Vega: A Computer Vision Processing Enhancement Framework with Graph-based Acceleration

Julian Gutierrez
Northeastern University
gutierrez.jul@husky.neu.edu

Shi Dong
Northeastern University
shidong@ece.neu.edu

David Kaeli
Northeastern University
kaeli@ece.neu.edu

Abstract

The popularity of Computer Vision (CV) algorithms has been on the rise given their growing dependence on machine learning and deep neural networks. The resulting improvement in inference accuracy has revolutionized a number of fields. However, given that CV algorithms consist of many different stages, each having different computing characteristics, their execution is frequently irregular and inefficient, unable to leverage the full potential of the computing platform. Presently, supporting real-time video processing for high resolution images on edge systems involves a significant amount of programming effort and performance tuning.

To overcome this challenge, we present Vega, a parallel graph-based framework that enables better utilization of multi-core edge computing platforms. Vega provides a highly flexible and user-friendly interface to execute a range of CV algorithms efficiently, leveraging multiple external libraries for performance. First, Vega maps independent stages of a CV algorithm to nodes in a pipeline graph. Next, it dynamically schedules nodes on a multi-core CPU using multi-threading.

From our experimental results, our framework improves performance of all selected algorithms by at least 1.75x and up to 4.82x on the same platform. We analyze the impact of using our framework in terms of hardware utilization, frame processing latency and throughput.

1. Introduction

Computer Vision (CV) and Image Processing (IP) are both active research fields in both academia and industry. While they have been studied extensively in the past, they enjoy renewed interest thanks to emerging applications including the Internet of Things (IoT) [1], self-driving vehicles [2] and virtual/augmented reality [3]. Each application domain leverages tools from a rich collection of CV algorithms, focusing on a specific area (e.g., face detection, image segmentation, image classification, object recognition and etc.). The goal in most algorithms is to extract useful information from an image with high accuracy. With advances in machine learning algorithms, and deep learning in particular, CV researchers are able to achieve new levels of accuracy using smart CV algorithms, outperforming conventional approaches [4].

Most CV algorithms consist of multiple stages, steps that form a image/frame processing pipeline. A typical CV pipeline includes stages for signal reconstruction, feature extraction and contextual understanding [5], processing a given image or video input. Each stage may perform one or more image processing tasks, transforming an input into the desired format that is fed to the next stage. To shorten the time to build multi-staged CV applications, computer vision libraries, e.g., OpenCV [6] and VLFeat[7] have been developed, providing a rich collection of algorithms with user-friendly programming interface.

To accelerate CV algorithms, previous studies have focused on accelerating stages that are highly data-parallel and compute intensive, mapping execution to multi-core processors [8, 9, 10]. However, these approaches have two limitations when processing a corpus of images (i.e., a video). First, the class of algorithms used typically involve stages that are I/O intensive versus compute intensive. When we execute multiple stages in sequence, some stages may suffer low utilization on a given platform, leading to imbalanced in resource utilization across stages. Second, accelerating compute-intensive stages can only shorten the processing latency of a single image, and can limit our ability to improve the throughput when processing a continuous stream of frames.

In this paper, we propose Vega, a computer vision processing framework equipped with graph-based acceleration. The proposed framework addresses the above limitations by leveraging a graph-based scheme to parallelize the processing of a video, while in addition, accelerating individual compute-intensive stages. By

URI: https://hdl.handle.net/10125/64560
978-0-9981331-3-3

decomposing an application into clearly defined set of stages, we are able to generate a graph, with stages as nodes and data flow dependencies as edges. Independent nodes can be executed in parallel for different frames, effectively leveraging thread-level parallelism available on a multi-core processor.

Our framework provides a set of user-friendly APIs, allowing CV applications to be mapped to parallel threads. This framework can be applied to image processing algorithms such as face detection, object detection and face recognition, or could be generally applied to other multi-staged algorithms. To demonstrate the framework, we present a case study focusing on face detection on a streaming video. To evaluate our proposed framework, we conduct a series of experiments across 6 CV algorithms, including face detection and object detection, run on a multi-core CPU. The results show that our framework can achieve up to 4.82x speedup and 3.03x speedup on average over a single-threaded baseline. We characterize utilization and develop a new scoring scheme that balances latency with throughput. We also discuss a number of trade-offs to consider when faced with hardware constraints when using our framework.

2. Related Work

The graph-based data flow processing frameworks are not new. Spark [11], a Big Data processing framework, generates acyclic data flow graphs which can later be analyzed for operator execution. Tensorflow [12] adopts a very similar approach as that in Spark, creating an acyclic graph based on the defined data flow, and then executing the graph using a session manager. The former is aimed at parallel processing on a distributed system for Big Data, and the latter mainly focuses on Deep Learning algorithms. OpenCV [6] also supports a graph-based implementation (G-API), which was added in version 4.0 to achieve better device utilization for a single algorithm, without the added benefits of a parallelized pipeline. Huang et al. propose a general approach to schedule multiple tasks in parallel by analyzing a task dependency graph [13]. Their work proposed an efficient parallelization method, but requires significant effort from the programmer in order to implement CV algorithms. Our framework simplifies the final implementation, making it easier for a user to implement their own CV algorithms.

Acceleration for computer vision algorithms is an active area of work in both academia and industry. OpenVX [14] is a computer vision library providing accelerated algorithms; VisionWorks [15] provides tools developed based on the OpenVX standard for computer vision (CV) on NVIDIA GPUs; OpenVINO [16], developed by Intel, maximizes performance of deep learning inference on multiple Intel platforms.

These frameworks focus on optimization of Deep Learning models, which translates to better overall performance of computer vision applications. Gutierrez proposed a parallel processing mechanism for real-time face detection leveraging both CPU and GPU [17]; Teichmann et al. propose a unified architecture, in which encoders are shared by different tasks to shorten inference time for autonomous driving [18]; Amamra et al. take advantage of the power of a GPU to enable real-time RGBD (red, green, blue and depth) data filtering [10]. All of these frameworks provide improvements in performance, but they do not implement user-friendly interfaces that increase portability of CV applications.

Scanner [19] focuses on processing video collections on cloud computing platforms and providing a robust set of tools. Their main focus is to parallelize data mining and video processing across multiple compute nodes. In contrast, our framework is tailored towards real-time performance of CV algorithms on edge computing platforms. We also focus on programmability, providing a user-friendly and high performance framework to deploy CV algorithms using multi-core CPUs.

3. Framework

Computer Vision algorithms have a common pattern: they are composed of a set of stages that execute in sequence to process a given image. Figure 1 shows an example of a sequence of high-level operations required to run a deep learning face detection algorithm on an input frame.

The first stage reads the input frame from a video file or camera. The second stage transforms the input frame into the data format compatible with a selected deep learning model. The third stage performs inference using the model. The fourth stage reads the output from the model and updates the final image with the bounding boxes of the detected faces. Finally, the fifth stage displays the final results in a window.

This is a common approach for many face detection applications, and our primary target in this paper. We reduce the complexity of the implementation by constructing the graph in the same order as stages are added in the pipeline. This removes the requirement for the user to explicitly add edges between the nodes in the graph. We additionally exploit the shared memory model provided by the C++ thread library. This model facilitates data access across threads. Our framework was developed with edge computing systems

in mind, and does not provide cluster level support. Our graph-based approach is lightweight and easy to incorporate into existing projects.

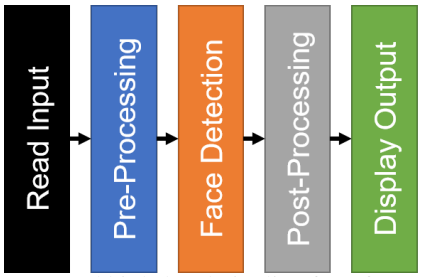


Figure 1: General high-level pipeline for a face detection algorithm.

3.1. Graph Definitions

We construct a directed acyclic graph to represent the connections between different stages in the CV algorithm. The pipeline allows us to process different stages for different frames concurrently. Figure 2 shows the structure of the face detector application, including all the stages and queues in the framework. The implementation of the input stage and output stage are embedded into the framework in order to provide flexibility for generating customized nodes on the user plane, as presented in Figure 2. The framework provides an API to launch the execution of the graph using multi-threading, while managing the completion of each node.

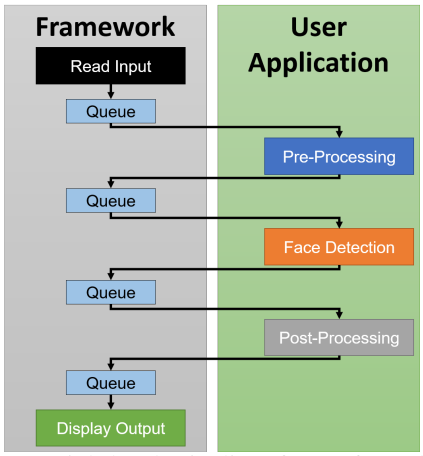


Figure 2: High-level pipeline for a face detection algorithm using the Vega framework.

3.1.1. Nodes Each node in the graph represents one stage, and each edge between two stages is represented by a lock-based queue. The node contains information such as the input and output queues, stage name, the node id and the number of instances. Depending on the specific task, one stage/node can contain more than one instance, which can be scheduled to run concurrently, improving performance for a single stage.

3.1.2. Queue A queuing mechanism is used to manage coherency and synchronization between stages. Synchronization for insertions and deletions in queues is done through mutex locks and conditional variables. For example, if a stage tries to retrieve data from its input queue and the data is not ready, the execution of this stage will suspend until data has been inserted into the queue by the previous stage. Each stage, other than first and last stage, in the pipeline is equipped with an input queue and an output queue. The implementation follows a producer-consumer pattern, supporting multiple producers and consumers associated with different stages. We avoid data transfers between threads by simply manipulating pointers.

3.2. Stage Instances

The stage object contains information similar to that of the node, with the difference that the stage is in charge of the parallel execution of the work. For every stage instance, a thread is launched to execute the main function of the stage. Different stages can inherit the properties of the base stage in order to execute different functionality. Each stage has an *initialization* phase, allowing the user to define any pre-stage requirements (e.g., loading the deep learning model weights) before the start of the execution.

The framework provides all the necessary I/O functionality by using predefined stages. The input stage provided by the framework is designed to read the input data and process any user input during execution. The framework also provides the output stage to display general information of the execution, display the output frames, store log files with the results, and take snapshots during the execution. Additional stages are provided for simplicity for the user, such as pre-processing and post-processing stages, where the user is only required to specify the execution function, without the need to create a new object.

3.3. Data Packet

The data packet contains all the necessary information for the execution of the algorithm on every frame. It contains the input frame in an OpenCV Mat object, a Mat object for the processed frame, and a Mat object for the output frame. All Mat objects

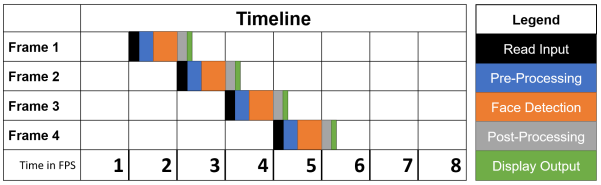


Figure 3: Timeline of the framework for the face detector algorithm, where all stages execute faster than the FPS requirement.

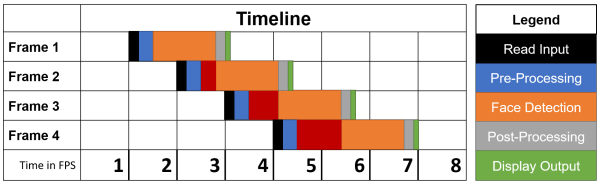


Figure 4: Timeline of the framework for the face detector algorithm, where one of the stages violates the FPS limit, causing a delay in the processing of subsequent frames.

are highly efficient, and if they are not used, the data required to store them is minimal.

A package type is used to define which type of input is being processed. IMG, VID and CAM are used to indicate the type of the package being processed, an image, video or a video from a camera, respectively. Timing variables are used to monitor the execution time for each stage and the total execution time for each frame.

3.4. FPS and Latency Constraints

Figure 3 shows an example of how the framework executes on a stream from a video camera, where the FPS is limited by the hardware. In this example, all stages execute under the frames per second (FPS) quality of service limit (FPS limit). The FPS limit given by a video is equal to 1000/FPS (in ms.). For example, a 30 FPS input video sets the maximum execution time for a stage to 33 ms. If all stages take less time than the FPS limit, the application is capable of running in real-time. If the input video is a file, the framework is capable of running faster than the input video rate, as long as the previous constraint is met.

If one of the stages is slightly over the FPS limit, this will impact the FPS of the entire application. For example, if a single stage takes 50 ms, the maximum FPS achievable by the application is 20 FPS. Figure 4 shows an example when this occurs in the third stage of the compute pipeline. Therefore, it is important that our application is capable of handling all the stages while maintaining this throughput rate. If hardware resources are underutilized, we can reduce the impact of slow stages in the pipeline by instantiating multiple copies of a stage. This gives the user the flexibility to improve the FPS rate of an implementation without having to compromise in terms of accuracy of the algorithm.

On the other hand, latency is another concern for real time processing. In a real-time scenario, the human eye is incapable of identifying delays in videos under 100 ms [17], which means, the total execution time should remain under this limit. Adding more instances can adversely affect the latency to process a single frame. This effect will be studied in Section 6.

3.5. Terminal Interface

The input stage is in charge of handling the terminal interface. It uses the nCurses library to easily process inputs provided by the user during the execution of the program. nCurses is a commonly used library for graphical interfaces, but minimizing the cost of displaying information was key to reduce the overhead of the framework. The interface displays general information about the execution, including the frame size, the frame rate and the current run-time.

This terminal also displays the data for the execution times of each stage if debug or verbose options were in the initial command. The framework additionally supports taking snapshots, recording video at any instance, and displaying the input and/or output video using an OpenCV window instance.

3.6. Additional Features

With the addition of multiple instances for a given stage, a reorder buffer was added to a quality of service (QoS) post processing stage. The reorder buffer guarantees the order of frames in the output will be the same as the input. This is necessary due to the non-deterministic behavior of threaded applications on a CPU. The program is unable to determine precisely when a thread has finished processing a frame. Multiple instances of the same stage can process frames with highly variable execution times, impacting the order of data packets in the output node.

The framework also provides support for CPU utilization measurements (supported on Linux) in order to provide an estimate of the hardware utilization and effectiveness of the framework. Fully utilizing the CPU hardware should not be a measure of efficiency, as it is better if we are able to process the video in real time without exhausting all the available hardware resources.

4. Case Study: Face Detection Implementation

Next, we discuss the implementation for our case study face detection algorithm. This algorithm uses the Caffe version of the SSD deep learning model described by Liu et al [20], running on the optimized OpenCV Deep Learning framework [21] for inference. The OpenCV Deep Learning framework uses a multi-threaded implementation, allowing the user to define the number of threads the framework can instantiate.

This case study demonstrates how we can use the proposed framework efficiently, as well as how easy it is to port an existing algorithm into the framework. The steps shown with this case study are representative of the general steps required to port other algorithms.

The first step required in the implementation of the algorithm using Vega is to expand the Data object with the data we will use to store the bounding boxes, as shown in the code snippet below.

```
class FaceDetectorData : public Data{
    public:
        vector<vector<int>> bboxes;
};
```

Next, we need to define what each stage in the algorithm will execute. The pre-processing stage for this algorithm executes normalization, and shifting by the mean value of the training dataset on the input frame. The output will be stored on the processed frame, which will be the input to the next stage. This behavior can be defined in the *execute* function of the pre-processing stage instance using the OpenCV contrib DNN API.

```
template <class Data>
void Preprocessing<Data>:: execute(Data*
 ↪  Packet){
    double inScaleFactor = 1.0;

    cv::Size size =
     ↪  cv::Size(Packet->frame.cols,
     ↪  Packet->frame.rows);

    cv::Scalar meanVal = cv::Scalar
     ↪  (104, 177, 123);

    Packet->processed_frame =
    cv::dnn::blobFromImage(
     ↪  Packet->frame, inScaleFactor,
     ↪  size, meanVal, false, false);
}
```

The Detector stage is the most complex stage, as we need to create a new stage object that inherits the properties of the Stage base class. We define the new class in the following snippet. We provide all the new variables and objects the stage requires to execute, as well as define the behavior for the *initialize* function (load Caffe model) and *execute* function (i.e., run inference). All of the data that the algorithm will require is in the data object, which can be accessed using the Packet pointer. Note that the Caffe model used in this case study was trained using 300x300 images, but we use a 1080p input frame to expose the algorithm's efficiency when processing full HD resolution images.

```
template <class Data>
class Detector : public Stage<Data>{
 public:
  const int num_threads =
   ↪  cv::getNumberOfCPUs();
  const std::string caffeConfig =
   ↪  "./models/deploy.prototxt";
  const std::string caffeWeight =
   ↪  "./models/res10.caffemodel";
  double conf_threshold = 0.7;
  cv::dnn::Net net;

  void initialize(){
    net = cv::dnn::readNetFromCaffe
     ↪  (caffeConfig, caffeWeight);
    net.setPreferableBackend
     ↪  (cv::dnn::DNN_BACKEND_OPENCV);
    net.setPreferableTarget
     ↪  (cv::dnn::DNN_TARGET_CPU);
    cv::setNumThreads (num_threads);
  }
  void execute(Data* Packet){
    net.setInput
     ↪  (Packet->processed_frame,
     ↪  "data");
    cv::Mat detection =
     ↪  net.forward("detection_out");
    cv::Mat detectionMat
     ↪  (detection.size[2],
     ↪  detection.size[3], CV_32F,
     ↪  detection.ptr<float>());

    for(int i = 0; i <
     ↪  detectionMat.rows; i++) {
      float confidence =
       ↪  detectionMat.at<float>(i, 2);

      if(confidence > conf_threshold) {
        int x1 = static_cast<int>
         ↪  (detectionMat.at<float>(i,
         ↪  3) * Packet->frame.cols);
        int y1 = static_cast<int>
         ↪  (detectionMat.at<float>(i,
         ↪  4) * Packet->frame.rows);
        int x2 = static_cast<int>
         ↪  (detectionMat.at<float>(i,
         ↪  5) * Packet->frame.cols);
        int y2 = static_cast<int>
         ↪  (detectionMat.at<float>(i,
         ↪  6) * Packet->frame.rows);

        vector<int> box={x1,y1,x2,y2};
        Packet->bboxes.push_back(box);
      }
    }
  }
};
```

The post-processing stage of this algorithm adds the bounding boxes created in the previous stage to the output image. This can be observed in the following code snippet.

```
template <class Data>
void Postprocessing<Data>::
 ↪  execute(Data* Packet){
    Packet->output_frame =
     ↪  Packet->frame.clone();

    for (int i = 0; i <
     ↪  Packet->bboxes.size(); i++){

        cv::rectangle
         ↪  (Packet->output_frame,
         ↪  cv::Point
         ↪  (Packet->bboxes[i][0],
         ↪  Packet->bboxes[i][1]),
         ↪  cv::Point
         ↪  (Packet->bboxes[i][2],
         ↪  Packet->bboxes[i][3]),
         ↪  cv::Scalar (0, 255, 0),2,
         ↪  4);
    }
}
```

Finally, we can define the graph in the main function, as shown in the following code snippet. It should be clear that constructing the graph from the user's perspective requires minimum complexity. The Face Detector Data object created at the beginning is used in the instantiation of the Graph, indicating the data type used. The nodes added to the graph are the nodes we have previously defined.

```
int main(int argc, char* argv[]) {

    Graph<FaceDetectorData>
     ↪  pipeline(argc, argv);

    // Define Pipeline
    pipeline.add_node<Preprocessing>
     ↪  ("Preprocessing", 1);
    pipeline.add_node<Detector>
     ↪  ("Face Detector", 1);
    pipeline.add_node<Postprocessing>
     ↪  ("Postprocessing", 1);

    pipeline.initialize_graph();
    pipeline.run_graph();
    pipeline.wait_graph();

    return 0;
}
```

5. Experimental Setup

All experiments were conducted on a single system on the Discovery Cluster hosted at the Massachusetts Green High Performance Computing Center [22, 23]. The computing system used has an Intel Xeon CPU E5-2680v4 at 2.40 GHz, 512 GB of RAM, 28 physical cores (56 logical cores), with NVIDIA GPUs which were not used during this study. The software used to produce our results for the framework are: GCC version 5.5.0, with NCurses, Threads, and OpenCV 3.4.3 with the contrib libraries.

The algorithms chosen represent a wide range of computer vision applications. They range from a simple black and white converter, to a deep learning object detection algorithm. All of the deep learning baseline codes were obtained from the LearnOpenCV's database [24] and then implemented using Vega. The following list of algorithms where used in our performance analysis:

- **baw**: A 1-stage black and white image conversion.
- **ipa**: A 3-stage image processing algorithm that applies a 11x11 Gaussian blur, a 75x75 bilateral filter, and a BGR to HSV conversion.
- **fd-caffe**: A 3-stage face detection algorithm using the SSD Caffe model.
- **fd-tensorflow**: A 3-stage face detection algorithm using the SSD 8-bit quantized Tensorflow model.
- **fd-haar**: A 3-stage face detection algorithm using the Stump-based 24x24 discrete Adaboost frontal face detector, based on the Haar Cascade model [25].
- **od-yolov3**: A 3-stage object detection algorithm using the YoloV3 model [26].

To quantify and evaluate the effectiveness of the framework, we leverage two metrics: i) the throughput (FPS), computed from the total execution time divided by the total number of processed frames, and ii) the average frame latency, computed from the average frame processing time (sum of all stage execution times) to process one frame.

In addition, we propose a new score S to quantitatively measure the overall performance of the algorithms. The score described below leverages the FPS and frame latency measured for a specific algorithm, considering the FPS of the input video and the human tolerable latency in real time video processing (100 ms). This score can be derived by the following formula:

$$F_S = \frac{1}{(1 + 0.5e^{-3(2\alpha/\beta - 1)})^2}$$

$$L_S = \frac{1}{(1 + 0.5e^{-5(\gamma/\rho)})^3}$$

$$S = F_S(\delta) + L_S(1 - \delta)$$

Where,

- F_S = The score leveraging FPS (FPS score),
- L_S = The score leveraging latency (Latency score),
- δ = Weight of the FPS score,
- α = Observed FPS,
- β = Input Video FPS (24 FPS),
- γ = Maximum allowed latency (100 ms for real-time), and
- ρ = Observed latency for a single frame.

The equations above are logistic functions, leveraging FPS and frame latency as parameters. The initial parameter values were chosen to help normalize the differences across different scales used for the input parameters, and the impact on the output video under different scenarios (e.g., high latency compared to lower FPS). The logistic function can output a score between 0 and 1. High FPS and low latency will lead to high FPS and frame latency scores, respectively. The final score is a sum of weighted FPS and frame latency scores. δ allows us to prioritize between the two scores, depending on the objectives of the implementation.

Given δ of 0.5, the highest score will be achieved when both F_S and L_S are approaching 1.0. This occurs when the FPS of the algorithm is higher than the input videos FPS and the frame latency is lower than 100 ms.

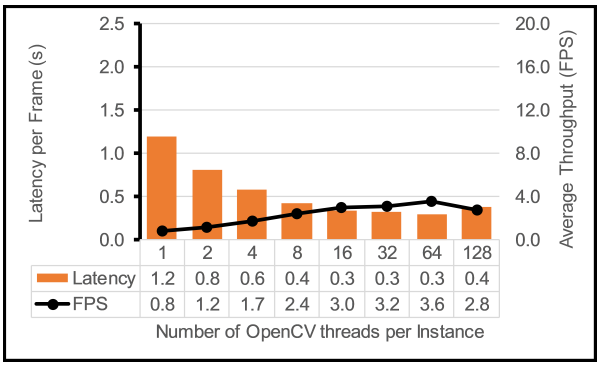
6. Performance Evaluation

Next, we discuss the experimental results of our Vega framework on two fronts: 1) an analysis on our case study and 2) an analysis of the real-world applications mentioned in Section 5. Working with deep learning frameworks, the accuracy of the model is paramount. We do not consider the accuracy of the deep learning models in this analysis, as Vega does not modify them. The baseline and Vega provide the same output across all tests.

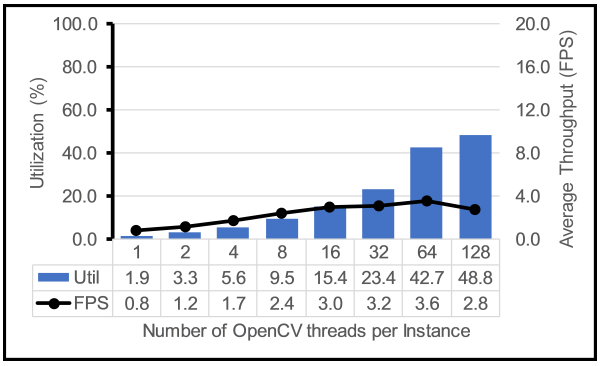
The following results were obtained using the same input video. The video has a 1080p resolution, with a 29.3 s duration at 24 FPS. The video is a recording of a crowd walking on a sidewalk, providing multiple facial detections in each frame. The results show the average metrics across the duration of the video.

6.1. Case Study: Face Detection

We use Vega to speed up the inference time for the Caffe face detection implementation. Considering the popularity of face detection algorithms, and their role in a large number of computer vision algorithms, we chose this implementation as our primary case study.



(a) Latency



(b) Utilization

Figure 5: Results for the Caffe face detection algorithm using 1 instance for the detection stage and varying the number of OpenCV threads.

6.1.1. OpenCV Threads As stated in Section 4, OpenCV's DNN framework is multi-threaded. We use Vega to study the effects of parallelizing the DNN inference used in one detection stage. As Figure 5.a suggests, increasing the number of threads leads to a reduction in the frame latency until we surpass the number of logical cores in the CPU. This over-allocation of threads degrades performance due to the increased context switching required by the OS.

The lowest frame latency achieved is 301 ms at 64 threads, which translates to a latency over the 100 ms limit. Figure 5.b shows the CPU utilization behavior for the same test. Utilization increases with the number of threads allocated to the OpenCV DNN framework, despite no correlation being observed between utilization and performance due to the higher utilization achieved with 128 threads (while achieving lower throughput). This is a result of intensive context

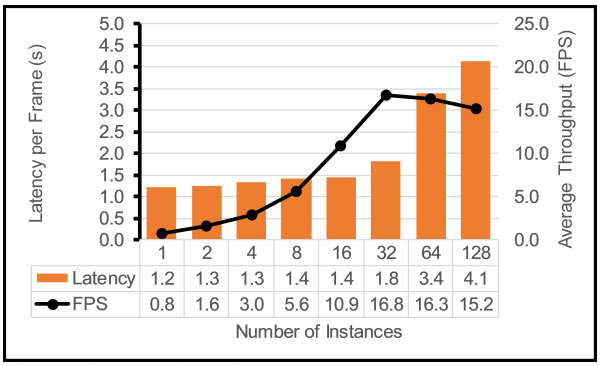


Figure 6: Face detection latency and throughput results from changing the number of instances for the detection stage with 1 thread per instance.

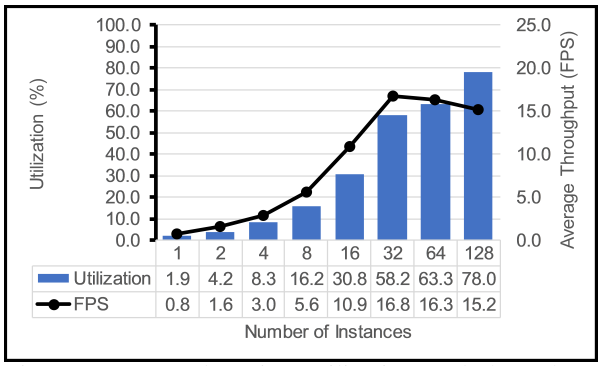


Figure 7: Face detection utilization and throughput results from changing the number of instances for the detection stage with 1 thread per instance.

switching, which in turn translates to wasteful compute time.

6.1.2. Instances Vega provides the functionality to add more instances per stage, processing different frames concurrently and improving the FPS of the application when a stage is unable to execute under the FPS limit of the video. Figure 6 demonstrates this effect when each instance uses 1 thread for the DNN forward propagation.

These results show that the addition of more instances translates to an increase in frame latency and improvement in FPS, until the number of instances surpasses the number of physical cores in the system. A 4.54x improvement in throughput is achieved when the number of instances is 32, despite obtaining a 5.6x degradation in latency compared to 1 instance using all threads.

Results are worse when each instance schedules a larger number of threads, resulting in added delay due to excessive context switching, sharing resources across multiple threads, and the larger synchronization cost due to the reorder buffer in the post-processing stage. Additionally, the latency per frame produces an unstable behavior, fluctuating drastically between frames. Frame latency is only a concern when real-time feeds are used (e.g., the live input stream from a camera). When the goal is to achieve the fastest processing of a video, throughput (FPS) is the metric to focus on.

Figure 7 shows the CPU utilization provided by the same configurations. Single-thread instances produce a similar pattern to the 1 instance test, changing the number of threads. A higher number of instances results in higher utilization, despite the best performance was achieved at a lower utilization.

Table 1: Score comparison with different δ for different input video resolution scales.

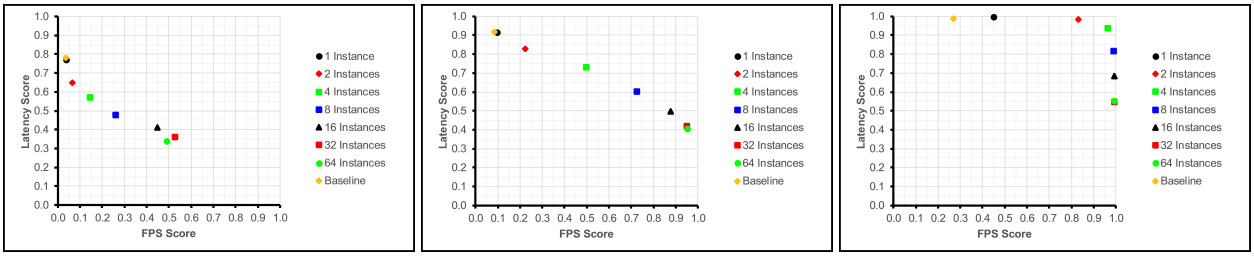
δ	**Framework**	100% **Scale**	75% **Scale**	50% **Scale**
0.0	Base	0.781	0.916	0.987
	Vega	0.771 (1i)[1]	0.914 (1i)	0.996 (1i)
0.5	Base	0.409	0.500	0.629
	Vega	0.445 (32i)	0.687 (16i)	0.950 (4i)
1.0	Base	0.038	0.084	0.270
	Vega	0.530 (32i)	0.952 (64i)	0.994 (16i)

6.1.3. Performance Metrics Figure 8 shows the scoring results for the face detection algorithm on the Vega framework, while changing the number of instances in the detector stage for different input resolutions (lower resolution, less computation). We address different input resolutions as a way to mitigate the lower scores due to the complexity of the model. The best overall score is given by the configuration closest to the top right corner. δ provides the flexibility to choose which metric to prioritize. These results can be observed in Table 1, where Vega achieves a better score for any delta except 0.0 for larger input resolutions. The input resolution has a larger impact on the score. For example, at a 1080p resolution image, no configuration is capable of achieving a balanced score above 0.5.

6.2. Speed-Up

Figure 9 shows the performance achieved by all the algorithms described in Section 5 on Vega when using a δ of 1.0 for scoring. We compare the average FPS across different configurations and display the

[1]Indicates the number of instances used for the best result.



(a) 100% Scale (b) 75% Scale (c) 50% Scale

Figure 8: Scores for the face detection algorithm in Vega, while increasing the number of instances at different input video resolutions.

best one for each algorithm. Converting the baseline implementations into the Vega framework provided immediate speed-up for 4 out of 6 implementations due to the concurrent execution of stages. The **ipa** algorithm suffered from poor performance when using our framework due to two compute intensive stages battling for the same hardware resources, and **ob-yolov3** performance degraded marginally (by less than 2%).

Once we removed the conflict in **ipa** by reducing the number of threads OpenCV was allowed to use per stage, we were able to improve the performance. This better performance was achieved using 3 instances of the Gaussian blur stage using 4 threads each, and 32 stages, with up to 16 threads each, for the bilateral filter stage.

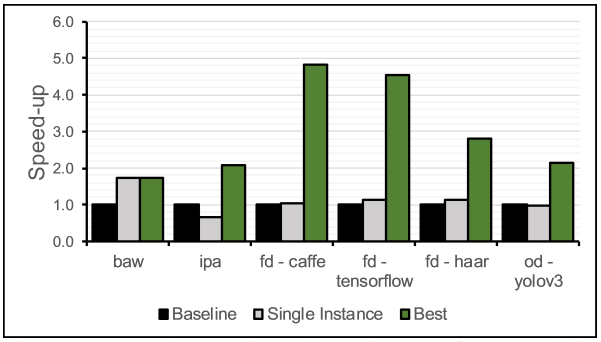


Figure 9: Speed-up achieved by using our framework in terms of total processing time.

For **baw**, the improvement gained by adding more instances was minimal due to the low complexity of the algorithm. Higher speed-up cannot be achieved because the algorithm is limited by the time to read the input video from the file system, the video decoding done in the input stage, and the overhead of synchronization in the framework (which is non-negligible for this algorithm). This limits the performance to 165 FPS with the best configuration.

For the remaining DNN models, we achieved the best performance by using 32 instances with 1 thread each. The performance improvement varies depending on the complexity of the model at hand. **ob-yolov3** improved the least due to the highly complex network structure. It is important to note that **ob-yolov3** is the only implementation where the input image was resized to fit the trained model structure due to incorrect results when testing using the full HD resolution input. Despite using lower resolution, this model is complex and requires a lot of computational power. **ob-yolov3** was only able to achieve 13.06 FPS with the best configuration, while the face detection models achieved 17.01, 16.79 and 15.03 FPS respectively. All Vega implementations achieved at least a 22% increase in CPU utilization, showing a more efficient usage of the available computing resources.

7. Conclusions and Future Work

In this paper we have presented Vega, a new C++ framework to help computer vision developers implement their algorithms using a user-friendly graph-based interface. Vega leverages an efficient parallel implementation in order to achieve better performance across a wide range of computer vision applications on edge computing systems.

We have evaluated Vega thoroughly using a case study of a face detection deep learning inference algorithm, and real-world applications where we achieved better performance across all applications using similar code complexity to that of the baseline, with added functionality. This allowed better hardware utilization for all configurations and increased the FPS, despite having a negative impact in terms of frame latency. To improve both FPS and frame latency together, the input resolution needs to be analyzed, as some algorithms are incapable of achieving a lower frame latency without additional resources.

Vega was created with flexibility in mind, allowing it to integrate different algorithms easily. Vega can be used for a wide range of CV applications, especially

when working on performance-sensitive applications that process high-resolution resolution video streams (e.g., 1080p) on edge computers. In the future, this framework could be incorporated into existing libraries (e.g. OpenCV) to facilitate access to more users.

To further expand the portability of this framework, one direction is to leverage the power of a GPU, and to combine this power with the flexibility of a multi-core CPU, arriving at a heterogeneous solution. We leave this as future work.

References

[1] D. Singh, G. Tripathi, and A. J. Jara, "A survey of internet-of-things: Future vision, architecture, challenges and services," in *2014 IEEE World Forum on Internet of Things (WF-IoT)*, pp. 287–292, March 2014.

[2] J. Levinson, J. Askeland, J. Becker, J. Dolson, D. Held, S. Kammel, J. Z. Kolter, D. Langer, O. Pink, V. Pratt, M. Sokolsky, G. Stanek, D. Stavens, A. Teichman, M. Werling, and S. Thrun, "Towards fully autonomous driving: Systems and algorithms," in *2011 IEEE Intelligent Vehicles Symposium (IV)*, pp. 163–168, June 2011.

[3] Y. Ohta and H. Tamura, *Mixed Reality: Merging Real and Virtual Worlds*. Springer Publishing Company, Incorporated, 1 ed., 2014.

[4] A. Voulodimos, N. Doulamis, A. Doulamis, and E. Protopapadakis, "Deep learning for computer vision: A brief review," in *Computational Intelligence and Neuroscience*, 2018.

[5] J. Clemons, C.-C. Cheng, I. Frosio, D. Johnson, and S. W. Keckler, "A patch memory system for image processing and computer vision," in *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-49, (Piscataway, NJ, USA), pp. 51:1–51:13, IEEE Press, 2016.

[6] K. Pulli, A. Baksheev, K. Kornyakov, and V. Eruhimov, "Real-time computer vision with opencv," *Commun. ACM*, vol. 55, pp. 61–69, June 2012.

[7] A. Vedaldi and B. Fulkerson, "Vlfeat: An open and portable library of computer vision algorithms," in *Proceedings of the 18th ACM International Conference on Multimedia (MM)*, (New York, NY, USA), pp. 1469–1472, ACM, 2010.

[8] T. P. Chen, D. Budnikov, C. J. Hughes, and Y. Chen, "Computer vision on multi-core processors: Articulated body tracking," in *2007 IEEE International Conference on Multimedia and Expo*, pp. 1862–1865, July 2007.

[9] J. Fung and S. Mann, "Openvidia: Parallel gpu computer vision," in *Proceedings of the 13th Annual ACM International Conference on Multimedia*, MULTIMEDIA '05, (New York, NY, USA), pp. 849–852, ACM, 2005.

[10] A. Amamra and N. Aouf, "Gpu-based real-time rgbd data filtering," *Journal of Real-Time Image Processing*, vol. 14, pp. 323–340, Feb 2018.

[11] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster computing with working sets," in *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'10, (Berkeley, CA, USA), pp. 10–10, USENIX Association, 2010.

[12] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: Large-scale machine learning on heterogeneous systems," 2015. Software available from tensorflow.org.

[13] T. W. Huang, C. X. Lin, G. Guo, and D. F. Wang, "Cpp-taskflow: Fast task-based parallel programming using modern c++," in *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2019.

[14] K. Group, "OpenVX," 2019. Online Source.

[15] NVIDIA, "VisionWorks," 2019. Online Source.

[16] Intel, "Intel distribution of openvino toolkit." Online Source.

[17] J. Gutierrez, "Exploring the benefits of heterogeneous computing to accelerate face detection deep learning inference," 2017.

[18] M. Teichmann, M. Weber, M. Zllner, R. Cipolla, and R. Urtasun, "Multinet: Real-time joint semantic reasoning for autonomous driving," in *2018 IEEE Intelligent Vehicles Symposium (IV)*, pp. 1013–1020, June 2018.

[19] A. Poms, W. Crichton, P. Hanrahan, and K. Fatahalian, "Scanner: Efficient video analysis at scale," *ACM Transactions on Graphics (TOG)*, vol. 37, no. 4, p. 138, 2018.

[20] W. Liu, D. Anguelov, D. Erhan, C. Szegedy, S. Reed, C.-Y. Fu, and A. C. Berg, "Ssd: Single shot multibox detector," in *European conference on computer vision*, pp. 21–37, Springer, 2016.

[21] S. Mallick, "Cpu performance comparison of opencv and other deep learning frameworks," Dec 2018. Online Source.

[22] "The massachusetts green high performance computing center." Available at: http://www.mghpcc.org/.

[23] "Research computing." Northeastern Information Technology Services. Online Source.

[24] V. Gupta, "Face detection opencv, dlib and deep learning (c++ / python)," Oct 2018. Online Source.

[25] R. Lienhart, A. Kuranov, and V. Pisarevsky, "Empirical analysis of detection cascades of boosted classifiers for rapid object detection," in *Joint Pattern Recognition Symposium*, pp. 297–304, Springer, 2003.

[26] J. Redmon and A. Farhadi, "Yolov3: An incremental improvement," *arXiv preprint arXiv:1804.02767*, 2018.