAs. The example assumes there are 4 CTAs in total, the width of CTAs is 128, and the partition size is 1024 (8 thread registers per partition).

for persistency of recurring model parameters. The lack of this feature leads to wasting a large portion of computation time on the latency of excessive off-chip memory loads. Also, these works require very large batch sizes to saturate the GPU resources, which can lead to convergence instability [18], poorer generalization [19], and lower accuracy.

Above discussion motivates the need for a solution that enables in-register parameter persistency in dynamic training environments. Since specification of computation graphs for a batch of inputs happens at runtime and processing these graphs will account for the training time, any such performance-oriented solution must avoid overheads that can cancel out or exceed provided benefits.

III. Virtual Persistent Processor Specialization

In this Section, we present our solution *Virtual Persistent Processor Specialization* (VPPS) that allows caching of model parameters onto registers during training an input batch in dynamic neural networks. Our solution has two parts that work in tandem. The first part constructs the source for the forward-backward propagation GPU kernel using model parameters and JIT compiles it at run time. The second part generates the script for the given input batch, transfers it to the device DRAM and executes the kernel accordingly. While the first part runs only once for a given model, the second part is executed for every given input batch of possibly-dissimilar computation graphs. Figure 3 presents an overview of our method. In this section, we elaborate on the mechanics of each part separately.

A. Forward-Backward Kernel Specialization

Before initiating the training loop, our framework needs to construct the forward-backward GPU kernel. For a given training batch, this kernel will solely be responsible for carrying out forward propagation, backward propagation and parameter update all in one invocation. Since parameters of the model are going to be cached within the register file, it is necessary for the kernel to be *specialized* and compiled for the given parameters dynamically at run time in order to enable static register indexing. Our design assembles a string of characters containing the CUDA C++ source for the kernel and its required functions, and supplies it to the Nvidia Runtime Compiler (NVRTC) for generating kernel binary. As it is shown in Figure 3, the kernel source is constructed from two distinct pieces. One of these pieces is directly made by our *parameter routine call generator* from the set of model parameters while the other piece is independent of the model specification.

1) Weight Matrix Distribution: For every weight matrix in the list of model parameters, our solution has to first assign matrix elements to registers belonging to threads in CTAs. To avoid over-subscribing or under-subscribing CTAs with different numbers of consumed registers, we virtually partition the registers available to each CTA threads into multiple partitions [2], and distribute pieces of model's weight matrices over these partitions in a round-robin fashion. Figure 4 visualizes this using an example. This strategy goes on until all the weight matrices are assigned. Using this method, not only inter-CTA register cache utilization imbalance is minimized, but also NVRTC creates the most similar looking matrix pieces at every level which helps reducing overall kernel binary size and instruction cache misses.

Similar to Persistent RNN [6], we require each row of a weight matrix to be stored in the registers of and processed by one specific warp. For example, in Figure 4 row 8 with the length 128 in Matrix B is distributed across 128 registers of warp 0 in CTA 3. This allows a coalesced load of weight matrices from GPU DRAM into the registers [3] and to eliminate the need for synchronization with other warps inside or outside of the CTA when performing matrix-vector multiplication. We also distribute weight matrix pieces into these partitions multiple rows at a time. In Figure 4, for example, two consecutive rows of matrix B are given to each warp. This helps reducing the number of remote atomic stores when performing transposed-matrix-vector multiplication [4]. In addition, while the width of the CTA should be small to avoid excessive internal usage of registers by threads themselves, since a thread can address up to 255 4-bytes-long architected registers in the most recent GPU micro-architectures, there need to be at least 256 threads resident on the SM to allow

[2]This partitioning is the same across all the thread blocks.
[3]In DyNet weight matrices are stored in a row-major order by default.
[4]Our implementation performs the transposed-matrix-vector multiplication without transposing the matrix view in registers.

full utilization of its register file with the size of 256 KB. Thus we select CTA width to be 256 [5].

Since we are fusing the backward propagation and forward propagation into one kernel, to avoid recurring accumulation of weight matrix gradients into off-chip DRAM, we give register partitions to gradient matrices as well. In the same fashion as we did for matrix elements themselves, pieces of gradient matrices are given to register partitions using a round-robin scheduling depicted in Figure 4. Clearly, the initialization and operation routines that will be generated for a matrix and its gradient differ, which we discuss in detail in Section III-A2. We will also discuss trade-offs involved with caching gradients on chip and introduce an automated decision-making optimization for it in Section III-C2.

There are two hyper-parameters for selecting a suitable partition size. First is the number of CTAs per SM. Having more CTAs results in more kernel occupancy and exposes more parallelism by allowing more warps to be resident on an SM. However, on the other hand, having more threads on an SM means lower number of registers will be available for caching the parameters. In our design, we allow for up to two CTAs (each with width 256) per SM depending on the aggregated size of weight matrices that need to be cached, and automate the decision-making in the framework. Having more than two thread-blocks per SM is impractical as it heavily limits the size of partitions.

The second hyper-parameter is the partition size (*CTA width* multiplied by *thread registers per partition* in Figure 4). Because we enforce storing each row by one warp, if we show the maximum length of a row in all the model's weight matrices with row_{max}, partition size can be formulated as:

$$P_{size} = TBSize \times rpw \times \lceil \frac{row_{max}}{warpSize} \rceil \qquad (1)$$

where $TBSize$ is the thread-block size and is fixed at 256, $warpSize$ is the warp size and is 32, and rpw is the minimum number of rows a warp gets to process. We essentially transform the decision on P_{size} to a decision on rpw. The more rows assigned to a warp the more the reuse of the input vectors on operations such as matrix-vector multiplication and the more data locality when executing the transpose matrix-vector multiplication; nonetheless, at the same time, the larger the granularity of the assigned tasks to thread-blocks, which increases the chance of inter-CTA load imbalance.

Profile-Guided Load Granularity Determination. The advantage of above transformation is that rpw has a limited number of valid integer options that are more than zero. For instance, a model with $row_{max} = 1024$ and one CTA per SM can have a maximum rpw of six [6]. In our framework, we have enabled a profile-guided approach that compiles and stores multiple kernels each using one of the valid options for rpw. During the training, it starts with the kernel with $rpw = 1$ and measures the average computation time for multiple training batches. Then incrementally uses the kernels for bigger rpw's and performs the measurements again. This goes on until the framework observes performance degradation or it arrives at the kernel with the largest rpw. For the rest of the training, the framework uses the kernel that performed the best during the profile stage. Note that the profiling takes only a small portion of the training and its effect averages out across many more training inputs and epochs. A similar approach has been adopted by Tensor Comprehensions [20] where an evolutionary search auto-tunes the training and gradually converges to the best-performing compiled kernel.

[5]One can argue that a CTA with width 256 can be replaced with two CTAs with the width 128. However, this did not provide any benefits in our applications due to the majority of tensors having the length more than 128.

[6]In our calculations for the kernel, we conservatively set aside 31 registers per thread for interpretation routines and 32 additional registers for caching vectors during matrix operations. This leaves us with 192 registers per thread for caching weight matrices when there is only one CTA on the SM.

```
// Compile-time definitions such as RegCache dimensions.
1  enum{ . . };
// Matrix operations.
2  template< . . > __device__ void mat_vec_mul( . . ) { . . }
3  template< . . > __device__ void tmat_vec_mul( . . ) { . . }
4  template< . . > __device__ void outer_prod( . . ) { . . }
// Routing matrix operation via nested switch statements.
5  template< . . > __device__ void handle_matrix_op( . . ) {
6    switch( blockIdx.x ) { . .
7      switch( mat_op ) { . .
8        switch( partitionIdx ) {
9        . . } } } } // Specializing CTAs for matrix ops.
// Routing and definition for common operations.
10 __device__ void component_wise_unary( . . ) { . . }
11 __device__ void component_wise_binary( . . ) { . . }
12 __device__ void loss_func( . . ) { . . }
13 . . . .
// Forward-backward propagation kernel.
14 __global__ void forward_backward_kernel( . . ) {
15   float RegCache[ PARTS_PER_CTA ][ TH_REGS_PER_PARTITION ];
16   . . // Other initializations.
17   register_load_initialization_routines( . . );
18   . . // Fetching the script into shared mem.
     // Iterating over CTA's script instructions.
19   for( uint instIdx = 0; instIdx < totalLength; ) { . .
       // routing the execution to the requested function.
20     switch( opType ) { . . } }
21   gradient_register_store( . . ); }
```

Fig. 5. A summarized view of the kernel source structure. Highlighted sections are parts specialized at runtime based on the model's weight matrices and their distribution over registers of CTAs, while the rest is static. Arrows on the left hand side indicate the call hierarchy.

2) Routine Call Generation: Using the partition size selected in the previous step, the framework can embed the register partition dimensions in the source code and generate calls with appropriate template and function parameters to the functions dealing with weight matrix operations. Figure 5 gives an overview of the structure of the source given to the JIT compiler. For in-register matrix operations, we have created device-side templated C++ functions that get included in the source for JIT compilation (lines 2-4 in Figure 5). These operations are matrix-vector multiplication for the forward pass, transposed-matrix-vector multiplication, and outer product of two vectors for extracting matrix gradients in the backward pass. In these functions, arguments such as partition index, the number of matrix rows per warp, and the number of iterations of the warp over a row have to be passed as template arguments in order to allow compile-time-known register array indexing. In addition, there are matrix routines that are executed only at the beginning or the end of the kernel (lines 17 and 21 in Figure 5). These routines include parameter load from their

master copy in DRAM into the registers, in-register gradient matrix initialization to zero, and application of gradients onto the master copy of parameters.

There are also other static kernel pieces used in the kernel source without any changes across different model specifications. These include the code for typical operations in neural networks (lines 10-13 in Figure 5), such as forward and backward device functions for activations, as well as the routine for script interpretation (lines 18-20 in Figure 5). Script interpretation routine is the inner part of the kernel that loops over the script commands, interprets them, and executes them with the specifications supplied in the script. We will go into more details of the script interpretation and execution in Section III-B2.

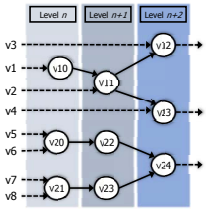
When all the pieces of the CUDA C++ kernel source are constructed, our framework assembles them within a string and passes them to the NVRTC for constructing the kernel binary. It is worth noting that our approach resembles both code specialization and interpreter specialization [21] for dynamically loaded code as follows.

B. GPU Script Generation and Execution

VPPS views each persistent CTA as a *virtual vector processor*. Each virtual processor needs a driving mechanism in order to execute operations according to the type and placement of nodes in the computation graph. This section discusses the procedure to produce this driving mechanism, i.e., the execution script, for one or a batch of computation graphs. Such a script contains exclusive instructions for each virtual processor and each processor is meant to utilize all its threads to carry out a given vector instruction. The script essentially guides persistent processors to read or write tensor contents from and into the global memory, execute the operations, or synchronize with each other if needed. This strategy is portable as it does not make any assumptions about the shape of the computation graph and therefore is well suited for dynamic neural net models.

1) Operation Scheduling and Script Generation: To generate the script from a given batch of computation graphs, our framework first sorts the nodes based on their maximum depth calculated from the leaf nodes—nodes with no incoming edges. This creates a correct total order of execution for nodes where parallelism between nodes within a level can be exploited due to their independence guaranteed through the sort. Figures 6(a) and 6(b) illustrate this using an example. This approach resembles depth-based batching [9], [17] for dynamic nets, however, unlike these two works, our method does not necessitate grouping similar operations together; all the nodes in a level, regardless of their types, are scheduled for a possibly concurrent execution. This creates a significant advantage for our approach from a task parallelism perspective even in small batch sizes.

After this sort, the framework traverses the computation graphs level by level starting from the leaf nodes (level zero). Upon visiting every node, depending on the operation associated with the node the framework encodes a complex



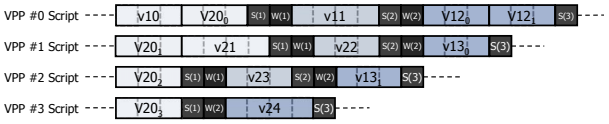
(a) Depth-based sorting the nodes.

```
Level n:   v10 = tanh( v1 )
           v20 = mat_vec_mul( v5, v6 )
           v21 = cwise_sum( v7, v8 )
Level n+1: v11 = cwise_mul( v2, v10 )
           v22 = sigmoid( v20 )
           v23 = tanh( v21 )
Level n+2: v12 = concat( v3, v11 )
           v13 = mat_vec_mul( v4, v11 )
           v24 = cwise_mul( v22, v23 )
```

(b) Extracting a correct total execution order for nodes in different levels.

```
Level n:
v10-> [ tanh.type : v10.size ] : [ v10.ptr ] : [ v1.ptr ]
v20-> for register partitions caching matrix parameter v5
        [ mat_vec_mul.type : partitionIdx ] : [ v20.ptr ] : [ v6.ptr ]
v21-> [ cwise_sum.type : v21.size ] : [ v21.ptr ] : [ v7.ptr ] : [ v8.ptr ]
Level n+1:
v11-> [ cwise_mul.type : v11.size ] : [ v11.ptr ] : [ v2.ptr ] : [ v10.ptr ]
v22-> [ sigmoid.type : v22.size ] : [ v22.ptr ] : [ v20.ptr ]
v23-> [ tanh.type : v23.size ] : [ v23.ptr ] : [ v21.ptr ]
Level n+2:
v12-> [ copy.type : v3.size ] : [ v12.ptr ] : [ v3.ptr ]
      [ copy.type : v11.size ] : [ v12.ptr + v3.size ] : [ v11.ptr ]
v13-> for register partitions caching matrix parameter v4
        [ mat_vec_mul.type : partitionIdx ] : [ v13.ptr ] : [ v11.ptr ]
v24-> [ cwise_mul.type : v24.size ] : [ v24.ptr ] : [ v22.ptr ] : [ v23.ptr ]
```

(c) Generating script's instructions for virtual CISC-like vector processors. The content within each square bracket pair is 4 bytes long.



(d) Distributing script instructions over Virtual Persistent Processors (VPPs). The total number of VPPs is assumed to be 4. It is assumed parameter `v4` is cached across all VPPs while parameter `v5` is cached over VPPs 1 and 2. `S` and `W` are signal and wait instructions respectively.

Fig. 6. An example (inspired by computation graphs in [5]) visualizing step-by-step operation scheduling and script generation in VPPS. This example schedules forward-propagation; back-prop scheduling is performed similarly.

instruction—which will eventually be decoded and executed by all the threads within the virtual CISC-like vector processor. Figure 6(c) shows this procedure. Each instruction starts with a 4-bytes-long preamble that encodes the type and input tensor size for the operation. Depending on the operation, generated instruction for a node may be up to 20-bytes-long, from which most of the bytes are typically used to address the node's input or output tensor content inside the global memory. For example, for a `tanh()` operation, the framework generates 12 bytes of instructions: 4 bytes for operation type and input tensor length, 4 bytes for the output tensor address, and 4 bytes for the input tensor address. Note that these addresses are, in fact, offsets to the base address for a globally-shared memory pool inside the DRAM [7] Using 4 bytes for tensor addresses lets us save on instruction length. Allocated memory pool size in our applications did not need to exceed 16 GB; naturally, applications that consume more memory—assuming they are trained with 32-bit floating point numbers—would necessitate using more bits to store the offsets.

Since a virtual processors may *produce* a tensor that another

[7]This is a reasonable assumption since DyNet, similar to neural net training frameworks such as TensorFlow [13] or Deep Speech 2 [22], uses a custom memory allocator to store tensors. These allocators typically request for a large portion of the GPU's DRAM upfront in order to circumvent the recurring overhead of CUDA runtime memory allocation and deallocation, which gives them a continuous virtual memory region to work with.

processor *consumes*, to enforce an order on the execution and data visibility between virtual processors, we utilize two special instructions named `signal` and `wait`. Each of these instructions is 4-bytes-long and specified by `S` and `W` in Figure 6(d). After generating the instructions for the nodes within a level l, each of the virtual processors that will participate in executing instructions at l are given a `signal` instruction with a particular barrier index. Before creating instructions for the the nodes in level $l+1$, virtual processors that will be executing instructions at $l+1$ are also given a `wait` instruction over that barrier. During the execution, these virtual processors will have to wait on that barrier until the required number of signals are arrived. This is to make sure that the dependencies for a node are all satisfied and visible to the executing virtual processors before initiating the operation. We essentially enforce a consumer-producer relationship among virtual processors involved in executing nodes in two consecutive levels. For barriers, threads inside the thread-block use `atomicAdd()` (alongside `__threadfence()` to ensure correctness [23]) on designated global memory locations.

After level by level traversal of the computation graphs in the batch, which collectively form a Directed Acyclic Graph, the framework traverses the levels in the reverse order to create the back-propagation instructions. Within each level during forward or backward traversal, in order to enable a fair load distribution across virtual processors and maximize parallelism, the framework keeps track of their accumulated assigned loads, and for every instruction dynamically targets the virtual processor with the minimum load. While we empirically set the load metric for most of the operations proportional to the aggregated length of tensors they read, we associate a relatively higher load for operations related to the cached matrices in order to better represent their computational intensity with respect to other operations.

2) Script-Guided Kernel Execution: For the generated scripts to be executed within the forward-backward kernel, they need to be copied to the device side. To maximize the copy throughput, we concatenate the scripts for all the virtual processors in a preallocated pinned host memory region and use a single copy command to initiate the transfer. Also, the set of scripts in the copied buffer is preceded by the prefix sum of the number of elements in each script in order to allow each virtual processor quickly index into its own set of instructions.

Upon invoking the kernel, virtual processors execute the script interpretation routine we had embedded in the kernel source. Threads inside the virtual processor collaborate and fetch the assigned script section into the SM's on-chip shared memory. Then, they start to loop over the script's instructions: sequentially decode each instruction and execute the associated operation. Figure 7 shows a fragment of the loop content. The decoder functions based on a `switch()` statement on the instruction type which routes the control flow to the appropriate operation. In some cases, the length of the script for a virtual processor might exceed its allocated shared memory size. This is handled by fetching, decoding, and executing consecutive pieces of its scripts in multiple rounds using another outer loop within the kernel.

```
uint instr = scriptBuff[ scriptIdx ];
uint opType = instr & 0x3F; // 6 right-most bits
uint tnsrSize = BFExtract( instr, 6, 12 ); // Next 12 bits
switch( opType )
case( COPY ) {
uint dstOffset = scriptBuff[ scriptIdx + 1 ];
uint srcOffset = scriptBuff[ scriptIdx + 2 ];
for( uint i = threadIdx.x; i < tnsrSize; i += CTA_WIDTH )
  globalTensorBasePtr[ dstOffset + i ] =
   globalTensorBasePtr[ srcOffset + i ];
scriptIdx += 3
} break;
case( IMMEDIATE_SCALAR_ADD ) {
uint dstOffset = scriptBuff[ scriptIdx + 1 ];
uint srcOffset = scriptBuff[ scriptIdx + 2 ];
float sclr = uint_as_float( scriptBuff[ scriptIdx + 3 ] );
for( uint i = threadIdx.x; i < tnsrSize; i += CTA_WIDTH )
  globalTensorBasePtr[ dstOffset + i ] =
   globalTensorBasePtr[ srcOffset + i ] + sclr;
scriptIdx += 4
} break;
. . .
```

Fig. 7. A fragment of the content of the virtual processor's script interpretation loop. Arrows on the sides indicate the execution flow for all the threads of the virtual processor. This fragment resides within the forward-backward propagation kernel and maps to line 20 in Figure 5.

CISC vs. RISC. Different operations for our kernel, if they are operating on tensors, have the DRAM memory addresses for source operands and the destination operand within them. If we were to virtualize CTAs as RISC processors instead of CISC, we would have to explicitly control the intermediate resources during the scheduling. For example, for a component-wise add, we would need to specify the on-chip memory and location the two input operands for the operation have to be fetched into. We would also need to specify where the result of the operation will be saved. Such a resource management introduces runtime overheads for the host *during* the training and negatively impacts the training throughput. On the contrary, a CISC abstraction for CTAs offloads resource management to NVCC and handles it at the kernel compile-time. Plus, the overall number of neural network operation types the kernel needs to support is limited, which does not provide an incentive for paying RISC abstraction cost.

C. Additional Optimizations

This section introduces optimizations and design decisions for additional performance enhancement.

1) Kernel Execution Asynchrony: Since assembling the execution script for a batch of inputs is independent of the results from the previous inputs, we allow it to happen concurrently with the GPU executing the script from the previous batch. In more details, while the GPU is executing the forward-backward kernel, the CPU moves on to the next batch, creates the graph from user-provided expressions and inputs, sorts the nodes level-by-level, and traverses the graph in the forward and backward directions to generate the script for virtual processors. And then it synchronizes with the device in order to be able to reuse the pinned host memory buffer for the host-to-device script transfer. This approach essentially enhances the training throughput by enabling concurrent execution of both CPU and GPU.

2) Gradient Accumulation Strategy: Caching gradients alongside weight matrices doubles the amount of required registers. If the GPU does not have sufficient registers, we avoid allocating them for gradient matrices and the kernel does not perform the outer-product operations—which constitute the gradient matrices. Instead, in a pre-allocated location in GPU's DRAM memory pool, we concatenate all the left hand side and all the right hand side tensors that participate in the outer-product for each weight matrix, and use dense matrix-matrix multiplication primitive in the CUBLAS library. This gives us the gradients for a matrix efficiently in one go. For each weight matrix, one such call must be made.

D. Design Automation

We implemented VPPS as a C++ library in DyNet [10]. This allows the user to include its headers and access its features with only a few modifications to an existing implementation that is written for DyNet's C++ front-end. The first function call to use VPPS is made prior to the training loop after the set of model parameters are defined by the user:

```
vpps::handle hndl( model );
```

This call JIT compiles the forward-backward kernel for our solution behind-the-scene using the parameters in the DyNet's model object sent to it. The second call is within the training loop after the computation graph is constructed (by DyNet's internal mechanism) for an input batch:

```
float staleLoss = hndl.fb( model, cg, lossExpr );
```

This tells the library to execute the forward-backward propagation kernel for the given computation graph (`cg`). We need to pass the model object again so that VPPS can query varying design specifications such as weight decay. The last argument of this call is a reference to the loss expression which will be the final node for our forward propagation. This call replaces dynet's calls for forward propagation, backward propagation, and parameter update.

As we mentioned in Section III-C1 we allow the GPU kernel execution to be asynchronous with respect to the host. Therefore, we designed VPPS such that what is returned by the second call is the calculated loss for the previous round, not the current one. To allow explicit synchronization with the GPU after a forward-backward propagation request, we provide a third function that waits for the GPU to finish the currently running kernel and returns its loss:

```
float latestLoss = hndl.sync_get_latest_loss();
```

It is expected that the third call is used occasionally, for example, when the user intends to ensure the training for an epoch is completely over. These three calls essentially abstract away the complexities of our design from the user.

Finally, our implementation uses the same method as in [9] to enable concurrent training of multiple computation graphs induced by multiple inputs. In this method, the loss nodes at the end of computation graphs, regardless of graph shapes, are accumulated using a summation operation in order to get the aggregated loss. This creates a super-graph for which the loss node is the aggregated loss. This aggregated loss can then be back-propagated to collect the gradients for nodes in the super-graph.

IV. Experimental Evaluations

We performed our evaluations on a system with Nvidia Titan V GPU (Volta architecture, CC 7.0, 80 SMs × 256 KB of register file) with reference clock rates, connected to the host, an Intel Xeon E5-1650 v2 running at 3.5 GHz, via PCIe 3.0 16x. The O.S. is CentOS Linux Release 7.4.1708 with GCC 4.9.3 and CUDA 9.1 installed.

We first present the experimental results for *Tree-structured LSTM Sentiment Analyzer* [5], [10] (*Tree-LSTM* for short) and analyze it in detail, and then discuss the performance of our method in other applications. Tree-LSTM is one of the most irregularly-shaped benchmark applications for which the structure of the neural network changes for every input based on sentence's parse tree. As its inputs, we used sentences and their associated trees from Stanford Sentiment Treebank [24]. Results for other benchmarks are presented in Section IV-E.

A. Performance Improvement

We compare the performance of our method on Tree-LSTM with the state-of-the-art solution: DyNet with on-the-fly batching [9] that is implemented in two variants. One variant groups the similar nodes in the constructed super-graph based on their maximum depth from a leaf node and is called depth-based batching (DyNet-DB). The other uses an active list to group similar *ready-to-be-executed* nodes and is named agenda-based batching (DyNet-AB). Figure 8 presents

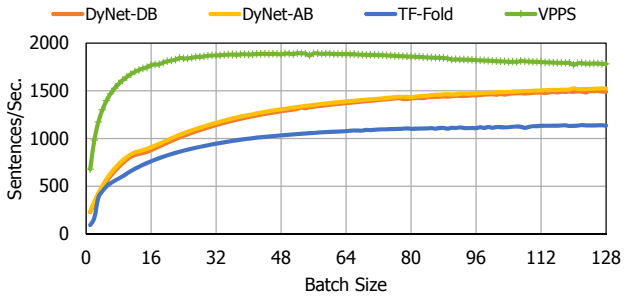


Fig. 8. VPPS training throughput for Tree-LSTM across different batch sizes compared with both variants of DyNet with on-the-fly batching [9] — depth-based batching (DyNet-DB) and agenda-based batching (DyNet-AB) — as well as TensorFlow Fold (TF-Fold). Hidden layer length and word embedding length are both fixed at 256 for all the frameworks.

the training throughput provided by VPPS and compares it with DyNet's for different batch sizes in the range of 1 to 128. As the Figure shows, VPPS performs better than DyNet specially in smaller batch sizes. VPPS's excellent performance with smaller mini-batches gives it a significant advantage since Machine Learning researchers typically favor smaller mini-batches due to better convergence properties [18], [19] and higher model update rates. On the contrary, DyNet requires very large batches to arrive at training rates close to VPPS's. This is because only large batches allow matrix-vector multiplications in DyNet to be converted into large matrix-matrix

multiplications, which incurs fewer overall memory loads. In addition, only large batches provide enough independent operations of the same type to merge in order to remedy SM underutilization. VPPS performance supremacy against the best-performing DyNet variant ranges from 1.16x (on batch size 128) to 2.92x (on batch size 2). When averaged across all the batch sizes, VPPS performs 1.48x faster than DyNet. Figure 12 also shows the training rates for TensorFlow Fold reference implementation of Tree-LSTM which are generally lower than both VPPS's (1.93x on average) and DyNet's (1.29x on average).

B. Off-chip Memory Load Analysis

Batch Size	1	2	4	8	16	32	64	128
VPPS	352.62	176.31	88.15	44.07	22.03	11.01	5.50	2.75
DyNet-AB	2.82k	2.18k	1.79k	1.48k	1.25k	1.04k	868	692

TABLE I
SIZE OF WEIGHTS LOADED (IN MEGABYTES) FOR TRAINING 128 INPUTS WITH DIFFERENT BATCH SIZES USING VPPS AND DYNET-AB.

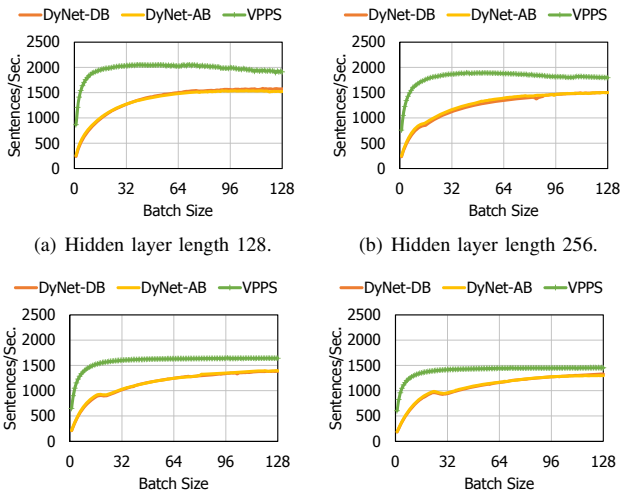
To further analyze the performance of VPPS against DyNet with on-the-fly batching, we measured the weight matrix loads from GPU memory incurred during the training of 128 inputs in VPPS and DyNet-AB, as shown in Table I. These results show that for all presented batch sizes, VPPS requires far fewer off-chip memory accesses. Also, as the batch size grows, we observe larger DRAM load reductions in DyNet-AB compared to VPPS, due to transforming more matrix-vector multiplications into matrix-matrix multiplications.

C. Analyzing the Sensitivity to the Size of Parameters

Figure 9 compares VPPS's training throughput with DyNet's for different hidden layer lengths. First, as we increase the hidden layer size, the throughput for both methods decrease. This is mainly because of added computational load to the training. Second, while VPPS's average training rate reduces by 8.5% when transitioning from hidden layer length 128 to 256, the training rate drop is 12.2% when using hidden layer length 384 instead of 256. This can be explained by examining the kernel occupancies. With hidden layer length 384, due to additional register pressure, VPPS produces kernels with occupancy 12.5% (one CTA per SM) instead of 25% (two CTAs per SM). We believe that this reduced parallelism exploitation opportunity contributes to the higher training rate decline. Third, with larger hidden layers and therefore larger weight matrices, we do not observe the slight performance decline with large batches anymore. This is because the GPU has larger computation loads and disallows CPU to act as the bottleneck for the training. Next section discusses this in more details. Fourth, even with large hidden layers where DyNet can create larger matrices, VPPS gives higher training throughput.

D. Execution Time Breakdown

To analyze the performance of our solution in different phases of execution during training, we measured the duration of CPU and GPU activities for different batch sizes. We plotted the results in Figure 10 where we normalized the durations



(a) Hidden layer length 128. (b) Hidden layer length 256.

(c) Hidden layer length 384. (d) Hidden layer length 512.

Fig. 9. VPPS training throughput on Tree-LSTM for different hidden layer lengths across different batch sizes against both variants of DyNet with on-the-fly batching [9]. Word embedding length is fixed at 128.

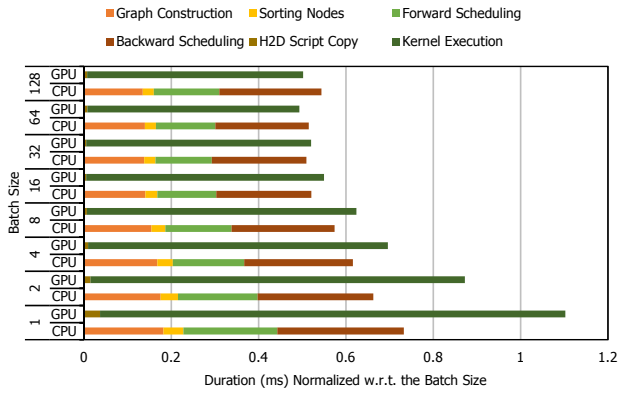
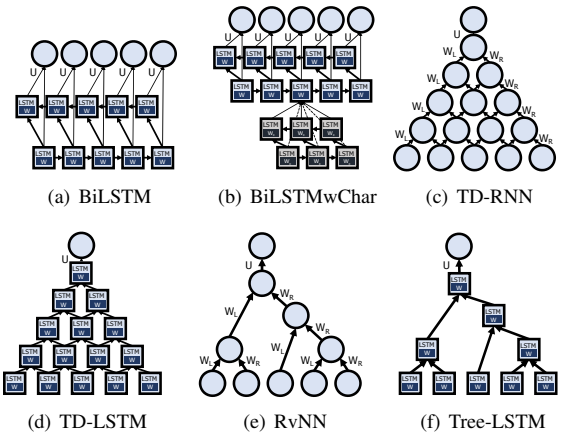


Fig. 10. Tree-LSTM Execution time breakdown in VPPS for different batch sizes. Word embedding length and hidden layer length for the LSTM are both 256. Since in our solution device operations are asynchronous with respect to the host, CPU and GPU execution durations are plotted side-by-side.

with respect to the batch size to retrieve per-input averages. Since in the actual VPPS execution, CPU and GPU work concurrently, we plotted CPU and GPU times side-by-side. It is clear from Figure 10 that in small batch sizes, GPU kernel execution on average takes longer than the preparations carried out by the host. This makes the GPU kernel execution the bottleneck for the performance. However, as the batch size gets larger, per-input averaged kernel duration shortens due to the exposure to more task parallelism. At the same time, we observe that CPU execution time, specifically the contribution of forward scheduling and backward scheduling, starts to slightly increase, which is largely due to bigger working spaces for data structures and higher cache misses. This makes the CPU the performance bottleneck at higher batch sizes and explains the small decline in VPPS's performance in Figures 8, 9(a) and 9(b).

E. Other Applications

In this section, we measure throughput boost provided by VPPS in five other dynamic neural network models, each of which exhibiting a different degree and form of architecture dynamicity.



(a) BiLSTM (b) BiLSTMwChar (c) TD-RNN

(d) TD-LSTM (e) RvNN (f) Tree-LSTM

Fig. 11. A simplified view of the unrolled architectures created by dynamic neural networks in our experiments. Note that these are only sampled views and different inputs can induce different shapes.

BiLSTM. *Bi-directional LSTM Named Entity Tagger* [10] based on [25] that uses a bi-directional LSTM to predict the tag of every word in the input sentence.

BiLSTMwChar. *Bi-directional LSTM Tagger w/ Optional Character Features* [10] similar to *BiLSTM* with a difference: for words with a frequency less than 5 in the corpus, another Bi-directional LSTM is run over the characters of the word to create its embedding.

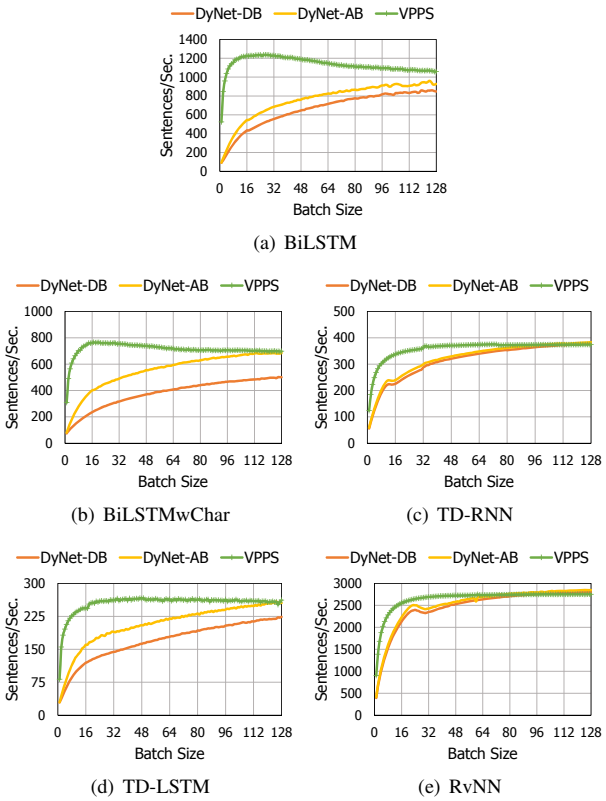
TD-RNN. A Time-Delay Neural Network [26] inspired by the work of Peddinti *et al.* [27] where adjacent embeddings are iteratively multiplied by recurrent left hand side and right hand side weights and added together to create new embeddings. Here we followed Socher *et al.* [24] proposition to reuse a single composition function. The outcome for an input sentence is passed through a multi-layer perception to predict the connotation of the sentence.

TD-LSTM. Similar to *TD-RNN* where the transformations with two Vanilla RNNs are replaced with LSTMs [8].

RvNN. A Recursive Neural Net [28] model similar to *TD-RNN* where a sparser binary tree is constructed using input sentence's parse tree. Inspired by [29] we have untied the weights for leaf nodes and internal nodes to separate transformation spaces.

Figure 11 gives a view of the unrolled network architectures these models create. While the shape of the computation graph in BiLSTM, TD-RNN, and TD-LSTM can vary due to different sentence lengths, in the rest of our applications the parse tree of the sentence (RvNN, similar to TreeLSTM) and the frequency of the words in the corpus (BiLSTMwChar) determine the shape of the computation graph as well.

Figure 12 plots the training throughput for above applications when using VPPS and DyNet. Similar to Tree-LSTM, all



(a) BiLSTM

(b) BiLSTMwChar (c) TD-RNN

(d) TD-LSTM (e) RvNN

Fig. 12. VPPS training throughput against both variants of DyNet with on-the-fly batching [9]. For RvNN and TD-RNN, hidden layer length and word embedding length are both fixed at 512 while for the rest of applications they are 256. MLP vector length for BiLSTM and BiLSTMwChar is 256. Character embedding length for BiLSTMwChar is 64. As inputs for our models, we used WikiNER English Corpus [30] for training BiLSTM and BiLSTMwChar, and Stanford Sentiment Treebank [24] for training the rest.

the benchmarks demonstrate the effectiveness of our approach compared to the state-of-the-art solution for the majority of the batch sizes. Specially in smaller batch sizes where DyNet fails to create large matrices and to utilize SM resources, our approach shows superior performance by eliminating the recurring cost of loading weight matrices. For BiLSTM at batch size 2 the throughput boost provided by VPPS is 6.08x compared to DyNet's best performing variant. Also, in models where the computation graph is comprised of limited types of operation nodes, i.e., TD-RNN and RvNN, it is easier for DyNet to batch the operations and arrive at VPPS's performance at smaller batches; while this is not the case for other applications.

F. JIT Compilation Overhead

	BiLSTM	BiLSTMwChar	TD-RNN	TD-LSTM	RvNN	Tree-LSTM
Prog. Compilation	28.66	28.27	73.85	11.43	74.61	11.10
Module Load	14.65	20.02	46.69	7.40	47.78	7.29

TABLE II
JIT COMPILATION DURATION (IN SECONDS) FOR THE FORWARD-BACKWARD PROPAGATION KERNEL IN VPPS.

Finally, Table II shows the JIT compilation duration for the forward-backward kernel for the benchmarks (with settings described in Sections IV-A and IV-E). Overall compilation

time is the aggregation of program compilation (CUDA C++ to PTX) and module load (PTX to binary). This overhead suggests that VPPS is more suited to be used at later stages of neural network development where specification of model parameters changes infrequently and the user can pay the JIT compilation price only once for a training session that may take hours or even days. Also, having a database for compiled kernels in a non-volatile memory such as disk or SSD is imaginable, although to the best of our knowledge serialization of kernel binaries produced by NVRTC is currently not supported; only intermediate PTX can be stored.

V. Related Work

In addition to the articles discussed throughput this paper, our method touches three distinct subjects.

a) Resource Virtualization: While VPPS virtualizes CTA's as persistent vector processors, other resource virtualization strategies have been proposed. vDNN [31] virtualizes available DRAM memory and utilizes host memory behind-the-scene in order to seamlessly give the illusion of a large global memory size for larger neural networks on the GPU. Jeon *et al.* [32] virtualize register file to break the curse of static physical register assignment in GPUs while RegMutex [33] utilizes a multi-stage allocation approach to address this issue. Zorua [34] virtualizes on-chip memories to balance the resource allocation. The main difference between these works and our work is that virtualization in VPPS gives a layer of abstraction that enables execution control in dynamic scenarios. Wireframe [35] virtualizes tasks by representing them as nodes in task graph and enforces a producer-consumer relationships between these tasks at the hardware. GPU Maestro [36] is another micro-architecture technique to enable dynamic resource management and partitioning by direct profiling in GPUs. Wu *et al.* [37] propose an SM-centric abstraction to allow flexible GPU task assignment. Adriaens *et al.* [38] propose spatial multi-tasking support on GPUs to allow multiple applications to use GPU resources simultaneously. SMK [39] and Warped-Slicer [40] take this idea one step further and share the resources within one SM between different applications. Finally, BiNoCHS [41] virtualizes the GPU interconnect by decoupling SMs and on-chip network routers to maximize communication throughput based on the on-chip traffic pattern.

b) Fusion, Thread Persistency and In-Register Caching: As mentioned, Persistent RNN [6] resembles kernel fusion [42]–[44] where model parameters are explicitly cached. VPPS extends this idea to dynamic scenarios and enables the benefits of kernel fusion for dynamic nets. XLA [45] is an extension for TensorFlow that recognizes and JIT compiles groups of inter-connected operations in static neural networks in order to carry them out with a single kernel. Tzeng *et al.* [46] utilized Persistent Threads (PT) [7] to eliminate load imbalance in irregular GPU workloads and Wald [47] used PT for active thread compaction. Chen and Shen proposed *Free Launch* [48] as a replacement for dynamic parallelism on GPUs. Free Launch deploys PT to reuse parent threads for nested tasks. VPPS, on the other hand, employs PT to maintain active state of registers and inhibit their invalidation. These use cases of PT have been broadly identified and discussed by Gupta *et al.* [49]. The collective register file size growth in recent GPU micro-architecture has led researchers to utilize it for caching and data reuse purposes in various applications such as binary field multiplication [50], similarity search [51], and segmented sort [52]. CCC [53] exploits shared memory as another software-controlled on-chip resource to cache tasks.

c) Hardware/Software Specialization: Although VPPS runs on existing hardware and does not require micro-architectural changes to the GPU, studying hardware specializations is insightful for an efficient design. TPU [15] utilizes 28 MiB of on-chip memory, which indicates the importance of data locality in Deep Learning applications. Eyeriss [54], DaDianNao [55] and ShiDianNao [56] emphasize on using on-chip storage to reduce DNN memory footprint. TETRIS [57] and Boroumand *et al.* [58] suggest accelerating the neural network inference phase with 3D memory. Chakradhar *et al.* [59], CATERPILLAR [60] and MAERI [61] suggest DNN accelerators with configurable building blocks that can be specialized for the given network architecture. EIE [62], SCNN [63], and Cambricon-X [64] propose hardware accelerators specialized for sparse and compress neural nets. Cnvlutin [65] discusses eliminating ineffectual operations in neural networks. Scalpel [66] specializes pruning of the network with respect to the underlying hardware architecture and DeftNN [67] offers synapse vector elimination and near-compute data fission. Proteus [68], on the other hand, focuses on numerical precision variability to augment DaDianNao. Reagen *et al.* [69] and Park *et al.* [70] propose full computing stacks to map the given Machine Learning model onto the specialized accelerator. On the software side, taco [71] specializes the CUDA kernel for the given operations while the main focus is efficient matching of sparse and dense tensors. Finally, CudaDMA [72] and Singe [73] specialize warps in a kernel for different tasks.

VI. Conclusion

This work presented Virtual Persistent Processor Specialization (VPPS) to allow recurring parameters of a dynamic neural network to persist on-chip throughout training an input batch. We demonstrated that our approach, unlike existing counterparts, does not need a very large batch size for efficient utilization of GPU resources. In addition, all described steps in the paper are automated, meaning Machine Learning researchers can naturally express their computation for a dynamic net without having to know the internal mechanics of the design. VPPS enables up to 6x training throughput boost over the state-of-the-art method.

Acknowledgment

This work is supported by National Science Foundation collaborative Grants No. 1629564 and 1629459. We gratefully acknowledge the support of NVIDIA Corporation with the donation of a Titan Xp GPU used for this research during the development phase.

REFERENCES

[1] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1*, ser. NIPS'12, 2012.

[2] A. Graves, A. r. Mohamed, and G. Hinton, "Speech recognition with deep recurrent neural networks," in *2013 IEEE International Conference on Acoustics, Speech and Signal Processing*, 2013.

[3] R. Collobert and J. Weston, "A unified architecture for natural language processing: Deep neural networks with multitask learning," in *Proceedings of the 25th International Conference on Machine Learning*, ser. ICML '08, 2008.

[4] F. A. Gers, J. Schmidhuber, and F. Cummins, "Learning to forget: continual prediction with lstm," *IET Conference Proceedings*, 1999.

[5] K. S. Tai, R. Socher, and C. D. Manning, "Improved semantic representations from tree-structured long short-term memory networks," *arXiv preprint arXiv:1503.00075*, 2015.

[6] G. Diamos, S. Sengupta, B. Catanzaro, M. Chrzanowski, A. Coates, E. Elsen, J. Engel, A. Hannun, and S. Satheesh, "Persistent rnns: Stashing recurrent weights on-chip," in *Proceedings of The 33rd International Conference on Machine Learning*, ser. Proceedings of Machine Learning Research, vol. 48. PMLR, 2016.

[7] T. Aila and S. Laine, "Understanding the efficiency of ray traversal on gpus," in *Proceedings of the Conference on High Performance Graphics 2009*, ser. HPG '09. ACM, 2009.

[8] J. Chung, C. Gulcehre, K. Cho, and Y. Bengio, "Gated feedback recurrent neural networks," in *Proceedings of the 32Nd International Conference on International Conference on Machine Learning - Volume 37*, ser. ICML'15. JMLR.org, 2015.

[9] G. Neubig, Y. Goldberg, and C. Dyer, "On-the-fly operation batching in dynamic computation graphs," in *Conference on Neural Information Processing Systems (NIPS)*, December 2017.

[10] G. Neubig, C. Dyer, Y. Goldberg, A. Matthews, W. Ammar, A. Anastasopoulos, M. Ballesteros, D. Chiang, D. Clothiaux, T. Cohn *et al.*, "Dynet: The dynamic neural network toolkit," *arXiv preprint arXiv:1701.03980*, 2017.

[11] S. Tokui, K. Oono, S. Hido, and J. Clayton, "Chainer: a next-generation open source framework for deep learning," in *Proceedings of workshop on machine learning systems (LearningSys) in the twenty-ninth annual conference on neural information processing systems (NIPS)*, vol. 5, 2015.

[12] A. Paszke and S. Chintala, "Pytorch," 2017.

[13] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean *et al.*, "Tensorflow: A system for large-scale machine learning," in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, 2016.

[14] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell, "Caffe: Convolutional architecture for fast feature embedding," in *Proceedings of the 22Nd ACM International Conference on Multimedia*, ser. MM '14, 2014.

[15] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa *et al.*, "In-datacenter performance analysis of a tensor processing unit," in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ser. ISCA '17. ACM, 2017.

[16] A. Arunkumar, E. Bolotin, B. Cho, U. Milic, E. Ebrahimi, O. Villa, A. Jaleel, C.-J. Wu, and D. Nellans, "Mcm-gpu: Multi-chip-module gpus for continued performance scalability," in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ser. ISCA '17. ACM, 2017.

[17] M. Looks, M. Herreshoff, D. Hutchins, and P. Norvig, "Deep learning with dynamic computation graphs," *arXiv preprint arXiv:1702.02181*, 2017.

[18] D. Masters and C. Luschi, "Revisiting small batch training for deep neural networks," *CoRR*, 2018. [Online]. Available: http://arxiv.org/abs/1804.07612

[19] N. S. Keskar, D. Mudigere, J. Nocedal, M. Smelyanskiy, and P. T. P. Tang, "On large-batch training for deep learning: Generalization gap and sharp minima," *CoRR*, 2016. [Online]. Available: http://arxiv.org/abs/1609.04836

[20] N. Vasilache, O. Zinenko, T. Theodoridis, P. Goyal, Z. DeVito, W. S. Moses, S. Verdoolaege, A. Adams, and A. Cohen, "Tensor comprehensions: Framework-agnostic high-performance machine learning abstractions," *arXiv preprint arXiv:1802.04730*, 2018.

[21] S. Thibault, C. Consel, J. L. Lawall, R. Marlet, and G. Muller, "Static and dynamic program compilation by interpreter specialization," *Higher-Order and Symbolic Computation*, vol. 13, 2000.

[22] D. Amodei, S. Ananthanarayanan, R. Anubhai, J. Bai, E. Battenberg, C. Case *et al.*, "Deep speech 2: End-to-end speech recognition in english and mandarin," in *Proceedings of the 33rd International Conference on International Conference on Machine Learning - Volume 48*, ser. ICML'16. JMLR.org, 2016.

[23] J. Alglave, M. Batty, A. F. Donaldson, G. Gopalakrishnan, J. Ketema, D. Poetzl, T. Sorensen, and J. Wickerson, "Gpu concurrency: Weak behaviours and programming assumptions," in *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '15. ACM, 2015.

[24] R. Socher, A. Perelygin, J. Wu, J. Chuang, C. D. Manning, A. Ng, and C. Potts, "Recursive deep models for semantic compositionality over a sentiment treebank," in *Proceedings of the 2013 conference on empirical methods in natural language processing*, 2013.

[25] Z. Huang, W. Xu, and K. Yu, "Bidirectional lstm-crf models for sequence tagging," *CoRR*, vol. abs/1508.01991, 2015.

[26] A. Waibel, T. Hanazawa, G. Hinton, K. Shikano, and K. J. Lang, "Readings in speech recognition," A. Waibel and K.-F. Lee, Eds. Morgan Kaufmann Publishers Inc., 1990, ch. Phoneme Recognition Using Time-delay Neural Networks, pp. 393–404.

[27] V. Peddinti, D. Povey, and S. Khudanpur, "A time delay neural network architecture for efficient modeling of long temporal contexts," in *INTERSPEECH*, 2015.

[28] R. Socher, C. C.-Y. Lin, A. Y. Ng, and C. D. Manning, "Parsing natural scenes and natural language with recursive neural networks," in *Proceedings of the 28th International Conference on International Conference on Machine Learning*, ser. ICML'11. Omnipress, 2011.

[29] O. İrsoy and C. Cardie, "Deep recursive neural networks for compositionality in language," in *Proceedings of the 27th International Conference on Neural Information Processing Systems - Volume 2*, ser. NIPS'14. MIT Press, 2014.

[30] J. Nothman, N. Ringland, W. Radford, T. Murphy, and J. R. Curran, "Learning multilingual named entity recognition from wikipedia," *Artif. Intell.*, vol. 194, 2013.

[31] M. Rhu, N. Gimelshein, J. Clemons, A. Zulfiqar, and S. W. Keckler, "vdnn: Virtualized deep neural networks for scalable, memory-efficient neural network design," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2016.

[32] H. Jeon, G. S. Ravi, N. S. Kim, and M. Annavaram, "Gpu register file virtualization," in *Proceedings of the 48th International Symposium on Microarchitecture*, ser. MICRO-48. ACM, 2015.

[33] F. Khorasani, H. A. Esfeden, A. Farmahini-Farahani, N. Jayasena, and V. Sarkar, "Regmutex: Inter-warp gpu register time-sharing," in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, 2018.

[34] N. Vijaykumar, K. Hsieh, G. Pekhimenko, S. Khan, A. Shrestha, S. Ghose, A. Jog, P. B. Gibbons, and O. Mutlu, "Zorua: A holistic approach to resource virtualization in gpus," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2016.

[35] A. Abdolrashidi, D. Tripathy, M. E. Belviranli, L. N. Bhuyan, and D. Wong, "Wireframe: Supporting data-dependent parallelism through dependency graph execution in gpus," in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-50 '17. ACM, 2017.

[36] J. J. K. Park, Y. Park, and S. Mahlke, "Dynamic resource management for efficient utilization of multitasking gpus," in *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '17, 2017.

[37] B. Wu, G. Chen, D. Li, X. Shen, and J. Vetter, "Enabling and exploiting flexible task assignment on gpu through sm-centric program transformations," in *Proceedings of the 29th ACM on International Conference on Supercomputing*, ser. ICS '15, 2015.

[38] J. T. Adriaens, K. Compton, N. S. Kim, and M. J. Schulte, "The case for gpgpu spatial multitasking," in *Proceedings of the 2012 IEEE 18th International Symposium on High-Performance Computer Architecture*, ser. HPCA '12, 2012.

[39] Z. Wang, J. Yang, R. Melhem, B. Childers, Y. Zhang, and M. Guo, "Simultaneous multikernel gpu: Multi-tasking throughput processors via fine-grained sharing," in *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2016.

[40] Q. Xu, H. Jeon, K. Kim, W. W. Ro, and M. Annavaram, "Warped-slicer: Efficient intra-sm slicing through dynamic resource partitioning for gpu multiprogramming," in *Proceedings of the 43rd International Symposium on Computer Architecture*, ser. ISCA '16, 2016.

[41] A. Mirhosseini, M. Sadrosadati, B. Soltani, H. Sarbazi-Azad, and T. F. Wenisch, "Binochs: Bimodal network-on-chip for cpu-gpu heterogeneous systems," in *Proceedings of the Eleventh IEEE/ACM International Symposium on Networks-on-Chip*, ser. NOCS '17, 2017.

[42] H. Wu, G. Diamos, S. Cadambi, and S. Yalamanchili, "Kernel weaver: Automatically fusing database primitives for efficient gpu computation," in *2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-45, 2012.

[43] A. Ashari, S. Tatikonda, M. Boehm, B. Reinwald, K. Campbell, J. Keenleyside, and P. Sadayappan, "On optimizing machine learning workloads via kernel fusion," in *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP 2015, 2015.

[44] J. Appleyard, T. Kocisky, and P. Blunsom, "Optimizing performance of recurrent neural networks on gpus," *arXiv preprint arXiv:1604.01946*, 2016.

[45] C. Leary and T. Wang, "Xla: Tensorflow, compiled," *TensorFlow Dev Summit*, 2017.

[46] S. Tzeng, A. Patney, and J. D. Owens, "Task management for irregular-parallel workloads on the gpu," in *Proceedings of the Conference on High Performance Graphics*, ser. HPG '10, 2010.

[47] I. Wald, "Active thread compaction for gpu path tracing," in *Proceedings of the ACM SIGGRAPH Symposium on High Performance Graphics*, ser. HPG '11. ACM, 2011.

[48] G. Chen and X. Shen, "Free launch: Optimizing gpu dynamic kernel launches through thread reuse," in *2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2015.

[49] K. Gupta, J. A. Stuart, and J. D. Owens, "A study of persistent threads style gpu programming for gpgpu workloads," in *2012 Innovative Parallel Computing (InPar)*, 2012.

[50] E. Ben-Sasson, M. Hamilis, M. Silberstein, and E. Tromer, "Fast multiplication in binary fields on gpus via register cache," in *Proceedings of the 2016 International Conference on Supercomputing*, ser. ICS '16, 2016.

[51] J. Johnson, M. Douze, and H. Jégou, "Billion-scale similarity search with gpus," *arXiv preprint arXiv:1702.08734*, 2017.

[52] K. Hou, W. Liu, H. Wang, and W.-c. Feng, "Fast segmented sort on gpus," in *Proceedings of the International Conference on Supercomputing*, ser. ICS '17. ACM, 2017.

[53] F. Khorasani, R. Gupta, and L. N. Bhuyan, "Efficient warp execution in presence of divergence with collaborative context collection," in *Proceedings of the 48th International Symposium on Microarchitecture*, ser. MICRO-48. ACM, 2015.

[54] Y. H. Chen, J. Emer, and V. Sze, "Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks," in *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, 2016.

[55] Y. Chen, T. Luo, S. Liu, S. Zhang, L. He, J. Wang, L. Li, T. Chen, Z. Xu, N. Sun, and O. Temam, "Dadiannao: A machine-learning supercomputer," in *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-47. IEEE Computer Society, 2014.

[56] Z. Du, R. Fasthuber, T. Chen, P. Ienne, L. Li, T. Luo, X. Feng, Y. Chen, and O. Temam, "Shidiannao: Shifting vision processing closer to the sensor," in *Proceedings of the 42Nd Annual International Symposium on Computer Architecture*, ser. ISCA '15, 2015.

[57] M. Gao, J. Pu, X. Yang, M. Horowitz, and C. Kozyrakis, "Tetris: Scalable and efficient neural network acceleration with 3d memory," in *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '17, 2017.

[58] A. Boroumand, S. Ghose, Y. Kim, R. Ausavarungnirun, E. Shiu, R. Thakur, D. Kim, A. Kuusela, A. Knies, P. Ranganathan, and O. Mutlu, "Google workloads for consumer devices: Mitigating data movement bottlenecks," in *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '18. ACM, 2018.

[59] S. Chakradhar, M. Sankaradas, V. Jakkula, and S. Cadambi, "A dynamically configurable coprocessor for convolutional neural networks," in *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ser. ISCA '10, 2010.

[60] Y. Li and A. Pedram, "Caterpillar: Coarse grain reconfigurable architecture for accelerating the training of deep neural networks," in *2017 IEEE 28th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, 2017.

[61] H. Kwon, A. Samajdar, and T. Krishna, "Maeri: Enabling flexible dataflow mapping over dnn accelerators via reconfigurable interconnects," in *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '18, 2018.

[62] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. A. Horowitz, and W. J. Dally, "Eie: Efficient inference engine on compressed deep neural network," in *Proceedings of the 43rd International Symposium on Computer Architecture*, ser. ISCA '16, 2016.

[63] A. Parashar, M. Rhu, A. Mukkara, A. Puglielli, R. Venkatesan, B. Khailany, J. Emer, S. W. Keckler, and W. J. Dally, "Scnn: An accelerator for compressed-sparse convolutional neural networks," in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ser. ISCA '17, 2017.

[64] S. Zhang, Z. Du, L. Zhang, H. Lan, S. Liu, L. Li, Q. Guo, T. Chen, and Y. Chen, "Cambricon-x: An accelerator for sparse neural networks," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2016.

[65] J. Albericio, P. Judd, T. Hetherington, T. Aamodt, N. E. Jerger, and A. Moshovos, "Cnvlutin: Ineffectual-neuron-free deep neural network computing," in *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, 2016.

[66] J. Yu, A. Lukefahr, D. Palframan, G. Dasika, R. Das, and S. Mahlke, "Scalpel: Customizing dnn pruning to the underlying hardware parallelism," in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ser. ISCA '17, 2017.

[67] P. Hill, A. Jain, M. Hill, B. Zamirai, C.-H. Hsu, M. A. Laurenzano, S. Mahlke, L. Tang, and J. Mars, "Deftnn: Addressing bottlenecks for dnn execution on gpus via synapse vector elimination and near-compute data fission," in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-50 '17, 2017.

[68] P. Judd, J. Albericio, T. Hetherington, T. M. Aamodt, N. E. Jerger, and A. Moshovos, "Proteus: Exploiting numerical precision variability in deep neural networks," in *Proceedings of the 2016 International Conference on Supercomputing*, ser. ICS '16, 2016.

[69] B. Reagen, P. Whatmough, R. Adolf, S. Rama, H. Lee, S. K. Lee, J. M. Hernndez-Lobato, G. Y. Wei, and D. Brooks, "Minerva: Enabling low-power, highly-accurate deep neural network accelerators," in *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, 2016.

[70] J. Park, H. Sharma, D. Mahajan, J. K. Kim, P. Olds, and H. Esmaeilzadeh, "Scale-out acceleration for machine learning," in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-50 '17, 2017.

[71] F. Kjolstad, S. Kamil, S. Chou, D. Lugato, and S. Amarasinghe, "The tensor algebra compiler," *Proc. ACM Program. Lang.*, 2017.

[72] M. Bauer, H. Cook, and B. Khailany, "Cudadma: Optimizing gpu memory bandwidth via warp specialization," in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '11. ACM, 2011.

[73] M. Bauer, S. Treichler, and A. Aiken, "Singe: Leveraging warp specialization for high performance on gpus," in *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '14. ACM, 2014.