Performance Evaluation and Improvement of Real-Time Computer Vision Applications for Edge Computing Devices

Julian Gutierrez
gutierrez.jul@husky.neu.edu
Northeastern University
Boston, Massachusetts

Nicolas Bohm Agostini
agostini@ece.neu.edu
Northeastern University
Boston, Massachusetts

David Kaeli
kaeli@ece.neu.edu
Northeastern University
Boston, Massachusetts

ABSTRACT

Advances in deep neural networks have provided a significant improvement in accuracy and speed across a large range of Computer Vision (CV) applications. However, our ability to perform real-time CV on edge devices is severely restricted by their limited computing capabilities. In this paper we employ Vega, a parallel graph-based framework, to study the performance limitations of four heterogeneous edge-computing platforms, while running 12 popular deep learning CV applications.

We expand the framework's capabilities, introducing two new performance enhancements: 1) an adaptive stage instance controller (ASI-C) that can improve performance by dynamically selecting the number of instances for a given stage of the pipeline; and 2) an adaptive input resolution controller (AIR-C) to improve responsiveness and enable real-time performance. These two solutions are integrated together to provide a robust real-time solution.

Our experimental results show that ASI-C improves run-time performance by 1.4x on average across all heterogeneous platforms, achieving a maximum speedup of 4.3x while running face detection executed on a high-end edge device. We demonstrate that our integrated optimization framework improves performance of applications and is robust to changing execution patterns.

ACM Reference Format:
Julian Gutierrez, Nicolas Bohm Agostini, and David Kaeli. 2021. Performance Evaluation and Improvement of Real-Time Computer Vision Applications for Edge Computing Devices. In *Companion of the 2021 ACM/SPEC International Conference on Performance Engineering (ICPE '21 Companion), April 19–23, 2021, Virtual Event, France.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3447545.3451202

1 INTRODUCTION

Computer Vision (CV) is a fast growing field of research in both academia and industry. Thanks to the latest GPU technology, we have seen impressive improvements in the performance and accuracy achieved by machine learning and deep learning (DL) algorithms [17]. These advances have spurred a resurgence of interest in CV applications. Self-driving vehicles [8] and vehicle surveillance [4] are two examples of emerging DL applications requiring real-time video processing, as shown in Figure 1.

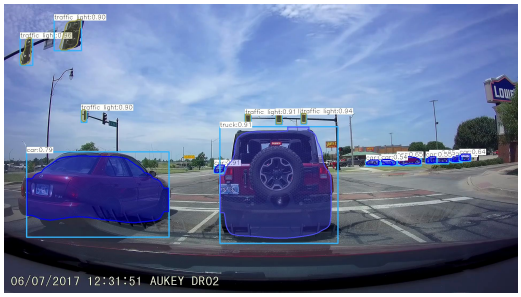


Figure 1: Object recognition using a Recurrent Convolutional Neural Networks implemented with Vega.

Real-time video processing places strict demands on a system's ability to process an input frame within a short time period, measured by the frames per second (FPS) rate. A video stream at 30 FPS translates to a processing time of 33.3 ms per input frame. Lower processing rates could result in the system dropping frames. Faced with the competing resource constraints in CV applications, the emergence of edge-computing makes this problem even harder. Previous work addressed this issue in different ways: 1) as *full systems* targeting an area of computer vision [13], 2) as *optimizations* to specific CV kernels such as [12], or 3) as full *Hardware-Software co-design* solutions [1].

Our work extends a full system solution called Vega [7], where we leverage the concepts of module replication [16] and introduce dynamic control that can replicate modules on the fly. We have selected this automated approach since we can enable inexperienced users to achieve better performance across a range of computational platforms, each possessing different compute capabilities.

We study the behavior of 12 applications implemented with Vega on four platforms. We then propose two optimizations to improve the responsiveness and performance across these devices: 1) an adaptive stage-instance controller (ASI-C) to improve performance by dynamically controlling the number of computing instances in each stage of the pipeline; and 2) an adaptive input-resolution controller (AIR-C) to improve responsiveness and enable real-time performance in hardware-limited scenarios.

2 DESIGN AND IMPLEMENTATION

Many CV algorithms are composed of a set of pipeline stages that execute in sequence to process a given image or frame from a video.

ICPE '21 Companion, April 19–23, 2021, Virtual Event, France

ACM ISBN 978-1-4503-8331-8/21/04…$15.00
https://doi.org/10.1145/3447545.3451202

Figure 2 shows an example of the flow through a set of processing steps required to run a deep learning-based face recognition on an input frame.

2.1 The Vega Framework

Vega constructs a directed acyclic graph (DAG) to represent the connections between different stages in the CV application. The DAG shows the concurrent execution of different stages across frames by using a unique thread to execute each stage. Figure 2 shows the structure of the face recognition application, including all the stages and queues in the framework. The implementation of the queue, input stage and output stage are embedded into the framework. This provides Vega users with an automated solution, leaving the user to only implement the custom nodes required by the custom application, as presented in Figure 2.

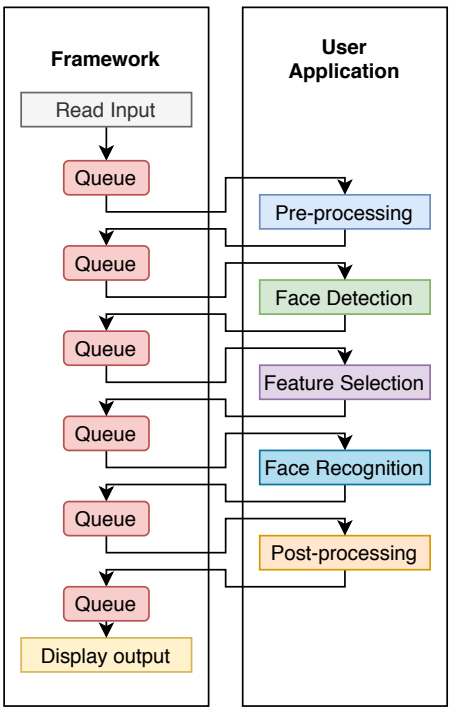


Figure 2: High-level pipeline for a face recognition algorithm using the Vega framework.

2.1.1 Queue. Our queuing mechanism is implemented as a ring buffer, and is used to manage synchronization between stages. Each stage, other than the first and last stage in the pipeline, is equipped with an input queue and an output queue.

2.1.2 Stage Instances. We refer to a copy of a pipeline stage as an instance of that stage. Multiple instances of a single stage enable parallel execution of the stage logic. A thread is launched for every stage instance. Vega provides the capability to add multiple stage instances to help hide the latency of slower stages.

2.1.3 Performance Metrics. Figure 3 shows an example of how the framework executes on a stream from a video camera, where frames arrive at a constant rate of 30 frames per second (FPS). In this example, all stages execute fast enough to meet the FPS rate of the input video. As long as each stage execution time is within this limit, the application is capable of achieving real-time response times. We use **FPS** as our main performance metric.

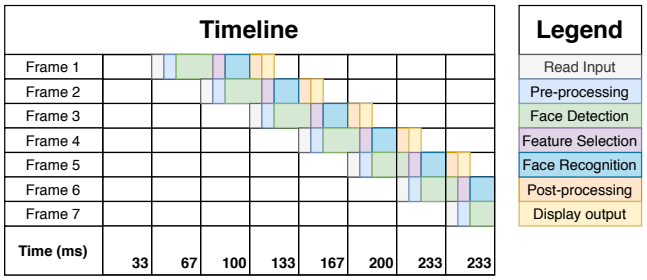


Figure 3: Execution timeline for our framework running a face recognition application, where all stages meet the FPS requirement.

2.2 Optimizations

In this section, we describe two new enhancements to the Vega framework. These enhancements modify Vega's run-time, allowing the user to leverage their benefits without changes to the user's code.

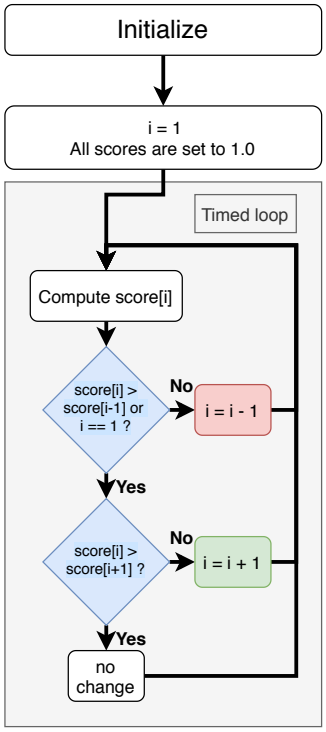


Figure 4: The flow of the Adaptive Stage Instance Controller, where *i* represents the current number of instances for the slowest stage in the pipeline.

2.2.1 Adaptive Stage-Instance Controller (ASI-C). This controller is capable of selecting the number of instances for each computing stage, based on monitoring performance dynamically as we increase the number of instances. A score, based on the observed FPS, is computed as we vary the number of instances. This score is used to control the number of instances, using the algorithm shown in Figure 4. The number of instances is increased if it results in a higher score. The controller considers thresholds for each comparison, allowing for small variations in the score due to dynamic changes in the run-time of the application. The maximum number of instances allowed is limited by the number of logical cores in the hardware, since it makes no sense to over-provision the available cores in a real-time application.

2.2.2 Adaptive Input Resolution Controller (AIR-C). This controller is in charge of adjusting the resolution of the input image to reduce the computation costs during later stages of the application. This results in an improvement in performance by reducing the latency and increasing the FPS rate[1]. We elect to use a PID (Proportional Integral Derivative) controller [2], which is a simple and effective method that can adjust to dynamic variations in a real-time system. By controlling each frame's input resolution, the PID controller adjusts the $FPS_{observed}$ to match the FPS_{input} of the input video. The output u_t of a basic discrete PID controller obeys the following equation:

$$u_t = K_p e_t + K_i T_s e_t + K_d \frac{e_t - e_{t-1}}{T_s}$$

Where u_t is the output of the controller at the t^{th} sampling instant, $e_t = FPS_{input} - FPS_{observed}$ is the value of the error at the t^{th} sampling instant, T_s is the time step (dt), and K_p, K_i and K_d are the proportional, integral and derivative gains, respectively. Figure 5 shows a block diagram of the PID controller used in this work. K_p, K_i and K_d gains can be manually tuned to fit the response time required by the user.

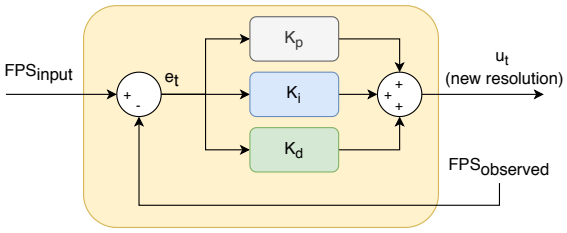


Figure 5: The basic operation of our PID controller.

2.2.3 Watch Dog State Logic. The controllers are run at fixed time intervals, as defined by the user. When both controllers are enabled, the watch dog enforces exclusion so that competing controllers do not lead to oscillations in performance. Figure 6 describes the state logic implemented for the watch dog, guaranteeing convergence of the controllers. We compare a long and short running average to compute the stability of the application.

State 2 and State 4 in Figure 6 describe states where one of the controllers has converged. In these two states, the margin of error to consider stable behavior is increased to avoid overreacting to small variations in the FPS, due to the dynamic nature of the application. The watch dog will first leverage the ASI-C to obtain the best possible FPS. It will then reach state 2, increasing the interval time to ensure stability has been achieved. Once the results are confirmed stable, state 3 executes the AIR-C. The AIR-C is used as the final controller because it can guarantee real-time performance, as long as the computations required by the algorithm scale with regards to the input image resolution.

[1]It should be noted that some CV applications do not benefit from this type of optimization, as some require a fixed input resolution (as discussed in section 3).

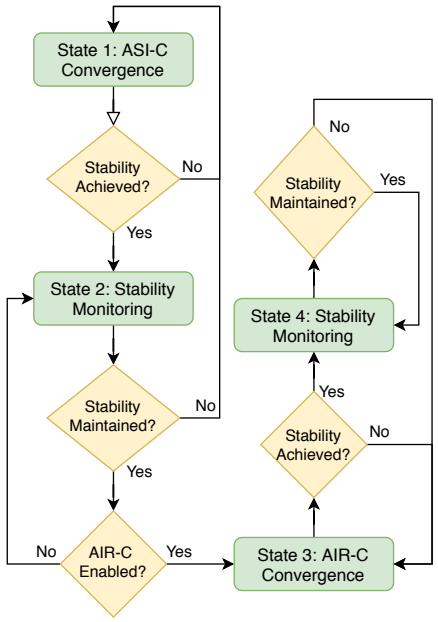


Figure 6: State machine logic used for the watch dog.

3 EXPERIMENTAL SETUP

Table 1: Platform hardware description.

Platform	Nano	Nx	Xavier	Desktop
GPU Cores (Gen)	128 (Maxwell)	384 (Volta)	512 (Volta)	2560 (Pascal)
CPU Logical Cores (Gen)	4 (ARM A57)	6 (ARM v8.2)	8 (ARM v8.2)	8 (i7 6700k)
Memory (GB and GB/s)	4 (25.6)	8 (51.2)	16 (137)	16 (25.6)
Perf (GFLOPS FP32)	235.8	1 058	1 410	8 873
Power (W)	10	15	25	~350

To evaluate our contributions, we utilized the platforms described in Table 1 and implement the experimental setup described in this Section. We selected algorithms that represent a wide range of CV applications, extracting all the deep learning baseline implementations from the Learn OpenCV's database [6], Dlib and OpenCV's open source algorithms.

The original version of the targeted application is referred to as the *baseline*. The baseline implements a sequential pipeline, processing one frame at a time. We create a version of the same application using the Vega framework and refer to it as the *vega-baseline*. This version implements a concurrent pipeline, which can overlap processing multiple frames. Finally, we enable the ASI-C controller on top of the *vega-baseline* and refer to it as *vega-opt*. The suite of application used in our analysis is summarized in table 2.

Table 2: Application description. NCS: Number of Compute Stages (excludes the input and output stages). DIR: Default Input Resolution. F: Fixed Resolution

Name	Description	NCS	DIR	Reference
od-ssd	Object detection algorithm using a MobileNet version of Single Shot MultiBox model (SSD)	3	300x300 (F)	[11]
od-rcnn	Object detection and semantic segmentation algorithm using the R-CNN model	3	1920x1080	[5, 15]
od-yolov3	Object detection algorithm using the YoloV3 model	3	416x416 (F)	[14]
td	Text detection algorithm using the East model	3	320x320 (F)	[19]
fd-opencv	Face detection (fd) algorithm using a ResNet version of the SSD model	3	1920x1080	[11]
fd-dlib	Face detection algorithm using the MMOD model	3	1920x1080	[9]
ad	Age detection algorithm using the MMOD fd model and a CNN model	4	1920x1080	[9, 10]
gd	Gender detection algorithm using the MMOD fd model and a CNN model	4	1920x1080	[9, 10]
fr	Face recognition (fr) algorithm using the MMOD fd model with a 128 feature descriptor and a Resnet model	5	1920x1080	[9]
pe	Pose estimation algorithm using OpenPose	3	368x368 (F)	[3]
vd	Vehicle detection algorithm using a CNN + MMOD model	3	1920x1080	[9]
color	Colorization algorithm using a CNN model	3	224x224 (F)	[18]

The input video used for all experiments consist of a total of 790 frames, with a native resolution of 1920x1080 and a frame rate of 30 FPS. The input resolution of the videos were scaled by 0.5 and 0.3 for the *Nx* and *Nano* platforms, respectively, in our evaluations. Each test was executed 5 times and averaged across runs.

4 PERFORMANCE EVALUATION

Next, we present experimental results of the Vega framework and our proposed optimizations. Working with deep learning frameworks, the accuracy of the models is paramount. But we omit accuracy results, as the ASI-C implementation does not change them. The *baseline* and *vega-opt* implementations provide the same recognition/detection outputs across all tests.

4.1 Stage Instance Increase Analysis

We use the *Desktop* platform to analyse the performance impact as we manually add more instances to the most time-consuming stages in each pipeline. Figure 7 shows how the speedup changes when we increase the number of instances for the compute stage(s). The *pe* and *od-ssd* benchmarks are the only applications to experience a decrease in performance when we increase the number of instances. Note that at two instances, the *od-ssd* algorithm does show improvement. All the remaining algorithms observed a speedup, reaching stable performance within 3-4 instances. Minor differences in execution time were observed when the number of instances was further increased.

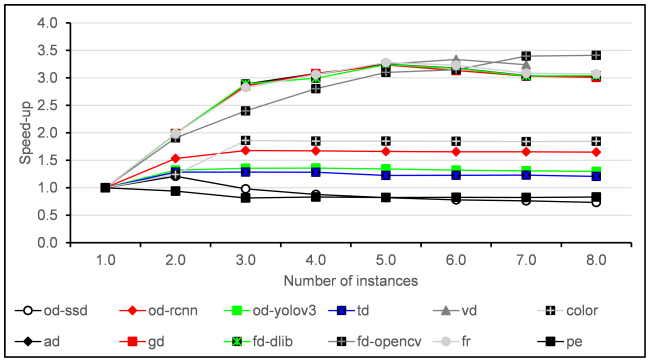


Figure 7: Performance comparison while varying the number of compute stage instances for each benchmark, as run on the Desktop platform.

Upon further analysis of the *od-ssd* and *pe* benchmarks, we observed a significant increase in the number of CPU context switches and CPU migrations as we increased the number of instances. Increasing the number of threads used by an application will naturally result in an increase in the number of CPU context switches and CPU migrations. *Od-ssd* with one instance produced 64.4 K CPU context switches and 600 CPU migration events. The same application run with four instances generated 7.9 M CPU context switches and 9 K CPU migrations.

The same experiments were conducted on the *Nx* platform. The lower performance of the *Nx* GPU limits the framework's ability to improve performance by adding instances. We achieve a 7% average reduction in execution time with two instances, after which performance only degrades as more instances are added. An increase in CPU context switching and CPU migrations was observed on this platform as well. We also observed a 90% (or higher) GPU usage across all benchmarks when using one instance. As a result, the potential performance improvement is limited by the GPU capabilities and the overhead associated with multi-threaded CPU scheduling.

4.2 ASI-C Performance Comparison

We compare the performance using our controller, with the best performance achieved by manually selecting the number of instances,

as shown in the previous section. This comparison quantifies the overhead associated with our controller. Figure 8 shows a comparison between the performance of our ASI-C controller and manually selecting the best number of stage instances.

For seven applications, *vega-opt* is within 8% of the hand-tuned run-time. This results in a small price to pay in performance when using ASI-C, as compared to manually testing across all possible number of instances. This controller additionally provides flexibility for scenarios where the performance fluctuates, as a result of the nature of real-time applications. ASI-C is capable of adjusting the number of instances accordingly, and continue to provide the best measured score. *Vd*, *Fd-dlib* and *fr* achieved lower performance because the ASI-C module was not always able to converge to the best solution.

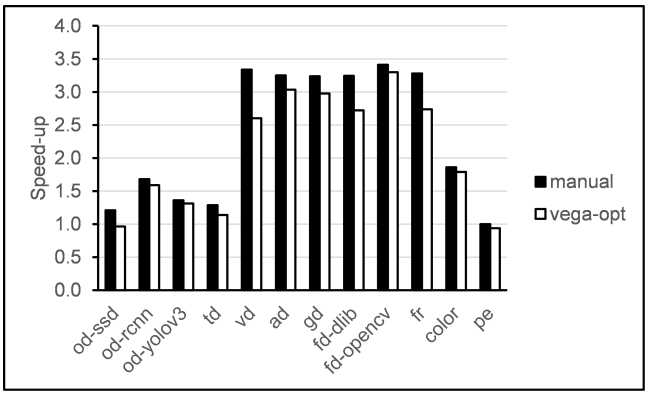


Figure 8: ASI-C speedup comparison to manual stage instance selection.

These fluctuations in the result of *vd*, *fd-dlib* and *fr* are caused by high variance in the stage execution times, resulting from the nature of multi-threaded benchmarks. These applications achieved the best manually-tuned performance with 5 or more stage instances. With *vega-opt*, the ASI-C will migrate to average lower FPS rate sometimes, resulting in a lower score for an otherwise good solution. This results in ASI-C converging to a sub-optimal solution. This problem can be mitigated by increasing the range of measurements taken to obtain the average FPS rate. However, this can have an adverse effect since it will take longer to compute the stability of the result.

4.3 AIR-C Performance Evaluation Under Load

Figure 9 displays the watch dog running with both optimizations enabled. We can see how the framework reacts when a task is added in the middle of the execution. A new CPU+GPU task starts at time point *A* in the execution. As expected, this new task impacts the overall FPS rate of the benchmark, as it is now sharing CPU and GPU resources. The watch dog notices the drop in the FPS rate and reverts to state 3, restarting the AIR-C. The AIR-C modifies the input resolution to compensate for the drop in performance, until stability is achieved. The new task completes at time point *B*, resulting in a release of the hardware resources and an increase in the FPS rate of our benchmark. This change restarts the AIR-C, enabling the input image resolution scale to be corrected to reach the original FPS.

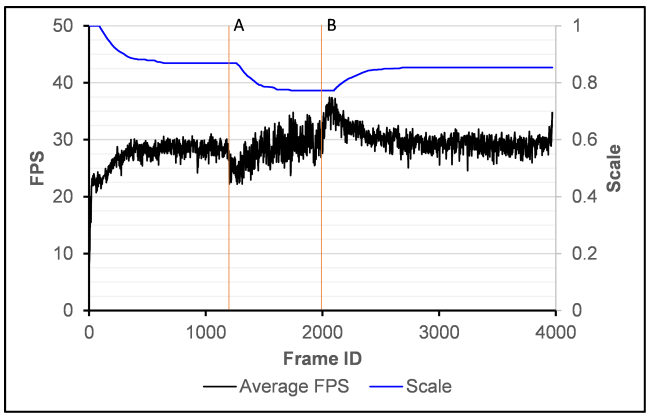


Figure 9: AIR-C performance evaluation for the *fd-dlib* face detection application on the *Desktop* platform.

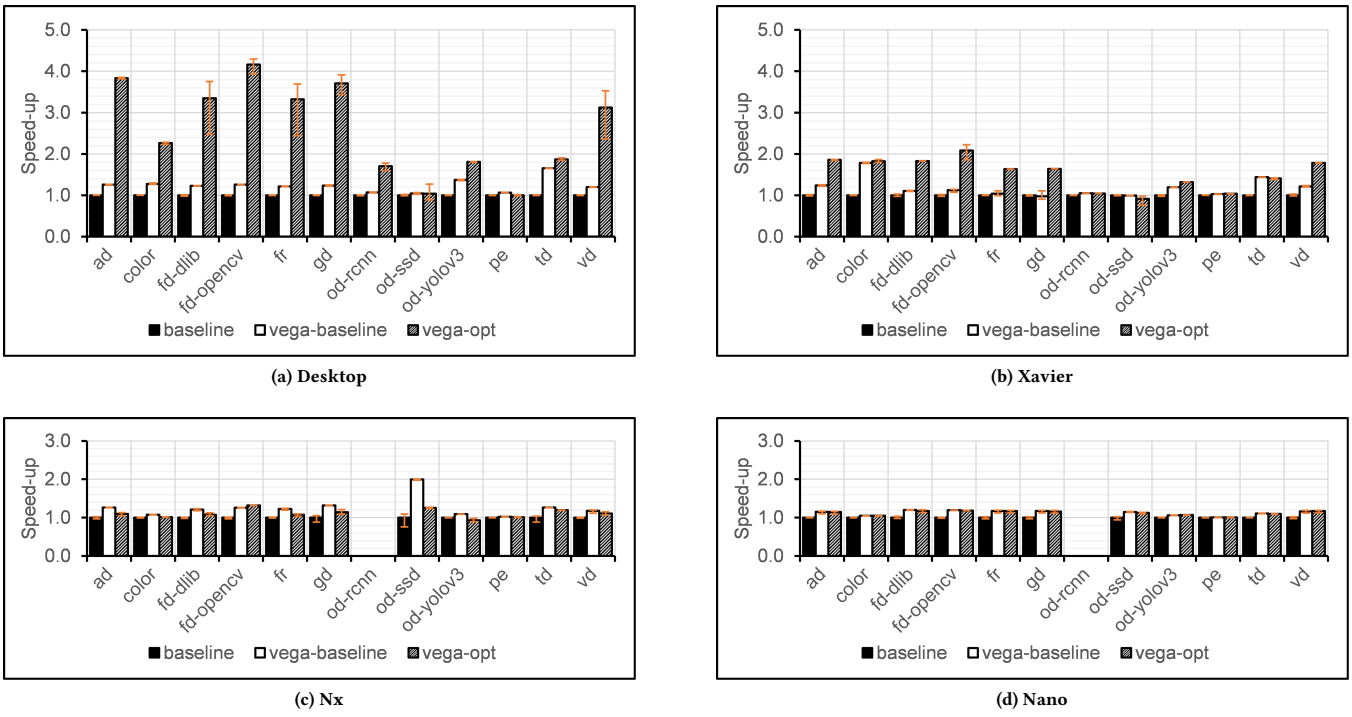
There is a delay between the moment the new task starts and the moment the AIR-C starts. This is because the watch dog logic is at state 4, where stability has been reached, so AIR-C is not running. The controller goes back to state 3 if the FPS rate changes by 15% or more.

4.4 Platform Performance Comparison

Figure 10 shows a performance comparison between the three implementations for each benchmark across all platforms. We observe high correlation between GPU usage and performance improvements when using *vega-opt*. For example, the GPU usage of the baseline code for *fd-dlib* on the *Desktop* represents 20.4% of the total execution time. Our optimized version resulted in 69.4% GPU usage, translating to a 3.4X increase in GPU usage, which tracks the performance gains we observed. The time spent on data transfers remains the same across all versions. This is true for the remaining applications.

Figure 10(a) shows the performance achieved with the *Desktop* platform. The *vega-baseline* achieved a 1.2x speedup on average. *Vega-opt* achieves a 2.1x speedup on average, with a maximum of 4.3x over the *baseline*. Pose estimation is the only application where *vega-baseline* performs better than *vega-opt*, requiring only one instance. Pose estimation reaches 98% GPU usage with *vega-baseline*, limiting the ASI-C capacity to improve performance.

Figure 10(b) shows the performance of the *Xavier* platform. Using *vega-opt*, we observed a 1.4x speedup on average, with a maximum of 2.2x. We observed similar trends on this platform compared to the *Desktop* platform. Figure 10(c) and (d) shows the performance of the *Nx* and *Nano* platforms, respectively. Using *vega-opt*, we observed a 1.1x speedup on average, with a maximum of 1.3x. The *vega-baseline* implementation performs better on average, compared to *vega-opt*, averaging 1.2x speedup. As expected, performance increases just by pipelining the algorithm, but the benefits of additional instances were not observed on these platforms. These results confirm our conclusions observed in Section 4.1.



(a) Desktop

(b) Xavier

(c) Nx

(d) Nano

Figure 10: Performance comparison of all of our platforms running the applications using the ASI-C controller (*vega-opt*).

5 CONCLUSION

In this paper we introduced ASI-C and AIR-C and applied them to CV applications, showing they can achieve significantly improved performance and enable real-time processing. Our study showed that combining these optimizations can further improve performance and responsiveness across a range of CV applications.

The performance impact of thread over-subscription on low-power devices is high, as the cost of scheduling and context switches dominate the weaker CPUs. By increasing the performance capabilities of the platform, the more easily performance can be improved by leveraging our framework optimizer. GPU usage was found to be a good indicator if an improvement can be achieved using ASI-C. If the GPU is already saturated by an application, adding more instances may result in reduced performance.

REFERENCES

[1] BK Anirudh, Vivek Venkatraman, Abhishek Rathan Kumar, et al. 2017. Accelerating real-time computer vision applications using HW/SW co-design. In *2017 International Conference on Computer, Communications and Electronics (Comptelix)*. IEEE, 458–463.

[2] Karl Johan Åström and Tore Hägglund. 1995. *PID controllers: theory, design, and tuning*. Vol. 2. Instrument society of America Research Triangle Park, NC.

[3] Zhe Cao, Gines Hidalgo, Tomas Simon, Shih-En Wei, and Yaser Sheikh. 2018. OpenPose: realtime multi-person 2D pose estimation using Part Affinity Fields. *arXiv preprint arXiv:1812.08008* (2018).

[4] Benjamin Coifman, David Beymer, Philip McLauchlan, and Jitendra Malik. 1998. A real-time computer vision system for vehicle tracking and traffic surveillance. *Transportation Research Part C: Emerging Technologies* 6, 4 (1998), 271–288.

[5] Ross Girshick, Jeff Donahue, Trevor Darrell, and Jitendra Malik. 2014. Rich feature hierarchies for accurate object detection and semantic segmentation. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 580–587.

[6] Vikas Gupta. 2018. Face Detection – OpenCV, Dlib and Deep Learning (C++ / Python). https://www.learnopencv.com/face-detection-opencv-dlib-and-deep-learning-c-python/

[7] Julian Gutierrez, Shi Dong, and David Kaeli. 2020. Vega: A Computer Vision Processing Enhancement Framework with Graph-based Acceleration. In *Proceedings of the 53rd Hawaii International Conference on System Sciences*.

[8] J. Becker et al. J. Levinson, J. Askeland. 2011. Towards fully autonomous driving: Systems and algorithms. In *2011 IEEE Intelligent Vehicles Symposium (IV)*. 163–168.

[9] Davis E King. 2015. Max-margin object detection. *arXiv preprint arXiv:1502.00046* (2015).

[10] Gil Levi and Tal Hassner. 2015. Age and gender classification using convolutional neural networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition workshops*. 34–42.

[11] Wei Liu, Dragomir Anguelov, Dumitru Erhan, Christian Szegedy, Scott Reed, Cheng-Yang Fu, and Alexander C Berg. 2016. Ssd: Single shot multibox detector. In *European conference on computer vision*. Springer, 21–37.

[12] Yury A Malkov and Dmitry A Yashunin. 2018. Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs. *IEEE transactions on pattern analysis and machine intelligence* (2018).

[13] Kari Pulli, Anatoly Baksheev, Kirill Kornyakov, and Victor Eruhimov. 2012. Real-time computer vision with OpenCV. *Commun. ACM* 55, 6 (2012), 61–69.

[14] Joseph Redmon and Ali Farhadi. 2018. Yolov3: An incremental improvement. *arXiv preprint arXiv:1804.02767* (2018).

[15] Shaoqing Ren, Kaiming He, Ross Girshick, and Jian Sun. 2015. Faster r-cnn: Towards real-time object detection with region proposal networks. In *Advances in neural information processing systems*. 91–99.

[16] Jaspar Subhlok and Gary Vondran. 1996. Optimal latency-throughput tradeoffs for data parallel pipelines. In *Proceedings of the eighth annual ACM symposium on Parallel algorithms and architectures*. 62–71.

[17] A. Voulodimos, N. Doulamis, A. Doulamis, and E. Protopapadakis. 2018. Deep Learning for Computer Vision: A Brief Review. In *Computational Intelligence and Neuroscience*.

[18] Richard Zhang, Phillip Isola, and Alexei A Efros. 2016. Colorful image colorization. In *European conference on computer vision*. Springer, 649–666.

[19] Xinyu Zhou and Cong et al.n Yao. 2017. East: an efficient and accurate scene text detector. In *Proceedings of the IEEE conference on Computer Vision and Pattern Recognition*. 5551–5560.