UNIVERSITY OF THESSALY

SCHOOL OF ENGINEERING

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

Supporting flexible acceleration for OpenCV functions through the vAccel Framework

Diploma Thesis

Ilias Lagomatis

Supervisor: Spyros Lalis

June 2024

ΠΑΝΕΠΙΣΤΗΜΙΟ ΘΕΣΣΑΛΙΑΣ

ΠΟΛΥΤΕΧΝΙΚΗ ΣΧΟΛΗ

ΤΜΗΜΑ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

Ύποστήριξη ευέλικτης επιτάχυνσης για OpenCV συναρτήσεις μέσω του vAccel Framework

Διπλωματική Εργασία

Ηλίας Λαγομάτης

Επιβλέπων: Σπύρος Λάλης

Ιούνιος 2024

Approved by the Examination Committee:

Supervisor **Spyros Lalis**
Professor, Department of Electrical and Computer Engineering, University of Thessaly

Member **Christos Antonopoulos**
Professor, Department of Electrical and Computer Engineering, University of Thessaly

Member **Nikolaos Bellas**
Professor, Department of Electrical and Computer Engineering, University of Thessaly

Acknowledgements

I would like to express my deepest gratitude to my supervisor, Professor Spyros Lalis, for his clear guidance and constant support throughout this work. His helpful feedback and encouragement have been crucial to completing this thesis.

I am also very thankful to Professor Christos Antonopoulos and Professor Nikolaos Bellas. Their teachings in key courses like Computer Organization and Design, Operating Systems, and High Performance Computing have greatly improved my understanding and skills. Their knowledge and advice have been essential to this work.

A special thanks goes to my colleague and friend, Anastassions Nanos, for the great teamwork on the vAccel Framework. His friendship and shared insights have made this journey both productive and enjoyable.

I would also like to sincerely thank Alexandros Patras, a researcher and PhD student at the Computer Systems Lab at the University of Thessaly. His help and guidance during my project in the labs were vital in solving many technical challenges.

Finally, I am deeply grateful to my family for their endless support and encouragement throughout my studies. Their belief in me has been a constant source of motivation and strength.

DISCLAIMER ON ACADEMIC ETHICS AND INTELLECTUAL PROPERTY RIGHTS

«Being fully aware of the implications of copyright laws, I expressly state that this diploma thesis, as well as the electronic files and source codes developed or modified in the course of this thesis, are solely the product of my personal work and do not infringe any rights of intellectual property, personality and personal data of third parties, do not contain work / contributions of third parties for which the permission of the authors / beneficiaries is required and are not a product of partial or complete plagiarism, while the sources used are limited to the bibliographic references only and meet the rules of scientific citing. The points where I have used ideas, text, files and / or sources of other authors are clearly mentioned in the text with the appropriate citation and the relevant complete reference is included in the bibliographic references section. I also declare that the results of the work have not been used to obtain another degree. I fully, individually and personally undertake all legal and administrative consequences that may arise in the event that it is proven, in the course of time, that this thesis or part of it does not belong to me because it is a product of plagiarism».

The declarant

Ilias Lagomatis

Diploma Thesis

Supporting flexible acceleration for OpenCV functions through the vAccel Framework

Ilias Lagomatis

Abstract

In the rapidly evolving field of computer vision, addressing the accuracy-speed trade-off in various operations is critical for applications ranging from autonomous vehicles to augmented reality. This trade-off entails that some implementations achieve high accuracy at the cost of reduced speed, while others prioritize speed, sacrificing accuracy. A promising solution involves dynamically scheduling the appropriate algorithm or device for each situation. Instead of developing proprietary methods for switching between algorithms or devices, we leverage the vAccel framework, which offers seamless interchanges between plugins, along with robust resource and hardware accelerator management mechanisms. In this thesis, we use as an indicative example the optical flow operation, which exemplifies the accuracy-speed trade-off, and demonstrate how the vAccel framework can be used to support the flexible selection of plugins at runtime. More specifically, we develop support that allows computer vision applications to select and invoke, with minimal programming effort, the optimal algorithm or device based on current conditions and available hardware resources, thereby enhancing the efficiency and effectiveness of the computation. By implementing and evaluating this approach, we aim to contribute to the development of more adaptive and intelligent systems in the field of computer vision.

Διπλωματική Εργασία

Ύποστήριξη ευέλικτης επιτάχυνσης για OpenCV συναρτήσεις μέσω του vAccel Framework

Ηλίας Λαγομάτης

Περίληψη

Στον ταχέως εξελισσόμενο τομέα της Όρασης Υπολογιστών, η αντιμετώπιση του trade-off μεταξύ ακρίβειας και ταχύτητας είναι κρίσιμη σε εφαρμογές όπως τα αυτόνομα οχήματα, η επαυξημένη πραγματικότητα κ.ά. Αυτό το trade-off συνιστά ότι κάποιες υλοποιήσεις επιτυγχάνουν υψηλή ακρίβεια με κόστος την μειωμένη ταχύτητα, ενώ άλλες δίνουν προτεραιότητα στην ταχύτητα, θυσιάζοντας την ακρίβεια. Μια πιθανή λύση περιλαμβάνει τον δυναμικό προγραμματισμό του κατάλληλου αλγορίθμου ή της κατάλληλης συσκευής για κάθε κατάσταση. Αντί όμως να αναπτύσσουμε καινούργιες μεθόδους για την εναλλαγή μεταξύ αλγορίθμων ή συσκευών, αξιοποιούμε το vAccel Framework, το οποίο προσφέρει απρόσκοπτες εναλλαγές μεταξύ Plugins, μαζί με ισχυρούς μηχανισμούς διαχείρισης πόρων και επιταχυντών υλικού. Σε αυτήν τη Διπλωματική Εργασία, χρησιμοποιούμε ως ενδεικτικό παράδειγμα τη λειτουργία οπτικής ροής (Optical Flow), η οποία αποτελεί παράδειγμα trade-off μεταξύ ακρίβειας και ταχύτητας, και δείχνουμε πώς το vAccel Framework μπορεί να χρησιμοποιηθεί για να υποστηρίξει την ευέλικτη επιλογή των Plugins κατά τη διάρκεια της εκτέλεσης. Πιο συγκεκριμένα, αναπτύσσουμε υποστήριξη που επιτρέπει στις εφαρμογές Όρασης Υπολογιστών να επιλέγουν και να καλούν, με ελάχιστη προγραμματιστική προσπάθεια, τον βέλτιστο αλγόριθμο ή τη βέλτιστη συσκευή βάσει των τρεχουσών συνθηκών και των διαθέσιμων πόρων υλικού, βελτιώνοντας έτσι την αποδοτικότητα και την αποτελεσματικότητα της υπολογιστικής διαδικασίας. Με την υλοποίηση και αξιολόγηση αυτής της προσέγγισης, σκοπεύουμε να συμβάλλουμε στην ανάπτυξη πιο προσαρμοστικών και έξυπνων συστημάτων στον τομέα της Όρασης Υπολογιστών.

Λέξεις-κλειδιά:

Accuracy-Speed Trade-off, vAccel Framework, Hardware Acceleration, Adaptive Algorithm, OpenCV

Table of contents

Chapter 1

Introduction

1.1 Motivation

Recent developments in hardware technology have significantly promoted the adoption of heterogeneous computer architectures, which are now prevalent not only in edge devices but also in datacenter-class machines. The shift towards incorporating multi-core CPUs, GPUs, and FPGAs into a single system has enabled remarkable performance improvements and energy efficiency gains. Systems such as NVIDIA's DGX A100 [1], which integrates GPUs for accelerated computing, and Intel's Agilex FPGAs [2], designed for adaptable acceleration, exemplify this trend. These advancements underscore a paradigm shift where heterogeneous computing is leveraged to meet the increasing demands for performance and efficiency across diverse computing environments.

These technological advancements, while promising enhanced energy-efficient computing, introduce several complexities for application developers. Firstly, writing code that can efficiently run across different hardware platforms becomes a challenging task. The diversity in architectures necessitates a deep understanding of each platform's capabilities and limitations to optimize performance effectively. Secondly, utilizing various implementation options in a dynamic, adaptive manner poses an even greater challenge. The need to switch between different hardware elements and computing kernels based on real-time requirements adds a layer of complexity that traditional development approaches struggle to address.

Moreover, compute-centric applications often face the challenge of having multiple implementations that solve a given problem (perform a given calculation) with varying degrees of accuracy and performance. Typically, implementations that offer lower accuracy tend to

have shorter execution times and consume less energy. While this variety increases the flexibility available to application developers, it also complicates the development process. Making the optimal choice from this array of options at any given time is crucial but difficult, as static design decisions can become highly suboptimal during runtime due to changing conditions and requirements.

1.2 Focus

This thesis addresses these dual challenges using an optical flow application as a case study. Optical flow algorithms are crucial for various applications, including video processing [3], autonomous vehicles [4], and motion detection [5], where they estimate the motion of objects between frames. These algorithms inherently involve an accuracy-speed trade-off. On the one hand, dense approaches, which compute motion for every pixel in the image, offer high accuracy but require extensive computation, leading to longer execution times and higher energy consumption. On the other hand, sparse approaches, which compute motion only for selected points, are faster and more energy-efficient but sacrifice some degree of accuracy.

The first challenge, the flexible use of different compute-kernel implementations that support various accuracy and performance trade-offs across different hardware elements, is tackled through the vAccel framework. This framework enables the seamless integration and utilization of heterogeneous hardware components, allowing developers to exploit the full potential of modern architectures with minimal effort.

The second challenge, the intelligent selection of the appropriate implementation at runtime for each kernel invocation, is managed by enhancing the vAccel backend system with additional intelligence. This enhancement enables the system to make informed decisions on-the-fly while providing corresponding plugin selection hits to the application, allowing the application to achieve optimal performance and energy efficiency with minimal programming effort for the developer.

1.3 Contribution

The main contributions of this thesis is the development and the evaluation of a plugin selection unit for the vAccel framework that operates at runtime. This unit dynamically selects the optimal optical flow algorithm and device based on the current application needs and the available hardware resources, abstracting the complexity from the user. The plugin selection unit evaluates the response of the plugins to choose between dense and sparse optical flow methods, based on computational constraints, too. For example, in scenarios with high motion complexity and significant scene changes, the unit would select a dense optical flow algorithm, ensuring high accuracy and detailed motion representation. Conversely, in situations with less movement or when real-time performance is critical, the unit would opt for a sparse method, maximizing speed while maintaining sufficient accuracy.

This dynamic selection process takes place in the backend plugins of the vAccel framework. The selection unit produces a hint that is returned to the frontend runtime library, which then determines on which of the available plugins will offload the calculation. Since the selection unit is embedded in the plugins, there is no need for an extra node in the network to implement this logic, thereby avoiding additional communication overhead. This architecture ensures that the optimal optical flow technique is chosen efficiently, leveraging the available hardware resources while maintaining seamless integration and execution from the user's perspective.

This innovation significantly enhances the performance and efficiency of the application, making the vAccel framework more adaptable and user-friendly. By automating the decision-making process, users can leverage the best-suited optical flow technique for their specific use case without needing in-depth knowledge of the underlying algorithms. This significantly reduces the barrier to entry for deploying advanced optical flow techniques in various applications, from video analysis and compression to autonomous navigation and real-time object tracking.

In addition, as another contribution, this work introduces a novel method for reading and writing arguments within the vAccel framework, emphasizing readability and better-looking code rather than performance. This enhances the framework's functionality and usability, making it easier for developers to understand and maintain the code. By prioritizing code clarity and structure, the vAccel framework becomes more accessible and user-friendly, supporting a wider range of applications and improving overall development efficiency.

1.4 Thesis Structure

The rest of the thesis is structured as follows. Chapter 2 provides an overview of related work. Chapter 3 and Chapter 4 give basic background information about the vAccel framework, optical flow and OpenCV library. Chapter 5 presents the extensions made to the vAccel framework to support programmer-friendly passing of arguments between the client and backend. Chapter 6 describes how vAccel is used to support the transparent acceleration of key OpenCV kernels through plugins for dense and sparse CPU-based and CPU-based implementations, while Chapter 7 discusses the support for flexible seletion of plugins at runtime. Finally, Chapter 8 summarizes the thesis and points to promising directions for future work.

Chapter 2

Related Work

The accuracy-speed trade-off is a fundamental challenge in many areas of technology and computer science. Various approaches have been developed that address this issue, focusing on optimizing algorithms, leveraging hardware acceleration, and utilizing innovative computing frameworks.

2.1 Approximate Computing

Approximate computing is an innovative approach that embraces inexact computations to enhance performance and reduce power consumption, making it particularly valuable in domains where perfect accuracy is not critical. By relaxing the need for exactness, approximate computing can significantly lower energy usage and increase processing speed, which is beneficial for applications such as multimedia processing, machine learning, and big data analytics [6]. Techniques such as voltage scaling, precision reduction, and probabilistic computing allow for these efficiency gains by tolerating minor inaccuracies in the final output [7]. This approach enables systems to execute complex tasks more efficiently, often with negligible impact on the perceived quality or functionality, thus providing a pragmatic balance between computational resource utilization and output fidelity [8]. Consequently, approximate computing presents a compelling solution for optimizing the performance of energy-constrained devices and enhancing the overall efficiency of computational workloads.

2.2 Adaptive Algorithms

Adaptive algorithms are designed to dynamically adjust their complexity based on the available computational resources and the accuracy requirements of the task at hand. These algorithms can modify parameters such as the depth and width of neural networks during inference to balance accuracy and speed. For instance, dynamic neural networks have been proposed to change their structure in real-time, thereby reducing computational load when full model capacity is not needed, and enhancing it when higher accuracy is required [9]. These adaptive techniques are particularly useful in environments where computational resources are limited or where the workload varies significantly.

2.3 Hardware Acceleration

Hardware accelerators such as GPUs (Graphics Processing Units), FPGAs (Field-Programmable Gate Arrays), and TPUs (Tensor Processing Units) play a crucial role in mitigating the accuracy-speed trade-off. GPUs, with their high parallel processing capabilities, are widely used in deep learning and other computationally intensive tasks to accelerate processing times while maintaining high accuracy [10]. FPGAs offer customizable hardware solutions that can be tailored to specific tasks, providing a balance between speed and power efficiency [11]. TPUs, designed specifically for machine learning workloads, offer significant improvements in processing speed and energy consumption, particularly for inference tasks in neural networks [12]. By offloading computationally heavy operations to these accelerators, applications can achieve real-time performance without compromising on accuracy.

2.4 Heterogeneous Computing

Heterogeneous computing involves the use of different types of processors within a single system to optimize performance and energy efficiency. By combining CPUs, GPUs, and other specialized accelerators, heterogeneous systems can dynamically allocate tasks to the most suitable processor based on the current workload and performance requirements [13]. This approach enables better management of the accuracy-speed trade-off by leveraging the strengths of each type of processor. For example, a system might use a GPU for parallelizable tasks, an FPGA for customizable hardware operations, and a CPU for general-purpose

computing, thereby optimizing overall performance [14].

2.5 This Work

This thesis is closely connected to the aforementioned related work. Firstly, vAccel's capability to handle multiple devices through its plugins aligns our work with the field of heterogeneous computing. In the context of heterogeneous computing, a significant engineering effort is typically required to achieve efficient interchanges between various devices, involving complex management of device-specific drivers, communication protocols, and data synchronization. However, the vAccel Framework simplifies this process by facilitating seamless transitions between plugins, and thus between devices. This inherent flexibility and ease of integration significantly reduce the overhead and complexity usually associated with heterogeneous computing environments, enabling more efficient utilization of diverse hardware resources.

Secondly, our focus on enhancing application performance by utilizing hardware accelerators, such as GPUs, ties our work to the domain of hardware acceleration. While hardware acceleration boosts the performance of compute-intensive applications, there are moments when the use of acceleration is unnecessary. For instance, in video processing applications where the input is simplistic (no scene changes), power-hungry hardware accelerators are not needed. A sparse implementation of the main algorithm could suffice, consuming less energy and avoiding the communication overheads associated with the device. In this thesis, we address this issue by inspecting the algorithm's output after each call and determining whether the next computation should be offloaded to a hardware accelerator.

Additionally, our approach is related to adaptive algorithms, as we employ optimization techniques that dynamically adapt methods based on current requirements. We extend this concept by incorporating not only various methods but also various devices. Using vAccel mechanisms, we introduce the concept of device priority. For example, after offline profiling reveals that a GPU performs better than a CPU for a specific method, we explicitly specify this priority order in vAccel. The framework will then automatically check if the preferred device is available for offloading the application call. If not, it will resort to the next available plugin.

Finally, while approximate computing enhances performance in terms of time and power

consumption, it achieves this by sacrificing accuracy. This trade-off can be advantageous in scenarios where perfect accuracy is not crucial, such as multimedia processing or certain machine learning tasks. However, for applications where accuracy is paramount, the results may be unsatisfactory. Our approach aims to identify situations where approximate computing can yield acceptable outcomes and offload the computation to a sparse or approximate implementation in those cases. Conversely, when the approximation does not meet the required accuracy standards, we resort to a traditional, precise implementation to ensure the integrity and reliability of the results.

Chapter 3

The vAccel Framework

vAccel is a flexible and modular framework designed to provide hardware acceleration for serverless workloads, particularly those running in virtual machines (VMs) or at the edge. The core of vAccel is its runtime library (vAccelRT), which exposes an API to user applications and dispatches requests to various backend plugins that handle specific hardware accelerators like GPUs, TPUs, and FPGAs.

The main design goals of vAccel include portability, security, compatibility, low overhead, and scalability. It allows applications to be deployed on different machines with varying hardware accelerators without the need to rewrite or recompile the code. This is achieved through a generic API that maps to specific hardware accelerators via plugins. The framework supports multiple environments execution, including standalone setups, Containers, VMs etc.

One key advantage of vAccel is its ability to run workloads on environments that don't have direct physical access to acceleration devices. Instead, it offloads compute-intensive operations from the guest VM to the host system where the hardware accelerators reside. This process is facilitated by a transport layer that is efficient and incurs minimal overhead, ensuring near-native performance for accelerated functions.

3.1 Design

The two most important concepts in the vAccel workflow are "Operations" and "Plugins" (or Backends). Figure 3.1 presents the abstract vAccel architecture. The vAccel API exports a variety of operations, such as mathematical calculations and machine learning tasks. When these operations are called, the Core Library offloads the computation to plugins that imple-

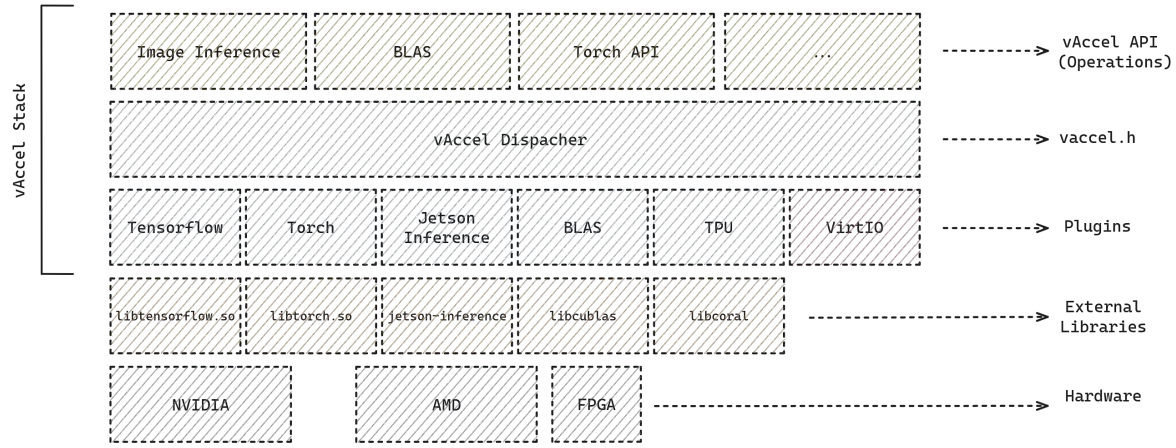


Figure 3.1: Overview of vAccel framework.

ment the requested operation.

A vAccel plugin is a shared object containing executable code, constructed using the vAccel Framework. During the development of a plugin, the developer specifies the supported operations, plugin name, device type, and other details via the `<vaccel.h>` header file. Once the plugin is built, its path (the path of the constructed shared object) is added to a specific environment variable to make it available for a vAccel program.

During the initialization of a vAccel program, the vAccel Core Library loads all specified plugins and saves their information. This process is completely transparent to the user. When the main program calls a vAccel Operation, the Core Library checks if any of the loaded plugins implement the requested operation by matching the operation ID. If a suitable plugin is found, the computation is offloaded to it. After completing the operation, the Library returns the result to the main program. This entire process is hidden from the user, ensuring a seamless experience.

An example main program may look as follows:

```c
#include <vaccel.h>

int main() {
    [...]

    int ret = vaccel_op(...);

    [...]
    return 0;
}
```

However, what happens under the hood is not obvious. Assume we have written and compiled this main program, generating an executable file. If this program is executed multiple times with a different backend path specified through the environment variable each time, we will observe entirely different executions for each run. For instance, the first execution might run the operation on a CPU, the second on a GPU, and the third on an FPGA, even though the executable file remains the same in all cases:

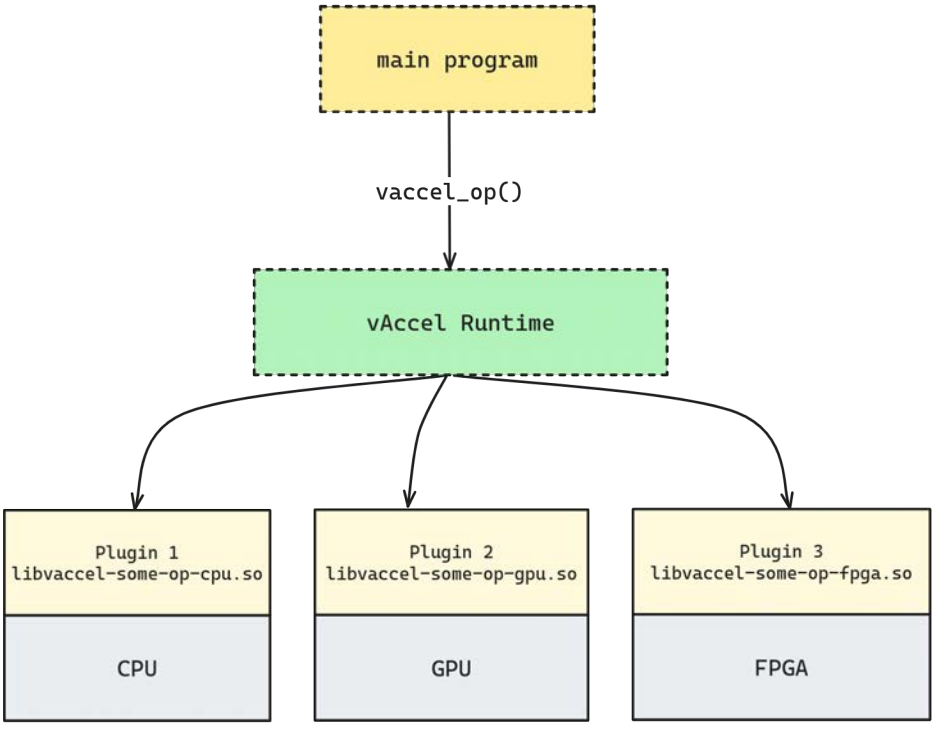


Figure 3.2: Example invocation flow via vAccel.

The following figure provides another overview of vAccel. In this example, vAccel includes 10 operations (0-9). However, by default, it does not contain implementations for any

of them. During initialization, the library loads all three plugins provided by the user (e.g., libvaccel-plugin1.so, etc.). The first plugin implements operations 2 and 3, the second plugin implements operation 4, and the third plugin implements operations 4 and 8. This means the main program can utilize all four operations.

It is also important to note that the main program can use two entirely different implementations of operation 4. This capability of vAccel, which we will discuss in more detail later, allows for agnostic interchanges between backends, enabling the development of a selection unit.

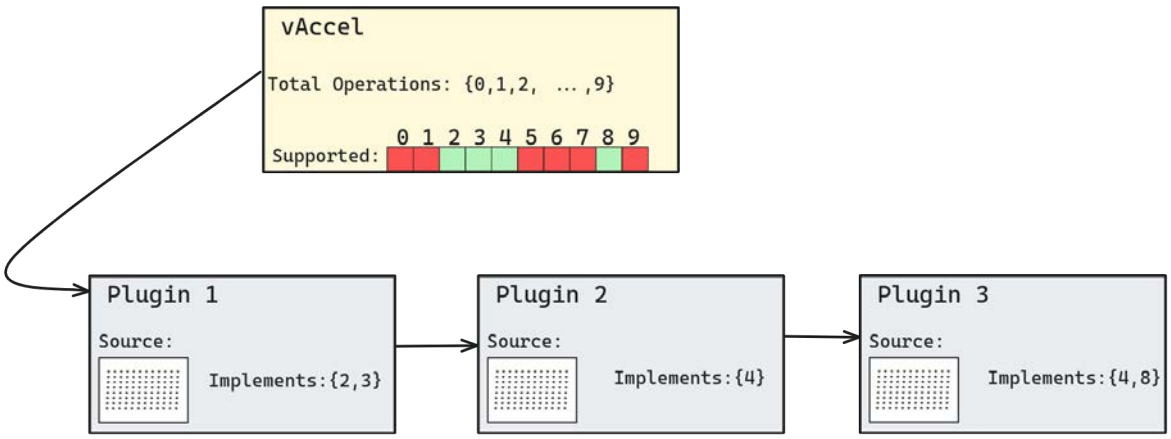


Figure 3.3: A vAccel-program execution with 3 available plugins and 4 supported operations.

The following is a summary of the operations that vAccel supports: *noop*; *sgemm*; *image [classification, detection, segmentation, pose estimation, depth estimation]*; *MinMax*; *Array copy*; *Matrix multiplication*; *Parallel acceleration*; *Vector Add*; *exec*; *exec with resource*; *OpenCV Generic*; *TensorFlow [model create, model destroy, model register, model unregister, session load, session run, session delete]*; *TensorFlow Lite [session load, session run, session delete]*. *Torch [jitload_forward function, SGEMM]*. Although there are existing implementations for these plugins, users have the flexibility to build their own implementations for different devices and specific requirements. This capability allows for greater customization and optimization, enabling users to leverage the unique features and performance characteristics of various hardware platforms. Furthermore, all these operations can run on vAccel with negligible overhead .

3.2 Example

3.2.1 Build and Installation

```
$ apt-get install build-essential cmake
$ git clone https://github.com/cloudkernels/vaccelrt --recursive
$ cd vaccelrt
$ mkdir build
$ cd build
```

Building the core runtime library

```
$ cmake ../ -DCMAKE_INSTALL_PREFIX=/usr/local
$ make
$ make install
```

As mentioned above, building the plugins is disabled, by default. Building one or more plugins can be enabled at configuration time of CMake by setting the corresponding variable of the following table on ”ON”.

Backend Plugin	Variable	Default
noop	BUILD_PLUGIN_NOOP	OFF
exec	BUILD_PLUGIN_EXEC	OFF

For example:

```
$ cmake -DBUILD_PLUGIN_NOOP=ON ..
```

will enable building the noop backend plugin.

Building a vAccel Application

We will use an example of image classification which can be found under the examples folder of the vAccel project.

The example can be built using the following directive in the build directory:

```
$ cmake -DBUILD_EXAMPLES=ON ..
$ make
```

A number of example binaries have been built:

```
$ ls examples
classify          detect           exec_generic     [...]
pynq_parallel     segment_generic  tf_inference     [...]
detect_generic    minmax_generic   pose_generic     [...]
tf_model          depth_generic    exec             [...]
pynq_array_copy   segment          sgemm_generic    [...]
```

3.2.2 Running a vaccel application

Having built the `classify` example, we need to prepare the vaccel environment for it to run:

1. Define the path to `libvaccel.so` (if not in the default search path):

   ```
   $ export LD_LIBRARY_PATH=/usr/local/lib:$LD_LIBRARY_PATH
   ```

2. Define the backend plugin to use for our application.

   In this example, the `noop` plugin will be used, for simplicity:

   ```
   $ export VACCEL_BACKENDS=/usr/local/lib/libvaccel-noop.so
   ```

3. Finally, we run:

   ```
   $ ./classify images/example.jpg 1
   ```

   which should dump the following output:

   ```
   $ ./classify images/example.jpg 1
   Initialized session with id: 1
   Image size: 79281B
   [noop] Calling Image classification for session 1
   [noop] Dumping arguments for Image classification:
   [noop]   len_img: 79281
   [noop] will return a dummy result
   classification tags: This is a dummy classification tag!
   ```

3.2.3 Code Overview

The following piece of code is source we built and ran above.

```
#include <stdlib.h>
#include <stdio.h>
#include <vaccel.h>
[...]

int main(int argc, char *argv[]) {
    int ret;
    char *image;
    size_t image_size;
    char out_text[512], out_imagename[512];
    struct vaccel_session sess;

    if (argc != 3) {
        fprintf(stderr, "Usage: %s filename #iterations\n", argv[0]);
        return 0;
    }

    ret = vaccel_sess_init(&sess, 0);
    if (ret != VACCEL_OK) {
        fprintf(stderr, "Could not initialize session\n");
        return 1;
    }

    printf("Initialized session with id: %u\n", sess.session_id);

    ret = read_file(argv[1], (void **)&image, &image_size);
    if (ret)
        goto close_session;

    ret = vaccel_image_classification(&sess, image, out_text, out_imagename,
                            image_size, sizeof(out_text), sizeof(out_imagename));

    if (ret)
        fprintf(stderr, "Could not run op: %d\n", ret);

    free(image);
```

```
    if (vaccel_sess_free(&sess) != VACCEL_OK) {
        fprintf(stderr, "Could not clear session\n");
        return 1;
    }
    return ret;
}
```

Although this source code is pretty straightforward, let's explain some important parts:

1. **Session Initialization**:

   ```
   struct vaccel_session sess;
   [...]
   ret = vaccel_sess_init(&sess, 0);
   ```

   `Session` is a fundamental concept in vAccel. It is utilized by the core library for bookkeeping and maintaining the state of a group of transactions. Whenever we want to write a vAccel program, we must start a session to allow the framework to manage the workflow of our program effectively.

2. **Operation Call**:

   ```
   ret = vaccel_image_classification(&sess, ...);
   ```

   This is the point where we invoke the `Image Classification` operation. vAccel will handle this call by searching through the available plugins to find one that implements this operation, thereby offloading the computation to the appropriate plugin.

3. **Clear Session**:

   ```
   if (vaccel_sess_free(&sess) != VACCEL_OK) {
       [...]
   }
   ```

   Finally, it is crucial to free our session to release memory and ensure efficient management of system assets. This step helps maintain the stability and performance of our vAccel program.

3.3 The 'Resource' concept in vAccel

In vAccel, the concept of a `Resource` is pivotal, particularly for handling and bookkeeping binary blobs such as ML models, shared objects, FPGA bitstreams etc. A resource in vAccel serves as a container for these blobs, enabling seamless management and utilization. One of the significant advantages of using resources in vAccel is that they can be stored once and programs that execute remotely can point to them each time they perform an operation. This removes the need to transfer the binary blob every time across the remote path. When a vAccel program is running on a remote host, the resource is transferred transparently, without requiring the user to write additional code or configure networking settings. This greatly simplifies the process of deploying and executing machine learning models across different environments. Users can register their models with a vAccel session, ensuring that when they invoke inference operations, vAccel will automatically utilize the previously registered model resource. This streamlined approach enhances efficiency and reduces the complexity associated with managing model deployments in distributed systems.

3.4 Evaluation of vAccel Execution vs Stock Execution

In this section, we present a detailed evaluation comparing vAccel execution to stock execution without the vAccel framework. Our primary focus is to demonstrate that the vAccel framework introduces negligible overhead while providing significant benefits in terms of resource management and execution efficiency. By performing identical operations with and without vAccel, we aim to show that the framework simplifies the deployment and management of computational tasks, but maintains performance.

The results are shown in Table 3.1 for indicative benchmark computations. As can be seen, the use of vAccel generally introduces acceptable overhead. The only exception is for the execution of the Image Classification Model on GPU because of its very small execution time.

3.5 Virtual Machines, Containers and vAccel

Containers and virtual machines (VMs) are technologies that allow for the isolation and efficient deployment of applications. They each provide distinct mechanisms for achieving

Operation	Device	Stock Time (ms)	vAccel Time (ms)	vAccel Overhead
Bert Model	CPU	58	59	1.7%
Bert Model	GPU	8	8	0%
Image Classification	CPU	51.5	54.5	5.5%
Image Classification	GPU	2	3	33%
KNN	GPU	128.38	128.45	0.05%
KMeans	GPU	258.9	262.83	1.5%

Table 3.1: vAccel overhead for various computation benchmarks.

this, and understanding how they work, along with their benefits and limitations, is crucial for modern software development and deployment.

Virtual Machines operate by emulating an entire computer system, including its own operating system (OS), on top of a physical host machine. This is achieved through a hypervisor, which manages and allocates resources from the host machine to multiple VMs. Each VM runs independently with its own OS and applications, which allows for robust isolation and security. Virtual machines are particularly useful for running different operating systems on a single physical machine, testing software in isolated environments, and efficiently utilizing server resources by hosting multiple VMs on a single server.

Containers, on the other hand, are a lightweight alternative to VMs. Rather than emulating an entire system, containers share the host OS's kernel and isolate applications at the process level. This is achieved through features like namespaces and control groups (cgroups) in the Linux kernel. Containers package an application along with its dependencies, ensuring that it runs consistently across different environments. Docker is one of the most popular containerization platforms. Containers are highly efficient because they avoid the overhead of running separate OS instances, leading to faster startup times and better resource utilization.

Both containers and VMs are **useful** for several reasons:

- **Isolation**: They provide strong isolation, which enhances security and allows multiple applications to run on the same physical hardware without interfering with each other.
- **Portability**: Applications packaged in containers can run consistently across various environments, from a developer's laptop to production servers.
- **Scalability**: They enable easy scaling of applications. Containers can be quickly repli-

cated across multiple instances, and VMs can be resized or migrated as needed.

- **Resource Efficiency**: Containers are particularly efficient in terms of resource usage because they share the host OS kernel, while VMs make better use of physical hardware by hosting multiple OS instances on a single server.

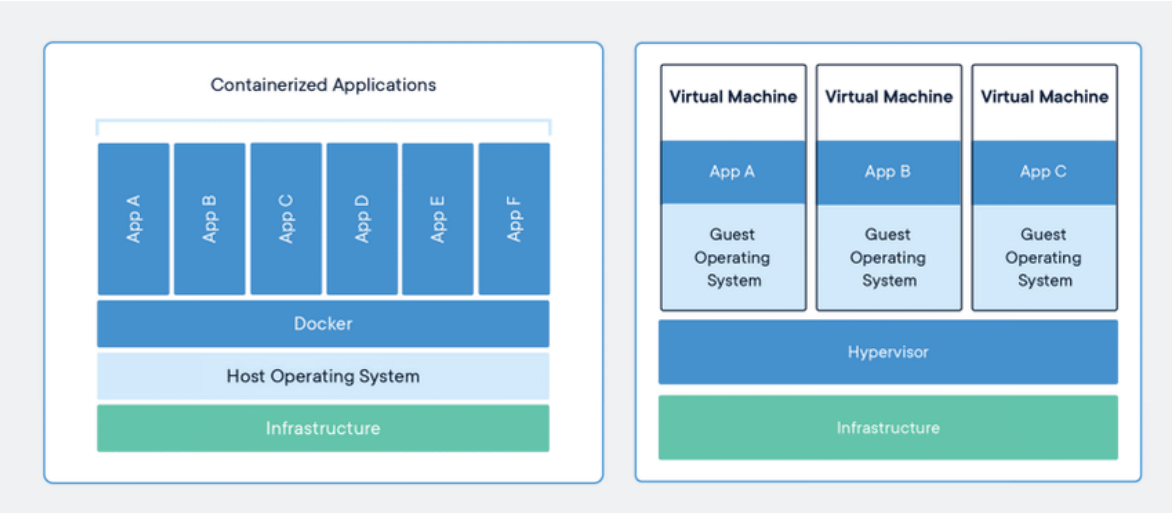


Figure 3.4: Application execution architecture when using containers vs virtual machines (from [15]).

Despite these benefits, both containers and virtual machines have **limitations** when it comes to accessing hardware devices, particularly GPUs. By default, containers and VMs do not have direct access to the host machine's GPU. This lack of access is due to the way virtualization and containerization abstract and isolate system resources. GPUs are essential for accelerating a variety of compute-intensive tasks, such as machine learning, scientific simulations, and graphics rendering. Without direct access to GPUs, the performance of applications that rely heavily on these resources can be significantly hindered.

This is where technologies like **vAccel** become useful. vAccel provides a mechanism for offloading computations to various hardware accelerators, including GPUs, from within a container or VM. By using vAccel, applications running in isolated environments can leverage the full power of the underlying hardware, overcoming the inherent limitations of traditional container and VM setups.

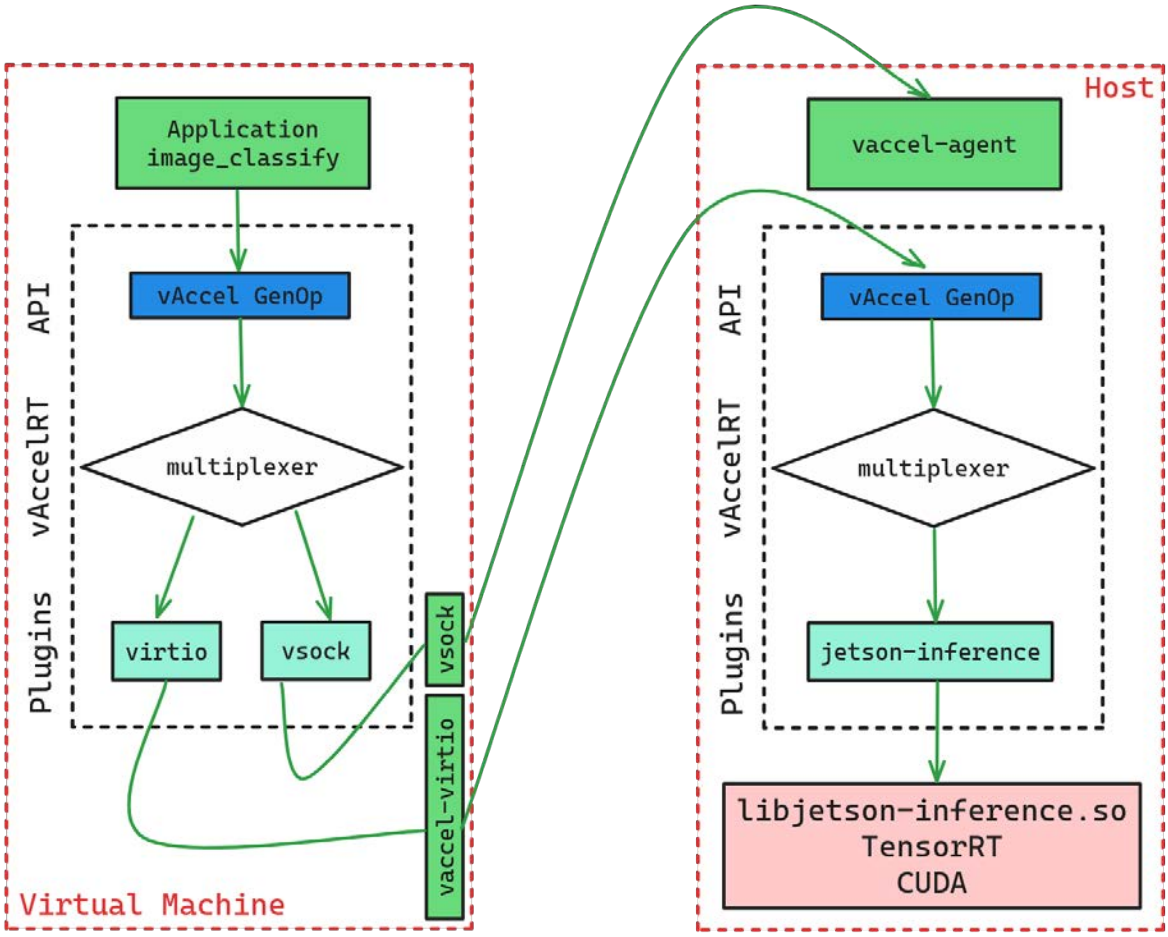


Figure 3.5: VM application execution flow.

3.5.1 Custom VirtIO Plugin

The custom VirtIO plugin is an integral part of the vAccel framework, designed to offload computation from a guest VM to the host. This plugin serves as a transport mechanism with straightforward semantics to facilitate data transport between the guest VM and the host, offering a low-overhead alternative to traditional VirtIO solutions by bypassing unnecessary paths, such as the network stack. It functions solely as a buffer passing layer, remaining agnostic to the type and contents of the request.

The transport mechanism comprises three key components:

- A Linux kernel module (virtio-accel)
- A vAccelRT VMM backend for both QEMU and Firecracker
- The actual vAccelRT plugin (vaccelrt-plugin-virtio)

The `virtio-accel` module operates as a Linux miscellaneous device (`/dev/accel`) within the guest VM, interfacing with userspace (vAccelRT) and implementing the guest-host communication via a VirtIO device. The `vaccelrt-plugin-virtio` communicates with `virtio-accel` through IOCTL calls. Once data is packed, the plugin issues the necessary IOCTL to `/dev/accel` to forward the request to the host. The device driver ensures that the data is accessible to the guest kernel-space and forwards the request to the VirtIO sublayer.

Data packing involves filling the buffer/length structure for each request argument. Users can either create custom pack/unpack functions for new operations or use the genop call without modifying the `vaccelrt-plugin-virtio` code.

The VirtIO implementation follows a split-driver model, consisting of a VirtIO frontend (guest kernel) and a VirtIO backend (host QEMU/Firecracker) that exchange data via shared memory rings (virtqueues). Upon receiving request data from `/dev/accel`, the VirtIO frontend handles the relevant structures, adds the data to the virtqueue, and notifies the host. The VirtIO backend then reads the request from the virtqueue and passes the necessary information to the host vAccelRT through the corresponding VMM backend. These VMM backends act as glue layers that prepare arguments and pass the actual request to vAccelRT. They are linked with vAccelRT and use the library's User API.

To illustrate the workflow, two primary steps must be followed to enable the offloading of computations from the guest VM to the host:

1. Select the host plugin upon starting the VM. This plugin contains the accelerator code necessary for executing the offloaded computations.

2. Load the `vaccelrt-plugin-virtio` as the guest plugin. This plugin facilitates the communication between the guest VM and the host through the `virtio-accel` module.

When a computation task is initiated within the Virtual Machine, vAccelRT uses the `vaccelrt-plugin-virtio` to create a request and forward the operation call to the host VMM. The host VMM (such as QEMU or Firecracker) processes the request and invokes the corresponding operation on the vAccelRT host library. This library then uses the loaded accelerator plugin to perform the computation on the host's accelerator hardware.

This workflow ensures efficient and low-overhead offloading of computational tasks from the guest VM to the host system, leveraging the capabilities of the accelerator plugin.

In this case, the significant advantage of the vAccel framework -apart from offloading the computation- is its transparency to users:

- **Source Code Compatibility**: Users do not need to modify their source code to benefit from the computation offloading. The vAccel framework handles the communication and offloading process transparently.
- **Seamless Execution**: The entire procedure of sending tasks to the host, processing them, and returning the results is managed by the vAccel framework, ensuring seamless execution from the user's perspective.

3.5.2 Evaluation

To demonstrate the improvements offered by vAccel, we present measurements for two examples: BERT model inference and image classification inference, both using Torch. Table 3.2 shows the execution time for both examples running inside a container (vanilla execution), the execution time when offloading computation to the Host CPU using the vsock plugin, and the respective overhead. Additionally, the table highlights the execution time when offloading computation to the host GPU via the vsock plugin and the speedup achieved compared to the vanilla execution in the container.

Operation	Container vCPU	vAccel-vsock Host CPU	Overhead	vAccel-vsock Host GPU	Speedup GPU vs vCPU
Img Classif	51.5 ms	66.5 ms	29.12%	5 ms	10.3 x
Bert	125 ms	138 ms	10.4%	12 ms	10.41 x

Table 3.2: Measurements to evaluate the performance of the vAccel-vsock plugin.

These results highlight the efficiency and performance advantages of leveraging vAccel to offload computations to a hardware accelerator on the host. They demonstrate that the benefits of reducing computation costs outweigh the associated communication overhead, making it a worthwhile addition. This validates the effectiveness of vAccel in optimizing computational tasks.

Chapter 4

Optical Flow and OpenCV Background

4.1 OpenCV Overview

OpenCV [16] (Open Source Computer Vision Library) is an open-source computer vision and machine learning software library. The library, which is cross-platform and free for use under the BSD license, has become a widely utilized tool in both academic and commercial settings for various computer vision applications.

4.1.1 History and Development

OpenCV was launched in 1999 with the aim of providing a common infrastructure for computer vision applications and accelerating the use of machine perception in commercial products. Over the years, the library has grown significantly. It is written primarily in C++ and includes extensive Python bindings, making it accessible to a wide range of users. OpenCV's modular structure is composed of several shared or static libraries, each focusing on different areas of computer vision.

4.1.2 Core Functionality

The core functionality of OpenCV includes a vast array of image processing and computer vision algorithms. These functions span from basic image manipulations to advanced machine learning techniques, enabling developers to create robust computer vision applications.

Image Processing Image processing is at the heart of OpenCV, with functions that cover:

- **Filtering:** Techniques such as Gaussian, median, and bilateral filtering for noise reduction and image smoothing.
- **Transformations:** Geometric transformations, including scaling, rotation, translation, and affine transformations.
- **Color Space Conversions:** Converting between different color spaces such as RGB, HSV, Lab, and grayscale.
- **Edge Detection:** Algorithms like Canny edge detection and Sobel operators for identifying edges in images.
- **Morphological Operations:** Operations such as erosion, dilation, opening, and closing to process shapes within images.
- **Histograms:** Calculation and equalization of histograms for image analysis and enhancement.

Video Analysis OpenCV provides robust tools for video analysis, which include:

- **Background Subtraction:** Methods to segment moving objects from a static background in video sequences.
- **Object Tracking:** Algorithms such as Kalman filters, Meanshift, and Camshift for tracking objects across frames.
- **Optical Flow:** Techniques to estimate the motion of objects between successive frames of video.

Feature Detection and Description Detecting and describing features is crucial for many computer vision tasks. OpenCV includes:

- **Keypoint Detection:** Algorithms like Harris corner detector, FAST, and AGAST for identifying keypoints in images.
- **Feature Descriptors:** Methods such as SIFT, SURF, ORB, and BRIEF for describing the appearance of keypoints.

- **Feature Matching:** Techniques for matching features between different images using descriptors and matching algorithms like FLANN and BFMatcher.

Object Detection OpenCV's object detection capabilities include:

- **Haar Cascades:** Pre-trained classifiers for face detection and other object detection tasks.
- **Deep Learning-based Detection:** Integration with deep learning frameworks to use models like YOLO, SSD, and Faster R-CNN for object detection.

Machine Learning OpenCV incorporates machine learning tools that support both traditional machine learning and deep learning, including:

- **Classifiers:** Support for various classifiers such as K-Nearest Neighbors (KNN), Support Vector Machines (SVM), Decision Trees, and Random Forests.
- **Clustering:** Algorithms like K-Means and Mean Shift for clustering data points.
- **Neural Networks:** Interfaces for creating and training neural networks, with support for popular deep learning frameworks like TensorFlow and PyTorch.

4.1.3 Applications

OpenCV's extensive functionality makes it applicable in various domains, including but not limited to:

- **Automotive Safety:** Advanced driver assistance systems (ADAS) that utilize vision techniques for lane detection, traffic sign recognition, and pedestrian detection.
- **Robotics:** Vision systems for autonomous navigation, object manipulation, and interaction with the environment.
- **Healthcare:** Medical imaging applications for diagnostics, surgical assistance, and treatment planning.
- **Augmented Reality:** Real-time overlay of digital content on the physical world, using vision-based tracking and alignment.

- **Surveillance:** Security systems that employ video analysis for monitoring and detecting suspicious activities or individuals.
- **Retail:** Applications in customer behavior analysis, product recognition, and inventory management.
- **Entertainment:** Use in games and interactive installations for gesture recognition, facial animation, and visual effects.

4.1.4 Integration with Other Frameworks

OpenCV's versatility is enhanced by its ability to integrate seamlessly with other frameworks and libraries. For instance:

- **Deep Learning Frameworks:** Integration with TensorFlow, PyTorch, Caffe, and ONNX for deploying deep learning models.
- **Hardware Acceleration:** Support for CUDA and OpenCL to leverage GPU acceleration for computationally intensive tasks.
- **IoT and Edge Devices:** Compatibility with platforms like Raspberry Pi and NVIDIA Jetson for deploying vision applications on edge devices.
- **Other Libraries:** Interoperability with libraries such as NumPy for numerical operations and SciPy for advanced scientific computing.

In summary, OpenCV is a powerful and flexible library that plays a critical role in the field of computer vision. Its comprehensive set of functionalities, ease of integration, and active community support make it an essential tool for developing innovative computer vision applications.

4.2 Optical Flow Calculation

Optical flow [17], [18] is a fundamental concept in computer vision that describes the pattern of apparent motion of objects, surfaces, and edges in a visual scene caused by the relative motion between an observer (e.g., a camera) and the scene. It plays a crucial role in various applications, including video compression, object tracking, motion estimation, and autonomous navigation.

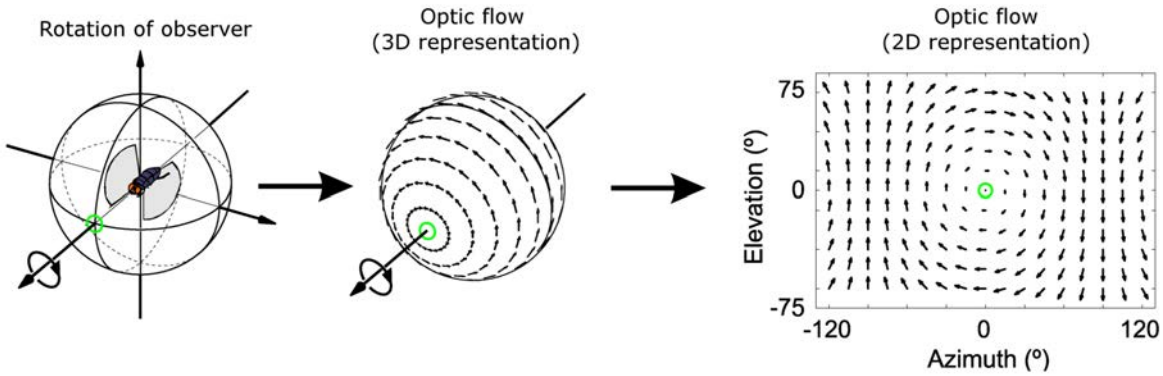


Figure 4.1: The optic flow experienced by a rotating observer (in this case a fly). The direction and magnitude of optic flow at each location are represented by the direction and length of each arrow (from [19]).

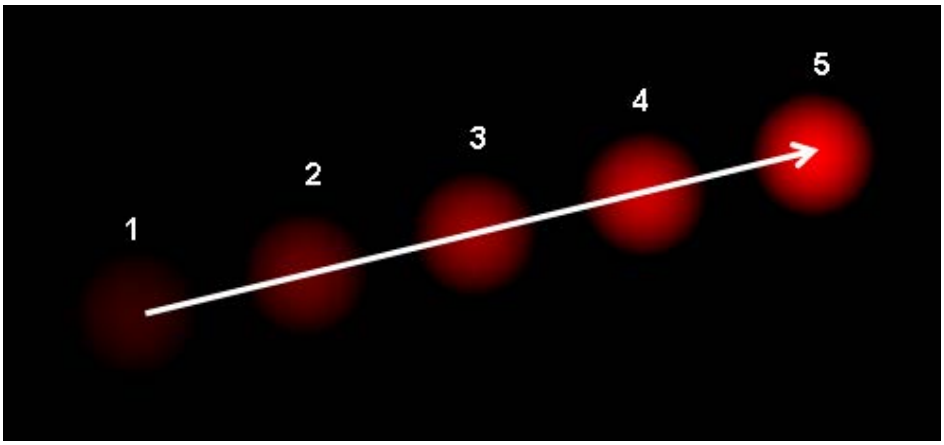


Figure 4.2: The optical flow vector of a moving object in a video sequence (from [20]).

4.2.1 Dense vs. Sparse Optical Flow

Optical flow estimation techniques can be categorized into dense and sparse methods. Dense optical flow provides a detailed motion field by estimating the motion vector for every pixel in the image. In contrast, sparse optical flow calculates motion vectors only for selected feature points, which are typically points of interest or keypoints within the scene.

Dense Optical Flow: Methods such as the Farneback algorithm use polynomial approximations to estimate the motion for each pixel, offering high accuracy and detailed motion information. This approach is computationally intensive and best suited for applications requiring precise motion analysis, such as motion segmentation.

Sparse Optical Flow: Methods like the Lucas-Kanade (PyrLK) algorithm track a limited number of feature points, resulting in faster computations. This approach is advantageous

for real-time applications like object tracking, especially in scenarios with moderate motion. Sparse methods are effective when the scene contains distinct and stable features that can be reliably tracked.

4.2.2 Optical Flow Tools in OpenCV

OpenCV provides a comprehensive suite of tools for computing optical flow [21], including both sparse and dense methods. For sparse optical flow, OpenCV offers the `cv::calcOpticalFlowPyrLK` function, which implements the Lucas-Kanade method using image pyramids to track feature points across frames. This method is suitable for applications where only a subset of points needs to be tracked efficiently. For dense optical flow, OpenCV includes the `cv::calcOpticalFlowFarneback` function, which computes the dense optical flow using the Gunnar Farnebäck's algorithm. This method is useful for obtaining a dense vector field of motion estimates over the entire image. For more advanced applications, the DeepFlow, SimpleFlow, and RLOF (Robust Local Optical Flow) algorithms are also available through OpenCV's `cv::optflow` module.

4.3 Challenges

Although various types and implementations of Optical Flow Algorithms exist, they all encounter the accuracy-speed trade-off [22]. To illustrate this, we will focus on the two most popular algorithms: **Farneback** (Dense) and **Pyramid Lucas-Kanade** (Sparse).

As mentioned above, Farneback calculates the motion of **every pixel** between two consecutive frames, providing a comprehensive overview of movement. In contrast, Pyramid Lucas-Kanade first identifies ”good features” in the initial frame and then tracks these features in the subsequent frame (Figure 4.3). Consequently, this sparse algorithm produces a set of vectors representing the motions of these features. Generally, this set is much smaller than the total number of pixels, and therefore much smaller than Farneback's output. Thus, while the Dense algorithm provides detailed overall information about movements between frames, the Sparse algorithm only captures a subset of this information.

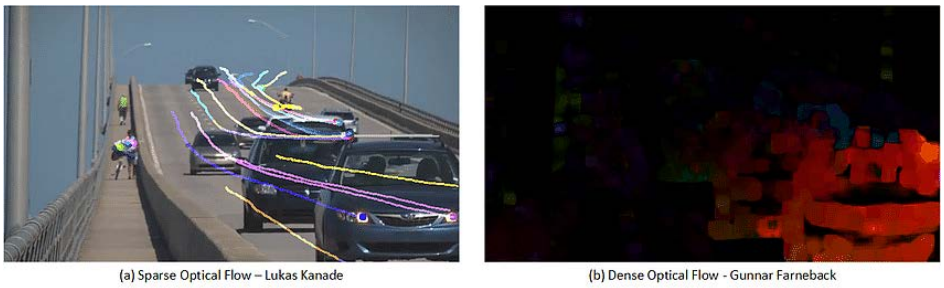


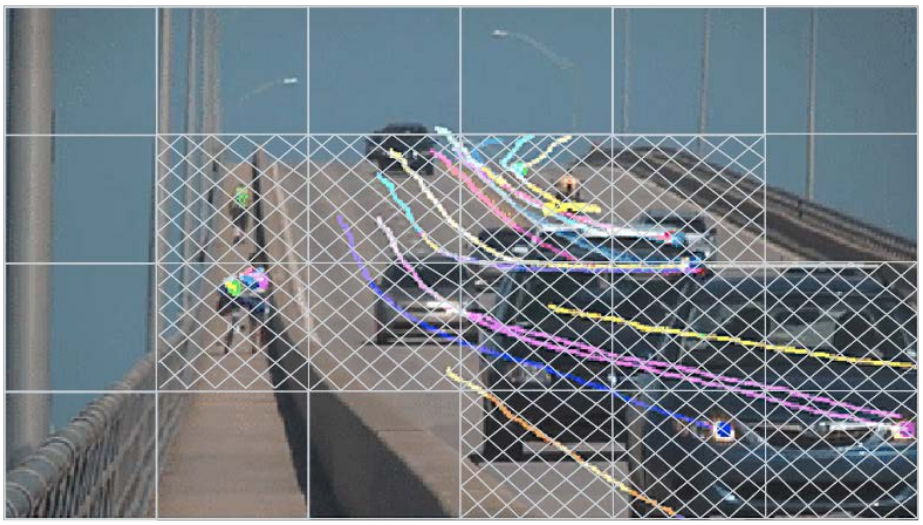
Figure 4.3: Optical flow output for sparse vs dense approach (from [23].)

Roughly speaking, Farneback achieves high accuracy but requires significantly more time and power to run. To fully understand the advantages and disadvantages of each method, we need to compare their results on the same video. However, the outputs of Sparse and Dense algorithms are dissimilar and not directly comparable. We can overcome this issue by homogenizing the outputs of the two methods. Instead of detecting moving features or pixels, we modify PyrLK and Farneback to **detect subframes of the image that contain motion**. This modification allows for a meaningful qualitative comparison of each algorithm's performance. The following figures demonstrate how PyrLK implements this detection.

Step 1: Read the new frame

Step 2: Split in subframes

Step 3: Locate moving subframes



Finally, data from various videos were collected and categorized. ”Static” refers to videos where consecutive frames are identical, indicating a stable image. ”Stereo” describes videos where the camera is stationary, with only a few subframes showing motion. ”Moving” videos feature intense changes and many moving subframes. ”Mixed” videos contain both static and moving scenes. Table 4.1 presents the results of PyrLK for each video category, with Farneback used as a reference.

Observing the results of Table 4.1, we notice 3 details:

1. First, the Sparse Algorithm performs -much- better in terms of **Speed**

	Quality Loss	Time Reduction	Energy Reduction
mixed 1	17.19%	97.99%	84.85%
mixed 2	31.8%	98.02%	84.28%
moving	87.13%	98.96%	88.03%
stereo	6.76%	99.73%	88.05%
static	0%	98.49%	86.69%

Table 4.1: Pyramid Lucas-Kanade Results, compared to Farneback

2. Second, the Sparse Algorithm performs better in terms of **Power Consumption**

3. Last, we see that PyrLK presents a negligible **Quality Loss** in videos with none or just a few moving subframes, but it loses significant information in videos with many features moving.

The points presented above clearly indicate that no single method is sufficient on its own. While Farneback claims to capture all image features, its performance is lacking. Conversely, PyrLK improves speed and reduces power consumption but has difficulty detecting many moving subframes in videos with intense changes. The **trade-off** between the two methods is evident: Farneback provides maximum accuracy but suffers from poor performance, whereas PyrLK excels in speed but delivers accurate results only for videos with low or no movement.

The results suggest a flexible approach: **we don't need to rely solely on Farneback or PyrLK for all situations. The optimal strategy is to use Farneback during moments of intense and complex scene changes, and switch to PyrLK when the movement is minimal.**

Chapter 5

Argument I/O Functions

5.1 Existing Infrastructure

To enable the vaccel framework to communicate with various plugins or multiple vaccel-agents across different hosts, it employs a generic method for transferring arguments. Unless the operation and plugin developers have written specific code to bypass this protocol, operations and plugins will communicate using the following data structure:

```
struct vaccel_arg {
    uint32_t size;
    void *buf;
};
```

Essentially, operation calls which use the generic method, have at least 4 arguments:

1. an array of `vaccel_arg`'s, which will be used for the input/read arguments
2. a second array of `vaccel_arg`'s, which will be used for the output/write arguments
3. an unsigned integer which represents the number of read arguments and
4. a second unsigned integer which represents the number of the write arguments.

Therefore, a vAccel Call could look like this, if we wanted to pass 2 integers as input, and receive 2 floats as output:

```
struct vaccel_arg read[2];
struct vaccel_arg write[2];
int in1 = 5;
```

```
int in2 = 10;
float out1, out2;

read[0].buf = &in1;
read[0].size = sizeof(int);

read[1].buf = &in2;
read[1].size = sizeof(int);

write[0].buf = &out1;
write[0].size = sizeof(float);

write[1].buf = &out2;
write[1].size = sizeof(float);

ret = vaccel_op(sess , ..., read , 2, write, 2) ;
```

Correspondingly, a Plugin function would look like this:

```
static
int vaccel_plugin(...,  struct vaccel_arg *read, size_t nr_read,
                            struct vaccel_arg *write, size_t nr_write) {

    int in1 = *read[0].buf;
    int in2 = *read[1].buf;
    float out1, out2;

    /* Process the input and calculate out1, out2 */

    memcpy(write[0].buf, &out1, sizeof(float));
    memcpy(write[1].buf, &out2, sizeof(float));
}
```

It is evident that this method of reading and writing arguments is not ideal. Additionally, it appears that this process could be automated, to become shorter and more straightforward. Another issue that may not be immediately apparent is the handling of non-serialized arguments. For example, if we want to add the following structure as an input argument, the framework would be unable to recognize that `arr` is merely an address. Consequently, sending this data remotely would be meaningless.

```
struct test_arg {
    int size;
    int *arr;
}
```

5.2 Design and Implementation

Our goals with this contribution are threefold: first, to simplify and clarify the process for a vAccel-operation developer to define read and write arguments and extract the plugin's output; second, to streamline the process for a vAccel-plugin developer to read the main program's provided arguments and write to the designated area defined by the frontend; and third, to address the issue of non-serialized arguments by incorporating support for them in the functions we will implement.

5.2.1 Main Program/Frontend

On the first hand, vAccel frontend programs perform three main operations related to arguments:

1. **add** Input arguments to the plugin
2. define **expected** arguments that will receive by the plugin
3. **extract** plugin's output arguments

These operations apply to both serialized and non-serialized arguments. Consequently, we need to provide the user with six functions instead of three (Figure 5.1).

5.2.2 Plugin/Backend

On the other side, vAccel backend programs perform two main operations related to arguments:

1. **extract** frontend's input arguments
2. **write** to previously defined (as expected) output arguments

Likewise, the operations apply to both serialized and non-serialized arguments. Therefore, we need to provide the user with four functions instead of two (Figure 5.2).

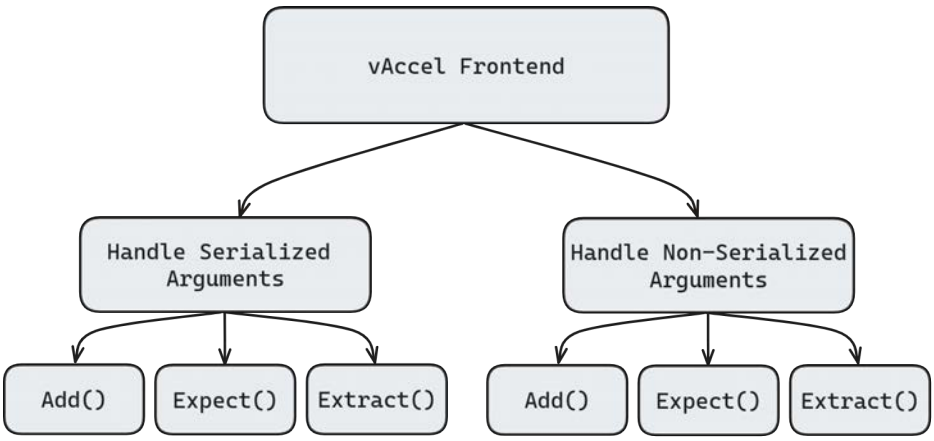


Figure 5.1: The Requirements of a vAccel frontend.

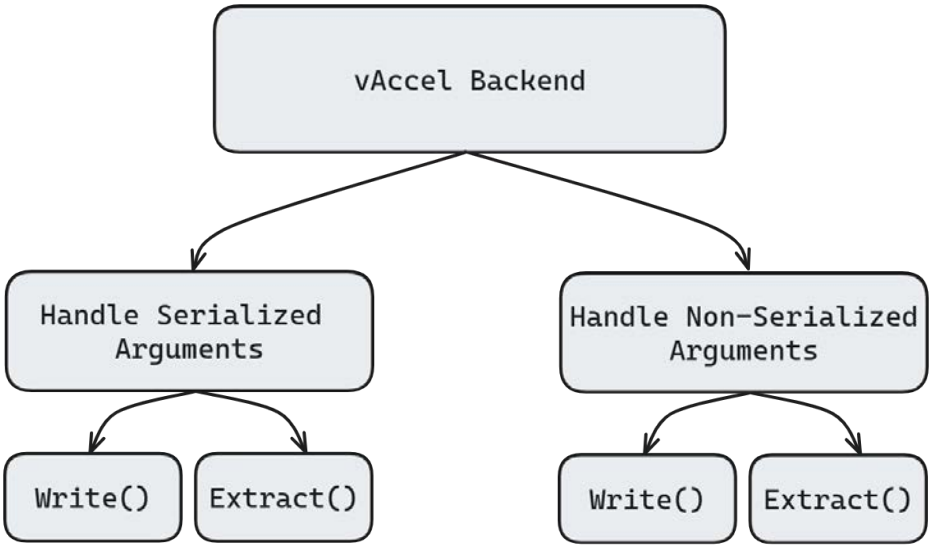


Figure 5.2: The requirements of a vAccel backend.

5.2.3 Non-serialized Arguments

While the use of serialized arguments (such as integers, characters, etc.) is straightforward, we must exercise greater caution with non-serialized arguments. These arguments need to be adequately prepared before being transferred remotely. This preparation involves processing the data to generate a serialized sequence of bytes that can be safely sent to another address space. For example, the previously mentioned structure `struct test_arg` is not properly formatted to be passed as an argument. If we wish to pass the following instance as an argument, we need to ensure it is correctly serialized, as shown in Figure 5.3.

```
struct test_data input {
    .size = 3,
    .arr = {12, 4, 9}
```

```
}
```

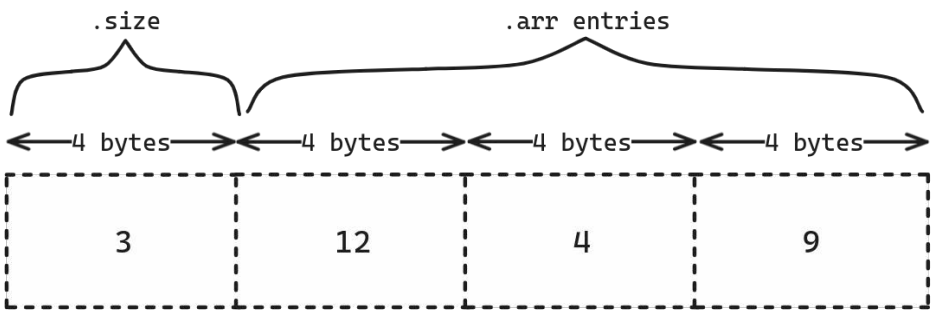


Figure 5.3: Representation of a serialized instance of `test_data` in memory.

Otherwise, the data would be rendered useless because it contains an address that is invalid in a different address space (Figure 5.4).

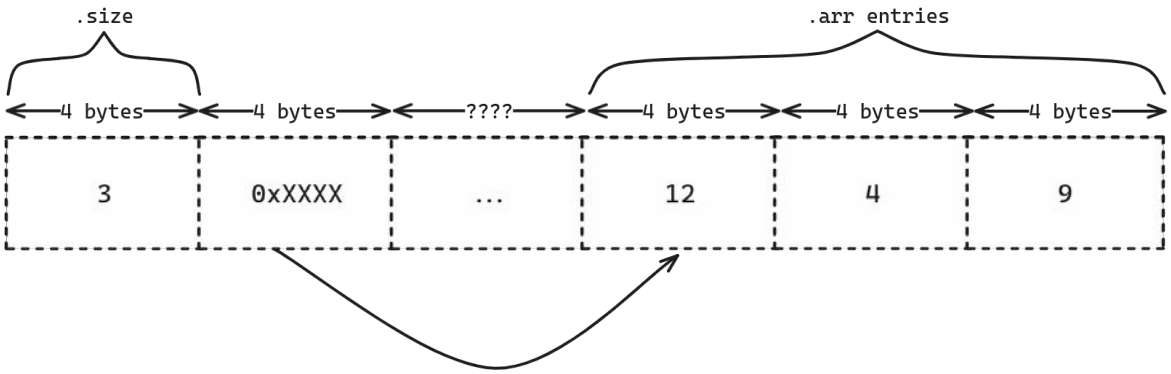


Figure 5.4: Representation of a `test_data` instance in memory, before serializing it.

To address this, the argument API we provide should include functions that allow developers to pass a function for serializing the given data. While developers could serialize their arguments directly in the main program and pass them as serialized arguments, incorporating serializer/deserializer functions presents an opportunity for future enhancements. By adding an **argtype** field to the **vaccel_arg** structure, we can specify the type of data to be passed as an argument. This allows us to define a set of important or popular structures within the vAccel Framework and subsequently implement functions to serialize and deserialize these structures. As a result, users can specify the argument type they wish to add, and vAccel will automatically use the corresponding functions to handle the data. This approach significantly simplifies the process of writing a vAccel program.

```
struct vaccel_arg {
    uint32_t argtype;
```

```
        uint32_t size;
        void *buf;
    };
```

5.2.4 API

Now, we will present the changes made to integrate the argument API into the Framework. First, the types of functions used to serialize and deserialize data are as follows:

```
/*
 * The signature of serializer functions
 *
 *  - buf:     pointer to data that will get serialized
 *  - nbytes:  buffer to write serialized data length
 *  - returns: pointer to the serialized data (contains malloc)
 */
void* serialize(void* buf, uint32_t* nbytes);

/*
 * The signature of deserializer functions
 *
 *  - buf:     pointer to serialized data
 *  - nbytes:  the size of the serialized sequence in bytes
 *  - returns: retrieved structure address (contains malloc)
 */
void* deserialize(void* buf, uint32_t nbytes);
```

Additionally, to facilitate the entire process of reading and writing arguments, the following structure has been introduced:

```
/*
 * The structure that will hold the list
 * of read/write arguments that will be passed.
 *
 *  - capacity:             the capacity of the list
 *  - list:                 pointer to the vaccel_arg entries
 *  - curr_idx:             index to insert the next entry
 *  - idcs_allocated_space: An array that holds 1 in positions
 *                          that memory is allocated using malloc(),
 *                          so it can be freed later automatically
```

```
 *
 */
struct vaccel_arg_list {
    uint32_t capacity;
    struct vaccel_arg *list;
    int curr_idx;
    int *idcs_allocated_space;
};
```

The following piece of code demonstrates the prototypes of the argument helper functions. It includes all the required operations mentioned above, handling both serialized and non-serialized arguments:

```
    /* Initializes the arg-list structure */
    struct vaccel_arg_list* vaccel_args_init(uint32_t size);

    /* Add a serialized argument in the list */
    int vaccel_add_serial_arg(
        struct vaccel_arg_list* args,
        void* buf,
        uint32_t size
    );

    /* Add a non-serialized argument in the list */
    int vaccel_add_nonserial_arg(
        struct vaccel_arg_list* args,
        void* buf,
        uint32_t argtype,
        void* (*serializer)(void*, uint32_t*)
    );

    /* Define an expected argument (serialized) */
    int vaccel_expect_serial_arg(
        struct vaccel_arg_list* args,
        void* buf,
        uint32_t size
    );

    /* Define an expected non-serialized argument */
    int vaccel_expect_nonserial_arg(
```

```
    struct vaccel_arg_list* args,
    uint32_t expected_size
);

/* Extract a serialized argument out of the list */
void* vaccel_extract_serial_arg(
    struct vaccel_arg* args,
    int idx
);

/* Extract a non-serialized argument out of the list */
void* vaccel_extract_nonserial_arg(
    struct vaccel_arg* args,
    int idx,
    void* (*deserializer)(void*, uint32_t)
);

/*
Write to an expected serialized argument.
Used inside plugin.
*/
int vaccel_write_serial_arg(
    struct vaccel_arg* args,
    int idx,
    void* buf
);

/*
Write to an expected non-serialized argument.
Used inside plugin.
*/
int vaccel_write_nonserial_arg(
    struct vaccel_arg* args,
    int idx,
    void* buf,
    void* (*serializer)(void*, uint32_t*)
);
```

```
/* Delete any allocated memory in the arg-list structure*/
int vaccel_delete_args(struct vaccel_arg_list* args);
```

5.3 Example

To demonstrate the usage of the new functions, we provide an example focusing on the process of handling serialized arguments. In this case, we want to pass two integers as input and receive a float from the plugin as output.

```
/* Initialize the argument list */
struct vaccel_arg_list* read = vaccel_args_init(2);
struct vaccel_arg_list* write = vaccel_args_init(1);

if (!read || !write)
    [...]

int in1 = 10;
int in2 = 15;
float out;

/* First Argument */
if (vaccel_add_serial_arg(read, &in1, sizeof(int)))
    [...]

/* Second Argument */
if (vaccel_add_serial_arg(read, &in2, sizeof(int)))
    [...]

/* Define an Expected output */
if (vaccel_expect_serial_arg(write, &out, sizeof(float)))
    [...]

/* Plugin */
ret = vaccel_op(..., read->list, read->capacity,
                            write->list, write->capacity);

if (ret)
    [...]
```

```
/* Extract the output (optional, could just read 'out') */
float* outptr = vaccel_extract_serial_arg(write->list, 0);

if (!outptr)
    [...]

/* Delete the lists */
if (vaccel_delete_args(read))
    [...]

if (vaccel_delete_args(write))
    [...]
```

5.4 Evaluation

If we compare the source code of this example with the initial version at the beginning of the section, it becomes evident how much the new functions enhance the workflow of writing a vAccel program. The new argument API significantly simplifies the process of handling data, making the code more readable and easier to maintain.

However, it is crucial to ensure that the new functions do not introduce any overhead that negatively impacts the overall performance of a vAccel program. To evaluate this, we conducted measurements using three examples. Initially, we ran these examples in their original form, without the argument helpers in vAccel. Subsequently, we modified the examples to utilize the new argument functions. Table 5.1 presents these measurements, highlighting the negligible overhead introduced by the API.

	Without Arguments-API	**With Arguments-API**	**Overhead**
Example 1	5.46 ms	5.60 ms	2.56%
Example 2	3.37 ms	3.39 ms	0.59%
Example 3	6 ms	6 ms	0%

Table 5.1: Comparing performance with and without the usage of the helper functions.

This minor overhead is primarily due to additional memory allocation during the initialization of the structure. To address this issue in future work, we could optimize memory

allocation by using a standard area in stack memory for the `vaccel_arg_list` structure by default. If the size required by the user is smaller than this predefined area, no additional memory allocation would be necessary. Only in the rare cases where the user requires more memory would we resort to using `malloc()`. This approach would minimize the overhead associated with memory allocation and improve the overall efficiency of the vAccel program.

Chapter 6

Integration of OpenCV functions in vAccel

6.1 Architecture

One of the operations provided by the vAccel Framework is `vaccel_opencv`. This function accepts generic arguments through `vaccel_arg` lists, enabling developers to delegate the operation to a plugin that implements this functionality (Figure 6.1). Despite being a single function, `vaccel_opencv` can execute all available OpenCV functions included in the plugin. This versatility is achieved by passing the operation ID of the desired OpenCV function as an argument. Specifically, vAccel assigns a unique ID number to each OpenCV function implemented by the plugin. Consequently, when an OpenCV call is made, the plugin identifies the operation from the ID and selects the appropriate function to execute (Figure 6.2). However, to simplify this process, we use wrapper functions that internally call the `vaccel_opencv` function with the corresponding ID. These wrapper functions abstract the complexity of dealing with operation IDs, letting us invoke OpenCV functions more intuitively and directly. By using these wrappers, developers can call specific OpenCV functions without needing to manually manage the IDs or construct `vaccel_arg` lists for each operation. For instance, if a developer needs to apply a Gaussian blur using OpenCV, instead of manually setting up the arguments and determining the correct operation ID, they can simply call a wrapper function like `vaccel_gaussian_blur`. This wrapper function will internally handle the construction of the `vaccel_arg` list, assign the appropriate ID, and call the `vaccel_opencv` function.

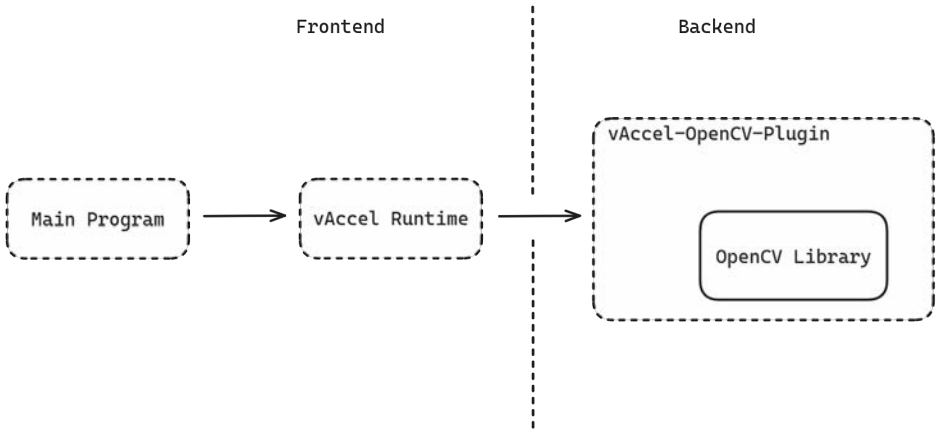


Figure 6.1: vAccel OpenCV plugin

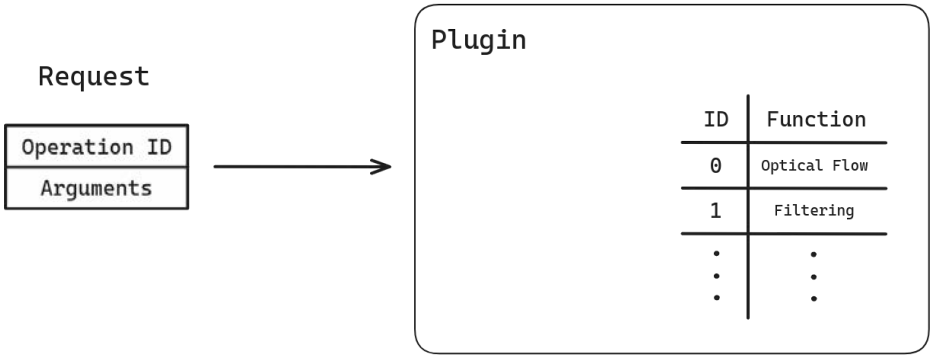


Figure 6.2: Select function based on ID

In our case, we aim to implement a function/operation that processes a sequence of frames by reading a video. For each pair of frames, the function will attempt to locate moving subframes. We want to achieve this using two different methods and on two different devices. As a result, we will have the combinations presented in Table 6.1.

	Device	**Method**
Backend Function 1	CPU	PyrLK (Sparse)
Backend Function 2	CPU	Farneback (Dense)
Backend Function 3	GPU	PyrLK (Sparse)
Backend Function 4	GPU	Farneback (Dense)

Table 6.1: Combinations of methods and devices for ”Moving Subframe Detection”.

Therefore, for each one of the combinations in Table 6.1, we add a function in the vAccel frontend that act as wrappers, packaging the arguments and offload the computation to a plugin. And afterwards, we shall implement the corresponding functions as vAccel Backends.

Here, it is important to note that we return the moving subframes instead of the flow for simplicity. While returning the flow would be straightforward, we needed to find a way to standardize the results of both the sparse and dense methods for quality comparison. By using moving subframes, we can homogenize the outputs of the two methods, allowing for a more consistent and meaningful comparison of the plugins' performance.

6.2 Implementation

6.2.1 Frontend

The function prototypes we provide to vAccel programs are the following:

```
void movingSubfrPyrLKCPU(
    Mat img1,
    Mat img2,
    int subfrGridDim,
    uint8_t* result
);

void movingSubfrFarneCPU(
    Mat img1,
    Mat img2,
    int subfrGridDim,
    uint8_t* result
);

void movingSubfrPyrLKGPU(
    Mat img1,
    Mat img2,
    int subfrGridDim,
    uint8_t* result
);

void movingSubfrFarneGPU(
    Mat img1,
    Mat img2,
    int subfrGridDim,
    uint8_t* result
```

```
);
```

Additionally, a program that uses these functions will have the following form:

```
cv::VideoCapture video(path);
if (!video.isOpened())
    [...]

cv::Mat gray_frame1, gray_frame2, frame;

video >> frame;

if (frame.empty())
    [...]

cv::cvtColor(frame, gray_frame1, cv::COLOR_BGR2GRAY);

while (1) {
    video >> frame;
    if (frame.empty())
        break;

    cv::cvtColor(frame, gray_frame2, cv::COLOR_BGR2GRAY);

    vaccel::movingSubfrMethodDevice(
        gray_frame1,
        gray_frame2,
        gridDim,
        result
    );

    std::swap(gray_frame1, gray_frame2);
}
```

Finally, the following code demonstrates the process of defining function arguments before invoking `vaccel_opencv`. Both serialized and non-serialized data are managed using functions introduced in the previous sections. In this example, we utilize the `serialize_cv_mat` function from the vAccel OpenCV Bindings [24] to serialize an OpenCV frame (`cv::Mat`).

```
struct vaccel_arg_list *read  = vaccel_args_init(5);
struct vaccel_arg_list *write = vaccel_args_init(1);
```

```
if (!read || !write)
    [...]

int ret = 0;

/* --- Read Args --- */
ret |= vaccel_add_serial_arg(read, op, sizeof(uint8_t));
ret |= vaccel_add_serial_arg(read, type, sizeof(int));
ret |= vaccel_add_nonserial_arg(read, &img1, 0, serialize_cv_mat);
ret |= vaccel_add_nonserial_arg(read, &img2, 0, serialize_cv_mat);
ret |= vaccel_add_serial_arg(read, subfrGridDim, sizeof(int));

if (ret)
    [...]

/* --- Expected Args --- */
if (vaccel_expect_serial_arg(write, resultGrid, sizeof(resultGrid)))
    [...]

/* --- Plugin --- */
ret = vaccel_opencv(sess, read->list,
            read->size, write->list, write->size);
```

6.2.2 Backend

Since we have demonstrated the frontend part, let's now explore the implementations that reside on the backend, specifically within the vAccel plugin. For simplicity, we will showcase 2 out of the 4 functions: the Moving Subframe Detection using PyrLK on CPU and the one using Farneback on GPU.

```
int movingSubfrPyrLKCPU(struct vaccel_session *session, struct vaccel_arg *read,
                    size_t nr_read, struct vaccel_arg *write, size_t nr_write) {

    Mat *fr1, *fr2;
    int subfrGridDim;

    fr1 = vaccel_extract_nonserial_arg(read, 0, deserialize_cv_mat);
```

```
    fr2 = vaccel_extract_nonserial_arg(read, 1, deserialize_cv_mat);

    subfrGridDim = *(int*) vaccel_extract_serial_arg(read, 2);

    uint8_t resultGrid[subfrGridDim][subfrGridDim];

    if (!feat_detection_done)
        /* Peform Feature Detection */

    cv::calcOpticalFlowPyrLK(*fr1, *fr2, p1, p2, status, err);

    pyrlkResultGrid(&resultGrid[0][0], subfrGridDim,
                fr1->rows, fr1->cols, p1, p2, status, err);

    vector<Point2f> goodPoints;
    for (uint i = 0; i < status.size(); i++)
        if (status[i] && err[i] < 10.0)
            goodPoints.push_back(p2[i]);

    if (goodPoints.size() <= 0.3 * numFeaturesToTrack)
        /* Many features are lost => Redetect */

    p1.clear();
    p2.clear();

    fr1->release();
    fr2->release();

    p1 = goodPoints;

    return vaccel_write_serial_arg(write, 0, resultGrid);
}

int movingSubfrFarneGPU(struct vaccel_session *session, struct vaccel_arg *read,
                        size_t nr_read, struct vaccel_arg *write, size_t nr_write) {

    Mat *fr1, *fr2, flow;
    int subfrGridDim;
    Ptr<cuda::FarnebackOpticalFlow> d_farnFlow;
```

```
    fr1 = vaccel_extract_nonserial_arg(read, 0, deserialize_cv_mat);
    fr2 = vaccel_extract_nonserial_arg(read, 1, deserialize_cv_mat);
    subfrGridDim = *(int*) vaccel_extract_serial_arg(read, 2);

    uint8_t resultGrid[subfrGridDim][subfrGridDim];

    d_farnFlow = cuda::FarnebackOpticalFlow::create(levels,
        pyrScale, false, winsize, iters, polyN, polySigma, flags);

    cuda::GpuMat d_fr1(*fr1);
    cuda::GpuMat d_fr2(*fr2);
    cuda::GpuMat d_flow;

    d_farnFlow->calc(d_fr1, d_fr2, d_flow);
    d_flow.download(flow);

    farneResultGrid(&resultGrid[0][0], subfrGridDim, flow);

    fr1->release();
    fr2->release();

    return vaccel_write_serial_arg(write, 0, resultGrid);
}
```

In the above implementations, the functions `pyrlkResultGrid()` and `farneResultGrid()` determine which subframes of the image contain moving features or pixels and return the response to the user. The ”result grid” is a 2D matrix where entries are 1 if the subframe is detected as ”moving” and 0 if there are no moving elements. For example, the result grid for the frame in Figure 6.3 is as follows:

```
0 0 0 0 0 0
0 1 1 1 1 0
0 1 1 1 1 1
0 0 0 1 1 1
```

Furthermore, it’s important to note some details regarding vAccel plugins. Since we have written our functions, we must explicitly define what operations they implement. Addition-

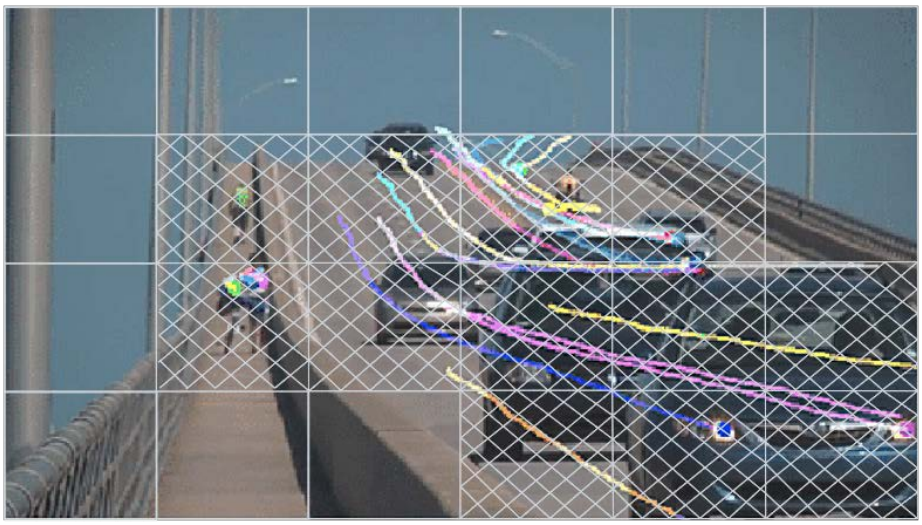


Figure 6.3: Moving subframe detection using PyrLK.

ally, we need to specify other information, such as the type of the plugin and the functions to initialize it. The following code is an example that shows how to provide these details to the core framework, illustrating how a vAccel plugin is created.

```
struct vaccel_op ops[] = {
    VACCEL_OP_INIT(ops[0], VACCEL_OPENCV, (void *) opencv_dispatch),
};

int opencv_plugin_init(void) {
    int ret = register_plugin_functions(ops,
                    sizeof(ops) / sizeof(ops[0]));
    if (!ret)
        return ret;

    return VACCEL_OK;
}

int opencv_plugin_fini(void) {
    return VACCEL_OK;
}

VACCEL_MODULE(
    .name = "opencv",
    .version = VACCELRT_VERSION,
```

```
    .init = opencv_plugin_init,
    .fini = opencv_plugin_fini,
    .type = VACCEL_PLUGIN_CPU | VACCEL_PLUGIN_GPU
)
```

1. `ops[]` array is used to export the functions we have written inside the plugin to vAccel, along with the operations they implement. Here, we declared that `opencv_dispatch()` (the function that will select function based on Operation ID) implements `VACCEL_OPENCV` operation.

2. `opencv_plugin_init()` is the function that will be called when the plugin is loaded by the core framework.

3. correspondingly, `opencv_plugin_fini()` is the function that will be called when the plugin is unloaded.

4. finally, `VACCEL_MODULE()` is a macro that will explicitly provide all this information to vAccel.

Thus, after following the aforementioned steps, we can construct the vAccel plugin in the form of a shared object. The path to this shared object can then be provided to a vAccel program using the `VACCEL_BACKENDS` environment variable. The vAccel core library will load the object and register the functions we implemented (indirectly, via `opencv_dispatch()`). The only remaining step to access the plugin is to call the functions implemented in the frontend: `movingSubfrPyrLKCPU()`,`movingSubfrPyrLKGPU()`,`movingSubfrFarneCPU()`, and `movingSubfrFarneGPU()`.

6.3 Evaluation

In this section, we will present a comprehensive analysis of the results for all the implemented functions, comparing them against the Farneback-CPU method. This analysis will cover three key aspects: Quality Loss, Time of Execution Reduction, and Energy Reduction. By examining these factors, we aim to provide a detailed evaluation of the effectiveness and efficiency of the different approaches, highlighting their strengths and weaknesses.

6.3.1 Dataset

Regarding the dataset used to evaluate the performance of the adapting mechanism, we selected five videos that encompass a wide range of scenarios. All the videos were captured by us.

First, the `static` video features a constant scene with no movement, resulting in identical pixels throughout the video. Next, the `stereo` video was captured by a stationary camera, leading to scenes with minimal changes. In this video, there is only one object moving on the screen, meaning that although the scene is not entirely static, only a small portion of the pixels are in motion.

The `moving` video captures a dynamic environment with continuous movement throughout the scene. This type of video challenges the adapting mechanism to handle significant and consistent pixel changes effectively. The `mixed 1` and `mixed 2` videos present more complex scenarios, where a combination of static and dynamic elements are present. These videos include both stationary and moving objects, providing a comprehensive test for the adapting mechanism to handle varying levels of motion and scene changes.

By using this diverse set of videos, we ensure that the evaluation covers all possible scenarios, enabling a thorough assessment of the adapting mechanism's performance under different conditions.

6.3.2 PyrLK-CPU vs Farneback-CPU

The first comparison, between PyrLK-CPU and Farneback-CPU is demonstrated in Table 6.2.

	Quality Loss	Time Reduction	Energy Reduction
mixed 1	17.19%	97.99%	84.85%
mixed 2	31.8%	98.02%	84.28%
moving	87.13%	98.96%	88.03%
stereo	6.76%	99.73%	88.05%
static	0%	98.49%	86.69%

Table 6.2: PyrLK-CPU results, compared to Farneback-CPU.

Based on the provided table, here are some comments on the differences:

- **Quality Loss**:
  - The Quality Loss varies significantly among the different scenarios.
  - For example, ”moving” has the highest Quality Loss at 87.13%, indicating a substantial degradation in quality when using PyrLK-CPU compared to Farneback-CPU.
  - On the other hand, ”static” has no Quality Loss (0%), suggesting that in static scenarios, PyrLK-CPU performs as well as Farneback-CPU in terms of quality.
- **Time Reduction**:
  - The Time Reduction is very high across all scenarios, with values ranging from 97.99% to 99.73%.
  - This indicates that PyrLK-CPU is significantly faster than Farneback-CPU, reducing the processing time by almost 98-99%.
- **Energy Reduction**:
  - The Energy Reduction also shows substantial savings, with rates between 84.28% and 88.05%.
  - This suggests that using PyrLK-CPU is much more energy-efficient compared to Farneback-CPU, with energy savings of over 84% in all scenarios.

Overall, while PyrLK-CPU offers considerable improvements in processing time and energy efficiency, it can lead to substantial quality loss in certain scenarios like moving scenes. However, in static scenarios, PyrLK-CPU provides both efficiency and quality compared with Farneback-CPU.

6.3.3 Farneback-GPU vs Farneback-CPU

The second comparison, between Farneback-GPU and Farneback-CPU is demonstrated in Table 6.3.

Based on the provided table, here are some comments on the differences:

- **Quality Loss**:

	Quality Loss	Time Reduction	Energy Reduction
mixed 1	0.05%	80.92%	34.87%
mixed 2	0.03%	81.29%	25.39%
moving	0.05%	82.23%	30.18%
stereo	0%	79.64%	29.06%
static	0%	79.68%	34.26%

Table 6.3: Farneback-GPU results, compared to Farneback-CPU.

  - The Quality Loss is minimal across all scenarios, with values close to 0%, indicating almost no degradation in quality when using Farneback-GPU compared to Farneback-CPU.

- **Time Reduction**:
  - The Time Reduction is significant across all scenarios, with values ranging from 79.64% to 82.23%.
  - This indicates that Farneback-GPU is considerably faster than Farneback-CPU, reducing the processing time by approximately 80%.

- **Energy Reduction**:
  - The Energy Reduction shows notable savings, with rates between 25.39% and 34.87%.
  - This suggests that using Farneback-GPU is more energy-efficient compared to Farneback-CPU, with energy savings of over 25% in all scenarios.

Overall, Farneback-GPU offers significant improvements in processing time and energy efficiency while maintaining almost identical quality to Farneback-CPU across all scenarios.

6.3.4 PyrLK-GPU vs Farneback-CPU

The last comparison, between PyrLK-GPU and Farneback-CPU is demonstrated in Table 6.4.

Based on the provided table, here are some comments on the differences:

- **Quality Loss**:

	Quality Loss	Time Reduction	Energy Reduction
mixed 1	17.19%	92.42%	41.08%
mixed 2	31.8%	94.7%	42.45%
moving	87.13%	94.39%	45.3%
stereo	6.76%	95.21%	44.83%
static	0%	93.24%	43.72%

Table 6.4: PyrLK-GPU Results, compared to Farneback-CPU

  - The Quality Loss varies significantly among the different scenarios.
  - For example, the ”moving” scenario has the highest Quality Loss at 87.13%, indicating substantial degradation in quality when using PyrLK-GPU compared to Farneback-CPU.
  - In contrast, the ”static” scenario has no Quality Loss (0%), suggesting that in static scenarios, PyrLK-GPU performs as well as Farneback-CPU in terms of quality.

- **Time Reduction**:
  - The Time Reduction is high across all scenarios, with values ranging from 92.42% to 95.21%.
  - This indicates that PyrLK-GPU is significantly faster than Farneback-CPU, reducing the processing time by approximately 92-95%.

- **Energy Reduction**:
  - The Energy Reduction shows considerable savings, with rates between 41.08% and 45.3%.
  - This suggests that using PyrLK-GPU is much more energy-efficient compared to Farneback-CPU, with energy savings of over 41% in all scenarios.

Overall, while PyrLK-GPU offers significant improvements in processing time and energy efficiency, it can lead to substantial quality loss in certain scenarios like scenes with intense changes. However, in static scenarios, PyrLK-GPU provides both efficiency and quality on compared with Farneback-CPU.

6.3.5 Summary

A summary of the information from the provided tables, displayed in three bar charts (Figures 6.4, 6.5, 6.6):

- **Quality Loss Comparison**:
  - PyrLK-CPU and PyrLK-GPU have significant quality loss in the ”moving” scenario.
  - Farneback-GPU has minimal quality loss across all scenarios, maintaining close to 0% in all cases.
- **Time Reduction Comparison**:
  - PyrLK-CPU shows the highest time reduction across most scenarios, reaching up to 99%.
  - Farneback-GPU shows a time reduction of around 80%, which is lower compared to PyrLK-CPU and PyrLK-GPU.
  - PyrLK-GPU provides a substantial time reduction, although slightly lower than PyrLK-CPU, ranging from 92% to 95%.
- **Energy Reduction Comparison**:
  - PyrLK-CPU shows high energy reduction, with the highest value being around 88%.
  - Farneback-GPU has lower energy reduction, around 30%.
  - PyrLK-GPU shows good energy reduction, better than Farneback-GPU but generally lower than PyrLK-CPU, ranging from 41% to 45%.

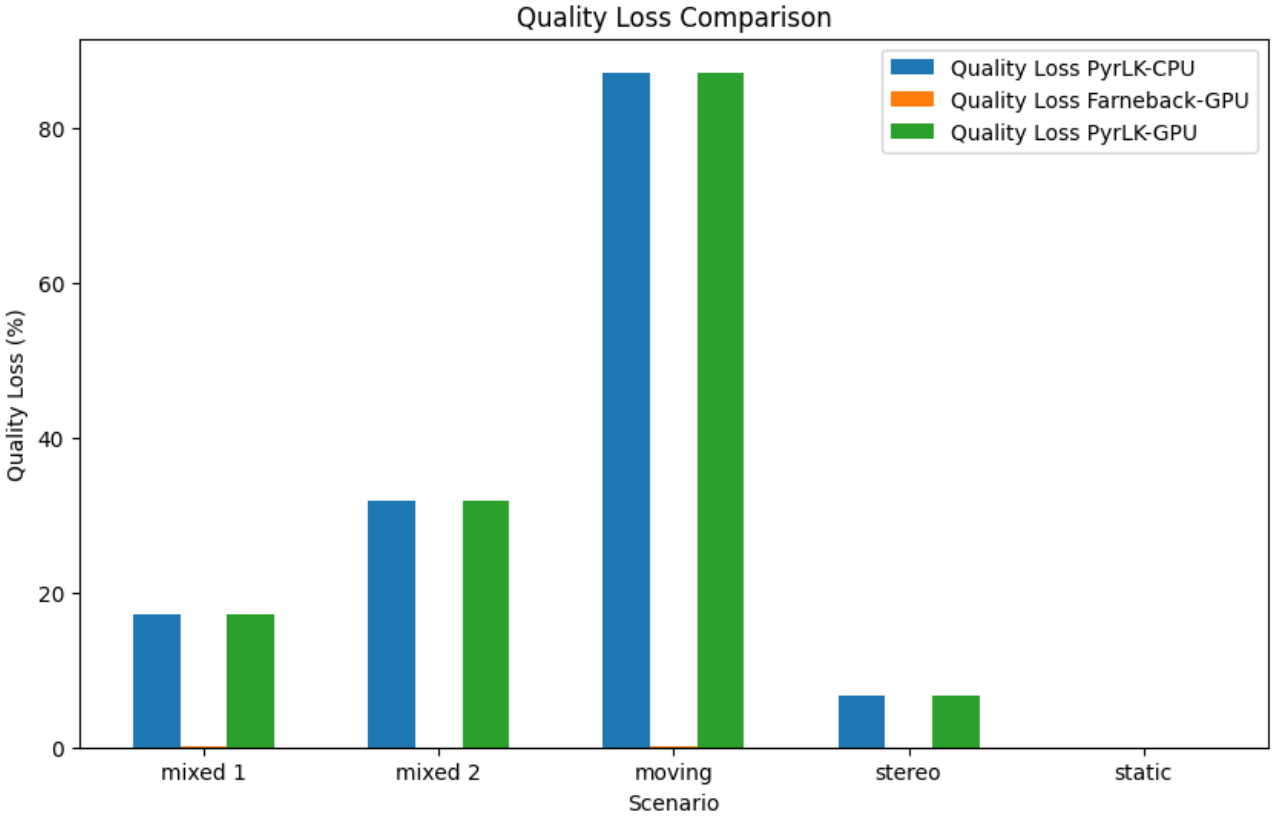


Figure 6.4: Quality loss comparison of the tested functions vs Farneback-CPU.

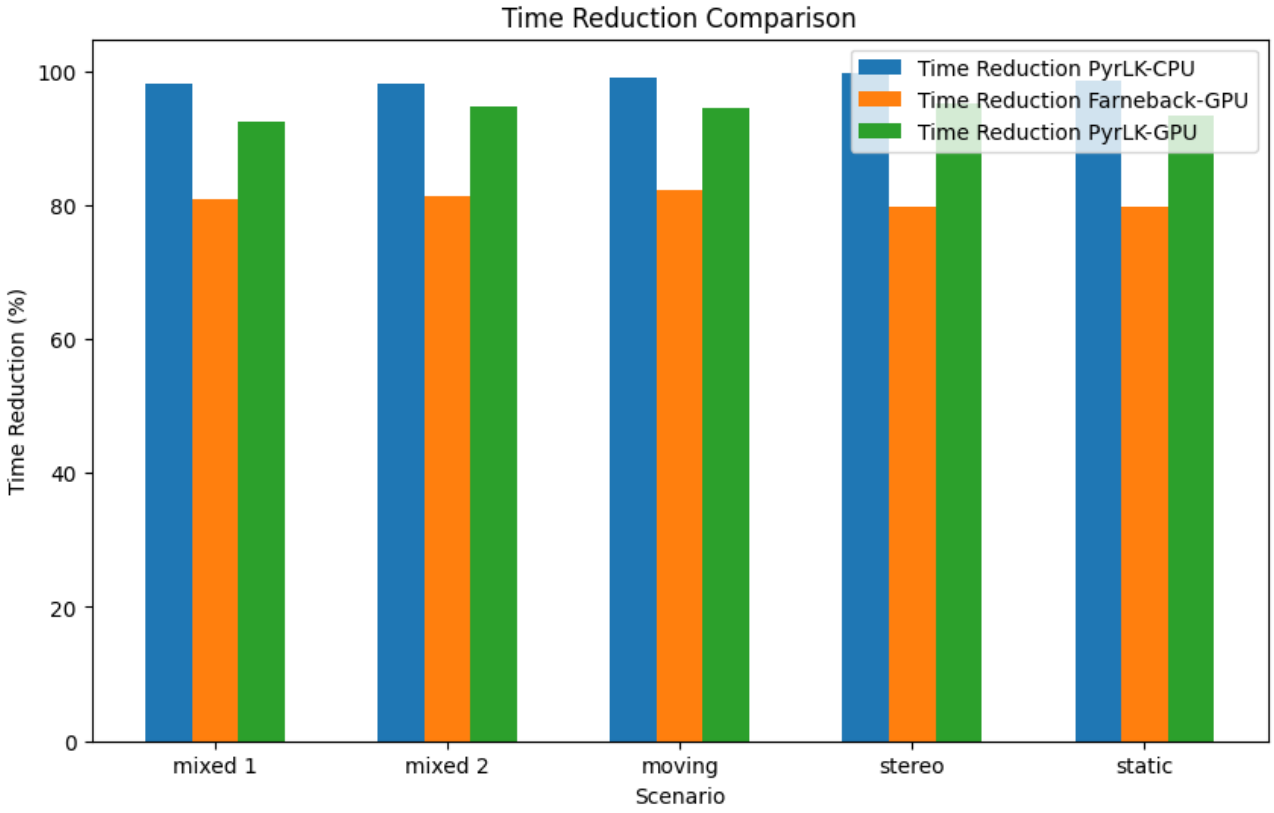


Figure 6.5: Time reduction of the tested functions vs Farneback-CPU.

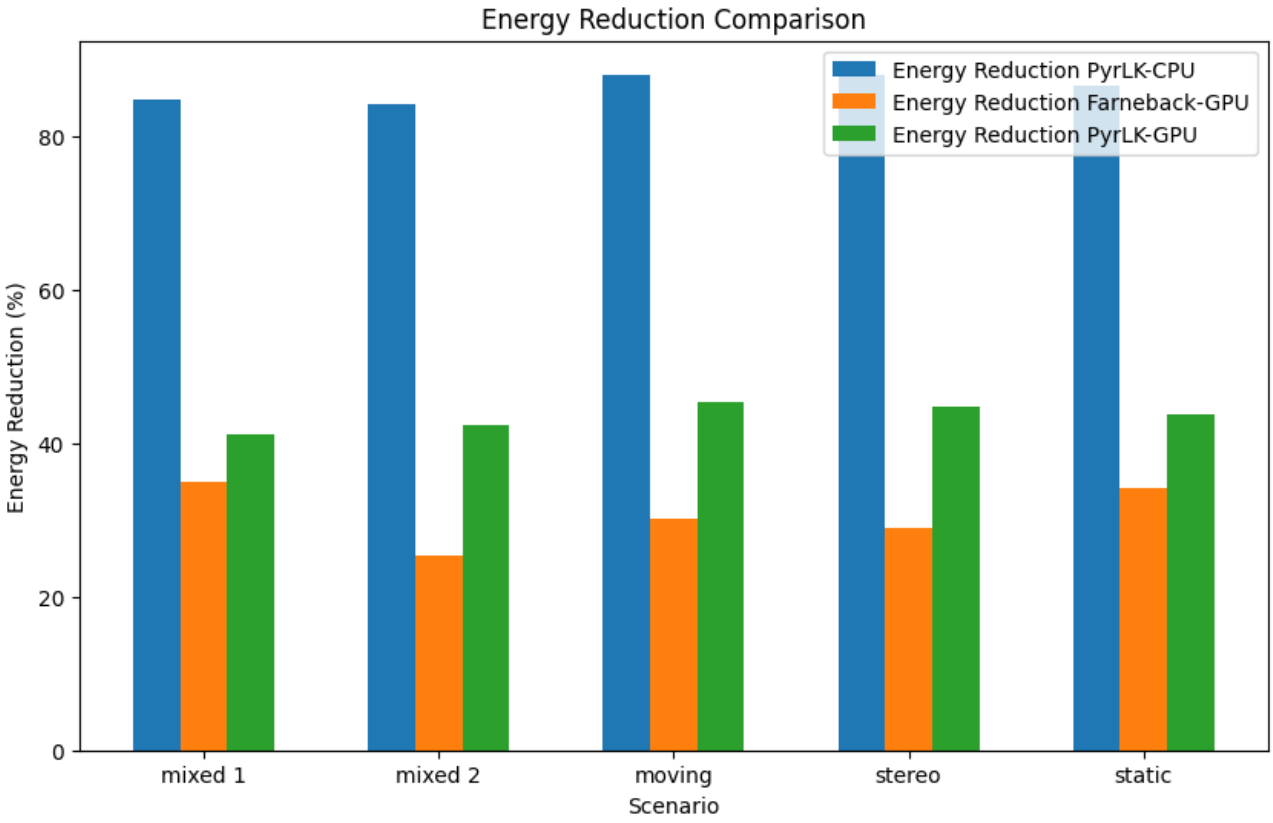


Figure 6.6: Energy reduction of the tested functions vs Farneback-CPU.

Chapter 7

Mechanism to Optimize Optical Flow Plugin Selection at Runtime

According to the measurements in the previous section, it is evident that no single function (method-device combination) is ideal for all scenarios. Our objective is to enable the framework to dynamically select the optimal function for offloading each optical flow (moving subframe detection) call. This selection will be based on specific criteria regarding the performance of the plugins. This section focuses on the design and development of this selection mechanism, ultimately leading to its integration into the vAccel framework. Our goal is to ensure transparency for the user. Therefore, we aim to provide a single function that internally manages the entire process of selecting the most appropriate offloading method for each call.

7.1 Interchange Plugins on Runtime

This work is made possible by the flexibility provided by vAccel, which allows us to update our session and switch between plugins at runtime. Specifically, when initializing a vAccel session, we can provide a hint to `vaccel_sess_init()`. This hint, represented as a bitmap, indicates the preferred types of plugins for the operation (Figure 7.1). If no specific type is defined (initialized as 0), vAccel will offload the computation to the first available plugin that implements the requested operation. Otherwise, it will scan the available plugins to check for a match with the current session hint, ultimately returning the first matching plugin. Correspondingly, when building a plugin, we can define its types using the

`VACCEL_MODULE()` macro, as shown earlier. This macro allows us to specify the characteristics and capabilities of the plugin, enabling vAccel to effectively match the plugin with the current session hint. The function that we can use at runtime to update the session hint, thereby switching between plugins, is `vaccel_sess_update()`. This function allows us to modify the session preferences dynamically, ensuring that the optimal plugin is selected based on the current operational requirements and available resources. By using `vaccel_sess_update()` internally (hidden from the user), we can seamlessly adapt to changing conditions and maintain high performance and accuracy for each computation. This ensures that the process of selecting the optimal plugin remains transparent, providing a smooth and efficient user experience while leveraging the full capabilities of the vAccel framework. Although vAccel provides some predefined types that can be used by the users, such as:

- `VACCEL_PLUGIN_CPU = 0x0001`,
- `VACCEL_PLUGIN_GPU = 0x0002`,
- `VACCEL_PLUGIN_FPGA = 0x0004` etc

there are no types that specifically cover our use case. Therefore, we can define our own types and use them in both the frontend and backend.

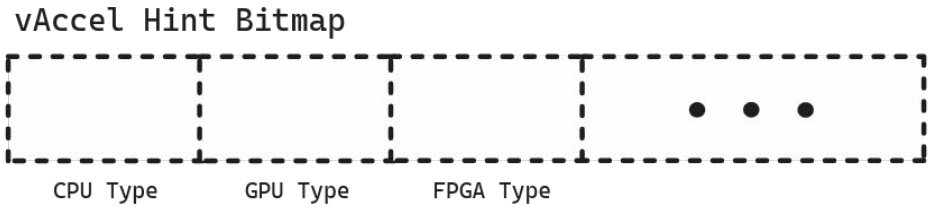


Figure 7.1: vAccel session hint, used to match application invocation calls to plugins.

7.2 Architecture

7.2.1 Criteria

To enable this mechanism, we must determine the criteria by which the selector will operate. A simple yet efficient way to decide which method is preferable for the current state

of the video is to inspect the result grid generated by the plugin. The result grid, as mentioned earlier, is a 2D bitmap where each (i,j) entry holds a value of 1 or 0, indicating whether the (i,j) subframe of the image is moving or not. We set a threshold for the ”Moving Rate” of each method’s response. If the current moving rate exceeds the threshold, we should prefer the dense method (Farneback). Conversely, if there is no significant movement in the frame, we should prefer the sparse method (PyrLK). Algorithms 1, 2 demonstrate this functionality.

Algorithm 1: Sparse Algorithm

Data: $frame1, frame2$

$resultGrid \leftarrow calcMovingSubfrSparse(frame1, frame2)$;

$movingRate \leftarrow calcMovingRate(resultGrid)$;

if $movingRate \geq Threshold$ **then**

  $switchToDense()$;

end

Algorithm 2: Dense Algorithm

Data: $frame1, frame2$

$resultGrid \leftarrow calcMovingSubfrDense(frame1, frame2)$;

$movingRate \leftarrow calcMovingRate(resultGrid)$;

if $movingRate \leq Threshold$ **then**

  $switchToSparse()$;

end

7.2.2 Selection Hint

Afterwards, we need to determine the optimal location for the selection unit. It is important to understand that the various plugins function as independent entities (Figure 7.2). We know that the frontend code will perform the session update to switch between plugins. However, there is a dilemma regarding where to place the Selection-Process. Initially, it seems simpler to place it in the frontend to keep it aligned with the session-update rationale. However, the selection process (moving rate calculation) involves some internal computation. In terms of the vAccel architecture, it is preferable to assign any computation-intensive tasks to backend resources and let the core framework handle resource management. This approach is

particularly important because the vAccel frontend represents a weaker node in the network, primarily responsible for offloading computation to more capable backend resources.

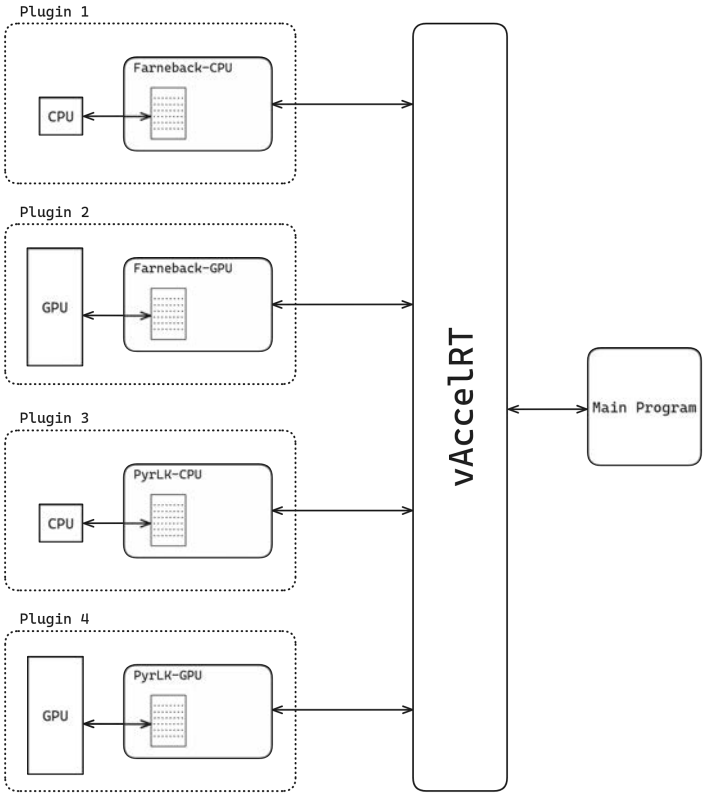


Figure 7.2: vAccel runtime, managing multiple plugins.

Nevertheless, the problem is not yet solved. If the decision regarding the best method is made in the backend, how would the frontend know? To address this, we introduce a second hint to solve this communication issue. This second hint, referred to as the ”selection hint” to avoid confusion with the vAccel session hint, will facilitate information transfer between the backend and frontend. The selection hint will be a bitmap that holds various insights, including the preferable method. After completing its operation, the plugin will determine the appropriate method to handle the next frame and write this information into the selection hint. The selection hint will be transferred to the frontend along with the output. The frontend will then extract the selected method and use this information to update the vAccel session (Figures 7.3, 7.4).

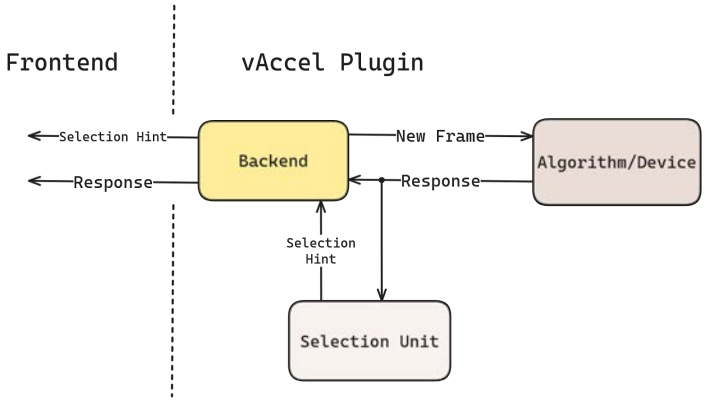


Figure 7.3: Plugin selection process taking place in the backend.

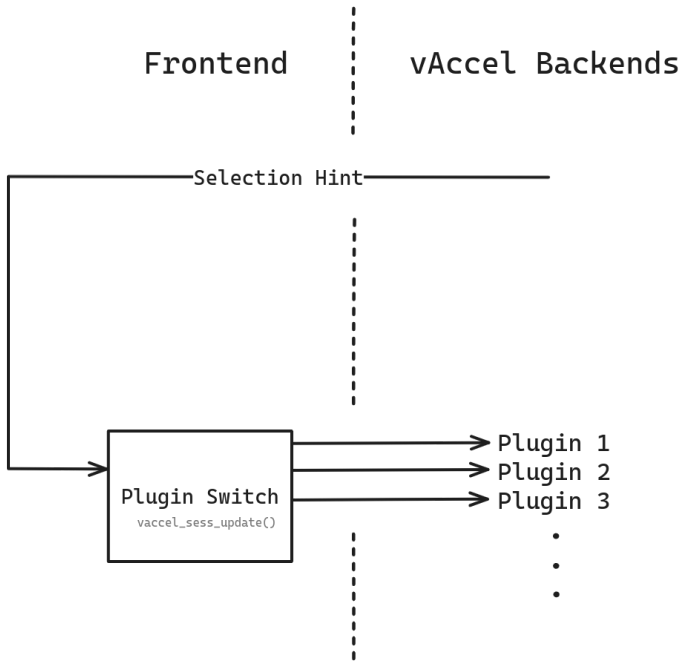


Figure 7.4: Plugin switch in frontend, using the selection hint.

Using the selection hint as a communication mechanism provides us with opportunities to optimize our application. As described earlier, the optical flow operation requires two input frames. However, when applying optical flow to an entire video, there is no need to send the first frame repeatedly. To address this, we will use the selection hint to manage a cache in the plugins. This cache is a memory area where the plugin saves the second frame, to be used as the first frame in the next call. As a result, the frontend only needs to send one frame in subsequent iterations, reducing communication costs.

However, if we switch plugins at runtime, the cached frame will be empty. The selection hint generator solves this problem by notifying the frontend when it needs to send both frames.

Additionally, the generated hint is sent back to the plugin after the call, enabling the backend to determine whether it should read the frame from the user or from the cache. This approach ensures efficient communication and optimized performance by minimizing redundant data transfer while maintaining flexibility in plugin switching.

7.2.3 Device Priority

Finally, another decision we need to make is regarding the device priority policy. The previous section demonstrated the performance of each method and each device. It is clear that Farneback on GPU runs more efficiently than Farneback on CPU, while PyrLK on CPU performs better than PyrLK on GPU (Figures 6.4, 6.5, 6.6). Therefore, if all four plugins are available, there would be no reason to select the GPU for PyrLK or the CPU for Farneback.

Instead of excluding these less efficient plugins, we will exploit a feature of vAccel regarding plugin priority. As mentioned earlier, if the preferred plugin type is supported by more than one plugin, vAccel will select the first available one. To address the availability issue, we will define the plugins in `VACCEL_BACKENDS` using the following priority. For PyrLK: (1) CPU, (2) GPU. For Farneback: (1) GPU, (2) CPU.

By doing this, the core library will automatically check if the preferred device is available and select it if it is. Otherwise, it will fall back to the second choice. This approach ensures that the most efficient device is used whenever possible, while maintaining flexibility and robustness in case of device unavailability.

7.3 Implementation

7.3.1 Backend

Since we have implemented the four plugins in the previous section, we will now modify them to have a single entry point. The appropriate function will be executed based on some internally used macros. The following directory tree is the source directory of the vAccel OpenCV Plugin project. The files `vaccel.cpp` and `vaccel.hpp` contain the pre-existing OpenCV operations. Additionally, the moving subframe detection functions have been added to `vaccel.cpp`. The other files have been created to hold all the support functions that implement the rationale of the Selection Unit, Caching, and Subframe Handling.

```
src
|-- cache.cpp
|-- cache.hpp
|-- sched.cpp
|-- sched.hpp
|-- subframe.cpp
|-- subframe.hpp
|-- vaccel.cpp
|__ vaccel.hpp
```

The following code uses that shows that entry point function.

```
static
int movingSubfr(struct vaccel_session *session, struct vaccel_arg *read,
                size_t nr_read, struct vaccel_arg *write, size_t nr_write) {

    Mat *fr1, *fr2;
    int gridDim;
    unsigned char hint;
    int ret = 0;

    hint = *(unsigned char*) vaccel_extract_serial_arg(read, 0);
    fr2 = vaccel_extract_nonserial_arg(read, 2, deserialize_cv_mat);
    gridDim = *(int*) vaccel_extract_serial_arg(read, 3);

    if (is_cold_start(hint))
        fr1 = vaccel_extract_nonserial_arg(read, 1, deserialize_cv_mat);
    else
        fr1 = read_frame_from_cache();

    uint8_t resultGrid[gridDim][gridDim];

    movingSubfrPlugin(*fr1, *fr2, gridDim, &resultGrid[0][0], hint);

    sched(resultGrid, gridDim, hint);

    if (!is_cold_start(hint))
```

```
        save_to_cache(fr2);

    ret |= vaccel_write_serial_arg(write, 0, resultGrid);
    ret |= vaccel_write_serial_arg(write, 1, &hint);

    if (ret)
        return VACCEL_EINVAL;

    return VACCEL_OK;
}
```

Here are some comments regarding the code:

- `is_cold_start()` scans the selection hint to check if the current call is the first call in this slot, and thereby if should avoid reading the cached frame
- `movingSubfrPlugin()` will use the appropriate method-device combination, based on the current plugin
- `sched()` generates a new selection hint by inspecting the plugin's response.

7.3.2 Frontend

On the other side, in the frontend part of the workflow, we had to interfere in a simpler manner. First, the function that we provide to the user is the following:

```
/* the function that uses the selection unit transparently*/
void movingSubfr(Mat &img1, Mat &img2, int gridDim, uint8_t* result);
```

By using this `movingSubfr()` the developer will enjoy scheduling in a seamless way. Here are some important parts of the internal implementation:

- **Update Session**

  As showm in Table 7.1, every plugin has its own type. `produce_vaccel_hint()` will read the `selection_hint` to check whether the preferable method is PyrLK or Farneback. In the first case, the generated hint will be `1001`. Otherwise, it will be 0110. The generated hint will then be assigned to `vaccel_sess_update()`.

  ```
  vaccel_hint = produce_vaccel_hint(selection_hint);
  vaccel_sess_update(sess, vaccel_hint);
  ```

- **Check if the first frame has to be sent**

  The `selection_hint` will also be used in `is_cold_start()` to determine if the first frame will have to be sent (if it's not cached in the plugin).

```
if (is_cold_start(selection_hint))
    ret |= vaccel_add_nonserial_arg(read, &img1, 0, serialize_cv_mat);
else
    ret |= vaccel_add_nonserial_arg(read, &empty, 0, serialize_cv_mat);

[...]
```

- **Define the Selection Hint and the Result Hint as Expected Arguments**

  We define explicitly that we expect to receive the new Selection Hint from the Plugin.

```
ret  = vaccel_expect_serial_arg(write, resultGrid, sizeof(resultGrid));

ret |= vaccel_expect_serial_arg(write, &selection_hint, sizeof(unsigned char));

if (ret)
    [...]
```

- **vAccel Plugin Call**

```
ret = vaccel_opencv(sess, read->list, read->size, write->list, write->size);

if (ret)
    [...]
```

- **Return the Result Grid to the user and delete the arguments**

  The new selection hint has now been received and saved to be used in the next call.

```
memcpy(result, resultGrid, sizeof(resultGrid));
vaccel_delete_args(read);
vaccel_delete_args(write);
```

7.4 Evaluation

The final step in our project is to evaluate the effectiveness of our mechanism. This will be achieved by presenting comprehensive measurements and data analysis.

Plugin	Type
PyrLK-CPU	0001 (0x0001)
Farneback-CPU	0010 (0x0002)
Farneback-GPU	0100 (0x0004)
PyrLK-GPU	1000 (0x0008)

Table 7.1: Plugin types and codes.

7.4.1 Evaluation of the Mechanism using CPU Plugins

As illustrated in Figure 7.5, the adaptive method exhibits an impressive balance across key performance metrics in the CPU-only scenario. Most notably, it achieves a significantly lower quality loss (2.07%) compared to the PyrLK-CPU method (28.58%), underscoring its ability to maintain high output quality. This considerable reduction in quality loss demonstrates the effectiveness of the adaptive method in scenarios where maintaining data integrity is critical.

In terms of time reduction, the adaptive method achieves a notable 68.07%, which is less than the PyrLK-CPU method's 98.64%, but still substantial. This indicates that while the adaptive method may not be the fastest, it strikes a favorable balance between processing speed and quality retention.

Additionally, the adaptive method achieves a robust energy reduction of 67.16%. Although this is lower than the PyrLK-CPU method's 86.38%, it remains highly significant, particularly given the drastically improved quality metrics. This demonstrates that the adaptive method provides considerable energy savings while maintaining superior quality.

In summary, although the adaptive method consuming slightly more energy and time (**less than twice that of PyrLK**), it achieves a significant reduction in quality loss. Specifically, the quality loss reduction achieved by the adaptive execution is approximately **14 times greater than that of PyrLK**.

7.4.2 Evaluation of the Mechanism using All Plugins

Regarding the execution where all Plugins are available, the adaptive method demonstrates a remarkable balance across key performance metrics, as depicted in Figure 7.6. Notably, it achieves a significantly lower quality loss (2.07%) compared to other methods, highlighting its superior capability in maintaining high output quality. This quality preservation

	Quality Loss	Time Reduction	Energy Reduction
mixed 1	1.07%	76.22%	54.96%
mixed 2	2.04%	62.65%	39.79%
moving	0.49%	3.85%	-3.80%
stereo	6.76%	99.12%	84.85%
static	0%	98.49%	84.41%

Table 7.2: Results of the adaptive execution when only CPU plugins are available vs Farneback-CPU.

is achieved while still offering satisfying time and energy reductions.

The time reduction for the adaptive method stands at a fair 93.93%, closely rivaling the PyrLK-GPU method (93.99%) and significantly outperforming the Farneback-GPU method (80.75%). This demonstrates that the adaptive method is highly efficient in reducing processing time, making it a viable choice for time-sensitive applications.

Moreover, the adaptive method also excels in energy efficiency, achieving a 52.04% reduction. While PyrLK-CPU leads with an 86.38% reduction, the adaptive method's energy savings are substantial, especially considering the minimal trade-off in quality.

	Quality Loss	Time Reduction	Energy Reduction
mixed 1	1.07%	95.17%	73.19%
mixed 2	2.04%	92.76%	65.77%
moving	0.49%	83.31%	27.49%
stereo	6.76%	99.94%	86.27%
static	0%	98.49%	83.06%

Table 7.3: Results of adaptive execution when all plugins are available, compared to Farneback-CPU.

In summary, the adaptive method proves to be a robust and well-rounded option, offering a harmonious balance between maintaining high quality, reducing processing time, and conserving energy. Its ability to deliver substantial quality improvements while still performing efficiently makes it an good choice for diverse application scenarios.

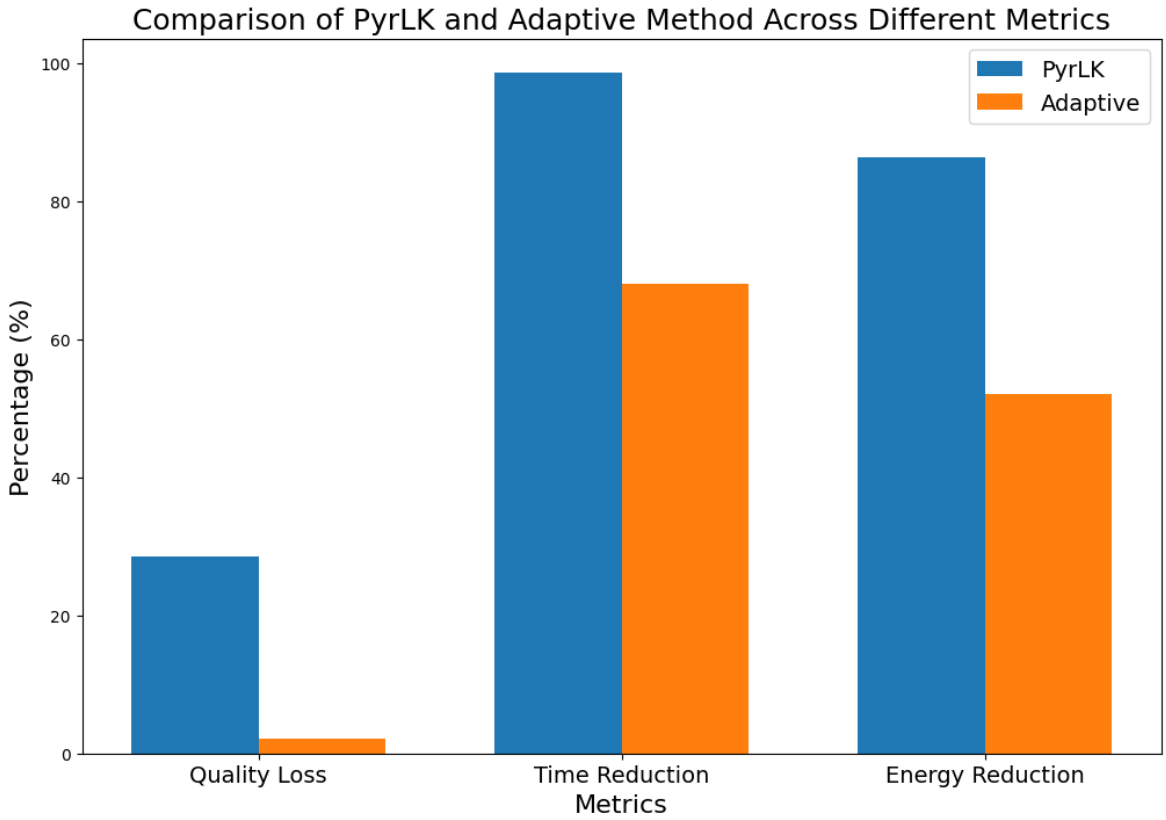


Figure 7.5: Comparison using only CPU plugins vs Farneback-CPU.

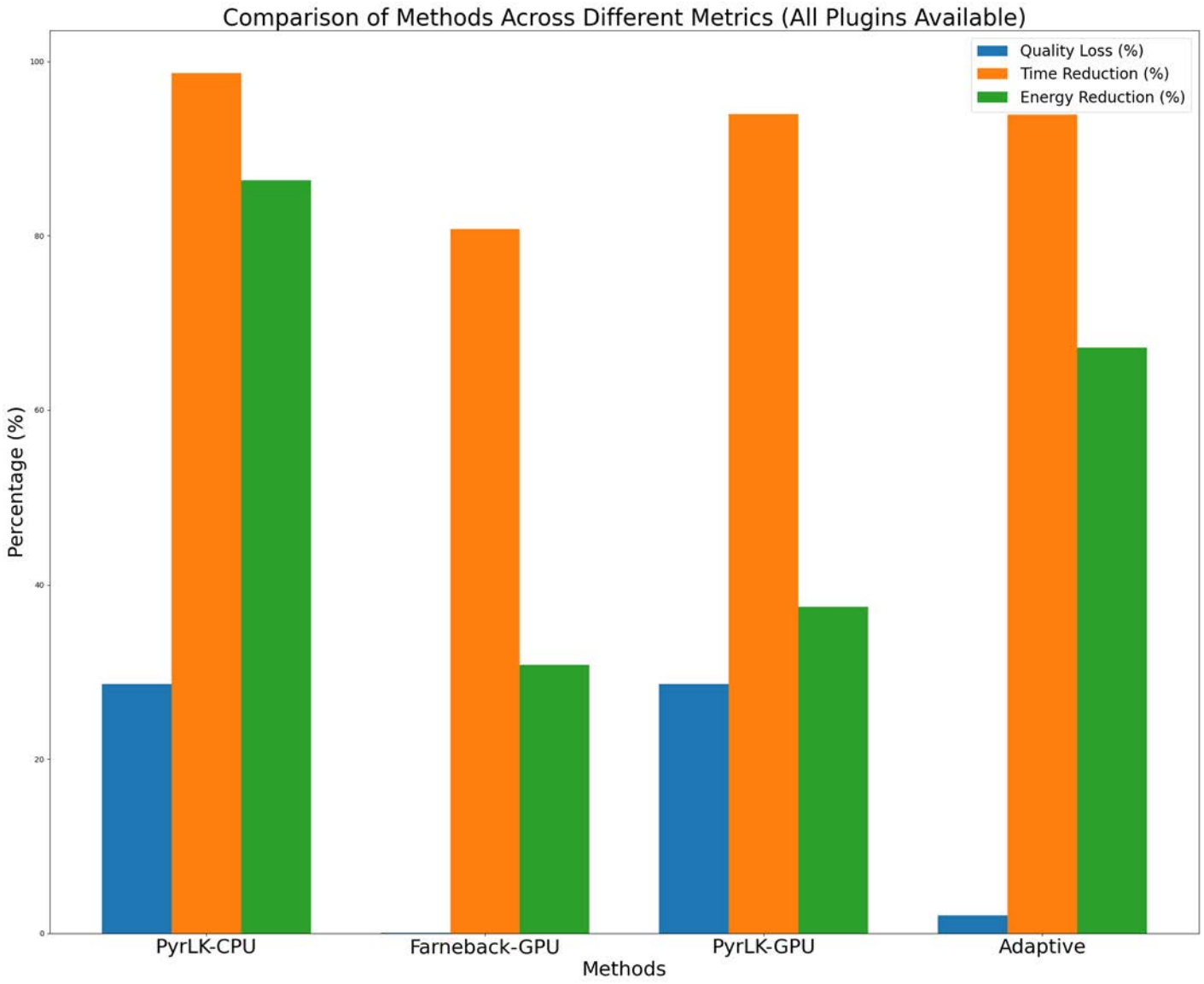


Figure 7.6: Comparison using both CPU and GPU plugins vs Farneback-CPU.

Chapter 8

Conclusion

In this thesis, we explored the utilization of the vAccel Framework to address the accuracy-speed trade-off, using optical flow as a use case. Our analysis focused on comparing the performance of different methods, including Farneback-CPU, PyrLK-CPU, Farneback-GPU, PyrLK-GPU, and the adaptive method, which was built using the selection unit, supported by vAccel core mechanisms.

The results revealed that each method has distinct strengths and trade-offs. The PyrLK-CPU method excels in reducing processing time and energy consumption, making it highly efficient for applications where speed and energy savings are paramount. However, this efficiency comes at the cost of a significant quality loss, which may not be acceptable in scenarios where data integrity is crucial. On the other hand, the adaptive method demonstrated a remarkable ability to maintain high output quality while still achieving substantial reductions in processing time and energy consumption. In the CPU-only scenario, the adaptive method reduced quality loss by approximately 14 times compared to PyrLK-CPU, while consuming only slightly more energy and time. This balance makes the adaptive method a superior choice for applications that prioritize quality without severely compromising on efficiency. When all plugins are available, the adaptive method continued to perform admirably, showcasing its versatility and robustness. It achieved a significant reduction in quality loss while maintaining competitive time and energy reduction rates. This method stands out as a highly efficient and well-rounded solution for diverse application scenarios, offering an optimal balance between quality, processing speed, and energy savings.

In conclusion, the vAccel framework proves to be a powerful tool for flexibly managing the accuracy-speed trade-off in computational tasks. Its capability to perform seamless plugin

switches, along with the ability to utilize framework features within our own libraries, significantly enhances its suitability for a wide range of applications. The adaptive method we developed, in particular, benefits greatly from these features, providing a balanced approach that ensures high-quality output with reasonable time and energy savings.

Future work can build upon these findings by exploring the integration of a generic mechanism within the framework, which would optimize the usage of all plugins transparently without requiring significant engineering effort from developers or even the need to provide explicit plugin selection hints. More specifically, plugin selection could be driven merely by high-level hints provided by the application, allowing the selection to be done fully automatically based on suitable heuristics. Also, machine learning methods could be explored, to train suitable models that will take the right plugin selection decisions at runtime based on certain input characteristics/patterns and the hints provided by the application.

Bibliography

[1] Nvidia dgx a100. https://images.nvidia.com/aem-dam/Solutions/Data-Center/nvidia-dgx-a100-datasheet.pdf.

[2] Intel agilex. https://www.intel.com/content/www/us/en/products/details/fpga/agilex.html.

[3] Video processing with matlab. https://www.mathworks.com/solutions/image-video-processing/video-processing.html.

[4] Autonomous vehicles. https://www.synopsys.com/automotive/what-is-autonomous-car.html.

[5] Motion detection. https://en.wikipedia.org/wiki/Motion_detection.

[6] M. Orshansky J. Han. Approximate computing: An emerging paradigm for energy-efficient design. 2013.

[7] S. Chakradhar K. Roy A. Raghunathan S. Venkataramani, V. K. Chippa. Approximate computing and the quest for computing efficiency. 2015.

[8] H. S. P. Wong S. Mitra. The role of approximate computing in future computer systems. 2014.

[9] Shiji Song Le Yang Honghui Wang Yulin Wang Yizeng Han, Gao Huang. Dynamic neural networks: A survey. 2016.

[10] David Luebke Simon Green John E. Stone John D. Owens, Mike Houston and James C. Phillips. Gpu computing. 2008.

[11] Jonathan Rose Ian Kuon. Measuring the gap between fpgas and asics. 2007.

[12] Norman P. Jouppi. In-datacenter performance analysis of a tensor processing unit. 2017.

[13] Dongarra J. Heterogeneous computing. 2009.

[14] Jeffrey Vetter Sparsh Mittal. A survey of methods for analyzing and improving gpu energy efficiency. 2015.

[15] Comparison of docker container and virtual machine architecture. https://researchgate.net/figure/Comparison-of-Docker-Container-and-Virtual-Machine-Architecture-13_fig1_343764931.

[16] Opencv documentation. https://docs.opencv.org/4.x/.

[17] B. K. P. Horn and B. G. Schunck. Determining optical flow. 1981.

[18] G. Farnebäck. Two-frame motion estimation based on polynomial expansion. 2003.

[19] Optical flow photo 1. https://upload.wikimedia.org/wikipedia/commons/thumb/5/55/Opticfloweg.png/600px-Opticfloweg.png.

[20] Optical flow photo 2. https://upload.wikimedia.org/wikipedia/commons/1/10/Optical_flow_example_v2.png.

[21] Opencv documentation: Optical flow. https://docs.opencv.org/4.x/db/d7f/tutorial_js_lucas_kanade.html.

[22] Optical flow photo 2. https://nanonets.com/blog/optical-flow.

[23] Sparse optical flow vs. dense optical flow. https://researchgate.net/figure/Sparse-Optical-Flow-vs-Dense-Optical-Flow-a-Sparse-Optical-Flow-Lukas-Kanade-b_fig2_361803148.

[24] vaccel opencv bindings. https://github.com/nubificus/opencv-bindings.