Accelerating Video Captioning on Heterogeneous System Architectures

HORNG-RUEY HUANG, DING-YONG HONG, and JAN-JAN WU, Academia Sinica
KUNG-FU CHEN, PANGFENG LIU, and WEI-CHUNG HSU, National Taiwan University

Video captioning is a core technology to many important applications, such as AI-assisted medical diagnosis, video question answering, storytelling through videos, and lip-reading. Video captioning employs a hybrid CNN + RNN model. Accelerating such a hybrid model on a heterogeneous system is challenging for two reasons. First, CNN and RNN exhibit very different computing behaviors, making the mapping between computation and heterogeneous devices difficult. Second, data dependency exists between the CNN and RNN within a video frame and between adjacent RNNs across video frames. These data dependencies prohibit the full parallelization of the hybrid model. The issues also include the utilization of accelerator resources, which is critical to maximizing the performance. In this work, we propose a fine-grained scheduling scheme for mapping computation and devices within a video frame, and a pipeline scheduling scheme for exploiting maximum parallelism between the execution of the video frames. In addition, we propose two capacity-guided scheduling methods. On the server, the concurrent kernel execution mechanism is exploited for improving GPU utilization. On the edge platform, we rearrange CNN computation among the CPU and EdgeTPUs guided by the EdgeTPU’s SRAM capacity so that balanced computation is achieved and off-chip memory overhead is minimized. Experimental results show that our scheduling scheme improves video captioning performance by up to 3.24× with CPU + GPU collaboration over the GPU-only execution. On an edge platform with an ARM CPU and two EdgeTPUs, our CPU + EdgeTPU scheduling exhibits outstanding performance, which achieves up to 54.9× speedup compared to using ARM CPU only and can perform video captioning of 59 frames per second.

CCS Concepts: • General and reference → Design; Performance; • Computer systems organization → Heterogeneous (hybrid) systems; • Computing methodologies → Parallel algorithms;

Additional Key Words and Phrases: Video captioning, heterogeneous system architectures, model scheduling, dynamic programming, pipelining

Extension of a conference paper: This article is an extension of a paper presented at IPDPS 2021 [19]. The additional material includes a new GPU-capacity-guided scheduling to increase GPU utilization using concurrent kernel execution, a new EdgeTPU-capacity-guided scheduling to maximize SRAM utilization, and new results on an edge device with EdgeTPU. This work was supported in part by the Ministry of Science and Technology, Taiwan, under grant numbers MOST-110-2221-E-001-007-MY2, and MOST-110-2221-E-001-004.

Authors’ addresses: H.-R. Huang, D.-Y. Hong, and J.-J. Wu, Institute of Information Science, Academia Sinica, 128 Academia Road, Section 2, Nankang, Taipei 115, Taiwan; emails: {r06922090, dyhong, wuj}@iis.sinica.edu.tw; K.-F. Chen, Department of Computer Science and Information Engineering, National Taiwan University, 1, Section 4, Roosevelt Road, Taipei 10617, Taiwan; email: b07902112@ntu.edu.tw; P. Liu and W.-C. Hsu, Department of Computer Science and Information Engineering, National Taiwan University, 1, Section 4, Roosevelt Road, Taipei 10617, Taiwan; emails: {pangfeng, hsuwe}@csie.ntu.edu.tw.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2022 Association for Computing Machinery.
1544-3566/2022/05-ART38 $15.00
https://doi.org/10.1145/3527609

ACM Transactions on Architecture and Code Optimization, Vol. 19, No. 3, Article 38. Publication date: May 2022.
1 INTRODUCTION

Deep neural networks (DNNs) have revolutionized machine learning in recent years. The emergence of deep learning is widely attributed to the availability of massive datasets and advances in computer hardware. Based on the underlying topology, DNNs are generally categorized into convolutional neural networks (CNNs), recurrent neural networks (RNNs), graph neural networks (GNNs), and autoencoders. They are specifically designed for different domain problems and have shown great success in many applications, such as image classification [13, 16, 18], object detection [30, 41, 43], natural language processing [9, 14, 55], and recommender systems [25, 31, 63].

With the success of the preceding network models, there is an increasing trend of combining multiple network models to increase deep learning capabilities—the hybrid model. For example, CNN and RNN have been combined to conduct video captioning [52, 53, 60, 64, 65], video question answering [22], automatic medical report generation [6], stock trading analysis [29], movie reviews analysis [42], and pollutant prediction [39].

Video captioning [52] uses a hybrid model to exploit computer vision and natural language understanding, then to generate natural language caption. First, it uses a CNN to extract information of objects, scenes, and actions from video frames as visual features. Then it uses an RNN to generate caption using the visual features recognized by the CNN. Figure 1(a) illustrates the execution flow.

There are two types of data dependencies in video captioning. First, an RNN uses the output of the CNN from the same iteration as its input. Second, an RNN also uses the output of the RNN from the previous iteration as its input. These dependencies are shown as directed edges in Figure 1(a), and they make parallel execution of video captioning a challenging issue.

It is crucial to use all types of resources in a heterogeneous environment to improve the efficiency of video captioning. A naive video captioning approach runs both CNN and RNN on a GPU without using any CPU [34, 47, 60]. However, a previous study [66] showed that GPUs may perform much worse than CPUs for RNNs using small batch sizes (e.g., web search and advertisement), due to lack of data reuse and excessive data transfer. Running different models on the computing devices they prefer can significantly reduce the video captioning time. Figure 1(b) compares the performance of CNN (MobileNet [16]) and RNN (LSTM [14]) running on a CPU and a GPU. The result shows that MobileNet runs more efficiently on a GPU than on a CPU. On the contrary, LSTM runs faster on a CPU than on a GPU due to the small batch size used in practice.

Today, heterogeneous system architectures (HSAs) are widely available. Machines from high-end servers to low-end edge devices are often equipped with both CPU and GPU or AI accelerators. Such a heterogeneous environment provides ample opportunity to improve the inference time of network models if the computation is carefully scheduled.

A good scheduling of resources on heterogeneous systems should address four issues. First, since the system is heterogeneous, we need to assign the network models to appropriate devices to reduce the computation cost. Second, we should choose the scheduling granularity carefully. The computation behaves quite differently if we schedule one model at a time (coarse-grained) or one operation at a time (fine-grained). Third, we need to carefully schedule the tasks so that resource contention is minimized. Two cases are considered: the computing device is exclusive or
Fig. 1. Video captioning. (a) Execution flow of the hybrid CNN + RNN model. (b) Performance comparison of CNN (MobileNet) and RNN (LSTM) on the Intel Core i7 CPU and Nvidia 1080 Ti GPU.

can be used by the tasks at the same time. Finally, we should reduce the communication overhead when two related tasks are mapped to different devices.

We categorize our video captioning optimization techniques for the preceding issues into intra-frame and inter-frame optimizations. The intra-frame optimization considers the data dependency between the CNN and RNN at the same iteration for both coarse-grained and fine-grained scheduling. We first build a cost model to estimate the computation cost of each operation (e.g., convolution, LSTM) and the communication cost among devices. Then, we derive an efficient dynamic programming to solve the scheduling problem using the predicted execution time from the cost model. The inter-frame optimization is difficult due to data dependency between RNNs of consecutive iterations. We use a pipeline scheduling algorithm to overlap the execution of the CNN and RNN of consecutive frames, so as to reduce the overall time. The preceding results are published in IPDPS 2021 [19]. In this article, we further improve the inter-frame optimization by two new capacity-guided pipeline scheduling algorithms, as introduced in the following.

The pipeline scheduling algorithm proposed maps the execution of CNNs to GPU exclusively. In other words, the CNNs cannot use the GPU at the same time. We observe that, however, the GPU is under-utilized when processing the CNN. To overcome this issue, we relax the constraint of resource exclusion and devise a GPU-capacity-guided pipelining method. Since there is no data dependency among the CNNs across video frames, the CNNs can be executed at the same time to exploit the unused GPU resources. Thus, we split the execution of the CNN model into multiple stages and schedule the CNNs of consecutive frames onto the GPU in a pipelining manner. As a result, the GPU utilization is improved and video captioning time is reduced.

We also evaluate video captioning on an edge device with an ARM CPU and EdgeTPU accelerators. Typically, video captioning suffers long execution time on edge devices, as it is often processed by the embedded CPU only. Sending videos to the server for captioning is a possible solution to speed up the computation. However, it would be unacceptable to send all data (e.g., live video from a surveillance camera) to the server due to excessive data transfer. Therefore, either the resolution or the frame rate is reduced before sending the video, thus trading the captioning accuracy off.

By exploiting the CPU + EdgeTPU system architecture, our scheduling method enables efficient video captioning for edge devices without the help of remote servers. Our solution has the following advantages. It retains accuracy. Moreover, it minimizes privacy concerns and network traffic. Finally, it can have less energy consumption than migrating the computation to the server.
Running video captioning with AI accelerators faces additional challenges compared to CPU + GPU scheduling. First, AI accelerators often support a limited set of operations—for example, EdgeTPU supports matrix multiplication and convolution, but no embedding operation. The unsupported operations must be executed by another processor, thus additional scheduling constraint. Second, the on-chip SRAM (i.e., parameter cache) must be effectively utilized to enable good performance. We propose an EdgeTPU-capacity-guided pipelining method to improve SRAM utilization by rearranging the CNN computation among the CPU and AI accelerators. As a result, a balanced computation is achieved and off-chip memory overhead is reduced.

The contributions of this article are as follows:

• An intra-frame scheduling algorithm for mapping operations to devices for both coarse-grained and fine-grained scheduling.
• A pipeline scheduling algorithm for exploiting inter-frame parallelism.
• A GPU-capacity-guided pipelining scheme for maximizing GPU utilization.
• An EdgeTPU-capacity-guided pipelining scheme for maximizing SRAM utilization.
• On the server platform, the GPU/CPU pipeline scheduling achieves up to 3.24× faster video captioning than using GPU only. The performance is further improved by up to 7% with the GPU-capacity-guided pipeline scheduling.
• On the edge platform, the EdgeTPU/CPU pipeline scheduling achieves up to 54.9× speedup over ARM CPU only. The DenseNet/LSTM-based model can process video captioning of 59 frames per second. To the best of our knowledge, this is the first effort that achieves real-time video captioning performance on the edge device for such a large model.

The rest of this article is organized as follows. Section 2 describes related work. Section 3, Section 4, and Section 5 formally describe the computation-resource mapping problem on heterogeneous systems and present our scheduling algorithms. Section 6 and Section 7 respectively evaluate our scheduling methods on the server and edge platforms. Section 8 concludes the article.

2 BACKGROUND AND RELATED WORK

2.1 DNN Parallelization

Many efforts have been devoted to the design of efficient parallel DNNs. Mirhoseini et al. [32] proposed an RNN model trained by reinforcement learning to determine device placement for parallel execution. Gao et al. [12] proposed a machine learning based method trained by a Markov decision process to assign DNN operations. Huang et al. [20] model the device scheduling problem on HSA as a graph problem. They use the partitioned Boolean quadratic programming solver [3] and critical earliest finish time algorithm [51] to assign operations to devices. The objective is to minimize the computation cost, the communication cost, and the cost of the critical path. These previous works only parallelize the execution of a single model. Hence, there are no dependencies among different input streams. As a result, applying these approaches directly on different frames will cause contention of computing devices in video captioning. In contrast, our pipeline scheduling can achieve parallelism across different frames (inter-frame) without device contention. Bagby et al. [5] and Appleyard et al. [4] proposed a skewing transformation to expose the parallelism of RNNs. However, their scheduling methods only consider one device (GPU or TPU) but not HSA devices.

Xiang and Kim [58] proposed DART, a DNN scheduling framework with real-time guarantee. DART allows multiple DNN models to run in parallel on multiple computing devices and pre-empts low-priority models to release computing resources for urgent ones. DART balances utilization among devices by solving a list partition problem with dynamic programming. Unlike DART, which dynamically schedules many independent models, we target hybrid models with dependent inputs and statically assign the models to HSA devices. Wu et al. [57] proposed a pipeline-based
scheduling scheme, which splits the convolution layers of a DNN model into different pipeline stages. Their work focuses on scheduling sequential graphs, whereas our work covers a broader class of graphs, including multi-edge graphs such as ResNet [13] and DenseNet [18].

Liu and Wu [27] proposed a concatenation algorithm to schedule tasks from multiple batches in a heterogeneous environment. The focus of the concatenation algorithm is quite different from the focus of our work. Each layer (i.e., operation) is a scheduling unit, and each scheduling unit must be mapped to a unique computing resource. The total number of required computing resources increases with the number of layers. The computations are overlapped between different batch samples, and the batch samples are assumed independent. Their algorithm cannot tackle dependent dataflow such as RNN. Our work considers a hybrid model (a CNN and a time-dependent RNN). Each model is a scheduling unit, and we overlap the execution between the CNN and RNN models.

2.2 DNN Acceleration on Edge Devices
Performing DNN inference on the computation-limited edge devices is challenging, especially when real-time performance is needed. Using deep learning accelerators to speed up DNN inference is a feasible solution, making accelerator design a hot research topic in recent years [2, 7, 17, 28, 36, 40, 46, 48]. Many works have used the Intel Movidius Neural Compute Stick [21] to accelerate their models, such as MVCNN [49] for 3D object classification on IoT devices [56], MobileNetv2 [45] in an autonomous racing car [38], and 3D CNN for voxel-based point clouds classification [59]. The Google EdgeTPU has been applied on SSD [30] for real-time face mask detection [37], U-Net [44] for semantic image segmentation [23], and a feed-forward network for a network intrusion detection system [15]. Wang et al. [54] proposed an FPGA-powered smart camera for live video analysis. These works focus on a single model, whereas we tackle the issues of running hybrid models on HSA.

2.3 Background of CUDA Concurrent Kernel Execution
In the CUDA model, a GPU kernel is made up of a set of thread blocks, and a thread block is made up of a set of warps (i.e., 32 threads). Each thread block is scheduled onto a streaming multiprocessor (SM) when a kernel is launched on the GPU.

When enqueueing a GPU kernel for execution, programmers can assign a CUDA stream ID to the GPU kernel to control its execution order. GPU kernels with the same stream ID are execution dependent and must be sequentially processed in the enqueued order. On the contrary, a kernel can be launched on the GPU and concurrently executed with other kernel streams when there exists available GPU resource and no stream ID dependency, thus achieving concurrent kernel execution [33]. Operations from different kernel streams can be executed in any order, which means the execution order between concurrent kernels is not guaranteed.

Concurrent kernel execution is often used for (1) overlapping data transfer and kernel execution to hide the long memory copy latency between the host and the GPU, and (2) maximizing GPU throughput by concurrently executing many narrow tasks (i.e., kernels that do not use all SMs) [62].

2.4 Background of EdgeTPU
The Google EdgeTPU [61] is a machine learning accelerator designed for AI inference. It is capable of executing 4 trillion operations per second and consumes 0.5 W for each of the trillion operations per second. EdgeTPU performs tensor operations using a systolic array. It supports a small set of tensor operations, such as matrix multiplication, 2D convolution, and ReLU. Many operations, such as embedding, are not supported by the EdgeTPU. The unsupported operations must be executed by another processor. Please refer to Coral [11] for the full list of the supported operations.
EdgeTPU only supports models that are fully 8-bit quantized. It has 8 MB of on-chip SRAM for caching model parameters (i.e., weights). At inference time, the model’s executable is placed into the SRAM first. Then, the model’s parameter data is stored to the SRAM one layer at a time. When the layer’s parameters cannot fit into the SRAM, they are stored at the off-chip memory and fetched when necessary. Figure 2 illustrates an example of the CNN inference time with different model sizes. The example shows that the inference time increases rapidly when the model size is larger than the SRAM capacity, indicating the significant overhead of the off-chip memory accesses. EdgeTPU allows parameters of multiple models to be cached in the SRAM together. The parameters are written to the SRAM according to the order of the models decided by the user.

Unlike Nvidia GPUs, EdgeTPU does not support concurrent kernel execution. When multiple models are issued at the same time, their execution is serialized.

3 COMPUTATION SCHEDULING ON HSA

To schedule the computation of video captioning on HSA, a simple approach is to treat CNN and RNN each as a schedule task and map them to their suitable computing devices. For example, according to the experimental results of Figure 1(b), CNN runs more efficiently on a GPU, and RNN runs faster on a CPU. Thus, CNN and RNN are assigned to GPU and CPU, respectively. As will be shown in Section 6.2, this simple coarse-grained scheduling can significantly reduce the total time of video captioning compared with GPU-only execution.

An alternative scheduling approach is letting all operations be schedulable tasks. The goal is to properly assign the tasks to computing devices and minimize the execution time. Such fine-grained scheduling (operation level) is more challenging than the coarse-grained one (model level) because more complex data dependencies may exist between the schedule tasks. In the following, we give a formal definition to the intra-frame scheduling problem and present our fine-grained scheduling algorithm.

3.1 Problem Definition

We consider a computational graph $G = (V, E)$, where $V = \{v_1, \ldots, v_n\}$ is the set of computation and $E$ is the set of dependency. We will consider two types of cost: computation time on nodes and communication time on edges. Each $v$ in $V$ must be mapped to a resource $f(v)$ by a mapping function $f$, and its computation time depends on the resource $r = f(v)$ it is mapped to. Formally, we should use $c_f(v_i)$ to denote the computation cost of running $v_i$ on $f(v_i)$, but for ease of notation, we will drop the subscript $f$ when the context clearly indicates that we are considering mapping function $f$. As a result, we use $c(i)$ to denote this computation time.

Every edge $e = (v_i, v_j)$ in $E$ requires a communication time. The communication time depends on the amount of data $v_i$ will send to $v_j$, and the bandwidth between $v_i$ and $v_j$, which is also a function $f$. As a result we will use $d(i, f(i), f(j))$ to denote this communication time for $v_i$ to send data to $v_j$. The communication cost depends on the amount of data $v_i$ will send to $v_j$ and the bandwidth between $v_i$ and $v_j$.
its data from resource $f(v_j)$ to resource $f(v_i)$. For ease of notation, we will also add the subscript $f$ and use $d_f(i,j) = d(i,f(i),f(j))$ to denote the same cost. Note that if $f(i) = f(j)$ (i.e., $v_i$ and $v_j$ are assigned to the same resource), we define $d$ to be 0, since they are in the same computing resource, and no communication is necessary.

For each $v_i$ in $V$, we will define a starting time $s(i)$, which indicates the earliest possible starting time for $v(i)$. This is again a function of $f$. We also define an end time $e(i)$ for $v_i$, which is by definition equal to $s(i) + c(i)$.

We now define the total time to run a model $G = (V,E)$ under a mapping function $f$. Note that every $v_i$ can start only if all $v_j$ preceding it—that is, $(j,i)$ in $E$—finish their computation and complete sending their data to $v_i$. As a result, the starting time of $v_i$ is in Equation (1), and the total execution time is $T(G) = \max_{v_i \in S} e(i)$, where $S$ is the set of sink nodes (i.e., those nodes without outgoing edges).

\[
s(i) = \max_{(j,i) \in E} (c(j) + d_f(j,i)) \tag{1}
\]

\[
e(i) = s(i) + c(i) \tag{2}
\]

Now we formally define the mapping problem as follows. Given a graph $G = (V,E)$, the computation time, and the communication time, find a function $f$ that minimizes the total time $T(G)$ in Equation (3).

\[
T(G) = \max_{v_i \in S} e(i) \tag{3}
\]

### 3.2 Cases

We consider two cases of computation models. The first case is a sequential graph, in which every $v_i$ receives data from $v_{i-1}$ only, for $1 \leq i \leq n$. An example of sequential graphs is MobileNet. The second case is a complex graph, in which every $v_i$ receives data from all $v_j$, where $j < i$, and $1 < j \leq n$. An example of complex graphs is DenseNet. Please refer to Figure 3 for illustrations.

#### 3.2.1 Sequential Graph

We now solve the scheduling problem for sequential graph. Since $E$ now has only edges connecting $v_i$ and $v_{i-1}$, Equation (1) now becomes Equation (4).

\[
s(i) = c(i-1) + d_f(i-1,i) \tag{4}
\]

We define the best ending time $E(i,r)$ for $v_i$ as the minimum ending time when $f(i) = r$. It is easy to see that we can derive the following recursion for $E(i,r)$, as in Equation (5) when $1 \leq i \leq n$. Note that $d(i-1,r',r)$ indicates the communication time to send the data of $v_{i-1}$ from $r'$ to $r$, where $v_i$ is assigned.

\[
E(i,r) = \min_{r' \in R}(E(i-1,r') + d(i-1,r',r)) \tag{5}
\]

\[
E(1,r) = c(1,r) \tag{6}
\]
The final solution $E^*$ is defined as Equation (7), since we need to consider the ending time of $v_n$ on all possible resources.

$$E^* = \min_{r \in R} E(n, r)$$

(7)

Equation (5) gives an optimal total time for sequential graphs with time complexity $O(nm^2)$, where $n$ is the number of computations and $m$ is the number of resources.

3.2.2 Complex Graph. We now solve the scheduling problem for complex graphs when the number of resource $m$ is 2. Since $E$ now has edges from all $v_j$ to $v_i$, Equation (1) now becomes Equation (8). Note that we cannot use the solution in Equation (5) to solve the problem of complex graphs. The reason is that there are many $i - 1$ incoming edges for a node $v_i$, and all of them will affect the starting time of $v_j$.

$$s(i) = \max_{j < i} (c(j) + df(j, i))$$

(8)

We need the following two observations to simplify the complex graphs solution. In other words, under certain conditions, we do not need to consider an edge $(j, i)$ in Equation (8), since it will not affect the final value of $s(i)$. We define this edge as redundant if its removal from Equation (8) will not affect $s(i)$:

- An edge $(i, k)$ is redundant if $f(i) \neq f(j) = f(k)$ for all $i < j < k$.
- An edge $(i, j)$ is redundant if $i < j$ and $f(i) = f(j)$.

We now describe a dynamic programming to solve the scheduling problems for complex graphs under the assumption that there are two types of resources. For ease of explanation, we will refer to the two resources $r_0$ and $r_1$ as white and black, respectively. Therefore, when we say $v_i$ is black, that means $f(i)$ is the black resource.

We define a function $F(i)$ to be the minimum time to complete $v_1$ to $v_i$ while $f(i) = 1$ is white and $f(i - 1)$ is black. Similarly, we define another function $F'(j)$ to be the minimum time to complete $v_1$ to $v_j$ while $f(j)$ is white and $f(j - 1)$ is black.

We now consider a black $v_i$ and a white $v_{i-1}$, and search backward and skip all white nodes until we find a black $v_{i-1}$. By definition, $v_j$ will be white and $j < i$. We observe that an edge $(a, b)$ is redundant if $1 \leq a < j$ and $j < b \leq i$. Please refer to Figure 4(b) for an illustration. This observation indicates that the effect of any node before $j$ cannot go through $j$ and affect the timing of those nodes between $i$ and $j$. As a result we can isolate the computation of $F(i)$ and $F'(j)$, as in Equation (9). Note that $d(k, 0, 1)$ indicates the communication cost for $v_k$, a white node (0), to send its data to $v_i$, a black node (1).

$$F(i) = \min_{1 < j < i} \left( \max_{j \leq k < i} \left( F'(j) + \sum_{j' = j}^{k} c(j') + d(k, 0, 1) \right) \right) + c(i)$$

(9)

In other words, we can enumerate all possible $F'(j)$ in front of $i$ to compute $F(i)$. Then for each given $j$, we only need to consider the delay due to only those white nodes $v_j, \ldots, v_{i-1}$ on the last black node $v_i$. Similarly, we can define $F''$ using $F$, as in Equation (10).

$$F'(i) = \min_{1 < j < i} \left( \max_{j \leq k < i} \left( F'(j) + \sum_{j' = j}^{k} c(j') + d(k, 1, 0) \right) \right) + c(i)$$

(10)

To sum up, we break $V$ into segments of white and black nodes and isolate the communication effect within each of the segments. Therefore, we can use $F$ and $F'$ alternatively to find the optimal solution. Equations (9) and (10) give an optimal solution for the scheduling problem on complex graphs when there are two types of resources, with time complexity $O(n^3)$. 

ACM Transactions on Architecture and Code Optimization, Vol. 19, No. 3, Article 38. Publication date: May 2022.
### Table 1. The Cost Model

| Operation        | Variable                                      | GPU                                     | CPU                                     |
|------------------|-----------------------------------------------|-----------------------------------------|-----------------------------------------|
| Embedding        | Output data size (x)                          | $y = 0.0933x - 3707$                    | $y = 0.1481x - 5979$                    |
| LSTM             | Input data size (x)                           | $y = 1.491x - 48980$                    | $y = 0.629x - 22290$                    |
| Fully connected  | Trainable parameters (x)                      | $y = 0.000162x + 11$                    | $y = 0.000524x + 17$                    |
| Pooling          | Input data size (x)                           | $y = 0.00012x + 154$                    | $y = 0.00163x + 203$                    |
| Add              | Output data size (x)                          | $y = 0.000092x + 67$                    | $y = 0.000341x + 96$                    |
| Concatenate      | Output data size (x)                          | $y = 0.00046x + 3.2$                    | $y = 0.00021x + 3.3$                    |
| Convolution      | Trainable parameters (x1) and output data size (x2) | $y = 0.0001641x^1 + 0.0117x^2 + 0.048$ | $y = 0.001226x^1 + 0.113x^2 + 0.224$ |
| Communication    | Output data size (x)                          | GPU to CPU                             | CPU to GPU                             |
|                  |                                               | $y = 0.000044x + 73$                    | $y = 0.00005x + 136$                    |

Unit of $x$ is byte; unit of $y$ is microsecond.

### 3.3 Parameter Estimation

The foundation of our theoretical analysis is an accurate estimation of the computation and communication time. The communication time is mainly affected by the size of the output data sent to the next layer; the computation time depends on the number of parameters and input/output data size. Table 1 summarizes the linear cost model. A CNN consists of five basic operations (fully connected, pooling, add, concatenate, and convolution), and the RNN consists of three basic operations (embedding, LSTM, and fully connected). The cost model is a linear regression that approximates the total time $y$ as a function of the primary factors $x$ [26]. We measure the total time and use the Scikit-learn [50] Python package to determine the coefficients of each cost function. The benefit of the cost model is that we can predict other similar deep learning models without any actual execution.

### 4 PIPELINE SCHEDULING

In the preceding section, we described the scheduling schemes for a single video frame. Now we consider the computation scheduling across video frames. As Figure 1(a) shows, the CNNs for different video frames are independent of each other and can be run in parallel. However, there exist data dependencies between adjacent RNNs. These data dependencies force RNNs to run in sequence, which prohibits the full parallelization of the hybrid model. To solve this problem, we propose a pipeline scheduling approach to overlap the execution of CNNs and RNNs. This section elaborates on how to exploit parallelism across video frames.

#### 4.1 Problem Definition

We consider a sequence of $n$ video frames that go through a CNN and an RNN as in Figure 1(a). From the previous discussion, we assume that all CNNs run on a GPU and all RNNs run on a CPU for high efficiency. As the result, we use $t_C$ to denote the execution time of a CNN and $t_R$ to denote the execution time of an RNN. Note that since all frames are of the same size, we assume that $t_C$ and $t_R$ are the same for all frames.

Now we specify the execution schedule of these CNNs and RNNs. Every CNN and RNN has an execution time interval. In other words, this model will run on a device during this time interval.

We observe Figure 1(a) and find three constraints for scheduling video captioning in a system with a CPU and GPU:

**CNN-RNN dependency:** A frame must go through its CNN before going through its RNN.
RNN-RNN dependency: The RNN for the $i$-th frame must run before the RNN for the $i + 1$-th frame. 

Exclusive resource usage: Two CNN intervals from two different frames cannot overlap since they both use the GPU, and two RNN intervals from two different frames cannot overlap since they both use the CPU.

We now define the scheduling problem as follows. Given the number of frames and the execution time of CNN and RNN $t_C$ and $t_R$, find a schedule to run all CNNs and RNNs for all frames without violating any constraints while minimizing the total execution time.

4.2 Concatenation Algorithm

We use a concatenation algorithm [27] to solve the pipeline scheduling problem. Before we illustrate the algorithm, we define ladders.

A ladder is a sequence of the same model (CNN or RNN) running on the same device sequentially. When one model finishes, the next model starts immediately, since we want to minimize the total time and respect the exclusive resource constraint at the same time. We also define the slope of a ladder as $\frac{1}{L}$, where $L$ is the execution time of a step in the ladder. If the model is a CNN, then the slope is $\frac{1}{t_C}$. If the model is an RNN, then the slope is $\frac{1}{t_R}$. The white and gray color indicate CNN and RNN ladder, respectively. Please refer to Figure 5(a) and (b) for an illustration.

4.3 Bottleneck

The key idea of the concatenation algorithm [27] is that we can concatenate two ladders together and maintain the smaller slope. For example, if $t_C < t_R$, the RNN has the smaller slope. Please refer to Figure 5(a) and (b) for an illustration.

We can delay the execution of CNNs so that they start right in front of RNNs and form a new ladder with the same slope, as illustrated in Figure 5(c).

Note that this delay of CNN maintains the smallest slope of the resulting ladder, which is the same as the RNN. As a result the algorithm can repeatedly concatenate all previous ladders with higher slope to the ladder with the smallest slope without violating any exclusive resource usage or data dependency [27].

However, if $t_C > t_R$, for instance, the CNN has the smaller slope. In this case, we can make the execution of RNNs earlier so that they start right after CNN, and form a new ladder with the same slope.

We observe that the execution time is dominated by the longest execution time of the ladder of the slowest model (i.e., by the ladder with the smallest slope). When $t_C < t_R$, the total execution time $T$ is dominated by $t_R$, as in Equation (11).

$$T = n \times t_R + t_C$$ (11)
It is easy to see that $T$ in Equation (11) is optimal, since all RNNs must run sequentially because of exclusive resource usage, and the first RNN can start only after the first CNN finishes, due to the CNN-RNN dependency.

Similarly when $t_C > t_R$, the total execution time $T$ is in Equation (12).

$$T = n \times t_C + t_R$$  \hfill (12)

It is easy to see that $T$ in Equation (12) is also optimal since all CNNs must run sequentially and the last RNN can start only after the last CNN finishes. Finally, we can combine Equations (11) and (12) into Equation (13).

$$T = n \times \max(t_C, t_R) + \min(t_C, t_R)$$  \hfill (13)

5 RESOURCE UTILIZATION

In this section, we describe the resource utilization problems of two accelerator architectures: GPU and EdgeTPU. Then we present two capacity-guided pipeline scheduling methods to solve the problems and improve the performance of video captioning.

5.1 GPU-Capacity-Guided Pipeline Scheduling

In Section 4, the CNN models across video frames are forced to run sequentially due to the exclusive resource constraint. During the CNN execution, we found that the GPU sometimes falls into low utilization. This gives us the optimization opportunity to rearrange the models to utilize the idle GPU resource. In the following, we explain the CNN execution issues using profile data and present how to extend the pipeline scheduling method to improve the GPU utilization.

5.1.1 CNN Execution Profiling. We use the hardware performance monitor to investigate the CNN execution behavior on the GPU. The Nvidia nvprof tool is used with the counter event, sm_efficiency, to collect the SM usage. Figure 6(a) shows the profile result of ResNet50 for one video frame, plotted with 100 sample points. The $x$-axis is the execution timeline, and the $y$-axis is the SM utilization rate. Nvidia GTX 1080 Ti GPU is used in this profiling.

In Figure 6(a), we observe two different execution phases. The first phase is the beginning 2.5 ms, in which the GPU is significantly underutilized. Profile data shows that ResNet50 runs in the Eigen library (the scalar_sum_op, scalar_rsqrt_op, and scalar_product_op function) during this time interval, and only one of the 28 SMs is used. Such a low utilization rate is also reported in other works [24, 67]. In the second phase (>2.5 ms), the GPU achieves a higher utilization rate but occasionally falls into low usage. Since video captioning processes a sequence of video frames, such resource under-utilization behavior will reappear for each frame until the application completes.

5.1.2 Methodology. Since the CNNs from different video frames are independent, they can be executed in parallel. Thus, we can schedule more CNNs to run concurrently and utilize the idle SMs by exploiting the concurrent kernel execution mechanism. However, the exclusive resource usage constraint in the pipeline scheduling algorithm of Section 4 prohibits the CNNs from execution at the same time. Therefore, we relax this constraint.

A simple scheduling scheme is to select a number of CNNs from subsequent frames, associate them with different CUDA streams, and issue them along with the CNN of the current frame. When a CNN is finished, a new CNN is fetched and issued. However, this approach has several drawbacks. First, when the GPU capacity is not sufficient to accommodate all CNNs, some of them will be queued and not launched on the GPU. Second, and most critically, since the execution order of the concurrent kernels is not deterministic, the CNN of the current frame can be finished after the CNNs of later frames. This delays the finish time of current CNN and also delays the start time.
of the RNN of the same frame. As a result, the total execution time is lengthened. Therefore, a sophisticated approach that can prevent these issues is called for.

To execute the CNNs in a controllable manner, we propose a GPU-capacity-guided pipeline scheduling algorithm, which is extended from the original pipeline scheduling approach. Figure 7 depicts the algorithm.

At first, we construct the same CNN ladder as in the original pipeline scheduling algorithm (Figure 7(a)). Then, we move the steps of the CNN ladder to an earlier location (e.g., the dashed line in Figure 7(b)). This makes the CNNs of subsequent frames start execution earlier. As a result, the execution of the later layers of the previous CNN is overlapped with the under-utilized, earlier layers of the next CNN. To determine the moved location, we search backward from the last CNN layer until the first overlapping location that can saturate the GPU capacity is found. Next, the RNN ladder is constructed (Figure 7(c)). Finally, we concatenate the CNN and RNN ladders (Figure 7(d)).

In the end, the CNN model is partitioned into two sub-models. These two CNN sub-models are assigned different CUDA stream IDs so that they can be executed on the GPU at the same
time. Compared with the simple scheduling scheme, the GPU-capacity-guided algorithm does not excessively issue CNNs. The ladder design enforces the execution order of the CNNs and the RNNs as well, making the execution of the hybrid model more controllable and predictable.

The proposed algorithm can improve video captioning when CNN is the bottleneck and GPU is not fully utilized. When the GPU is already fully utilized (e.g., DenseNet-LSTM) or RNN is the bottleneck (e.g., MobileNet-LSTM), the algorithm chooses the original model without splitting CNN.

We use \( c_1 \) and \( c_2 \) to denote the first and second CNN sub-models, and \( t_{c_1} \) and \( t_{c_2} \) respectively denote the execution time of \( c_1 \) and \( c_2 \) (please refer to Figure 7 for an illustration). In the original pipeline scheduling, the execution time of one step of the sequential CNN ladder is \( t_{c_1} + t_{c_2} \). In contrast, it is reduced to \( t_{c_1} \) in the overlapped CNN ladder. We can use \( t_{c_1} \) as the \( t_C \) term in Equation (13) to calculate the total execution time.

Figure 6(b) illustrates the profile result of GPU utilization after applying the GPU-capacity-guided pipeline scheduling. The result shows that the execution of the CNN sub-model \( c_2 \) from previous frame does overlap with the CNN sub-model \( c_1 \) of the next frame.

5.2 EdgeTPU-Capacity-Guided Pipeline Scheduling

Now we discuss the issues of accelerating video captioning on the edge device. The edge device contains a CPU and EdgeTPU(s). Recall that the RNN model consists of three basic operations: embedding, LSTM, and fully connected. Since EdgeTPU does not support embedding, it must be executed by the CPU. Due to this restriction, a straightforward pipeline scheduling method maps the entire RNN model to the CPU and the CNN model to the EdgeTPU. However, as will be shown in Section 7.2, this method performs poorly due to the long execution time of the RNN on the CPU.

Instead, we split the CNN and RNN within one video frame into three models: CNN, embedding, and the RNN without embedding (Figure 8(a)). For simplicity, we use RNN to denote the RNN without embedding. The execution of embedding is still mapped to the CPU. In the following, we elaborate on the scheduling methods for the rest of the model. Two cases are considered: the device has one or two EdgeTPU(s). The resource exclusion constraint is relaxed for scheduling flexibility.

5.2.1 Methodology. We assume both CNN and RNN run more efficiently on the EdgeTPU than on the CPU (please refer to the inference time comparison shown later in Figure 15). In addition, when CNN and RNN are mapped to the same EdgeTPU, the RNN parameters are placed in the SRAM first, then the CNN parameters.

One EdgeTPU. According to the preceding assumption, we map both CNN and RNN to the EdgeTPU for high efficiency. The same EdgeTPU is mapped since there is only one EdgeTPU. As a result, the inference of one video frame executes the CNN on the EdgeTPU first, then goes to the CPU to deal with embedding, and then goes to the EdgeTPU to execute the RNN. The pipeline
scheduling proposed in Section 4 is applied to overlap the inferences across video frames. However, this method has several problems.

First, EdgeTPU has 8 MB of on-chip SRAM for model parameter caching. Because CNN and RNN run on the same EdgeTPU, the total parameter size may exceed the SRAM capacity. For instance, the model size of RNN (LSTM) is 6.25 MB and CNN (DenseNet) is 10.3 MB. Since we store the RNN parameters first, RNN is fully cached. However, CNN is partially cached, in which only 1.75 MB of the parameters are cached in the SRAM and 8.55 MB are stored in the off-chip memory. The inference time is significantly increased due to the considerable off-chip memory overhead, because a large portion of the CNN parameters is not cached.

Second, although the pipelining scheme is used to overlap the execution of the CNN and RNN models, their execution is serialized because EdgeTPU does not support concurrent kernel execution. As a result, the inference time is lengthened.

Third, due to the effects of the preceding two problems, the execution time of embedding on the CPU is relatively shorter than that of CNN + RNN on the EdgeTPU. As a result, the CPU is mostly idle.

To tackle these problems, we propose an EdgeTPU-capacity-guided pipeline scheduling method. When the CNN + RNN parameters exceed the EdgeTPU’s SRAM capacity and the CPU is under-utilized, we can migrate the computation of a certain CNN layers to the CPU so that the off-chip parameter size is reduced. In addition, we can exploit the CPU to overlap the computation with EdgeTPU tasks. As a result, the total execution time can be reduced. To this end, we split the CNN model into two sub-models (c1 and c2), combine c2 and embedding into one model, and map the combined model to the CPU. Figure 8(b) illustrates the execution of the proposed method.

Depending on the CPU computing power, the CPU may become the bottleneck and increase the total execution time if the combined model is too large. Furthermore, the models of c1 and RNN are still sequentially executed, since only one EdgeTPU is used.

Two EdgeTPUs. Since there are two EdgeTPUs and two remaining models (CNN and RNN) to be scheduled, we assign the CNN and RNN each to one EdgeTPU. As a result, the CNN, embedding, and RNN are computed by different devices. Pipeline scheduling is applied to overlap their execution.

Unlike the single EdgeTPU scheduling, the CNN and RNN are executed in parallel rather than being serialized. Furthermore, the CNN has a complete 8 MB of SRAM to cache parameters, thus better CNN inference performance.

There are two observations. First, the SRAM of the EdgeTPU that runs the RNN model becomes under-utilized, because the RNN (LSTM) only uses 6.25 MB of the 8-MB space. Second, the CNN model may still exceed the SRAM capacity (e.g., the 10.3 MB of CNN (DenseNet)).

Therefore, we can exploit the unused SRAM space to store the uncached CNN parameters, thus reducing the off-chip memory accesses. The proposed EdgeTPU-capacity-guided pipeline scheduling is also applied here. We split the CNN model into two sub-models: c1 and c2. Instead of moving c2 to the CPU, we migrate the computation of c2 to the other EdgeTPU and utilize the available SRAM space. As a result, better SRAM utilization can be achieved and execution time can be reduced. Figure 8(c) illustrates the execution with two EdgeTPUs.

6 EXPERIMENTAL RESULTS ON THE SERVER PLATFORM
6.1 Experimental Settings

All methods are implemented using Python 3.7 with Keras 2.0.9 on TensorFlow [1] backend. We select the MSVD [68] dataset including 1,938 videos with 300 frames per video as the benchmark. The evaluation is performed on a CPU + GPU server platform, which has an Intel Core i7-8700 3.7-GHz processor with 64 GB of DRAM, and an NVIDIA GTX 1080 Ti graphics card containing...
Fig. 9. Time ratio of CNNs (MobileNet, ResNet, DenseNet) on Nvidia GPU compared to RNN (LSTM) on x86 CPU.

Fig. 10. Video captioning execution on the server platform.

Fig. 11. The timing results on the server platform (x86 CPU + Nvidia GPU).

28 SMs, 3,584 CUDA cores, and 11 GB of device memory. NVIDIA CUDA Toolkit 9.0 and cuDNN library 7.3.0 [8] are used in the TensorFlow backend.

We use three CNNs (DenseNet121 [18], ResNet50 [13], and MobileNetv1 [16]) and one RNN (LSTM) to evaluate the proposed scheduling methods. Figure 9 shows the ratio of running time of the CNNs to the running time of the RNN. This ratio provides useful information on how to scheduling these CNNs and RNNs. We use the CNN-RNN name pair to denote the hybrid model (e.g., DenseNet-LSTM) and omit the number of layers of CNNs for brevity unless otherwise mentioned.

6.2 Comparison of Scheduling Methods

We use GPU-only, HSA (coarse-grained), HSA (fine-grained), and HSA (pipelining) to denote the four scheduling methods. Recall that the coarse-grained method schedules an entire CNN/RNN as an unit and the fine-grained method divides the CNN into five basic operations (fully connected, pooling, add, concatenate, and convolution), and the RNN into three basic operations (embedding, LSTM, and fully connected) for possibly more flexible and efficient scheduling, since the basic operations in a CNN (RNN) may prefer different processing devices. The execution of these methods is illustrated in Figure 10. The white and gray color indicate GPU and CPU execution, respectively.

Figure 11 shows the performance of MobileNet-LSTM, ResNet-LSTM, and DenseNet-LSTM with 40 iterations using different scheduling methods. The y-axis is the speedup over the GPU-only execution (baseline).

We first observe that by simply running RNNs on CPU, the HSA (coarse-grained) method already has a $2.11 \times$ speedup over the GPU-only method in MobileNet-LSTM. This is a good indication that finding suitable devices to run these models is crucial.
We then observe that HSA (fine-grained) outperforms HSA (coarse-grained), with an improvement of 11%. This is because, compared with the coarse-grained scheduling, the fully connected layer in the RNN is moved from CPU to GPU for more efficient execution in the fine-grained scheduling. This result demonstrates that our accurate cost model together with the dynamic programming technique can identify the compute-intensive operations and schedule them to run on suitable devices.

Figure 11 indicates that HSA (pipelining) achieves $3.24 \times$ speedup over the GPU-only method. There are two reasons for the significant speedup. First, the RNN runs on a more suitable computing device (CPU). Second, by pipelining the execution of adjacent iterations, the MobileNet and LSTM can run on GPU and CPU concurrently.

We also observe that the speedups of all scheduling methods decrease when the size of CNN increases. We find that the time ratio over LSTM ranges from 0.66 for MobileNet to 2.66 for DenseNet in Figure 9. It is easy to see that large CNNs dominate the total execution time of video captioning, making the effect of optimizing scheduling less significant. Nevertheless, our result is still very promising because DenseNet is widely recognized as the state-of-the-art large CNN that provides high accuracy [47, 52, 60]. Even with large CNNs such as DenseNet, our scheduling achieves $1.69 \times$ speedup in Figure 11.

### 6.3 Evaluation of the GPU-Capacity-Guided Pipeline Scheduling

In the following, we use HSA (pipelining-cap) to denote the GPU-capacity-guided pipeline scheduling method. Recall that HSA (pipelining-cap) splits the CNN model into two sub-models, $c_1$ and $c_2$, and overlaps their execution (please refer to Figure 7(b) for illustration). Thus, we vary the overlapping ratio by changing the split locations to study the performance. The overlapping ratio is calculated as $\frac{t_{c_2}}{t_{c_1}+t_{c_2}}$, where $t_{c_1}$ and $t_{c_2}$ are the execution time of sub-models $c_1$ and $c_2$.

MobileNet, DenseNet, and three ResNet variants (50/101/152 layers) are used in this experiment. In Figure 12, the x-axis is the overlapping ratio, and the y-axis is the speedup of HSA (pipelining-cap) over HSA (pipelining). For all CNNs, we select the last layer (i.e., fully connected) as one split location, which achieves an overlapping ratio of about 1% and is the result of the leftmost five points in Figure 12. For ResNets, other layers (conv2d) are selected with overlapping ratios ranging from 5% to 35%. For MobileNet and DenseNet, two extra layers (conv2d) are selected with overlapping ratios between 13% and 50%.
Several factors affect the performance of HSA (pipelining-cap) compared with HSA (pipelining). The first factor is \textit{bottleneck}. HSA (pipelining-cap) only changes the CNN execution time, \( t_C \). Thus, according to Equation (13), it only affects the performance when the execution time is dominated by CNN (\( t_C > t_R \)) but not by RNN (\( t_R > t_C \)). The second factor is \textit{GPU capacity}. When GPU capacity is not sufficient for overlapping the execution of GPU kernels, the kernels are forced to run in sequence. Furthermore, concurrent kernel execution incurs overhead due to extra pressure on the scheduler. The third factor is \textit{overlapping ratio}. With sufficient GPU capacity, the greater the ratio is, the higher the improvement can be achieved. The fourth factor is \textit{synchronization}. For each video frame, HSA (pipelining) requires one synchronization: between the CNN and RNN model. On the contrary, HSA (pipelining-cap) requires two synchronizations: between CNN sub-models \( c_1 \) and \( c_2 \), and between CNN sub-model \( c_2 \) and RNN. Thus, HSA (pipelining-cap) incurs higher synchronization overhead.

As Figure 13 shows, the speedup of MobileNet-LSTM decreases when increasing the overlapping ratio. Profile data in Figure 13 indicates that MobileNet’s two sub-models are concurrently executed and its execution time is shortened. However, since RNN is the bottleneck of MobileNet-LSTM, it does not benefit from concurrent execution of the CNN kernels. In addition, it imposes additional synchronization overhead with HSA (pipelining-cap) compared to HSA (pipelining).

We also observe no performance improvement in DenseNet-LSTM. Note that DenseNet is composed by chaining the complex graph (Figure 3(b)) four times; we only choose the joint points as the candidate split locations because the synchronization overhead is minimized. In DenseNet, overlapping the execution of the sub-models (complex graphs) exceeds the GPU capacity. Furthermore, extra synchronization and hardware scheduling overheads are incurred, thereby making HSA (pipelining) outperform HSA (pipelining-cap).

In ResNet-LSTMs, we first observe that the speedup increases with the overlapping ratio. The performance gain results from shortened CNN execution time due to overlapping the execution of the CNN sub-models. The profile data in Figure 6(b) verifies this result. When the overlapping exceeds the GPU capacity, we observe that the speedup decreases. Overall, up to 7%, 6%, and 5% improvements are achieved with HSA (pipelining-cap) for ResNet50-LSTM, ResNet101-LSTM, and ResNet152-LSTM, respectively.

7 EXPERIMENTAL RESULTS ON THE EDGE PLATFORM

7.1 Experimental Settings

The Google Coral Dev Board [10] is used as the edge platform, which consists of an ARM Cortex-A53 1.5-GHz processor with 1 GB of DRAM and an on-board EdgeTPU accelerator (PCIe interface).
In addition, we plug a USB EdgeTPU into the board as the secondary accelerator, forming a two-EdgeTPU edge platform. Both EdgeTPUs have 8 MB of SRAM and support models of 8-bit integer only. We use the same benchmarks of the server platform, except that the CNN and RNN models are quantized to int8. The models are compiled with EdgeTPU compiler version 15.

Figure 14 illustrates the model size after quantization, reported by the EdgeTPU compiler. The size includes the model’s executable and the weights. Figure 15 shows the inference time ratio of individual model to the RNN execution time on EdgeTPU. The result indicates that RNN and CNNs all run faster on the EdgeTPU than on the ARM CPU. Furthermore, ResNet takes the longest time among the CNNs on the EdgeTPU. As shown in Figure 14, ResNet has the largest model size compared to MobileNet and DenseNet. Since EdgeTPU has only 8 MB of SRAM, a significant amount of the ResNet parameters are fetched from the off-chip memory, which degrades the performance.

In the following, we present the scheduling results of using one EdgeTPU only (i.e., on-board EdgeTPU) and two EdgeTPUs. Note that when one EdgeTPU is used and both CNN and RNN are mapped to EdgeTPU, RNN parameters are placed in the SRAM first and then the CNN parameters.

### 7.2 Comparison of Scheduling Methods with a Single EdgeTPU

We first evaluate the scheduling methods with one EdgeTPU: ARM-only, HSA (fine-grained), and HSA (pipelining-cap). The execution of these methods is illustrated in Figure 16(a) through (c), respectively.

ARM-only denotes running the entire model on the ARM CPU. We use this as the baseline.

According to Figure 15, both CNN and RNN run more efficiently on the EdgeTPU than on the ARM CPU. Thus, our coarse-grained scheduling maps the execution of both models to the EdgeTPU. The same mapping is also made by our fine-grained scheduling method. Thus, we only show HSA (fine-grained) for both scheduling methods.

HSA (pipelining-cap) denotes the EdgeTPU-capacity-guided pipelining method, which migrates the computation of the last CNN layer (fully connected) to the ARM CPU. We observe that moving the last layer achieves the best performance. Moving more layers (conv2d) significantly increases the CPU time and degrades the performance.

Figure 17 shows the performance results. For DenseNet-LSTM, the result indicates that HSA (fine-grained) is 24.1× faster than ARM-only. Furthermore, HSA (pipelining-cap) achieves a significant speedup of 26.9× over ARM-only. Such improvement results from running CNN and RNN on the highly efficient EdgeTPU instead of the inefficient ARM CPU. The optimizations of re-arranging the fully connected layer and overlapping the computations of the CPU and EdgeTPU tasks make HSA (pipelining-cap) outperform HSA (fine-grained) by 11%.
Fig. 16. Video captioning execution on the edge platform. The white color indicates the execution on ARM CPU. The gray and black colors indicate the execution by the on-board and USB EdgeTPU, respectively.

Fig. 17. The timing results on the edge platform (ARM CPU + single EdgeTPU).

Fig. 18. HSA (pipeline-2stage) timing result, which runs CNN on EdgeTPU and RNN on ARM CPU.

Similar results are observed in MobileNet-LSTM and ResNet-LSTM. HSA (pipelining-cap) achieves 11.8× and 15.3× speedup over ARM-only, respectively, and 13% faster than HSA (fine-grained).

Case study: HSA (pipeline-2stage). It is interesting to study the case that the RNN is not split, and run the RNN entirely on the ARM CPU and CNN on the EdgeTPU. We denote this pipeline scheduling as HSA (pipeline-2stage), and Figure 16(d) illustrates the execution flow.

The performance result in Figure 18 shows that HSA (pipeline-2stage) runs significantly slower than HSA (fine-grained) in MobileNet-LSTM and DenseNet-LSTM. This is mainly attributed to the significantly long execution time of RNN on the ARM CPU. Even if pipeline scheduling is employed to overlap the execution of RNN and CNN, it is still much longer than the sequential execution of both CNN and RNN on the EdgeTPU with fine-grained scheduling.

Interestingly, HSA (pipeline-2stage) outperforms HSA (fine-grained) with ResNet-LSTM. As Figure 15 shows, running ResNet solely on the EdgeTPU already suffers long inference time due to excessive off-chip memory accesses. This situation is exacerbated in fine-grained scheduling because RNN and ResNet are both scheduled to the same EdgeTPU, and RNN parameters are cached in the SRAM first. This causes more ResNet parameters to be moved away to the off-chip memory.

On the contrary, although running RNN on the ARM CPU increases the execution time of RNN, it occupies no SRAM space. More ResNet parameters can be cached in the SRAM, and thus the overhead of off-chip memory accesses is reduced. Furthermore, the execution of CNN and RNN can be overlapped to hide the ResNet time. As a result, HSA (pipeline-2stage) exhibits better performance than the fine-grained scheduling method.

7.3 Comparison of Scheduling Methods with Two EdgeTPUs

We evaluate the scheduling methods with two EdgeTPUs: HSA (fine-grained-dual) and HSA (pipelining-cap-dual). HSA (pipelining-cap-dual) denotes the EdgeTPU-capacity-guided pipelining
method, which splits CNN into two sub-models (c1 and c2) and migrates the computation of c2 to the other EdgeTPU. In both methods, CNN and RNN are respectively executed on the on-board and USB EdgeTPUs. Figure 16(e) and (f) illustrate the execution of the two methods.

At first, we set c1 to the entire CNN model and c2 to none—that is, CNN is not split. The performance results in Figure 19 indicate that HSA (pipelining-cap-dual) achieves the best performance, with 37.6×, 19.2×, and 46.1× speedups over ARM-only in MobileNet-LSTM, ResNet-LSTM, and DenseNet-LSTM respectively, and 58%, 15%, and 19% faster than the sequentially executed HSA (fine-grained-dual).

Comparing the results of single EdgeTPU, we first observe that all models are improved with two EdgeTPUs. This is as expected—since more SRAM space is used, better performance is achieved.

We also observe that the improvement is significant in MobileNet-LSTM (e.g., from 10.4 to 23.8 with fine-grained scheduling), and is only slightly improved in ResNet-LSTM (e.g., from 13.4 to 16.7 with fine-grained scheduling). The reason is that MobileNet is only partially cached when there is one EdgeTPU. The off-chip memory overhead increases the total execution time. On the contrary, MobileNet is fully cached when two EdgeTPUs are used (please see the model size in Figure 14). The off-chip memory overhead is eliminated, and thus significant improvement is observed.

Significant improvement is also observed in DenseNet-LSTM. It is due to most of the DenseNet parameters being cached with two EdgeTPUs. In ResNet-LSTM, although more parameters are cached on the EdgeTPUs, a significant portion of the parameters are still stored in the off-chip memory. The overhead dominates the inference latency and limits the performance gain. The comparison of one and two EdgeTPUs is summarized later in Figure 20, using HSA (fine-grained) as the baseline.

Split ratio. Now we compare the performance of HSA (pipeline-cap-dual) with different c2 by varying the CNN split ratio. The ratio is calculated as \( \frac{t_{c2}}{t_{c1}+t_{c2}} \), where \( t_{c1} \) and \( t_{c2} \) are the execution
time of $c_1$ and $c_2$. We evaluate ResNet-LSTM and DenseNet-LSTM because their CNNs are still partially cached.

Figure 21 shows that both models are improved when 10% to 40% of CNN layers are migrated. The maximum speedup is achieved when the SRAMs of both EdgeTPUs are fully utilized, in which the most CNN parameters are cached. With more CNN layers being moved, $c_1$ becomes too small, which leads to the under-utilization of the EdgeTPU’s SRAM. Moreover, the total number of off-chip memory accesses starts to increase. As a result, the performance starts to degrade. In the worst case, the entire CNN is migrated, and the performance is the same as using only one EdgeTPU.

Compared to the results without CNN split, the speedup of ResNet-LSTM over ARM-only is significantly improved from $19.2 \times$ to $27.3 \times$. The best result is in DenseNet-LSTM, which achieves $54.9 \times$ speedup over ARM-only.

### 7.4 Execution Performance and Accuracy

To build the quantized video captioning models for the edge platform, we first train the CNN and RNN models of 32-bit floating points on the server. We then apply post-training quantization to convert both models into 8-bit integers. We use BLEU [35] as the metric to evaluate the quality of video captioning, and Table 2 lists the scores. The result indicates that the BLEU score drops by only 0.04 points with the quantized models compared to the original ones.

Table 2 also lists the execution performance of the best-performing method (i.e., pipeline scheduling) using processing frame rate as the metric. MobileNet-LSTM achieves 0.55 BLEU scores with video captioning performance of at least 48 frames per second on the edge platform. ResNet-LSTM only achieves 25 frames per second with two EdgeTPUs due to its long inference latency. For the large and high-accuracy DenseNet-LSTM model, our scheduling can perform video captioning of 29 frames per second with one EdgeTPU and 59 frames per second with two EdgeTPUs. Hence, our method is a highly desirable solution for real-time video captioning on edge devices.

### 8 CONCLUSION

Video captioning is a multidisciplinary application that employs a hybrid CNN + RNN model. There are several challenging issues to run hybrid models efficiently on heterogeneous systems.
First, we consider the scheduling granularity for assigning a model (coarse-grained) or an operation (fine-grained) to appropriate devices to minimize the computation cost. Second, we address the data dependency between the CNN and RNN within a video frame and between the adjacent RNNs across video frames. Third, we address resource utilization of accelerators. To minimize computation cost, we use dynamic programming to find the best mapping of computations and devices within a frame. To exploit parallelism, we propose a pipelining method to overlap the execution of CNN and RNN of consecutive frames. To maximize resource utilization, we exploit GPU concurrent kernel execution. On edge platform, we carefully arrange the CNN computations among the CPU and AI accelerators, thus balancing the computation and minimizing off-chip memory accesses. Experimental results show that the proposed scheduling methods achieve outstanding video captioning performance and enable real-time video captioning on the edge device.

ACKNOWLEDGMENTS

The authors thank the reviewers for their valuable comments and suggestions to improve the quality of this article.

REFERENCES

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, et al. 2016. TensorFlow: A system for large-scale machine learning. In Proceedings of the USENIX Conference on Operating Systems Design and Implementation. 265–283.
[2] Manoj Alwani, Han Chen, Michael Ferdman, and Peter Milder. 2016. Fused-layer CNN accelerators. In Proceedings of the International Symposium on Microarchitecture. 1–12.
[3] Andrew Anderson and David Gregg. 2018. Optimal DNN primitive selection with partitioned Boolean quadratic programming. In Proceedings of the International Symposium on Code Generation and Optimization. 340–351.
[4] Jeremy Appleyard, Tomás Kociský, and Phil Blunsom. 2016. Optimizing performance of recurrent neural networks on GPUs. arXiv:1604.01946 (2016).
[5] Tom Bagby, Kanishka Rao, and Khe Chai Sim. 2018. Efficient implementation of recurrent neural network transducer in TensorFlow. In Proceedings of the IEEE Spoken Language Technology Workshop.
[6] Siddharth Biswal, Cao Xiao, M. Brandon Westover, and Jimeng Sun. 2019. EEGtoText: Learning to write medical reports from EEG recordings. In Proceedings of the 4th Machine Learning for Healthcare Conference. 513–531.
[7] Yu-Hsin Chen, Joel Emer, and Vivienne Sze. 2016. Eyerriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks. In Proceedings of the International Symposium on Computer Architecture. 367–379.
[8] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. 2014. cuDNN: Efficient primitives for deep learning. arXiv:1410.0759 (2014).
[9] Kyunghyun Cho, Bart van Merriënboer, Dzmitry Bahdanau, and Yoshua Bengio. 2014. On the properties of neural machine translation: Encoder-decoder approaches. In Proceedings of the Workshop on Syntax, Semantics, and Structure in Statistical Translation. 103–111.
[10] Coral. 2020. Coral Dev Board. Retrieved April 8, 2022 from https://coral.ai/products/dev-board/.
[11] Coral. 2021. Operations Supported by the EdgeTPU. Retrieved April 8, 2022 from https://coral.ai/docs/edgetpu/models-intro#supported-operations.
[12] Yuanxiang Gao, Li Chen, and Baochun Li. 2018. Spotlight: Optimizing device placement for training deep neural networks. In Proceedings of the 35th International Conference on Machine Learning. 1676–1684.
[13] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition. 770–778.
[14] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long short-term memory. Neural Computation 9, 8 (1997), 1735–1780.
[15] Andrew G. Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. 2017. MobileNets: Efficient convolutional neural networks for mobile vision applications. arXiv:1704.04861 (2017).
[18] Gao Huang, Zhuang Liu, Laurens Van Der Maaten, and Kilian Q. Weinberger. 2017. Densely connected convolutional networks. In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition. 2261–2269.

[19] Horng-Ruey Huang, Ding-Yong Hong, Jan-Jan Wu, Fangfei Liu, and Wei-Chung Hsu. 2021. Efficient video captioning on heterogeneous system architectures. In Proceedings of the IEEE International Parallel and Distributed Processing Symposium. 1035–1045.

[20] Zi Xuan Huang, Shen Yu Fu, and Wei Chung Hsu. 2019. Efficient dynamic device placement for deep neural network training on heterogeneous systems. In Embedded Computer Systems: Architectures, Modeling, and Simulation. Springer, 51–64.

[21] Intel. 2017. Neural Compute Stick. Retrieved April 8, 2022 from https://software.intel.com/content/www/us/en/develop/articles/intel-movidius-neural-compute-stick.html.

[22] Yunseok Jang, Jae Song, Chris Dongjoo Kim, Youngjae Yu, Youngjin Kim, and Gunhee Kim. 2019. Video question answering with spatio-temporal reasoning. International Journal of Computer Vision 127, 10 (2019), 1385–1412.

[23] Andreas M. Kist and Michael Döllinger. 2020. Efficient biomedical image segmentation on EdgeTPUs at point of care. IEEE Access 8 (2020), 139356–139366.

[24] Cheng Li, Abdul Dakkak, Jinjun Xiong, Wei Wei, Lingjie Xu, and Wen-Mei Hwu. 2020. XSP: Across-stack profiling and analysis of machine learning models on GPUs. In Proceedings of the IEEE International Parallel and Distributed Processing Symposium. 326–327.

[25] Chong Li, Kunyang Jia, Dan Shen, C. J. Richard Shi, and Hongxia Yang. 2019. Hierarchical representation learning for bitapirte graphs. In Proceedings of the International Joint Conference on Artificial Intelligence. 2873–2879.

[26] En Li, Zhi Zhou, and Xu Chen. 2018. Edge intelligence: On-demand deep learning model co-inference with device-edge synergy. In The Workshop on Mobile Edge Communications. 31–36.

[27] Pangfei Liu and Jan-Jan Wu. 2019. Task scheduling techniques for deep learning in heterogeneous environment. In Proceedings of the International Symposium on Computing and Networking Workshops. 141–147.

[28] Shaoli Liu, Zidong Du, Jinhua Tao, Dong Han, Tao Luo, Yuan Xie, Yujin Chen, and Tianshi Chen. 2016. Cambricon: An instruction set architecture for neural networks. In Proceedings of the International Symposium on Computer Architecture. 393–405.

[29] Shuanglong Liu, Chao Zhang, and Jinwen Ma. 2017. CNN-LSTM neural network model for quantitative strategy analysis in stock markets. In Neural Information Processing. Lecture Notes in Computer Science, Vol. 10635. Springer, 198–206.

[30] Wei Liu, Dragomir Anguelov, Dumitru Erhan, Christian Szegedy, Scott Reed, Cheng-Yang Fu, and Alexander C. Berg. 2016. SSD: Single shot MultiBox detector. In Proceedings of the European Conference on Computer Vision. 21–37.

[31] Chao Li, Abdul Dakkak, Jinjun Xiong, Wei Wei, Lingjie Xu, and Wen-Mei Hwu. 2020. XSP: Across-stack profiling and analysis of machine learning models on GPUs. In Proceedings of the IEEE International Parallel and Distributed Processing Symposium. 326–327.

[32] Azalia Mirhoseini, Anna Goldie, Hieu Pham, Benoit Steiner, Quoc V. Le, and Jeff Dean. 2018. A hierarchical model for device placement. In Proceedings of the International Conference on Learning Representations. 1–11.

[33] Nvidia. 2021. CUDA C++ Programming Guide v11.4.0. Retrieved April 8, 2022 from https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html.

[34] Silvio Olivastri, Gurkirt Singh, and Fabio Cuzzolin. 2019. End-to-end video captioning. In Proceedings of the International Conference on Computer Vision. 1474–1482.

[35] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 2002. BLEU: A method for automatic evaluation of machine translation. In Proceedings of the Annual Meeting on Association for Computational Linguistics. 311–318.

[36] Eunhyeok Park, Dongyoung Kim, and Sungjoo Yoo. 2018. Energy-efficient neural network accelerator based on outlier-aware low-precision computation. In Proceedings of the AAAI Conference on Artificial Intelligence 34, 4 (2020), 5045–5042.

[37] Keondo Park, Wonyoung Jang, Woochul Lee, Kisung Nam, Kihong Seong, Kyuwook Cha, and Wen-Syan Li. 2020. Efficient video captioning on heterogeneous system architectures. In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition. 779–788.

[38] Anwar Ur Rehman, Ahmad Kamran Malik, Basit Raza, and Waqar Ali. 2019. A hybrid CNN-LSTM model for improving accuracy of movie reviews sentiment analysis. Multimedia Tools and Applications 78 (2019), 26597–26613.
[43] Shaoqing Ren, Kaiming He, Ross Girshick, and Jian Sun. 2015. Faster R-CNN: Towards real-time object detection with region proposal networks. In Proceedings of the 28th International Conference on Neural Information Processing Systems, Vol. 1. 91–99.

[44] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. 2015. U-Net: Convolutional networks for biomedical image segmentation. In Medical Image Computing and Computer-Assisted Intervention. Lecture Notes in Computer Science, Vol. 9351. Springer, 234–241.

[45] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. 2018. MobileNetV2: Inverted residuals and linear bottlenecks. In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition. 4510–4520.

[46] Yongming Shen, Michael Ferdman, and Peter Milder. 2017. Maximizing CNN accelerator efficiency through resource partitioning. In Proceedings of the International Symposium on Computer Architecture. 535–547.

[47] Jingkuan Song, Yuyu Guo, Lianli Gao, Xuelong Li, Alan Hanjalic, and Heng Tao Shen. 2019. From deterministic to generative: Multimodal stochastic RNNs for video captioning. IEEE Transactions on Neural Networks and Learning Systems 30, 10 (2019), 3047–3058.

[48] Prakalp Srivastava, Mingu Kang, Sujan K. Gonugondla, Sungmin Lim, Jungwook Choi, Vikram Adve, Nam Sung Kim, and Naresh Shanbhag. 2018. PROMISE: An end-to-end design of a programmable mixed-signal accelerator for machine-learning algorithms. In Proceedings of the International Symposium on Computer Architecture. 43–56.

[49] Hang Su, Subhransu Maji, Evangelos Kalogerakis, and Erik Learned-Miller. 2015. Multi-view convolutional neural networks for 3D shape recognition. In Proceedings of the IEEE International Conference on Computer Vision. 945–953.

[50] Gaël Varoquaux, Lars Buitinck, Gilles Louppe, Olivier Grisel, Fabian Pedregosa, and Andreas Mueller. 2015. Scikit-learn: Machine learning without learning the machinery. GetMobile: Mobile Computing and Communications 19, 1 (2015), 29–33.

[51] Aravind Vasudevan and David Gregg. 2017. Mutual inclusivity of the critical path and its partial schedule on heterogeneous systems. arXiv:1701.08800 (2017).

[52] Subhashini Venugopalan, Marcus Rohrbach, Jeff Donahue, Raymond Mooney, Trevor Darrell, and Kate Saenko. 2015. Sequence to sequence—Video to text. In Proceedings of the International Conference on Computer Vision. 4534–4542.

[53] Subhashini Venugopalan, Huijuan Xu, Jeff Donahue, Marcus Rohrbach, Raymond Mooney, and Kate Saenko. 2015. Translating videos to natural language using deep recurrent neural networks. In Proceedings of the Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies. 1494–1504.

[54] Shang Wang, Chen Zhang, Yuanchao Shu, and Yunnxin Liu. 2019. Live video analytics with FPGA-based smart cameras. In Proceedings of the Workshop on Hot Topics in Video Analytics and Intelligent Edges. 9–14.

[55] Ronald J. Williams and David Zipser. 1995. Gradient-based learning algorithms for recurrent networks and their computational complexity. In Backpropagation: Theory, Architectures, and Applications, Yves Chauvin and David E. Rumelhart (Eds.). Lawrence Erlbaum Associates, Hillsdale, NJ, 433–486.

[56] Cristian Wisultschew, A. Otero, Jorge Portilla, and Eduardo de la Torre. 2019. Artificial vision on edge IoT devices: A practical case for 3D data classification. In Proceedings of the Conference on Design of Circuits and Integrated Systems. 1–7.

[57] Hsin-I. Wu, Da-Yi Guo, Hsu-Hsun Chin, and Ren-Song Tsay. 2020. A pipeline-based scheduler for optimizing latency of convolution neural network inference over heterogeneous multicore systems. In Proceedings of the International Conference on Artificial Intelligence Circuits and Systems. 46–49.

[58] Yecheng Xiang and Hyoseung Kim. 2019. Pipelined data-parallel CPU/GPU scheduling for multi-DNN real-time inference. In Proceedings of the IEEE Real-Time Systems Symposium. 392–405.

[59] Xiaofan Xu, João Amaro, Sam Caulfield, Andrew Forembks, Gabriel Falcao, and David Moloney. 2017. Convolutional neural network on neural compute stick for voxelized point-clouds classification. In Proceedings of the International Congress on Image and Signal Processing, BioMedical Engineering, and Informatics. 1–7.

[60] Yang Yang, Jie Zhou, Jiangbo Ai, Yi Bin, Alan Hanjalic, Heng Tao Shen, and Yanli Ji. 2018. Video captioning by adversarial LSTM. IEEE Transactions on Image Processing 27, 11 (2018), 5600–5611.

[61] Amir Yazdabakhsh, Kiran Seshadri, Berkin Akin, James Laudon, and Ravi Narayanaswami. 2021. An evaluation of edge TPU accelerators for convolutional neural networks. arXiv:2102.10423 (2021).

[62] Tsung Tai Yeh, Amit Sabne, Putt Sakdhnagool, Rudolf Eigenmann, and Timothy G. Rogers. 2017. Pagoda: Fine-grained GPU resource virtualization for narrow tasks. In Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. 221—234.

[63] Peng Yu, Yanqiao Zhu, Qiang Liu, Shu Wu, Liang Wang, and Tieniu Tan. 2020. TAGNN: Target attentive graph neural networks for session-based recommendation. In Proceedings of the International ACM SIGIR Conference on Research and Development in Information Retrieval. 1921–1924.

[64] Huanyu Yu, Shuo Cheng, Bingbing Ni, Minsi Wang, Jian Zhang, and Xiaokang Yang. 2018. Fine-grained video captioning for sports narrative. In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition. 6006–6015.
Haonan Yu, Jiang Wang, Zhiheng Huang, Yi Yang, and Wei Xu. 2016. Video paragraph captioning using hierarchical recurrent neural networks. In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition. 4584–4593.

Minjia Zhang, Samyam Rajbhandari, Wenhan Wang, and Yuxiong He. 2018. DeepCPU: Serving RNN-based deep learning models 10x faster. In Proceedings of the USENIX Annual Technical Conference. 951–965.

Hongyu Zhu, Mohamed Akrout, Bojian Zheng, Andrew Pelegris, Anand Jayarajan, Amar Phanishayee, Bianca Schroeder, and Gennady Pekhimenko. 2018. Benchmarking and analyzing deep neural network training. In Proceedings of the IEEE International Symposium on Workload Characterization. 88–100.

Begüm Çtamak, Menekşe Kuyu, Aykut Erdem, and Erkut Erdem. 2019. MSVD-Turkish: A large-scale dataset for video captioning in Turkish. In Proceedings of the Signal Processing and Communications Applications Conference. 1–4.

Received September 2021; revised January 2022; accepted March 2022