Towards Scalable Video Analytics at the Edge

Theodore Stone[1] Nathaniel Stone[1] Puneet Jain[2] Yurong Jiang[3] Kyu-Han Kim[4] Srihari Nelakuditi[1]
[1]University of South Carolina [2]Google* [3]LinkedIn* [4]Hewlett-Packard Labs

***Abstract*—Breakthroughs in deep learning, GPUs, and edge computing have paved the way for always-on, live video analytics. However, to achieve real-time performance, a GPU needs to be dedicated amongst a few video feeds. But, GPUs are expensive resources and a large-scale deployment requires supporting hundreds of video cameras – exorbitant cost prohibits widespread adoption. To ease this burden, we propose `Tetris`, a system comprising of several optimization techniques from computer vision and deep-learning literature blended in a synergistic manner. `Tetris` is designed to maximize the parallel processing of video feeds on a single GPU, with a marginal drop in inference accuracy. `Tetris` performs CPU-based tiling of active regions to combine activities across video feeds. resulting in a condensed input volume. It then runs the deep learning model on this condensed volume instead of individual feeds, which significantly improves the GPU utilization. Our evaluation on Duke MTMC dataset reveals that `Tetris` can process 4x video feeds in parallel compared to any of the existing methods used in isolation.**

I. Introduction

Deep learning inference accuracy has recently surpassed humans for several image recognition tasks. Given that a plethora of new applications could be enabled using real-time video analysis, it is natural to raise the question of scalability and hardware requirement of deep learning based video analytics. A system that analyzes a large number of camera feeds needs to overcome two fundamental challenges in the *connectivity* and the *compute*. The connectivity challenge relates to the requirement of high-bandwidth and low-latency networks. Edge computing addresses the connectivity issue to some extent by locating compute nodes closer to the data source – avoiding long round-trip network delays. However, edge computing presents new challenges in the application scalability. Avoiding round-trips to the cloud results in reduced computing power. An edge compute node is expected to be far less powerful than the cloud, containing a few GPUs at best. Therefore, this paper seeks to answer how we can enable real-time video analytics in a cost efficient manner at the edge, as depicted in Figure 1.

Industry trends suggest an edge compute node to be as powerful as a workstation, supporting up to two GPUs per node. Any video analytics task such as image classification or object localization would rely on these GPUs for the heavy-lifting. To seamlessly enable video analytics at the edge, it is desirable that the number of compute nodes required is minimized. Past works in this area focused on minimizing inference time per video frame, giving little attention to the parallel inferences on a GPU. But having a dedicated compute node per camera is neither cost-effective nor maintainable.

*The work was done while the authors were at Hewlett-Packard Labs.

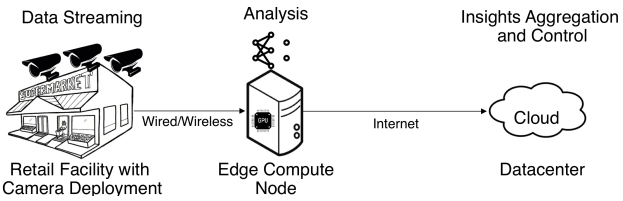

Fig. 1: A typical deployment of video-analytics

Therefore, we set out to explore how well we can harness the parallelism in a GPU to process multiple feeds simultaneously.

Our primary goal is to squeeze the most out of a given GPU; specifically in terms of the number of parallel inferences drawn on a large corpus of video feeds. We define processing to be real-time if inferences are drawn at a rate greater than the native frame rate. We declare that our design goal is met if the median inference rate for popular deep-learning tasks across a large number of video feeds exceeds their native frame rate.

Contributions: This work explores the opportunities available in scaling video analytics, when only a handful of GPUs are present at the edge computing node. We notice that existing techniques in isolation only provide marginal improvements. However, due to their complementary nature, they can be integrated to achieve significantly higher throughput. We harness this synergy into a comprehensive system `Tetris`, which performs active region identification across all video feeds and packs them into a condensed volume. We pass these condensed volumes through optimized CNNs and carefully orchestrated system pipeline – achieving substantial parallelism. In designing `Tetris`, we make minimal assumptions and consciously confine within the bounds of a production deployment.

Outcome: The final outcome of this work is captured in Figure 2. It shows the maximum expected throughput per video feed for multiple feeds, for an application involving detection/localization of people, with and without `Tetris`, Clearly, as the number of feeds increases, the relative gain with `Tetris` gets wider. Specifically, without `Tetris` we can expect to process 3, 4, and 7 feeds respectively at 30, 20, and 12 FPS, whereas at the same rates `Tetris` can handle 8, 16, and 36 feeds, yielding up to 5x gain at lower frame rates. How this outcome is achieved forms the crux of the paper.

II. Motivation

Video analytics on a GPU could be scaled in various ways. In this section, we present an exhaustive set of opportunities and discuss their merits and demerits. We harness their collective

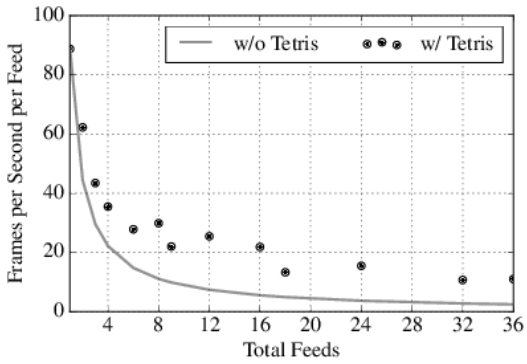

Fig. 2: Performance of the proposed system `Tetris` at varying numbers of total feeds processed, compared to the baseline system.

power in our final system, `Tetris`. Before we discuss all the opportunities, first we briefly describe the GPU related terminology and deep learning based inference process.

A. Background

Batching: A key aspect of GPU programming is ensuring that the GPU has sufficient data to operate. While there are many ways to maximize data rate to the GPU such as overlaying data transfers and computation using CUDA streams (explained shortly), larger transfers are usually handled more efficiently than smaller ones. Batching is a way to achieve this and involves taking several inputs to the deep model and combining them into a single batch. The batch is sent to and processed by the GPU in one fell swoop. Data in a batch is stored as a single contiguous array in memory; in the case of image data this requires each image to be of the same size.

Kernel: It is a piece of code that executes on the GPU. Kernels are designed to perform a specific task in parallel on multiple data. The kernels' work is split up into groups of threads called *blocks*. Each thread performs the same operation on a portion of the data. The number of blocks and threads per block are configurable by programmer and affect kernel performance. Kernels are launched by a CPU onto a CUDA stream.

CUDA Stream: A CUDA stream is a sequence of operations that operate in issue order on the GPU. These operations are either memory transfers to and from the GPU or kernel launches. By default, all CUDA operations occur in the default stream, but the user can choose to use different streams to achieve higher level concurrency. This most commonly allows memory transfers to be overlaid with computation, but it is also possible to overlay multiple kernel executions under the right circumstances, an analysis of which is presented in [1].

Network: A deep neural network model consists of a number of computational layers. A subset of these layers, most notably convolution and fully connected layers, contain intrinsic parameters (kernel weights and biases in the former, inner product weights and biases in the latter) which are learned during a supervised training period. In addition to these intrinsic, static parameters, neural networks also contain dynamic data to record the state of the intermediate calculations. These two sources, which we will refer to as network *parameters* and network *data* respectively, constitute the vast majority of the memory footprint of a deep neural network.

B. Opportunities

We now discuss various opportunities available for making video analytics scalable on a GPU. They are categorized as: 1) *multi-threading*; 2) *model pruning*; 3) *identifying active regions*; and 4) *batching active regions*. In each case, we provide some supporting preliminary experimental results.

All experiments shown below are performed on the Duke multi-target, multi-camera (MTMC) dataset [2], which consists of videos of pedestrians from 8 cameras on the Duke University campus. It contains more than 2000 unique pedestrians in a one hour time frame. We adopt FRCNN (Faster-RCNN) [3], which is a VGG-16 [4] network based object detector and is capable of localizing more than 20 distinct types of objects. The detector pipeline is implemented using Caffe and OpenCV. Hereafter when we refer to an *inference*, we mean a single forward pass on FRCNN yielding detection results for some (possibly batched) input. All experiments are conducted on HP Z840 workstation with 32 GB main memory and an Intel Xeon processor containing 10 physical (20 virtual) cores and a dedicated NVidia Titan XP GPU with 12 GB memory.

1) Multi-Threading: Directly using multiple CPU threads, each owning a copy of the deep model on the GPU, is not feasible due to the large memory requirement of a typical deep model. For instance, FRCNN detection model contains 235 million parameters, requiring up to 2.0GB of GPU memory for storing the model, input data, and intermediate states. Replicating deep-learning model across more than 8 threads, causes out-of-GPU-memory error. Figure 3 shows the number of analyzed frames per second, hereafter referred to as *inferences per second* (IPS), as a function of the number of threads, by applying FRCNN on the Duke dataset. Because each thread uses its own copy of the model, adding more threads to the GPU also adds more models, which end up vying for limited GPU resources. This contention results in a significant performance drop with each additional thread/model.

Shared Queue: A way to avoid unnecessary copies of network parameters and reduce memory requirement is to have CPU threads share a common deep learning model. While this approach scales in terms of memory, it enforces serial execution on the GPU. The total delay if n video feeds are present could be up to $t \times n$, where t is the inference time per frame. Figure 4 shows IPS when only one model is maintained in memory. Sharing deep-learning model allows a large number of threads/feeds at the cost of a significant drop in IPS.

Synchronous Model: A bottleneck of the above approach is that the queue enforces a serial behaviour. It is natural to ask why not invoke multiple synchronous calls on the same model. Most deep learning frameworks, including Caffe, maintain

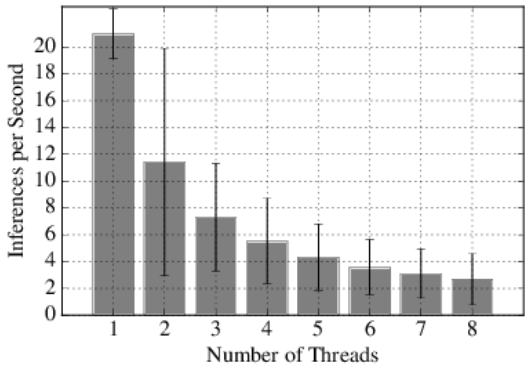

Fig. 3: Inferences per second (with 99% confidence interval) for FRCNN when running multiple deep learning model instances.

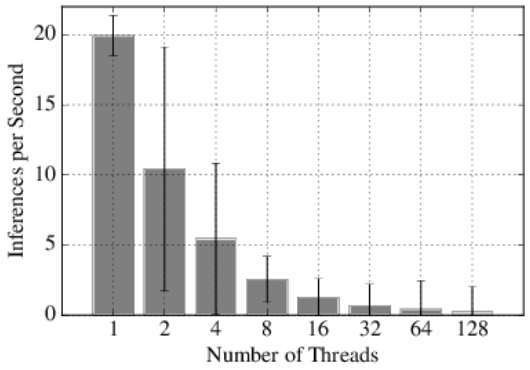

Fig. 4: Inferences per second (with 99% confidence interval) for FRCNN with multiple feeds using a shared deep-learning model.

intermediate data over the course of an inference, precluding this approach due to race conditions and data corruption.

CUDA Streams: CUDA streams provide a means to execute multiple GPU operations in parallel. In our case, each CUDA stream may load a copy of the deep-learning model – similar to the multi-threaded scheme. This approach is advantageous in that it simplifies the problem of receiving gains from CUDA streams, typically an in-depth process of finetuning specific kernel parameters within the scope of a single application. While this somewhat blunt take on CUDA streams may not guarantee optimal kernel concurrency, it opens the door for further concurrency between model executions. Like the multi-threaded scheme, our approach is primarily limited by the space needed to maintain a large number of model parameters in GPU memory – which in theory could be improved by sharing them across streams. However, the framework (e.g., Caffe) does not support this currently.

Fig. 5 shows that the inference time of FRCNN improves on average by 11% for an unpruned model with two CUDA streams. The improvement for the pruned variant (described shortly) is slightly greater, as its smaller kernels are more easily overlaid within the GPU due to their lighter use of GPU resources (CUDA cores/shared memory). We do not notice further gains beyond two CUDA streams as the available GPU resources saturate when more than two models are present.

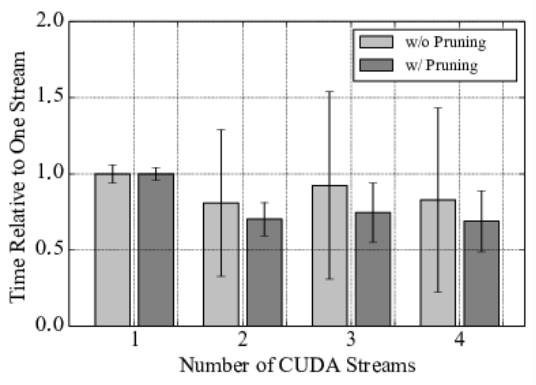

Fig. 5: Performance gains from CUDA streams for a detection with pruned and unpruned FRCNN. Note that the lighter weight pruned FRCNN benefits more from CUDA streams.

2) Model Pruning: The high accuracy of deep neural networks could be attributed to a large number of parameters present in the model. However, majority of these parameters are redundant and could be represented using sparse matrices, saving up to 95% of memory. Memory gains from the sparsity do not immediately translate to inference performance because sparsity imposes new overheads by reducing GPU parallelism, data locality, and uniform memory access [5]. Instead, pruning seeks to reduce the model's size by eliminating unneeded parameters through structured sparsity. This effectively replaces larger, sparse matrices with smaller dense ones, which can be efficiently processed on the GPU [6]–[8].

We use a pruned version of FRCNN as our base model. This pruned model consists of a VGG-16 head obtained using the pruning strategy discussed in [8]. In addition, we reduce the size of the final decision-making fully connected layers and relearn those parameters with finetuning. This latter change was motivated by the fact that we focus on single class detections (people only) while general FRCNN can handle over 20 different classes. The size of this pruned model is approximately 322 MB, about 6.35 times smaller than the unpruned model. We compare the performance of unpruned and pruned models on VOC 2012 validation dataset in Figure 6. It is evident that our pruned model, while saving memory, does not suffer any significant degradations in accuracy.

3) Active Regions: While analyzing the time taken to make an inference on a video frame, we find that a significant portion of the time in a typical deep inference is spent performing convolutions on the input image volume. Figure 7 shows the time spent on different components of the above pruned version of the FRCNN model. It is clear that convolutions take roughly 3 times longer than both the time for the fully connected layers and the time spent on the CPU.

Because the convolution operation scales linearly with the volume of the input image (width, height, and depth), its time can be improved by diminishing the area of these input images.

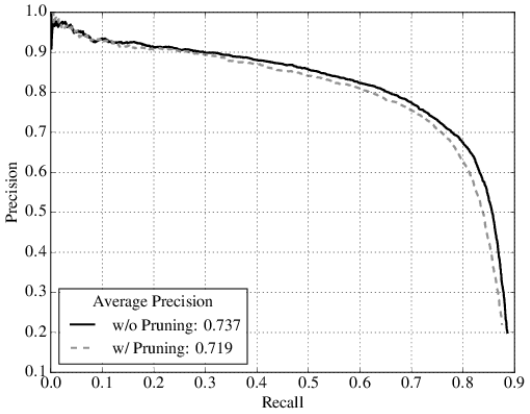

Fig. 6: Pruned version of FRCNN experiences a minimal drop in accuracy when evaluated on the VOC 2012 validation dataset.

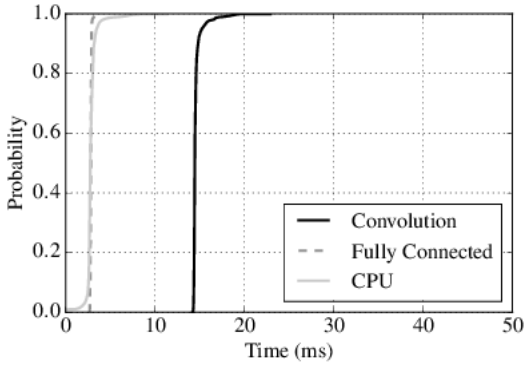

Fig. 7: Time taken to perform convolutions dominates the other times in our pruned version of FRCNN. Typically FRCNN convolutions take roughly 60% of the time while fully connected layers take 40%.

This can be efficiently accomplished in video analytics by limiting processing to only those regions of a camera feed that are of interest, namely, those containing activity. For a static camera, only the subset of a given video feed that is changing from one frame to the next will ever contain activity, meaning that we can limit our processing to only these regions. Note that active regions are generally noisy and do not guarantee the existence of an object of interest, hence the need for the deep inference. In practice, these regions are always significantly smaller than the original image, as shown in Figure 8, indicating that substantial gains are possible.

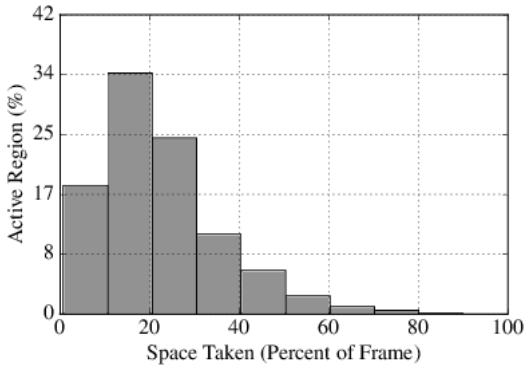

Fig. 8: Percent of the image covered by active regions

Such a method is only beneficial if active regions within a feed can be identified at little cost compared to the inference time. This can be accomplished with a computationally simple frame subtraction approach, where the elementwise difference in pixel intensities between consecutive frames is used to indicate activity. In practice, we often wait for several frames to pass before performing the subtraction in order to have a significant difference between subsequent frames. Differences are further accentuated through thresholding followed by erosion and dilation operations. The final active regions are found by taking the bounding boxes of convex hulls formed on these residual regions. These operations, being sufficiently lightweight to run at a faster rate than that of frames received from a typical live camera feed, do not impact the system throughput and cause only a negligible increase in the latency. Figure 9 compares the time to identify active regions with the inference time. It is apparent that active region detection time is insignificant.

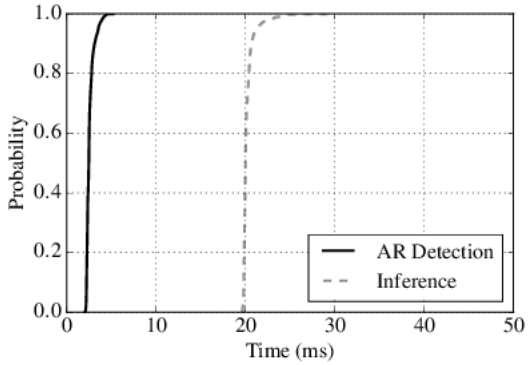

Fig. 9: CPU can identify active regions much faster than they can be processed by the deep inference model.

4) Batching Active Regions: The next question that arises is how to efficiently process multiple active regions. A naive implementation may simply perform a separate inference on each active region, possibly performing more than one inference per image but on a smaller combined area. For detector networks expecting fixed inputs, this approach can still be used by performing image warping on each active region. Figure 10 shows that this approach is mostly faster than not using active regions due to the reduced image area. A significant opportunity for improvement is present in this naive scheme, however. Because active regions are relatively small compared to the total image size, and because only one active region is processed at a time, the GPU is underutilized in this scheme.

The results of this underutilization are clearly illustrated in Figures 11 and 12, which show the distribution of active region sizes (areas) in the Duke dataset and our GPU performance for a range of image sizes, respectively. The GPU is only saturated with images larger than 128K (2^{17}) square pixels, meaning that images smaller than this have poor *pixel* processing times. These active regions may process at a faster rate overall than the entire frame due to their reduced size, but do not experience a superior *time per pixel* compared the larger alternative. Coincidentally, active regions in our dataset are

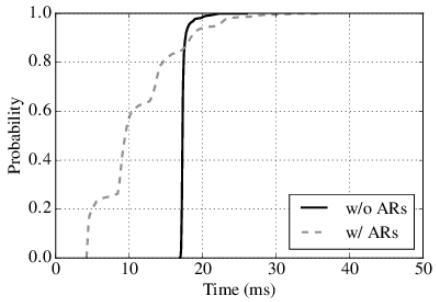

Fig. 10: On average it is faster to process active regions rather then entire image.

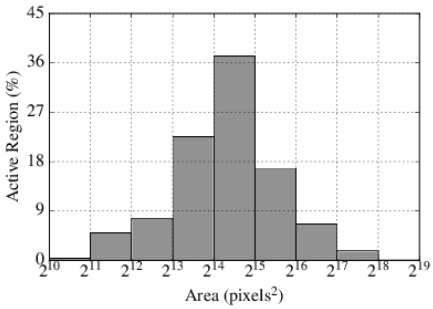

Fig. 11: Distribution of active regions areas.

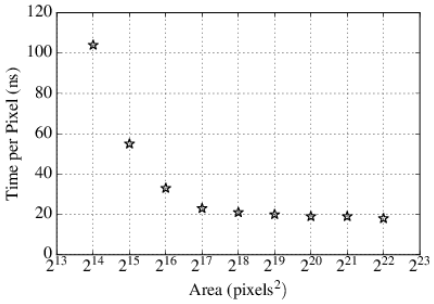

Fig. 12: Larger areas are in general processed more efficiently than smaller ones.

frequently much smaller than 128K (2^{17}) square pixels, having an average area between 4K (2^{14}) and 8K (2^{15}) square pixels. This size discrepancy results in a rarely fully-utilized GPU for the naive scheme, opening the door for greater improvements.

GPU can be better utilized by using batching to bundle multiple active regions together. This scheme is especially attractive when processing multiple video feeds, as this plurality of frame sources will better guarantee the existence of multiple active regions for batching. We consider two strategies for batching active regions: *stacking* and *tiling*.

Active Region Stacking: We define a stack of active regions as a 3D volume of slices placed one above the other as in a conventional batch. We cannot apply this traditional batching directly, however, as active regions are variably sized, preventing their placement within a contiguous swath of memory as is required for a batch. As such, each active region must be padded to a constant size (the size of the largest region) in order to be stacked. This arrangement wastes space proportional to the variability of the active region area for a given batch. The performance of this scheme is shown in Figure 13. While stacking performs better than the case without active regions, wasted space due to padding effectively negates any benefits from batching that this strategy could enjoy over processing active regions without batching.

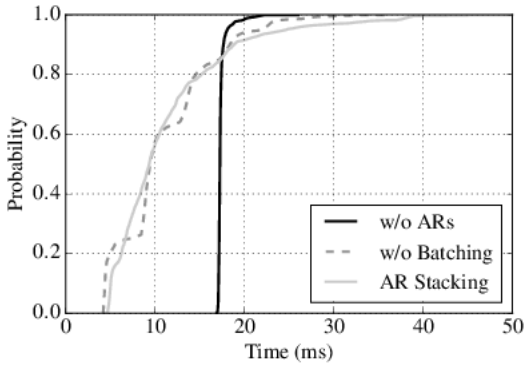

Fig. 13: Stacking active regions performs similarly to processing each active region independently without batching.

Active Region Tiling: The second strategy we consider for active region batching is tiling, where active regions are placed onto a single image surface like tiles onto a floor. The problem then is to find a tiling scheme that minimizes the amount of wasted space given nonuniform tiles. This can be done by formulating this task as a variant of the bin packing problem for two-dimensional objects. To this end, we use the first-fit decreasing algorithm [9] to find an approximate solution to the bin packing problem, which is NP-hard. In practice, this heuristic algorithm performs well and has been shown to use no more than 11/9 OPT + 6/9 bins, where OPT is the optimum number of bins for the packing [9]. For detector networks that support variable sized images, we impose no upper bound on the allowable bin size by dynamically resizing the bin during construction. For those that require a constant size input, we can fix the bin size to the expected dimensions and overflow the packing into additional bins if needed.

Figure 14 gives an example of a packed image slice using tiling method on frames from multiple video feeds. The performance of this strategy relative to the default and no batching approaches is shown in Figure 15. We observe that tiling is consistently superior to the other two.

Fig. 14: Example slice of a batch with bin packing.

To dig into the source of superior performance of tiling, we show in Figure 16 the distribution of wasted space for each active region strategy and the default case where active regions are not used. For the latter case we compare the active region area of each frame to the total frame size to measure space wastage. Unsurprisingly, the default scheme is the least efficient, as the majority of most frames contain relatively little activity, as seen previously. The scheme with no batching occupies the other extreme, as no space is wasted

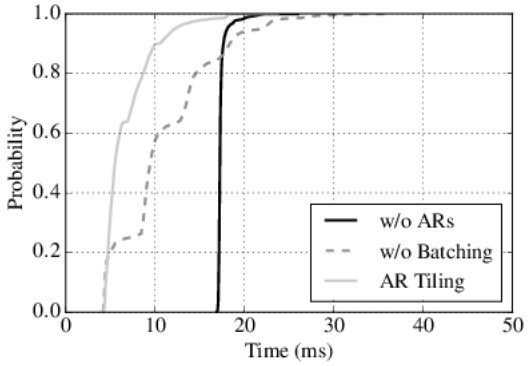

Fig. 15: Tiling active regions performs much better.

by performing separate inference on each active region. The two active region batching methods fall between these two, with tiling outperforming stacking. Based on these results, we employ tiling to batch active regions found in the video feeds.

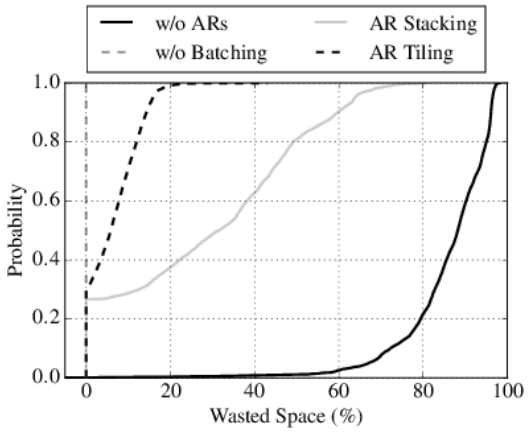

Fig. 16: Processing each active region independently incurs no wasted space, while stacking wastes much space. Tiling active regions is a good compromise between the two.

Summary: The essence of the opportunities for scaling video analytics on a singe GPU can be summarized as follows. Multi-threading is beneficial only if we can fit more than one model in GPU memory, i.e., only smaller models can benefit from multi-threading. CUDA streams can be used to provide performance benefits if multiple models are present in the GPU memory by putting each model on its own CUDA stream. Both these opportunities are made possible with pruning, particularly when dealing with large models. Pruning is a salutary practice if the drop in accuracy is almost negligible, which is possible with knowledge of the application. Batching multiple feeds is a good idea when each input volume is small, which is the case with active regions. Based on these observations, in the following, we design an efficient system for detecting people by leveraging the synergy in combining the techniques of pruning, multithreading, CUDA streams, and tiling of active regions. We refer to this system as `Tetris`, as it puts all these ideas together in a coherent manner.

III. System Implementation and Evaluation

In this section, we describe the implementation and evaluate `Tetris`. Our main aim is to assess throughput gains provided by `Tetris` in a variety of settings. A secondary, but no less important, goal is to determine how `Tetris` affects the accuracy of the underlying deep learning model. Specifically, we verify that our approach yields substantial throughput gains without a significant drop in application accuracy.

A. Implementation

In our implementation of `Tetris`, we consider the problem of object detection, which is a common primitive underlying many video analytics applications. We choose FRCNN [3] as our detector network, pruned as described previously, and extended to support batching. Several modifications are also done in Caffe in order to support CUDA streams. The network entity is given a reference to a CUDA stream to be set at runtime, while kernel launches, memory copies, and device synchronize operations are modified to be done on the CUDA stream being utilized by the corresponding network.

The flow of `Tetris` system is depicted in Figure 17. One or more camera feeds feed into the active region detector which in turn provides active regions to the bin packer. The bin packer appropriately records the transformation of each active region and sends the batch to the GPU, where it is processed by the deep learning model. Finally, the results are mapped back to their original feeds and positions for application specific processing. An elaboration of this workflow is shown below.

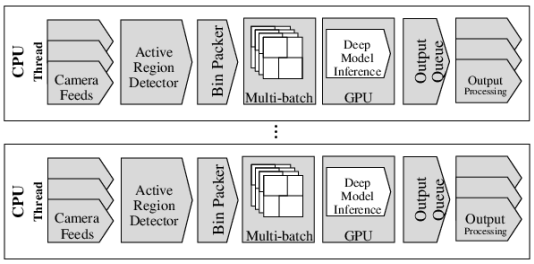

Fig. 17: `Tetris`: Video frames from multiple cameras are fed into the active region detector that identifies activity in each frame. Active regions are combined into a batch by the bin packer and sent to the GPU for the deep inference. After the deep inference, results are split up according to their origin feeds for application specific processing.

As we employ active region tiling, some processing is needed to reconstruct the correct output from a tiled input created by bin packing. This is accomplished by recording the transformation mapping each active region on the original frame to its position within the batch slice. Note that this transformation includes the identifier of the feed producing the active region in cases where more than one video feed is used.

At the end of the deep inference on the batch, the output slice is re-partitioned based on the active regions given as input. Each active region partition may contain one or more

location results. Spatially, these location results should be fully contained within their corresponding active region; those that encroach on other active regions are clipped appropriately. When a correspondence is established between each location result and its active region, the location result is transformed back into the space of the original image using the inverse of the transformation previously recorded. In the unlikely event that a location result spans two or more active regions to the extent that its true assignment is ambiguous, it is ignored. Although other strategies for handling this rare case are possible, we leave their exploration to future work.

B. Evaluation

Before we present our evaluation results, first we describe the experimental setup. All experiments are conducted on an HP Z840 workstation using the training/validation set of the Duke MTMC dataset. The videos from cameras 2 through 8 (we disregard camera 1 as parts of its video seem to be shaky) are each partitioned into three 10 minute segments yielding a total of 21 videos, totaling 3.5 hours. Each video is assigned an activity level to provide a sense of system performance under varying numbers of people in the scene. For simplicity, we only consider three levels of activity, low, medium, and high. Activity levels for each are determined by analyzing general intensities of the frame subtractor run over each video. These segments are then classified based on average activity into the three discrete bins. 11 videos are classified as low activity, 8 as medium, and only 2 as high activity.

One Thread Multiple Feeds

We first examine the throughput of our system when utilizing one shared model for multiple video feeds. In this scenario, each deep inference is performed on a tiled batch constructed from the current active regions gathered from all feeds. We compare our scheme to the base case, where a single model performs one inference per frame on a typical batch constructed from each feed without active regions (frames from each feed are stacked). To measure the performance of both cases, we use inferences per second made by the entire system rather than inferences per second as seen by each video feed, for a fair comparison. Figure 18 shows the result.

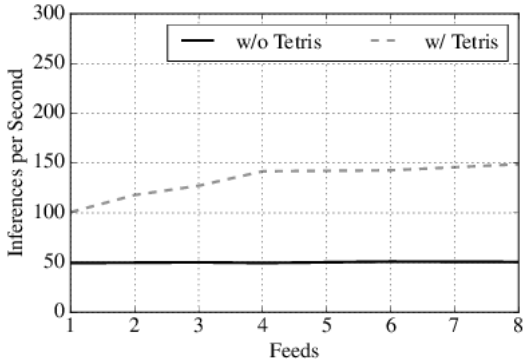

Fig. 18: Performance of `Tetris` with one thread and varying feeds.

The baseline version of the detection model has a constant throughput of 50 IPS even with varying number of feeds, indicating that it does not scale when handling multiple video feeds, which is to be expected since the GPU is saturated at one feed. `Tetris` begins at 100 IPS with a single feed thanks to active regions and gains by nearly 50% when processing four or more feeds, at which point GPU saturates. Overall, `Tetris` yields up to 3x throughput gain with a single thread.

Multiple Threads Multiple Feeds

We next assess system throughput in the case of two or more threads, each with its own model. Based on the previous result, we fix the number of video feeds per thread at four. This essentially means that each thread will be constructing tiled batches from active regions from four video feeds. Once again, we compare with the baseline system in the same configuration, with two or more threads each processing four video feeds. Each inference done in the baseline system is performed on a typical batch constructed by stacking each frame from all feeds. The throughput of these multithreaded configurations is measured with and without CUDA streams (when we do use CUDA streams, the model for each thread is assigned a unique stream) and compared in Figure 19.

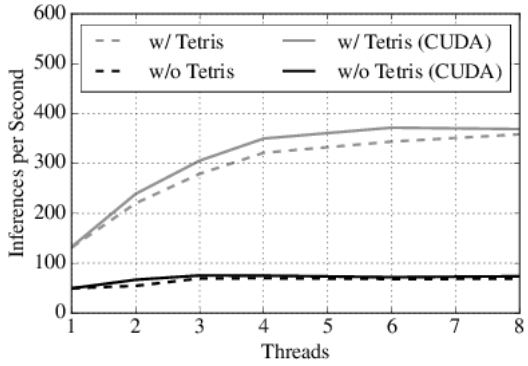

Fig. 19: Performance of system at four feeds and varying threads.

It is apparent that both systems benefit from having multiple threads. Gains here come from parallelism on the CPU, but more importantly, filling in idle gaps on the GPU created as other threads work on the CPU. Figure 19 also contrasts the performance with and without CUDA streams. Note that `Tetris` benefits from CUDA streams due to its reduced processing volume, which equates to smaller, more easily parallelizable, GPU kernels. The throughput of the baseline version goes up from 50 IPS to 80 IPS with 3 threads, whereas `Tetris` improves from 150 IPS to around 380 IPS with 6 threads, amounting to more than 4.5x gain over the baseline.

Levels of Activity

We now assess the relative performance of `Tetris` at 3 activity levels previously discussed. Figure 20 shows the average inference time for 4 video feeds from each activity level. Inference times without `Tetris` are relatively constant, as

the baseline system is only marginally influenced by the video content (these influences arise in the RPN of FRCNN). As expected, gain with `Tetris` is low at high activity level due to larger active region areas. Interestingly we see little difference between low and medium levels of activity. In practice, video activities mostly fall into medium and low activity levels, so the gains from `Tetris` hold in real deployments.

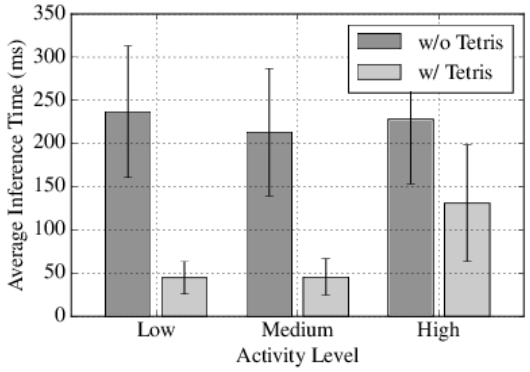

Fig. 20: Performance with and without Tetris at varying levels of activity at one thread and four feeds. Without Tetris, performance is relatively constant as it is not influenced by activity. Small differences present are due to the variable video content. With Tetris, greater activity results in larger inference time due to larger active regions.

Inference Accuracy

We evaluate the accuracy of our system on the medium activity videos, for a single thread and varying numbers of feeds, and compare it to that of the baseline. Figure 21 shows these results for 1 thread at 2 and 4 feeds, (top and bottom, respectively). We observe that the benefits of our system come at a slight cost in accuracy, which increases with the number of feeds per thread. This indicates that the accuracy loss is incurred by the active region packing process. Still, trading off no more than 5% drop in average precision for higher than 4x gain in terms of overall throughput is cost-effective for most applications.

The overall outcome of this work is summarized in Figure 2. Here, the baseline system is assumed to batch all current frames from each feed into a single deep inference. In case of `Tetris`, the best configuration of threads and feeds per thread is adopted for the given number of total video feeds. These results indicate that depending on the frame rates of the video feeds, `Tetris` can serve up to 5x number of feeds compared to the baseline, particularly at lower frame rates. Consequently, `Tetris` can yield substantial savings in deployment cost.

IV. Limitations and Discussion

Adverse Artifacts of Tiling: Frame subtraction and bin-packing modify the input volume in unexpected ways. The modified input volume could affect convolution feature map in the later stages. For instance, the conv5 layer of the VGG-16 network has a receptive field of 163×163 for an input size 227×227. Providing only a part of the video frame as input could

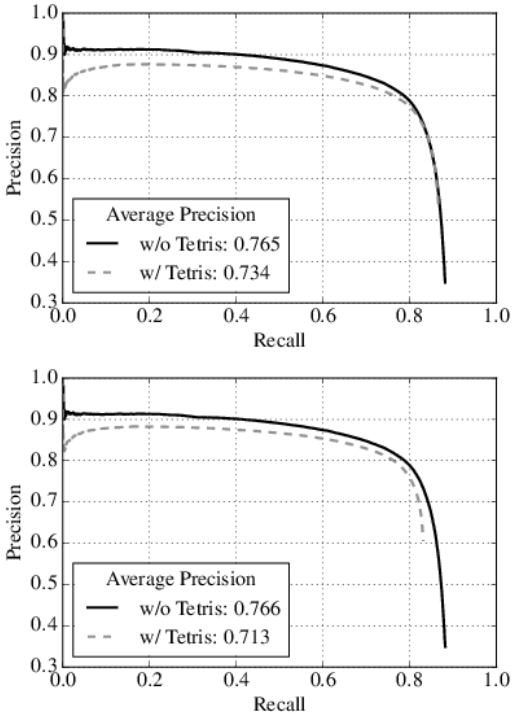

Fig. 21: Accuracy of baseline and our system for 1 thread at 2 (top) and 4 (bottom) feeds evaluated on medium activity videos. We observe that as the number of feeds increase, accuracy drops.

modify conv5 feature map, thereby affecting the subsequent stages. However, one could finetune the deep learning model on artificial images such as ours to overcome this problem.

Tradeoff between Inference Accuracy and Throughput: Our evaluation has shown that `Tetris` can yield more than 4x gain in inference throughput at the cost of 5% drop in inference accuracy. While we believe this is a worthy tradeoff, it is unclear whether this is optimal or desirable, as it depends on requirements of the applications. We need to investigate this in-depth to make a compelling case for `Tetris`.

V. Related Work

Ubiquitous real-time video analytics remains an open and exciting research problem, and is fueled by recent advances in hardware and deep learning. CNNs have been shown to outperform most existing computer vision techniques for classification [10], [11], detection [3], [12], and tracking problems [13], [14]. Many methods to optimize deep network training and inferences have been proposed. For instance, DeepCompression [15] uses Huffman encoding to substantially reduce the in-memory footprint of the network weights. BinaryConnect [16] and XNorNet [17] propose binary network weights to improve training and inference time on CPUs. DistBelief [18] is a framework to train large deep-learning models on a distributed cluster of CPUs. Nickel [19] is a framework by Nvidia to parallelize neural network training on

a set of GPUs. Our work is orthogonal to all these approaches and can be applied to further reduce the inference time.

Apart from improving accuracy and computation delay, several deployment models of video-analytics have been proposed. Cloudlet [20] aims to bring computation closer to IoT devices. VideoStorm [21] is a cloud-based framework to process a large number of queries on live video streams. ParkMaster [22] is an application of edge-based video-analytics to find free parking spots. MCDNN [23] is an approximate resource allocator that attempts to serve maximum number of heterogeneous video streams by trading-off accuracy for resource use. MCDNN poses resource allocation as an optimization problem amongst various DNN models while our work optimizes processing pipeline and DNN models to achieve the same task. Similar to our work, DeepMon [24] applies background subtraction and uses mobile device GPU to perform object classification. `Tetris` is also an edge-based video-analytics system; however, it is designed for real-time inferences on live video feeds, exploiting the parallelism on a GPU.

Scheduling policies of the CUDA library providing insights on execution of GPU kernels are exposed by [1] and [25]. [26] is a multi-camera video analytics system on the GPU, resembling `Tetris w/o binpack` in parts. SSL [5] is a framework to learn structured sparsity in a network that can be exploited by cuSparse library to improve inference time by up to 3x. We assessed the applicability of SSL to `Tetris` and found that, with a significant engineering effort, our performance could be further improved by 1.6x (applying a 3x speedup to convolutions, which account for 60% of the inference time).

A rich set of industries are emerging around video-analytics. Pilot AI is a start-up specializing in deploying deep-learning models on resource-constrained embedded hardware. Dextro [27] uses deep-learning to summarize multiple live video feeds. [28] derives real-time insights using video-analytics on surveillance cameras. Such deployments can adopt `Tetris` to improve the scalability of their systems.

VI. Conclusion

We present an approach to improve the scalability of video analytics at the edge by enabling a single GPU to perform deep inferences on a greater number of live camera feeds. Our approach includes batch processing of data using CUDA streams to increase concurrency between multiple feeds. It employs frame subtraction to detect active regions, paired with a tiling algorithm to efficiently condense these regions into a single batch. We demonstrate that this approach significantly improves the efficiency of processing multiple feeds on a single GPU compared to the current state of the art. Our evaluation with Duke MTMC dataset [2] shows average throughput gains of more than 4x by trading off 5% drop in inference accuracy. We plan to investigate this tradeoff further and apply `Tetris` to a variety of video analytics applications.

References

[1] N. Otterness, M. Yang, T. Amert, J. Anderson *et al.*, "Inferring the scheduling policies of an embedded CUDA GPU," in *OSPERT*, 2017.

[2] E. Ristani *et al.*, "Performance measures and a data set for multi-target, multi-camera tracking," in *Benchmarking Multi-Target Tracking*, 2016.

[3] S. Ren, K. He, R. Girshick, and J. Sun, "Faster r-cnn: Towards real-time object detection with region proposal networks," in *NIPS*, 2015.

[4] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *CoRR*, vol. abs/1409.1556, 2014.

[5] W. Wen, C. Wu, Y. Wang, Y. Chen, and H. Li, "Learning structured sparsity in deep neural networks," in *NIPS*, 2016.

[6] H. Hu *et al.*, "Network trimming: A data-driven neuron pruning approach towards efficient deep architectures," *CoRR*, 2016.

[7] X. Jin *et al.*, "Training skinny deep neural networks with iterative hard thresholding methods," *CoRR*, vol. abs/1607.05423, 2016.

[8] J. Luo, J. Wu, and W. Lin, "Thinet: A filter level pruning method for deep neural network compression," *CoRR*, vol. abs/1707.06342, 2017.

[9] G. Dósa, *The Tight Bound of First Fit Decreasing Bin-Packing Algorithm Is FFD(I) 11/9OPT(I) + 6/9*. Springer Berlin Heidelberg, 2007.

[10] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *NIPS*, 2012.

[11] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *CVPR*, 2016.

[12] W. Liu *et al.*, "Ssd: Single shot multibox detector," in *ECCV*, 2016.

[13] L. Bertinetto, J. Valmadre, J. F. Henriques *et al.*, "Fully-convolutional siamese networks for object tracking," in *ECCV*, 2016.

[14] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma *et al.*, "ImageNet Large Scale Visual Recognition Challenge," *IJCV*, 2015.

[15] S. Han *et al.*, "Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding," *ICLR*, 2016.

[16] M. Courbariaux *et al.*, "Binaryconnect: Training deep neural networks with binary weights during propagations," in *NIPS*, 2015.

[17] M. Rastegari, V. Ordonez *et al.*, "Xnor-net: Imagenet classification using binary convolutional neural networks," in *ECCV*, 2016.

[18] J. Dean, G. Corrado, R. Monga, K. Chen, M. Devin, M. Mao, A. Senior *et al.*, "Large scale distributed deep networks," in *NIPS*, 2012.

[19] Nvidia, "Nickel," https://github.com/NVIDIA/nccl, 2017.

[20] M. Satyanarayanan, P. Simoens, Y. Xiao, P. Pillai *et al.*, "Edge analytics in the internet of things," *IEEE Pervasive Computing*, 2015.

[21] H. Zhang, G. Ananthanarayanan, P. Bodik *et al.*, "Live video analytics at scale with approximation and delay-tolerance." in *NSDI*, 2017.

[22] G. Grassi, P. Bahl, K. Jamieson, and G. Pau, "Parkmaster: An in-vehicle, edge-based video analytics service for detecting open parking spaces in urban environments," in *IEEE/ACM SEC*, 2017.

[23] S. Han, H. Shen, M. Philipose, S. Agarwal, A. Wolman, and A. Krishnamurthy, "MCDNN: An approximation-based execution framework for deep stream processing under resource constraints," in *MobiSys*, 2016.

[24] L. N. Huynh *et al.*, "Deepmon: Building mobile gpu deep learning models for continuous vision applications," in *MobiSys*, 2017.

[25] R. Vuduc *et al.*, "On the limits of GPU acceleration," in *Proceedings of the 2nd USENIX conference on Hot topics in parallelism*, 2010.

[26] P. Guler, D. Emeksiz *et al.*, "Real-time multi-camera video analytics system on gpu," *Journal of Real-Time Image Processing*, 2016.

[27] Dextro, http://dextro.co/, 2017.

[28] Intelli-Vision, https://www.intelli-vision.com/, 2017.