ABSTRACT

Title of Dissertation: Methods and Tools for
Real-Time Neural Image Processing

Jing Xie

Dissertation Directed by: Professor Shuvra S. Bhattacharyya (Chair/Advisor)
Dept. of Electrical and Computer Engr., and
Institute for Advanced Computer Studies

Professor Rong Chen (Co-Chair/Co-Advisor)
Department of Diagnostic Radiology
and Nuclear Medicine,
University of Maryland School of Medicine

As a rapidly developing form of bioengineering technology, neuromodulation systems involve extracting information from signals that are acquired from the brain and utilizing the information to stimulate brain activity. Neuromodulation has the potential to treat a wide range of neurological diseases and psychiatric conditions, as well as the potential to improve cognitive function.

Neuromodulation integrates neural decoding and stimulation. As one of the two core parts of neuromodulation systems, neural decoding subsystems interpret signals acquired through neuroimaging devices. Neuroimaging is a field of neuroscience that uses imaging techniques to study the structure and function of the brain and other central nervous system functions. Extracting information from neuroimaging signals, as is required in neural decoding, involves key challenges due to requirements of real-time, energy-efficient, and accurate processing and for large-scale, high resolution image data that are characteristic

of neuromodulation systems.

To address these challenges, we develop new methods and tools for design and implementation of efficient neural image processing systems. Our contributions are organized along three complementary directions. First, we develop a prototype system for real-time neuron detection and activity extraction called the Neuron Detection and Signal Extraction Platform (NDSEP). This highly configurable system processes neural images from video streams in real-time or off-line, and applies techniques of dataflow modeling to enable extensibility and experimentation with a wide variety of image processing algorithms.

Second, we develop a parameter optimization framework to tune the performance of neural image processing systems. This framework, referred to as the NEural DEcoding COnfiguration (NEDECO) package, automatically optimizes arbitrary collections of parameters in neural image processing systems under customizable constraints. The framework allows system designers to explore alternative neural image processing trade-offs involving execution time and accuracy. NEDECO is also optimized for efficient operation on multicore platforms, which allows for faster execution of the parameter optimization process.

Third, we develop a neural network inference engine targeted to mobile devices. The framework can be applied to neural network implementation in many application areas, including neural image processing. The inference engine, called ShaderNN, is the first neural network inference engine that exploits both graphics-centric abstractions (fragment shaders) and compute-centric abstractions (compute shaders). The integration of fragment shaders and compute shaders makes improved use of the parallel computing

advantages of GPUs on mobile devices. ShaderNN has favorable performance especially in parametrically small models.

Methods and Tools for Real-Time Neural Image Processing

by

Jing Xie

Dissertation submitted to the Faculty of the Graduate School of the
University of Maryland, College Park in fulfillment
of the requirements for the degree of
Doctor of Philosophy
2023

Advisory Committee:
Professor Shuvra S. Bhattacharyya, Chair/Advisor
Professor Rong Chen, Co-Chair/Co-Advisor
Professor Manoj Franklin
Professor Donald Yeung
Professor Ramani Duraiswami, Dean's Representative

Dedication

To my family. To my father Xihai Xie, my mother Yongdong Jing, my grandmother Jianqing Chen, and my grandfather Yeliang Jing. To my girlfriend Wenqing Chen.
To faculties and staffs at UMD. To Professor A. Yavuz Oruç. To Professor Behtash Babadi.
To Professor Rajeev Barua. To Professor Ankur Srivastava. To Emily Irwin.
To professors at XJTU. To Professor Xiayu Xu. To Professor Junbo Duan. To Professor Jue Wang. To Professor Qinwu Zhou
To my friend Jiaming Wang, Jiongyu Zhang, Guangyao Shi, Shuangqi Luo, Yutao Chen, Guoliang Liao, Jiahang Wu, Ruoxi Li and all other friends.
To my neighbors Probey Michele, and Frocke John

Acknowledgements

During my Ph.D. study in the US, many people provide support and help for the completion of this dissertation.

First, I would like to thank my research advisor Professor Shuvra Bhattacharyya, and my co-advisor Professor Rong Chen, for their significant support and guidance throughout the journey of my Ph.D. study.

I want to extend my sincere thanks to my dissertation committee members. I am very appreciative of Professor Donald Yeung, Professor Manoj Franklin, and Professor Ramani Duraiswami for serving on my committee and providing encouragement, support, insightful questions, and valuable feedback on my dissertation.

I would also like to thank the OPPO software engineering team. To my mentor Yuzhong Yan, teammates, Abhishek Saxena, Jiangong Chen. To my manager Qiang Qiu.

I would like to extend my appreciation to my lab mates Dr. Xiaomin Wu, Dr. Yaesop Lee, Dr. Lei Pan, Dr. Honglei Li, and Dr. Eung-Joo Lee for their help and insightful discussions. And also great thank to Dr. Jiahao Wu, Dr. Yanzhou Liu for guiding me during different periods of my Ph.D.

Last but not least, I thank my family for their support during my study.

The research underlying this thesis was supported in part by the National Institutes of Health (NIH) / National Institute of Neurological Disorders and Stroke (NINDS) under Grant No. R01NS110421 and the Brain Research through Advancing Innovative Neurotechnologies (BRAIN) Initiative.

Table of Contents

Chapter 1

Introduction

This thesis explores methods and tools for design, optimization, and implementation of real-time neural image processing systems. The thesis presents three main research contributions. The first contribution is a novel system for neuron detection and signal extraction from calcium imaging signals. The second contribution is software tool for parameter optimization that can be applied to arbitrary neural signal processing systems. The third contribution is a lightweight and efficient inference engine for real-time applications on mobile devices. The inference engine can be applied to deep-learning based-neural imaging processing systems, as well as to a wide variety of other types of deep-learning-based applications.

Neuromodulation regulates nerve activity by means of monitoring neural activity and applying stimulation to adjust the release of neurotransmitters, or other signaling molecules, in the brain or peripheral nervous system. Neuromodulation has a variety of applications, including chronic pain management, mood regulation, and the treatment of neurological and movement disorders, such as Parkinson's disease.

Neural decoding is one of two key parts of a neuromodulation system. The other key part is that which applies stimulation. Neural decoding involves extracting neural signals, and interpreting neuron activities to map brain activity to the outside world, which includes motor and sensory domains. Calcium imaging is a neuroimaging technique to

capture neuron cell activities. A distinguishing characteristic of calcium imaging is that it provides both high spatial resolution and high temporal resolution.

Closed-loop neuromodulation systems have significant advantages compared to open-loop neuromodulation [1]. For example, closed-loop neuromodulation helps scientists find biomarkers that impact brain functionality. Closed-loop neuromodulation also leads to more efficient and user-friendly brain-machine interface systems. In a closed-loop neuromodulation system, stimulation must be delivered in a timely manner in relation to the current state of neural activity, creating a need for a real-time and accurate neural image processing system. The image processing speed, power consumption, and memory requirements are in general all important in determining the effectiveness of image processing functionality that is operated within a closed-loop neuromodulation system.

Moreover, neural decoding systems usually have many parameters. The parameters lead to a complicated design space where different configurations (design points) offer various trade-offs involving crucial operational metrics, such as accuracy and time-efficiency. When neural decoding is applied within a closed-loop neuromodulation system, accuracy typically needs to be optimized under strict real-time constraints. On the other hand, in open-loop systems or in offline neural signal analysis, accuracy can be optimized subject to looser constraints on processing speed. Because of the complexity of the design spaces associated with neural decoding systems, and the number of relevant operational metrics, including accuracy and speed, it is often not feasible to effectively optimize parameter settings manually. This motivates the importance of automated methods and tools for parameter optimization when implementing neural decoding systems.

In the developments of this thesis, we extensively apply concepts of dataflow-based

design of embedded software. When constructing complex image processing systems under challenging constraints, such as the constraints imposed by closed-loop neuromodulation, dataflow-based design methodologies are useful in the development of effective tools and design processes [2, 3]. In dataflow-based methodologies, applications are represented as directed graphs, called dataflow graphs. In dataflow graphs, computational tasks, or dataflow actors, correspond to graph vertices. Actors can be executed, or fired, whenever they have enough data, provided that the notion of data-sufficiency is properly established as part of the actor's design. Actors can be designed for a wide range of functionality, such as functionality for motion correction, object detection, and signal decoding. Well-constructed dataflow-based application representations enable important forms of design optimization for embedded signal and information processing systems [3].

Deep learning and neural networks (NNs) are promising for use in neural image processing applications as they are known to work well for tasks such as denoising, object detection, field-of-view segmentation, and removal of motion artifacts. Deep learning models can improve the performance of neural imaging processing greatly — for example, by requiring minimal assumptions, parameter tuning, and pre/post-processing compared to traditional methods that do not use deep learning. NNs enable training on small image windows to learn implicit features of neurons and applying the knowledge on full images.

The design of deep learning algorithms for neural imaging has been explored extensively [4, 5, 6, 7, 8, 9]. Effective inference of neural imaging processing models requires optimizing inference speed to enable time-sensitive, closed-loop, real-time applications that respond to neural activities. Moreover, the development of neural imaging methods such as calcium imaging has made it possible to capture the activity of numerous

cells at once, with precise details of individual cells, within a certain field-of-view. This trend presses the need to create real-time image processing methods for large-scale, high resolution image data [10].

Certain types of resource-constrained computing devices, such as smartphones, are useful for deploying neuromodulation systems [11, 12, 13], as they are cost-effective and enable out-of-the-clinic treatments and experiments while potentially having low intrusion on patients' daily lives. The ease of use and low cost of smartphones make it possible to avoid the expense and development delays associated with custom-designed integrated circuits for various medical applications. However, modern image processing NN models require high computational resources, which makes them unsuitable for real-time neural imaging processing associated with closed-loop neuromodulation systems on resource-constrained devices. Additionally, resource-constrained image processing applications introduce the need for high energy efficiency due to battery constraints, and the need for high data locality due to the time and energy costs to transfer large volumes of image data between processors. This motivates real-time, fast, power-efficient inference of NN-based neural image processing applications on mobile devices.

Instead of designing a new specialized NN model for neural image processing, we develop ShaderNN, an NN inference engine for real-time imaging applications targeted to mobile devices, and we demonstrate the utility of ShaderNN for neural image processing applications. ShaderNN is lightweight, fast, and power-efficient, which helps designers address the challenging constraints of resource-constrained neural image processing system design. As a wide variety of NN operators are implemented/supported in ShaderNN, the inference engine is not limited to neural imaging applications, and can be applied to NN

models in other application areas as well.

The research reported in this thesis focuses on three perspectives — design, optimization, and on-device implementation (deployment) — to develop efficient neural imaging processing systems. First, in the design perspective, we explore the advantages, including expandability, cross-platform portability, and high-level design optimization, that dataflow brings, and we address the challenges of real-time neural imaging processing with rigorous application of dataflow-based system design methods. We demonstrate our results through a prototype software system for processing calcium imaging signals associated with neural activity. Second, we investigate methods and tools to optimize the overall performance of neural imaging processing systems. We develop new optimization methods to jointly tune the hyper-parameters of neural imaging processing systems, and optimize trade-offs between speed and accuracy for different application scenarios. Lastly, to enhance the performance of on-device neural imaging applications, we develop an inference engine to implement neural image processing systems that operate in real-time with low inference latency, high energy efficiency, and high data locality.

The remainder of this thesis is organized as follows. In Chapter 2, we develop a novel software framework, called the Neuron Detection and Signal Extraction Platform (NDSEP), for real-time neural image processing on calcium imaging video streams. The framework addresses important challenges associated with real-time neural image processing, as described earlier in this chapter, and provides a useful tool for further research and development of neuromodulation systems. In Chapter 3, we present a parameter optimization framework , called NEural DEcoding COnfiguration (NEDECO), that can automatically tune the hyper-parameters of neural imaging processing systems including

NDSEP. This framework helps designers optimize trade-offs between performance metrics, such as speed and accuracy, according to practical application scenarios — for example, open-loop or closed-loop neuromodulation. In Chapter 4, we develop an inference engine to implement NN-based real-time neural imaging processing systems on mobile devices. This inference engine helps to address the challenges of deploying deep-learning-based real-time neural imaging applications on resource-constrained mobile devices.

Chapter 2

Real-time Neuron Detection and Neural Signal Extraction Platform for Miniature Calcium Imaging

2.1 Chapter Overview

Real-time neuron detection and neural activity extraction are critical components of real-time neural decoding. In this chapter, we propose a novel real-time neuron detection and activity extraction system using a dataflow framework to provide real-time performance and adaptability to new algorithms and hardware platforms. The proposed system was evaluated on simulated calcium imaging data, calcium imaging data with manual annotation, and calcium imaging data of the anterior lateral motor cortex. We found that the proposed system accurately detected neurons and extracted neural activities in real-time without any requirement for expensive, cumbersome or special-purpose computing hardware. We expect that the system enables cost-effective, real-time calcium imaging based neural decoding, leading to precise neuromodulation.

The work in this chapter was developed in collaboration with Yaesop Lee, another Ph.D. student at the University of Maryland. My contribution to the work presented in this chapter has been focused on the motion correction actor design, the overall dataflow-based model development, simulated data generation and performing experiments on the simulated data, and ALM data.

Material in this chapter was published in partial, preliminary form in [14].

2.2 Introduction

Real-time neural decoding predicts behavioral variables based on neural activity data, where the prediction is performed at a pace that keeps up with the speed of the activity that is being monitored. Neuromodulation devices are becoming one of the most powerful tools for the treatment of brain disorders, enhancing neurocognitive performance, and demonstrating causality [15, 16]. A precise neuromodulation system integrates neural activity monitoring, real-time neural decoding, and neuromodulation. In precise neuromodulation, a decoding device predicts a behavioral variable based on neural data streams in real-time. Based on the decoding results, neuromodulation parameters such as timing, frequency, duration, and amplitude are changed. Precise neuromodulation systems with closed-loop real-time feedback are superior to the fixed (open-loop) neuromodulation paradigm [17, 18, 1].

A recent direct brain stimulation study demonstrated significant advantages of precise neuromodulation over open-loop neuromodulation [1]. This study applied direct brain stimulation with decoding capability to patients with epilepsy to improve their memory. The study found that stimulation increased memory function only if delivered when the decoding device indicated low encoding efficiency, while stimulation decreased memory function if delivered when the decoding device indicated high encoding efficiency. An open-loop neuromodulation system with a fixed stimulation paradigm may not always facilitate improving memory function.

Miniature calcium imaging (e.g., see [19, 20, 21]) is a neuroimaging tool that can observe all cells in the field of view in behaving animals, has high spatial and temporal resolution (single cell spatial resolution and sub-second temporal resolution), and enables chronic imaging. In this chapter, we focus on 2-photon calcium imaging. A closed-loop real-time neural decoding system based on miniature calcium imaging will lead to a powerful precise neuromodulation system. The first step in development of such a neural decoding system is to have an accurate and fast *Real-time Neuron Detection and Activity Extraction* (*RNDAE*) system. In our context, an RNDAE system takes as input a video stream S that is generated by a miniature calcium imaging device, which is mounted on the head of a behaving animal. The output produced by the RNDAE system is a set of neuron masks $\{n_1, n_2, \ldots, n_m\}$ that is detected in S, where m is the number of detected neurons, along with the neural signal $s_i(k)$ that is extracted for each neuron n_i. The neural signal $s_i(k)$ gives the neural activity associated with neuron n_i for each input video frame k, as represented by the video stream S. See Table S2 in the supplementary material for the definitions of variable and symbols in this article.

The tremendous rate at which miniature calcium imaging devices produce data imposes major challenges in the design and implementation of an RNDAE system. For example, during 10 minutes of imaging, such a device generates 1G of data under a frame rate of 10 Hz. Additionally, intensive processing within and across video frames in the input data stream is required for accurate detection of neurons and extraction of the associated neural signals. Furthermore, since algorithms and hardware platforms relevant to neural signal processing are evolving rapidly, the design of an RNDAE system should be architected in a manner that supports flexible adaptation to different component algorithms,

and retargeting to different processing devices. These requirements for complex processing on high-rate video data and flexible support for hardware/software design modifications make development of RNDAE systems a very difficult task.

In this chapter, we develop a novel RNDAE system, called *Neuron Detection and Signal Extraction Platform* (*NDSEP*), that is designed to address the challenges described above. NDSEP provides an experimental platform for neuron detection and neural signal extraction that provides real-time performance, and adaptability to new algorithms and hardware platforms. NDSEP also provides a valuable foundation for research and development of precise neuromodulation systems. The architecture of NDSEP is based on principles of signal processing oriented dataflow models of computation (e.g., see [22, 3]).

In dataflow programming, computational tasks can be executed whenever they have sufficient data. This property provides great flexibility to compilers, software synthesis tools or system designers to coordinate task execution in ways that are strategic with respect to the relevant implementation constraints and objectives. The data-driven semantics of task execution in dataflow is fundamentally different from procedural programming languages, such as C and Java, where the programmer specifies a sequential control flow between tasks in addition to the tasks themselves. This sequential approach to programming hides concurrency between tasks, whereas well-designed dataflow representations expose concurrency explicitly. A trade-off is that dataflow representations can be highly non-intuitive to apply to arbitrary types of applications; however, they have been shown to be well-suited in the broad area of signal and information processing (e.g., see [3]).

Motivated in part by its utility for efficient implementation on parallel computing platforms, system design using dataflow methods is widely used for complex signal and

information processing applications. The high level signal flow structure that is exposed by well designed dataflow models is valuable for design optimization in the context of important metrics, including those related to processing speed, memory management, and energy efficiency [3]. Additionally, dataflow provides a precise, abstract representation of computational modules and the interaction between modules within a given signal processing application. The formal, abstract representation provided by dataflow is of great utility in migrating implementations across platforms, and also for efficiently expanding, upgrading or otherwise modifying an implementation that is targeted to a given platform. Throughout the presentation of NDSEP in this chapter, we therefore emphasize the ways in which dataflow techniques are employed to help address the complex and multi-faceted challenges, motivated above, that are involved in RNDAE system development.

The major contribution of our chapter is the rigorous application of dataflow-based system design methods to real-time neural decoding. There are many systems, such as CaImAn-CNMF [23] and STNeuroNet [24], for neuron detection, which may achieve higher accuracy than our current implementation. However, these algorithms are not dataflow-based, and therefore they do not provide the advantages of expandability, cross-platform portability and high-level design optimization described above. All of these features are useful for flexible experimentation with and practical deployment of neural decoding methods. The main contribution of this effort can therefore be viewed as an overall system design, not just a single component.

2.3 Background and Related Work

In this research, we apply advanced methods for dataflow-based system design to address the challenges motivated in Section 2.2 for RNDAE technology. In this section, we first review related work on neuron detection and neural signal extraction, and then we present background on dataflow methods for signal processing system design.

2.3.1 Real-time Neuron Detection

Neuron detection centers on identifying the source (neurons) in the image field of view (FOV). A straightforward method for neuron detection is to manually delineate neuron masks. This manual labeling process is labor intensive. For semi-automated/automated neuron detection, a PCA/ICA based method [25] is proposed. This algorithm first runs PCA to reduce data dimensionality, and then uses ICA to segment data into statistically independent spatial and temporal signals. Constrained nonnegative matrix factorization (CNMF) based methods for neuron detection are described in [26, 27]. Deep learning based neuron detection methods are proposed in [28]. Although these semi-automated/automated neuron detection methods are powerful, they are not suitable for real-time applications because of long running time. That is to say, the methods mentioned above are not in real-time which makes a huge difference with our method, which is in real-time, that will be described later.

Motion correction is a crucial step for accurate neural detection. For real-time applications, motion correction must be integrated as part of the neural detection and neural signal extraction system, as the input arrives directly without any preprocessing.

The motion correction problem can be solved by image registration [29]. However, these registration algorithms require a running time on the order of seconds to minutes per frame [30]. Real-time applications require optimized and efficient motion correction.

2.3.2 Dataflow-based System Design

Dataflow provides a valuable foundation for design and implementation of novel signal and information processing systems under complex constraints (e.g., see [3]). When dataflow is used as an abstraction for signal processing system design, applications are represented as directed graphs, called *dataflow graphs* [22]. Vertices in dataflow graphs, called *actors*, represent computational tasks, such as digital filters, matrix operations, or image transformations, and each edge represents a first-in, first-out (FIFO) buffer that stores data as it passes from the output of one actor to the input of another. Each unit of data within such a buffer is referred to as a *token*.

Dataflow actors abstract the detailed implementation of the corresponding computational tasks, while imposing important constraints on how the actors interface with the surrounding graph, regardless of the implementation. These dataflow interface constraints include two major aspects. First, a dataflow actor can execute (*fire*) only when certain well-defined conditions on the buffers associated with its input and output edges are satisfied. These conditions are typically formulated in terms of the token populations on the buffers — that is, some minimum amount of data is required on each input buffer (to provide the input for the next firing), and some minimum amount of empty space is required on each output buffer (to store the output generated by the firing). When the firing conditions

described above are satisfied, the actor (or its next firing) is said to be *enabled*.

Second, when an actor is fired, it must actually produce and consume on each output and input port, respectively, a number of tokens that is consistent with the assumptions that were used to determine that the firing was enabled.

A distinguishing feature of dataflow is that the "program" (dataflow graph) does not specify the order in which actors will execute, nor (in the case of a hardware platform with multiple processors) the processing resource on which each actor is mapped. Instead, the mapping of actors to processors and execution ordering of the actors are left up to the system designer or design tool. The mapping together with the ordering of actors that share the same processor is referred to as the *schedule* for the dataflow graph. A general rule of dataflow schedule construction is that an actor can only be fired (executed next in the evolution of a schedule) when it is enabled, as described above.

The schedule typically has great impact on most or all key implementation metrics, including throughput, latency, and memory requirements. The decoupling of a dataflow graph G from the schedule together with the high-level signal flow structure exposed by G provides great flexibility to designers and design tool developers in constructing schedules. This flexibility is important for optimizing a schedule with respect to the specific constraints, objectives, and processing devices that are relevant to the given application. In this work, we seek to enable and exploit this flexibility by applying dataflow-based concepts consistently throughout the RNDAE system design process.

Formally, a dataflow graph is represented as a directed graph $G = (X, E)$, where X is the set of actors and E is the set of edges. For each edges in $e \in E$, we denote the source and sink vertices of e as $src(e)$ and $snk(e)$, respectively. Each edge e has a nonzero-integer

delay associated with it, which gives the number of initial tokens that are stored in the corresponding FIFO before the dataflow graph begins execution. A *self-loop* edge is an edge e_s whose source and sink actors are identical ($src(e_s) = snk(e_s)$).

Figure 2.1 shows a simple dataflow graph with three actors ($X = a, b, c$), and two edges $e_1 = (a, b)$, and $e_2 = (b, c)$. The "D" on the edge (b, c) represents a unit delay. If the delay on an edge exceeds 1, then we typically annotate the edge with "N D", where N is the delay of the edge. If the delay is zero, then we omit the "D" symbol, and do not provide any annotation on the edge associated with delay. For example, the absence of a "D" symbol on (a, b) in Figure 2.1 indicates that this edge has no delay.

Self-loop edges are often omitted from drawings of dataflow graphs. However, their presence must be taken into account by some forms of analysis and optimization. For example, self-loop edges in general limit the amount of data parallelism that can be exploited when scheduling a given actor (e.g., see [31]).

For further background on dataflow fundamentals for signal processing systems, we refer the reader to [22, 3]. For background on more general foundations of dataflow, we refer the reader to [32, 33].

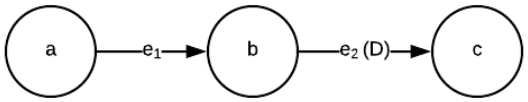

Figure 2.1: An example of simple dataflow graph.

2.4 Proposed Method

Our NDSEP system is developed and tested for use on video streams that are acquired from mice using miniature calcium imaging devices. We especially focus on 2-photon calcium imaging. The NDSEP system is therefore suitable for use in monitoring neural activity in real-time — for example, to help inform the scientist performing an experiment about how to adapt experiment options so that subsequently-acquired data is most relevant to the experiment objectives.

The system design of NDSEP incorporates two distinct modes of operation, which we refer to as the *initialization mode* and *real-time mode*. The purpose of the initialization mode is to optimize system- and actor-level parameters in relation to the image characteristics associated with a given experiment. Calcium imaging data for a given experiment have certain distinctive characteristics that are influenced by the experimental setup, including the imaging devices, neuron types, and specific animal subjects that are involved. To maximize neuron detection and signal extraction accuracy, it is important to tune, in relation to these distinctive characteristics, certain parameters associated with the neural signal processing algorithms that are employed. Image characteristics that are relevant in this tuning process include the size of the neurons being monitored, and the brightness of the firing neurons relative to the background.

For concreteness and for insight into specific optimizations that we applied to facilitate real-time performance, we describe in this section selected details on actor implementations in the current version of NDSEP. These details include, for example, specific OpenCV functions that are applied within the actors, and associated parameter

settings for these functions. However, we would like to emphasize that the NDSEP framework is independent of any specific approach for implementing algorithms or any specific algorithms for image analysis. For example, one could replace the calls to OpenCV functions with calls to a different library that provides similar capabilities or with customized code that is developed by the actor designer. As another example, one could replace the Neuron Detection actor, which implements the `SimpleBlobDetector` algorithm, with another actor that implements the Holistically-nested Edge Detection (`Hed`) or `MaskRCNN` algorithm. The modular, model-based design of NDSEP facilitates use cases such as these for experimentation with alternative algorithms and actor implementations. Such experimentation is useful for developing insight into trade-offs between neural decoding accuracy and real-time performance, which are critical to the overall utility of a neural decoding system.

In Section 2.5, we evaluate NDSEP using datasets involving both simulated data and real data. The real world dataset is acquired from mice models. Two-photon calcium imaging was used to image the calcium fluorescence of Anterior Lateral Motor (ALM) cortex. Thus, in the remainder of the chapter, we refer to the real data as the ALM dataset. More details about the ALM data we use is given in Section 2.5.

The remainder of this section is organized as follows. First, we provide background on a specific form of dataflow modeling, called *parameterized synchronous dataflow* (*PSDF*), which is well-suited to the computational structure of NDSEP. Next, we present the key actors (dataflow-based software components) that are involved in NDSEP. We then present the overall system design for NDSEP, including relevant details of the initialization mode and real-time mode.

2.4.1 PSDF Modeling

A variety of of specialized dataflow modeling techniques have been developed for different classes of signal processing applications (e.g., see [3]). For design of NDSEP, we apply the PSDF model due to its utility in representing signal processing applications in which dynamic modifications to system parameters play an important role. PSDF enables the joint, dataflow-based modeling of (1) subsystems (*adapting subsystems*) whose parameters can be modified dynamically along with (2) subsystems (*controlling subsystems*) whose results are used to determine new values of relevant parameters in the adapting subsystems [34].

A number of different variants of dataflow have been developed with an emphasis on supporting dynamic parameter reconfiguration (e.g., see [35]). Among these, we apply PSDF because PSDF is well-supported in the software tool, called the lightweight dataflow environment (LIDE) [36], which we use in this work for dataflow graph implementation. Adapting NDSEP to other forms of dynamic-parameter-integrated dataflow models is an interesting direction for future work in exploring implementation trade-offs.

In the PSDF modeling approach that we use in NDSEP, the system-level dataflow graph is composed of two communicating subgraphs called the *subinit graph* and *body graph*. These graphs are used, respectively, to model the controlling subsystems and adapting subsystems described above. In NDSEP, the body graph represents the core signal processing functionality for neuron detection and activity extraction, while the subinit graph represents functionality for dynamically computing new values for selected parameters in the body graph. In particular, each output port p of the subinit graph is associated at design time with one or more ordered pairs $((A_1(p), P_1(p)), (A_2(p), P_2(p)), \ldots$

$(A_{n(p)}(p), P_{n_p(p)}(p))$. where $n(p)$ is the number of such ordered pairs associated with p, each $A_i(p)$ is an actor in the body graph, and each $P_i(p)$ is a parameter of actor $A_i(p)$. When the PSDF graph executes, each iteration of the subinit graph is followed by the transmission of values from each output port p to update each parameter $P_i(p)$ of each actor $A_i(p)$.

More details on the PSDF-based application model for NDSEP are discussed in Section 2.4.3.

2.4.2 Signal Processing Modules in NDSEP

In this section, we discuss the design of the signal processing actors that are employed in the body graph of NDSEP.

A common approach used in the implementation of the actors in NDSEP is that actors produce and consume pointers to images rather than directly producing and consuming image pixels on their incident dataflow edges. That is, in cases where images are communicated across a dataflow edge e, we transfer only a pointer to each communicated image through the FIFO buffer associated with e rather than writing and reading the entire image to and from the buffer. The same approach is used when communicating matrices across actors. This approach allows us to adhere to the dataflow principles described in Section 2.3.2 without requiring large overhead for FIFO buffers that carry streams of images or matrices.

The system-level dataflow graph for NDSEP, including all of the actors discussed in this section, is developed using the LIDE tool mentioned in Section 2.4.1. For background

on LIDE, we refer the reader to [36].

2.4.2.1 Motion Correction

Motion correction is the first step of image processing in NDSEP. In real calcium imaging data taken from moving mice, significant motion can result due to the drift of the implanted imaging device. This kind of shaking in general may result in motion translation as well as slight rotation, thereby distorting the acquired video stream. The goal of motion correction in NDSEP is to remove such motion translation and rotation from image frames.

Through profiling of execution time across different actors in the NDSEP system, we determined early on in the design process that motion correction contributes significantly to overall system execution time. More details on system-level profiling are provided in Section 2.5. Because of the critical role of motion correction in determining overall system efficiency, we applied a significant portion of our design effort on optimizing the accuracy/speed trade-off for this part of NDSEP.

For motion correction, NDSEP utilizes the Enhanced Correlation Coefficient (ECC) algorithm [37] for *motion detection*, which is a core part of motion correction. We selected ECC because it provides parameter settings that give significant flexibility in exploring trade-offs between accuracy and processing speed. Such exploration is useful in the design of RNDAE systems, where the objective is to provide acceptable accuracy in real-time rather than maximum accuracy at any cost. ECC is also invariant to photometric distortions in brightness and contrast.

We employ the ECC function provided by the OpenCV library [38], and call this

function within the LIDE-based actor implementation for the Motion Correction actor.

In addition to using the ECC algorithm, as described above, we apply two major techniques to improve the real-time performance of the Motion Correction actor. First, before comparing frames for motion detection, we downsample the frames by a factor of 1.67 in each dimension so that the number of pixels is reduced to one quarter of the original pixel count. The downsampled image is currently applied only to the detection process so that any distortion introduced by it is localized to the detection step. Applying downsampling strategically in other parts of NDSEP is a useful direction for future work.

Second, while our motion correction approach takes both translation and rotation into account, we apply rotation selectively, only in cases where translation-based motion correction fails. This optimization is motivated by empirical observations that in our experimental context, rotation is encountered relatively infrequently in frames that are captured by the neuron imaging device. For example, in the ALM dataset, the rotation frequency detected by NDSEP is 1.62%, the mean and max rotation angle are 0.0232 degree, 0.0733 degree, respectively. We choose rigid motion correction because of algorithm efficiency for real-time applications. When a single frame is acquired quickly (less than 50 ms), the influence of motion across the frame is relatively uniform, and a rigid correction can give good results [39, 40]. Translation is more common. Furthermore, detection and correction of rotation is more computationally expensive compared to translation. For example, we found that the "Euclidean" mode for the OpenCV ECC function, which detects both translation and rotation, takes on average about three times longer to compute compared to the "Translation Only" mode.

Figure 2.2 illustrates a flowchart of the optimized motion correction approach in

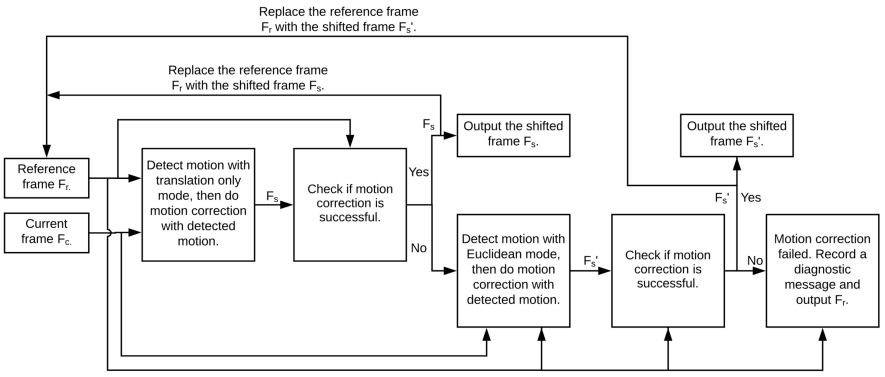

Figure 2.2: Flowchart for Motion Correction actor operation in NDSEP.

NDSEP, which is based on differences in frequency of occurrence and computational complexity associated with translation and rotation. As illustrated in Figure 2.2, we first apply motion detection with the Translation Only mode. If motion is detected from this operation, then the current frame F_c is shifted to compensate for the detected translation, and the correlation between the shifted frame F_s and the reference frame F_r is evaluated. On the other hand, if no motion is detected, the correlation is carried out between F_c and F_r. If the computed correlation C_1 meets or exceeds a threshold τ_1, then F_r is replaced with F_s or F_c, respectively, F_s is produced on the output edge of the actor, and the current actor firing is complete.

On the other hand, if the correlation C_1 is less than τ_1, then motion detection for both translation and rotation is applied using the more costly Euclidean mode of OpenCV ECC. If motion is detected from the Euclidean mode, then a shifted version F_s' of F_r is derived based on the detection result. Then the correlation C_2 between F_r and F_s' is carried out, where $F_r = F_s'$ if motion was detected from the Euclidean mode, and $F_r = F_c$ otherwise.

Again, a thresholding check, using another threshold τ_2, is used determine how

to interpret the correlation result. If $C_2 \geq \tau_2$ (similar to the case of C_1 exceeding the threshold), then F_r is replaced with F_s' or F_c, respectively; F_s' is produced on the output edge of the actor; and the actor firing is complete. Otherwise, a diagnostic message is sent to a log file associated the overall experiment, and F_r is produced on the output edge to complete the firing.

The diagnostic message generated in this last case identifies the input frame index, and indicates that motion correction has failed at this index. Such information, which is accumulated in an experiment log file by all relevant actors, can be useful to the system designer in continually improving the robustness of individual actors and the overall system.

The thresholds τ_1 and τ_2 defined above, which determine whether to accept the motion correction result or not, are computed adaptively to track any relevant changes in image characteristics. This is due to dynamic variation in the characteristics of calcium imaging frames. For example, some of the datasets are noisy or have contaminated backgrounds. This lowers the average correlation value. For the ALM dataset that we employed in Section 2.5.3, the noise/contamination level is stable during relatively short time periods than those of non-aligned frames in clear datasets. Therefore, in NDSEP, correlation values are only compared with close neighbors. For this purpose, NDSEP stores the 100 most recent correlation values in a queue, which we refer to as the *correlation history queue* (*CoHisQ*). Every time CoHisQ changes, the mean value $\overline{CoHisQ}$ and standard deviation σ_{CoHisQ} across all elements in the queue are calculated. Each threshold $\tau \in \{\tau_1, \tau_2\}$ is computed from $\overline{\mathrm{CoHisQ}}$ and σ_{CoHisQ} using:

$$\tau = \overline{CoHisQ} - p(\tau) \times \sigma_{CoHisQ}, \tag{2.1}$$

where $p(\tau)$ is an empirically defined parameter for each of the two thresholds. In our experiments, we employ $p(\tau_1) = 2$, and $p(\tau_2) = 10$. The threshold values in τ range from approximately [0.3, 0.95] in our experiments resulting in motion correction success.

The threshold computation approach and its associated parameters provide an example of an RNDAE-system design issue for which there are many possible solutions. The modularity and extensibility of NDSEP, based on its dataflow-based foundations, facilitate experimentation across different solutions for such design issues.

2.4.2.2 Preprocessing

The Preprocessing actor is designed to remove image distortion that is caused by the imaging device and imaging environment. To remove distortion caused by the imaging device, the actor incorporates a Gaussian filter and median filter. Furthermore, background subtraction is used to remove background effects along with performing image equalization. As described in Fig. 2.3, the output of the preprocessing actor does not affect the neural signal as it only helps to get the positions of neurons. This process helps to eliminate the bright background that results from firing of neurons in deeper areas of the brain. These deeply-located neurons are not of interest in the targeted class of experiments, so it is useful to subtract their potentially strong effect on the image background. To enhance the image's contrast, we normalized the image intensities by using $(I - I_{min}) \times 255/(I_{max} - I_{min})$,

where I indicates the current pixel's intensity, and I_{max} and I_{min} represent the maximum intensity and minimum intensity of the image, respectively. As part of the Preprocessing actor, we employed the `GaussianBlur` and `MedianBlur` functions from OpenCV. For the GaussianBlur and MedianBlur functions, we employed a filter size of 3×3 in order to minimize the possible distortion of the small neurons. The filter size can be reconfigured based on the distortion level and characteristics of the data. Presently, we only consider the possible distortion and removal that might occur to small neuron sizes.

2.4.2.3 Neuron Detection

The Neuron Detection actor takes an image frame as input, detects the presence of neurons in the image frame, and outputs the position and size of each detected neuron. The output is produced in the form of an $n_d \times 3$ matrix δ, where n_d is the number of detected neurons. Each row in the matrix corresponds to a detected neuron. We refer to δ as a *neuron detection matrix*. For each row index i, $n_d[i][1], n_d[i][2]$, respectively, give the x coordinate and y coordinate for the center of the ith detected neuron, and $n_d[i][3]$ gives the neuron's radius.

For its core computational task, the Neuron Detection actor applies the `SimpleBlobDetector` function from OpenCV [38]. The function detects closed contours ("blobs"), which are assumed to outline the detected neurons. Among the contours, the function can filter out the blobs by intensity, size, and shape. The function finds blobs using the parameter `thresholdStep`, which denotes the minimum intensity difference between the inside and outside of a blob. By using the parameter, it filters out the blobs

that have low intensity difference compared to their backgrounds — that is, it removes less active blobs. Using the parameters A_{min} and A_{max}, which are related to the sizes of the blobs to detect, the function computes a set of detected blobs, along with their centers and radii. The parameters A_{min} and A_{max} specify the minimum and maximum sizes (in terms of the number of pixels contained) of the blobs to detect. Using parameters for circularity, inertia and convexity, the function filters out non-neuron-like shapes. The radius of a blob is computed to be the distance between the center of the blob and the furthest point from the center. In this context, a blob can be viewed as a set of connected pixels in an input image that have some minimum intensity (exceed a threshold on the pixel value), and satisfy size constraints that are carefully configured to help ensure that the corresponding image regions represent neurons within the imaged brain region. Since the `SimpleBlobDetector` function is not for segmentation but for detection, it returns the center of each blob's position and its radius. By using `SimpleBlobDetector` instead of a segmentation function, we can gain comparable detection results with faster speed.

To this end, the `SimpleBlobDetector` function is configured to filter blobs by size based on two size-related parameters, which we denote by A_{min} and A_{max} ($A_{min} < A_{max}$).

The values of A_{min} and A_{max} are determined as part of the initialization mode for the NDSEP system. The initialization mode includes an automated training process, which configures parameters such as A_{min} and A_{max}. Another parameter of the `SimpleBlobDetector` that is configured during the initialization mode is the `thresholdStep` parameter, which controls the step size for determining the set of pixel-intensity thresholds that are used during the blob detection process [38]. More details on the

initialization mode are discussed in Section 2.4.3.

The `minThreshold` parameter for the `simpleBlobDetector` function is set to zero in all of our experiments.

After blob detection within a given firing of the Neuron Detection actor, a set of neurons $\eta = \mu_1, \mu_2, \ldots, \mu_k$ is identified in the input image along with their positions and radii. During real-time operation of the actor, the positions and radii of these neurons are produced as output in the form of a neuron detection matrix, as described above.

During its training process, however, further processing using the set η is performed before producing output. The Neuron Detection actor is equipped with a parameter that is used to select whether it operates in training mode or real-time mode. In NDSEP, Neuron Detection operates in its training mode during a well-defined system initialization phase (discussed further in Section 2.4.3), and then operates for the remainder of the given experiment in its real-time mode.

In the remainder of this section on the Neuron Detection actor, we discuss the further processing that is performed during the training process, after η has been determined.

First, if the current firing is not the first firing within the experiment, the neuron positions in η are compared with those in δ_p, which is the detection matrix derived from the previous actor firing. The previous matrix δ_p is maintained as a state variable of the Neuron Detection actor. This state variable is maintained and used only during the training mode. By a state variable, we mean a data object that is local to the actor and that persists across firings of the actor. Actor state can be modeled in signal processing dataflow graphs with self-loop edges (e.g., see [41]).

If the position of a neuron within η is found to be sufficiently close to a neuron

position in δ_p, then that neuron is removed from η. In our current design, "sufficiently close" in this context means that the difference in position can be d pixels in both the x and y dimensions. The parameter d can be determined by considering how close it should be to be considered as a neuron but has slight a motion. That is to say, user can define the criteria of a distance which if the distance is closer than d pixels, then the system will regard the neurons as a single neuron, but it the close two neurons is not closer than d pixels, then the neurons will be two different neurons, overlapping neurons. After removing all neurons from η that are sufficiently close to corresponding neurons in δ_p, the remaining neurons in η are interpreted to be *newly discovered* neurons in the training process. Thus, all of the remaining neurons in η are appended to those in δ_p. The resulting δ_p may be unchanged from the previous firing (if there were no newly discovered neurons) or it may contain one or more new neurons. The resulting δ_p is produced as the output of the training mode firing, and it is also retained as the updated value of the corresponding state variable in the actor.

2.4.2.4 Signal Extraction

Each firing of the Signal Extraction actor takes as input a motion-corrected image frame F_{mc}, and a neuron detection matrix δ, which gives the positions and radii of the neurons that have been detected in F_{mc}. The output of the firing is a vector β that gives the relative intensity of each detected neuron.

Each ith element of β corresponds to a distinct neuron, and is calculated as $\beta[i] = (F(i)(F_{mc}) - F0)/F0)$, where $F(i)(F_{mc})$ is the average intensity (average pixel value)

across all pixels in the circle centered at $(\delta[i][1], \delta[i][2])$, and having radius $\delta[i][3]$ in F_{mc}th image frame. To calculate F0, we followed [42] using the average ROI intensities across a window of time that immediately precedes a particular experimental event.

Throughout a given experiment, the Signal Extraction actor produces a sequence of vectors $\beta_1, \beta_2, \ldots, \beta_L$, where L is the total number of image frames in the input video sequence for the experiment (excluding the frames using for system initialization/training). Each β_i is a ν-element vector, where ν is the total number of neurons that have been detected throughout the training process for the Neuron Detection actor. The sequence $\beta_1[i], \beta_2[i], \ldots, \beta_r[i]$ thus provides a sampled representation of the relative pixel intensity for each ith detected neuron ($1 \leq i \leq \nu$).

2.4.3 System Design

Figure 2.3 shows how the different actors described in Section 2.4.2 are integrated into the dataflow graph for the NDSEP system. The dataflow graph is based on the PSDF model of computation (see Section 2.4.1). As discussed earlier in this section, the system has two distinct modes of operation — the initialization mode (also known as the training mode) and the real-time mode.

The initialization mode is used to configure selected actor parameters in the body graph using a set of *training frames*. The training frames are captured from a calcium imaging device that is implanted within a given animal subject. The resulting set of optimized parameters is then applied to perform accurate real-time processing during neuron image acquisition and analysis experiments involving the same device and animal

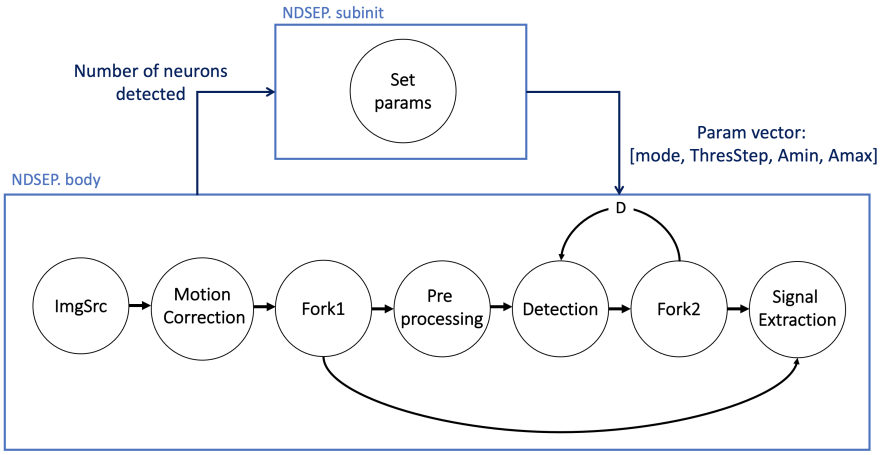

Figure 2.3: System-level dataflow graph for NDSEP.

subject. This real-time processing corresponds to the real-time mode of NDSEP. The set of frames that is processed during the real-time mode for a given experiment is referred to as the set of *analysis frames*. For more details, see Section 2.4.2.3 and Section 2.4.3.2.

2.4.3.1 Auxiliary Actors

All of the core signal processing actors in Figure 2.3 have been discussed in Section 2.4.2. Four additional actors — namely, the actors labeled ImgSrc, Fork_1, Fork_2, and SetParams — are also used, as shown in Figure 2.3.

Each firing of the ImgSrc actor reads the next image from the input video sequence from disk into memory, and outputs a pointer to the memory block that contains the image. This disk-based interface is used in our current NDSEP prototype, since our focus is on functional validation and on optimizing trade-offs between accuracy and real-time performance. For integration into a complete experimental system, the ImgSrc actor can readily be replaced by an actor that provides direct software interfacing to the image

acquisition device.

The actors labeled Fork_1 and Fork_2 are *fork* actors, also referred to as *broadcast* actors. Each firing of a fork actor consumes one token on its input, and produces a copy of the token on each of its outputs. Since images and matrices are communicated by reference (through pointers) in NDSEP (see Section 2.4.2), the fork actors require minimal execution time compared to the core signal processing modules in the system.

The fourth auxiliary actor, SetParams, is discussed in Section 2.4.3.2.

2.4.3.2 Adapting the Neuron Detection Actor

The SetParams actor is used during the initialization mode to adaptively optimize parameters of the Neuron Detection actor. The objective is to calibrate the selected parameters to the given calcium imaging device and animal subject so that neuron detection accuracy is enhanced compared to use of generic parameter settings. The parameters are adapted progressively as the training frames are processed in the initialization mode. Specific parameters that are configured by the SetParams actor are the A_{min}, A_{max}, and `thresholdStep` parameters for neuron detection (see Section 2.4.2.3). Before running the initialization mode, we manually "pre-initialize" these parameters by considering the size of the input image and rough size of the neurons. The initialization mode then uses the pre-initialized parameter values as a starting point and optimizes the three values through an iterative process (see Section 2.4.2.3).

Many different approaches for adapting neuron detection processes can be envisioned for use in NDSEP. Presently, we use a relatively simple adaptation approach that

progressively loosens the filtering constraints of the blob detector used in the Neuron Detection actor. The constraints are loosened until a pre-determined target number T_n of neurons is detected. Presently, we use the empirically determined value $T_n = 5$. Incorporating more sophisticated parameter adaptation processes into NDSEP is a useful direction for future work.

We conducted some simple experiments to help validate our current adaptation approach. With the simulated data, when we tried an approach that progressively tightens the constraints, we observed more false positives than our proposed approach, which progressively loosens the constraints. For example, with the most noisy set of simulated data, which is described in 2.5.1, we observed 11% more false positives with progressive tightening compared to our proposed approach. In this dataset, the progressive tightening approach also led to a few false negatives, whereas there were no false negatives resulting from our proposed approach.

2.4.3.3 Real-time Mode

For the real-time mode of NDSEP, the iteration count for the body graph is set to the total number of analysis frames. Thus, the subinit graph is effectively disabled through duration of the real-time mode. This is because the body graph continues executing for the specified number of iterations before control returns to the subinit graph, at which point the neural decoding process terminates.

In real-time mode, each analysis frame is processed by correcting motion, detecting neurons, and then extracting relative pixel intensities for each neuron. The relative pixel

intensities are used, as described in Section 2.4.2.4, to populate the vector elements for the next time step in the extracted signals for the neurons.

The output of the real-time mode is the sequence of vectors $\beta_1, \beta_2, \ldots, \beta_L$, where L is the number of analysis frames. This sequence encapsulates a sampled version of the signal extracted for each neuron. The sequence can be saved for subsequent off-line analysis or connected to another computational subsystem for further real-time processing, as would be the case if NDSEP were embedded within a precise neuromodulation system.

2.5 Experiments

In this section, we present results obtained through experiments using the proposed platform, NDSEP. We first present experiments involving simulated data, and then experiments involving real data. The experiments involving real data include results on neural imaging data that has already been processed with motion correction, as well as results on "raw imaging data" (without motion correction already applied).

2.5.1 Simulated Data

In this experiment, simulated calcium imaging datasets were used to assess the proposed platform because simulated data had ground-truth. The simulation described interactions among a set of leaky integrate-and-fire neurons with additive noise. The neuron model ([43]) is as follows:

$$\frac{\partial V}{\partial t} = \frac{V_{rest} - V}{\lambda} + \theta \times \lambda \times (-0.5) \times \epsilon, \tag{2.2}$$

where V is the membrane potential, V_{rest} is the rest potential, ϵ is a Gaussian random variable with mean 0 and standard deviation 1, λ is the membrane time constant, and θ is a parameter to control the noise term. Spikes received through the synapses cause changes in V. A neuron fires if V is greater than a threshold. After firing, a neuron cannot generate a second spike for a brief time (refractoriness). Such a neuron model can represent many kinds of postsynaptic potentials or currents described in the literature [44].

Our simulation included 100 neurons. We divided these 100 neurons into two groups: group A and B. Neurons in group A have no parent nodes, while neurons in group B have one or two neurons in group A as parent nodes. If a parent node fires, the membrane potential of the target node will increase by $w = 0.2$. This simulation represented a scenario in which neurons in group A were responsive to external stimulus, and firing of neurons in group A facilitated firing of neurons in group B. We generated simulated spike trains with 1800 time points.

100 neuron masks from the Neurofinder 00 dataset [45] were chosen as ground-truth neuron masks. For a given frame t, if neuron i fired, then the intensities of pixels inside neuron mask i were set to be 128. After this, we performed exponential smoothing to simulate calcium signal decay. jGCaMP7f, which is a calcium sensor, has half decay time around 265 ms [46]. To simulate imaging jGCaMP7F using a two-photon microscope at 30Hz, the half decay time in our simulation was set to 8 frames. To simulate motion,

we introduced a global shifting in the x and y axes. The random translation motion in x or y followed a uniform distribution in [-10, 10]. Also, to simulate rotation in frames, datasets were generated with different random rotation range and occurrence probability. We used P_{rot} to denote the rotation occurrence probability, and α_{rot} to denote the rotation range. For example, if $\alpha_{rot} = 5.16$, and $P_{rot} = 10$, then rotation within $[-5.16^o, 5.16^o]$ is randomly applied to the simulated data, with a rotation occurrence probability of 10%.

In addition to the random translation motion, three kinds of drift are simulated. A slow and constant drift is simulated according to the trajectory shown in Figure 2.4 (B). Motions are simulated around the ground truth position within 10 pixels, following the same range that we used for random translation motion. Slow and constant drift was incorporated by moving the frame 1 pixel at each time step in the same direction until it hits the boundary, which is taken to be 10-pixels the in x or y direction away from the ground truth. Then the direction of the drift changes according to the trajectory. We simulate small and large drift by controlling the range of random translation motion. Motion in x or y follows a uniform distribution in $[-3, 3]$ or in $([-10, -7] \bigcup [7, 10])$ to simulate small drift and large drift, respectively.

Then we added temporally autocorrelated zero mean noise with standard deviation $\sigma = 0.4$ [47] as well as shot noise with zero mean and $\sigma = 1.0$ [48] to some of the simulated datasets, called noisy simulated datasets (NoiseSim). All noisy simulated datasets had $P_{rot} = 25$, and $\alpha_{rot} = 6.3153$, along with the same global translation motion described above, as shown in Figure 2.4.

We applied the proposed system to the simulated data. The system was evaluated in terms of neuron mask detection accuracy, signal-to-noise ratio, and running time. For

neuron mask detection accuracy, because ground-truth neuron masks were available, we compared each detected neuron mask with the corresponding ground-truth neuron mask, and calculated the recall (the fraction of matched pixels divided by the number of pixels in the ground truth) and precision (the fraction of matched pixels divided by the number of pixels in the detected neuron mask).

For signal-to-noise ratio, for each detected neuron ϕ, we first correlated the detected $\Delta F/F$ of ϕ with the ground-truth spike trains. Given an image frame I and some region r (connected subset of pixels) in the frame, $\Delta F/F$ is a measure of the relative pixel intensity in r relative to a baseline. The metric is similar to that used for elements of the vector β defined in Section 2.4.2.4. Here, F represents the baseline pixel intensity, and $\Delta F = R - F$, where R is the average pixel intensity for all pixels in r. When $\Delta F/F$ is used in the context of a neuron, the region r consists of all pixels contained in the neuron. Then we calculated the average correlation coefficient R_s between the neuron time course and the ground-truth spike trains, across all neurons. Next, for each neuron ϕ, we randomly selected a region of the image frame with size close to the size of ϕ. For each randomly selected region r, we correlated $\Delta F/F$ of r with the ground-truth spike train of ϕ. This yielded a correlation coefficient $\rho(\phi)$. Then we calculated the average correlation coefficient across all ϕ — that is, the average value of $\rho(\phi)$. To avoid selection bias in choosing random regions, we repeated the above process 1000 times and calculated the average value R_n. The signal-to-noise ratio was then computed as $10 \log_{10}(R_s/R_n)$.

The motion corrected images were compared with the ground truth images. The ideal case here is that the Motion Correction actor detects the ground-truth x and y motion along with the rotation angle. There are 3 matrices used to evaluate the performance of motion

correction. For each dataset, M_x, M_y, and M_{rot} denote x-displacement, y-displacement, and angle error, respectively. $Rate_{fail}$ denotes the failure rate of motion correction. When motion correction fails, the frame is not motion-corrected (see Section 2.4.2.1).

Table 2.1 shows our measured results for motion correction with $\alpha_{rot} = 3.43$ and different values of P_{rot}. Table 2.2 shows results with $P_{rot} = 40$ and different values of α_{rot}. In Table 2.1, we see that as P_{rot} increases, the error also increases. However, the mean errors of x-displacement and y-displacement are very small, about 1 pixel in each dimension. The mean rotation error $mean(\alpha_{rot})$ is close to zero. Although some rotations are not detected (the rotation detection rate is not 100%), such cases are rare. As Table 2.2 shows, only when α_{rot} reaches 7.46^o does the motion correction failure rate begin to rise in some sparsely-occurring cases (1.56% failure rate). However, from our observations, such a large value of the rotation angle is rare in practice. Table 2.3 shows the performance of NDSEP in noisy situations. As the noise level increases, the x, y and angle detection error increases. Even in the s05c15 case, which includes frames that contain large amounts of noise, motion correction still performs effectively. The mean error of x, y-displacement remains consistently around 1 pixel while the mean rotation error is also comparable to the no-noise case. Compared with the average neuron size in simulated data, which has 6.8387 pixel width and 6.7634 pixel height, the 1 pixel x, y-displacement means NDSEP motion correction is effective for simulated data.

To further evaluate the motion correction process in NDSEP, slow and constant drift is added to the no-noise case s00c00, and to noisy cases s01c05 and s05c15. $R_{fail}(\%)$ is 5, 2.11, 4.33, respectively, for these three cases; $mean(M_x)$ is 0.6518, 0.8707, 0.7366, respectively; and $mean(M_y)$ is 1.4400, 1.4121, 1.1849, respectively. The results shown

Table 2.1: Motion correction accuracy with different P_{rot} without noise.

$P_{rot}(\%)$	0	5	10	15	20	25	30	40	50
$mean(M_x)$	0.8002	1.1696	0.9322	0.9911	0.6672	1.1643	1.0612	1.0956	1.0225
$max(M_x)$	1.6443	1.9522	1.8845	1.7710	1.9838	1.7457	1.9203	1.9470	1.9464
$mean(M_y)$	0.8145	0.7509	0.7655	0.7720	0.7710	0.7903	0.7476	0.7395	0.7146
$max(M_y)$	1.1627	1.2483	1.3923	2.6277	1.4583	1.4413	1.3937	1.4621	1.4736
$mean(M_{rot})(\times 10^{-4})$	0.0657	0.2590	0.2666	0.3568	0.5596	0.5988	0.9873	1.2820	2.0379
$max(M_{rot})$	0.0041	0.0056	0.0042	0.0095	0.0091	0.0067	0.0068	0.0038	0.0040
$R_{fail}(\%)$	0.5	0.44	0.33	0.38	0.33	0.38	0.22	0.33	0.06

Table 2.2: Motion correction accuracy with different α_{rot} without noise.

$sin(\alpha)$	0.06	0.09	0.11	0.13	0.15	0.17
α	3.4398	5.1636	6.3153	7.4696	8.6269	9.7861
$mean(M_x)$	1.0956	1.0257	0.9840	0.9125	1.1493	0.5187
$max(M_x)$	1.9470	1.8579	1.8476	2.7967	2.4186	1.1522
$mean(M_y)$	0.7395	1.8579	0.7586	0.7543	0.7357	0.8116
$max(M_y)$	0.7395	1.8579	1.3687	1.3776	1.3771	1.1597
$mean(M_{rot})(\times 10^{-4})$	1.2820	1.4704	0.8297	0.6719	0.7182	0.6089
$max(M_{rot})$	0.0038	0.0045	0.0035	0.0030	0.0041	0.0039
$R_{fail}(\%)$	0.33	0.27	0.33	1.56	3.80	7.16

above are comparable to the random motion drift cases in Table 2.1 and Table 2.2, indicating that NDSEP-Motion Correction is able to correct slow and constant drift. Similarly, small and large drift are applied to the no-noise case s00c00 and noisy case s03c10 and s05c15. For small drift in cases s00c00, s03c10,and s05c15: $R_{fail}(\%)$ is 0 for all three cases; $mean(M_x)$ is 2.28, 0.85, 0.82; and $mean(M_y)$ is 0.41, 0.84, 0.70, respectively. For large drift: $R_{fail}(\%)$ is 4.5, 0.83, 26.2; $mean(M_x)$ is 0.61, 0.89, 1.17; and $mean(M_y)$ is 1.45, 1.73, 1.06, respectively. For large motions with intensive noise, the failure rate of 26.2% is higher than other cases, while in terms of successful correction output, $mean(M_x)$ and $mean(M_y)$ remain comparable to the small drift cases. We anticipate that image frames with such intensive noise and motion do not occur often in practice. In the clear small motion case, 2.28 is a little bit higher than others, but it's still smaller than half size of the neurons, which is about 7 pixels, in the simulated dataset.

From the results described above, we conclude that the NDSEP Motion Correction can accurately detect motion translation and rotation — with or without the presence of

Table 2.3: Motion correction accuracy on simulated noisy datasets ($P_{rot} = 25$, $\alpha_{rot} = 6.3153$) with different noise levels.

Noisy case	s01c05	s01c10	s01c15	s03c05	s03c10	s03c15	s05c05	s05c10	s05c15
$mean(M_x)$	0.7605	1.0310	1.2810	0.5778	1.4274	1.1609	1.1774	1.1628	1.2028
$max(M_x)$	1.7837	2.2656	2.8494	2.3209	2.9944	2.5531	2.7935	2.3158	2.1030
$mean(M_y)$	0.9026	0.8362	0.9485	0.8211	0.9484	0.9668	0.8385	0.9608	0.8041
$max(M_y)$	1.6227	1.8998	2.2747	1.3876	1.6665	2.1496	2.2238	2.1455	1.6457
$mean(M_{rot})(\times 10^{-4})$	0.6755	0.9516	1.1771	0.7090	1.1146	2.3755	1.5048	1.2410	1.8007
$max(M_{rot})$	0.0049	0.0042	0.0050	0.0060	0.0081	0.0082	0.0071	0.0080	0.0079
$R_{fail}(\%)$	0.56	0.56	0.83	0.89	1.50	0.50	0.67	0.89	0.33

noise — in most cases.

For the simulated data, all 97 of the active neurons were detected by NDSEP. Three neurons should not be detected, since all three of these were inactive for the duration of the image sequence. In Figure 2.5 (A) and (B), the ground truth neurons are depicted as bright blobs, which are overlaid on the mean map of all 1800 frames in the simulation-derived dataset in the case of s03c05. Since different neurons have different firing rates they generally appear with different levels of brightness in the mean map. Each circle with a red perimeter in Figure 2.5 represents a detected neuron.

Figure 2.5 (C) shows signals that have been extracted by NDSEP for 5 randomly chosen neurons. The figure also shows the corresponding ground truth signals. The signals shown in red correspond to spike events. These signals have a value of 1 when the corresponding neuron fires and 0 when the neuron is not firing. Blue signals indicate $\Delta F/F$ values for the detected neurons. From the results in Figure 2.5, we see that NDSEP accurately detected the spike events. For these results, we computed $R_s = 0.354$, $R_n = 0.000011$, and $10\log_{10}(R_s/R_n) = 45.08$.

2.5.2 Neurofinder Data

We also performed experiments using the Neurofinder collection of datasets [45]. These datasets represent publicly available, real neuron imaging data that has already been preprocessed with motion correction. The Neurofinder data provides ground truth for the position of each neuron. There are five datasets in the Neurofinder database. The first one (Dataset 00) contains segmented neurons using fluorescently labeled anatomical markers. It is possible that neurons are not firing in Dataset 00, but are still labeled. This is inappropriate for neuron detection based on neural activity. Datasets 02 and 04 have around 8%–41% potentially mislabeled neurons [24]. Among the five datasets, 00, 02, and 04 are not suitable for evaluating the neuron detection performance of NDSEP. Therefore, we used Datasets 01 and 03 for our experiments. See Table S1 in the supplementary material for the details about the neurofinder data.

Figure 2.6 shows results of our experiments with the two Neurofinder datasets. The results show that NDSEP is effective at detecting relatively active neurons. In Figure 2.7, the extracted signals for ten randomly selected neurons from each dataset are plotted. Most of the signals in Figure 2.7(B) exhibit the signal characteristics that are described in [29]. Technically, these experiments pertain to the combination of the Preprocessing and Neuron Detection (PND) actors of NDSEP since the experiments do not involve motion correction (the Neurofinder input data is already motion-corrected) or signal extraction. We refer to this actor combination concisely as NDSEP-PND.

Next, we report the precision, recall, and F1 scores achieved by NDSEP-PND across each of the two Neurofinder datasets. The precision is the fraction of true neurons (true

positives) detected among detected neurons. The recall is the fraction of actual neurons that are detected. The F1 score is defined as the harmonic mean of precision and recall: $\mathrm{F1\_score} = 2 \times (u \times v)/(u + v)$, where u is the precision and v is the recall. For Dataset 01, the precision, recall, and F1 scores are 0.6431, 0.5275, and 0.5848, respectively. For Dataset 03, the precision, recall, and F1 scores are 0.7166, 0.7259, and 0.7212, respectively.

The results on the Neurofinder datasets demonstrate that NDSEP-PND has accuracy that is comparable with the top 5 neuron detection algorithms from the comparative experimental study reported on in [49]. These top 5 previously-developed algorithms are: HNCcorr [50]; Sourcery; UNet2DS; Suite2p [51] + Donuts [52]; and HNCcorr [50] + Conv2din. Out of the 6 algorithms (the five previously-developed ones together with NDSEP-PND), the result of NDSEP-PND has the fifth highest accuracy (in terms of recall and precision) for Dataset 01, and for Dataset 03, NDSEP-PND also has the fifth highest accuracy. At the same time, NDSEP-PND achieves real-time performance, while the other 5 methods are not real-time systems. This is a critical advantage of NDSEP-PND in the context of our work. Also, in relation to the other state-of-the-art algorithms in the Reference [45] challenges, Reference [53] (recall=0.56, precision=0.85, f1=0.67 for nf 01) and Reference [24] (recall=0.65 precision=0.57 and f1=0.61 for nf 01 and recall=0.56 precision=0.54 f1=0.55 for nf 03), NDSEP-PND outputs comparable results for both datasets.

2.5.3 Anterior Lateral Motor Cortex Data

As a representative real-world application, an Anterior Lateral Motor Cortex (ALM) dataset [54] is used to evaluate NDSEP. This dataset includes 11189 frames of calcium imaging that record the anterior motor cortex in mice, where the mice are performing a tactile delay-response task. The motion correction failure rate $Rate_{fail}$ of NDSEP on the ALM dataset is 0.

Figure 2.8 shows the results of applying NDSEP to the ALM dataset. Figure 2.8(A) shows the detected neuron masks overlaid on the mean signal map. We randomly picked 10 neurons (Figure 2.8(B)) and plotted $\Delta F/F$ (Figure 2.8(C)). $\Delta F/F$ captures the characteristics of the neural activity. A neuron is detected only after it fires at least once. For example, $\Delta F/F$ of neurons 8, 9, 10 were 0 until their first spiking activity began. In this experiment, NDSEP neuron detection detects 50 neurons. Compared with the ALM ground truth mask which has 69 neurons, the recall and precision of NDSEP based neuron detection are 72.46% and 69.44%, respectively. Also, the F1 score is 70.92%.

2.5.4 Execution Time Measurements

Table 2.6 shows measured execution times for all of the core signal processing actors in NDSEP, as well as the ImgSrc actor. The total running time is measured by using time calculation functions. The times shown in Table 2.6 are the single frame execution times, which are calculated by: $\mathrm{Single\_Frame\_Time} = \mathrm{Total\_Processing\_Time}/\#\mathrm{Frames\_in\_Dataset}$. The mean and standard deviation of the execution time is calculated by repeating the associated experiment 10 times under the same conditions. All execution time measurements

are shown in milliseconds (msec). Only the SetParams and fork actors are not considered in these execution time experiments. SetParams is used only during the initialization mode of the application, and not during real-time mode. Thus, the execution time of the SetParams actor is not relevant to real-time performance. The fork actor is excluded because it is of minimal complexity and has negligible impact on overall performance.

All of the execution time measurements were taken on a MacBook Pro laptop computer. The computer was equipped with a 2.5 GHz Intel Core i7 CPU, the Mac OS High Sierra 10.13.1 operating system, and 16 GB memory.

The experiments on execution time were performed on all of the four datasets employed in Section 2.5.1 through Section 2.5.3. Since image registration has already been applied in the two Neurofinder datasets, we disabled the Motion Correction actor in the experiments with these two datasets. For the simulated dataset, the two Neurofinder datasets, and the ALM dataset, average execution times were taken across 1800 frames, 2245 frames, and 11189 frames, respectively. The results in Table 2.6 show average execution times *per frame* for each actor / dataset combination.

Generally, the Motion Correction actor dominated the overall execution time for the datasets in which the actor was used. The signal extraction actor exhibited the largest variation in execution time. We anticipate that this is because of the strong dependence of this actor's execution time on the number of detected neurons. Also, not only the number of detected neurons but also how active the neurons are affects the detection actor's execution time. As shown in Table 2.6, the total execution time, summed across all actors, was less than 22 milliseconds per frame for all datasets that we experimented with except for the Neurofinder 03 dataset which has over 600 neurons detected. Considering the image

acquisition rate, for all four datasets, NDSEP is shown to provide adequate performance for real-time operation without the need for expensive, cumbersome or special purpose computing hardware.

2.5.5 Comparison with Other Platforms

Using the simulated dataset and ALM dataset, we compared NDSEP-based motion correction with CaImAn-NoRMCorre (Non-Rigid Motion Correction) as well as with ImageJ-SIFT (Scale Invariant Feature Transform). In addition, a neuron detection comparison was made among NDSEP-based neuron-detection, CaImAn-CNMF (CaImAn-Constrained Nonnegative Matrix Factorization), and CellSort (also known as PCA/ICA). For details on CaImAn-NoRmCorre, SIFT, CaImAn-CNMF, and CellSort, we refer the reader to [23, 55, 56].

2.5.5.1 Motion Correction Comparison

The Motion Correction comparison is made on the simulated dataset. In CaImAn-NoRMCorre, we experimented with both the rigid and non-rigid mode. Although CaImAn-NoRMCorre is not natively designed to correct rotations, rotations can be recognized when the grid size is small (the resolution is high). However, excessively small grid sizes make matches hard to find and greatly increase computational cost. In our experiments, we used an empirically-determined grid size of [32, 32], which we found to provide an efficient balance between the aforementioned trade-offs. We also applied the cubic shifting method, and a small overlapping region with size [16, 16]. All other parameters are

set recommended value in [23]. As with CaImAn-NoRMCorre, we tuned parameters in SIFT to maximize the accuracy. Most of the parameter values that we used are those recommended in [55]. We set the maximal alignment error is to 2 pixels to increase the accuracy, instead of 10% of the image size as recommended in [55].

CaImAn-NoRMCorre is not able to align most of the rotations, as shown in Figure 2.9(A), especially in the rigid mode. The high spike values in Figure 2.9(B) correspond to the non-corrected motions. The non-rigid mode has lower spikes, which correspond to rotations, than the rigid mode. However, the non-rigid mode exhibits higher error when correcting translation-only frames. SIFT achieves a relatively high accuracy for both translations and rotations. But Figure 2.9(B) shows that the root mean square error (RMSE) value increases as the number of frames increases. This means that the error accumulates and the accuracy drops as the system continues processing. This feature makes SIFT inefficient for our real-time motion correction context. Unlike CaImAn-NoRMCorre, the NDSEP system efficiently corrects nearly all simulated rotations while discarding the unusual uncorrected frames.

2.5.5.2 Neuron Detection Comparison

In our comparison of neuron detection performance, experiments are made on two datasets, simulated data and ALM data. On simulated data, two approaches, NDSEP-neuron detection and CaImAn-CNMF are compared. On ALM data, three approaches, NDSEP-neuron detection, CaImAn-CNMF, and CellSort, are evaluated. To eliminate the influence of different motion correction approaches, the input datasets are first motion-corrected. In

particular, the simulated data input is the motion correction ground truth used to calculate the RMSE value in Figure 2.9(B). The ALM input data is motion corrected by CaImAn-NoRMCorre with the rigid mode following the parameters used to register the simulated data. When comparing different neuron detection approaches, we compare only the spatial component — that is, only the locations of the detected neurons.

In CaImAn-CNMF, the number of neurons N_{det} to be detected is predefined. In our experiments, for simulated data, we set $N_{det} = 97$ because there are totally 97 neurons in the ground truth mask. N_{det} is set to 80, a little higher than the number of neurons in the ALM ground truth. This setting is used to increase the recall rate. The parameter τ of the Gaussian kernel is set to half the size of a single neuron. The resulting values for the simulated and ALM datasets are $\tau = 3.4$ and $\tau = 8.0$, respectively. The optional parameter P used for normalization by noise and user feed component centroids is disabled for simulated data, since the temporally-autocorrelated noise we have cannot be removed in this way. Other parameters are tuned according to [23]. In CellSort, which is only tested on the ALM dataset, the value of mu is set to 0.5, which enables use of both temporal and spatial information for segmentation. Other CellSort parameters are set as recommended in [56].

Table 2.5 shows that both CaImAn-CNMF and NDSEP detect all of the 97 neurons when the input frames are free from noise. However, for noisy input, the CaImAn-CNMF detection rate $CaImAn - CNMF_{det}$ drops as the noise intensity increases, while NDSEP consistently provides 100% accuracy for both noise-free and noisy input. Thus, our experiments demonstrate that CaImAn-CNMF neuron detection is vulnerable to noise, while NDSEP is much more robust. Furthermore, as shown in Table 2.6, NDSEP requires

significantly less processing time compared to CaImAn-NoRMCorre, SIFT and CaImAn-CNMF.

On the ALM dataset, CaImAn-CNMF correctly detects 53 among 69 neurons; the resulting recall is 76.81%, while the precision is 72.60%. CellSort segments 79 neurons of which 57 of are correct (true positives), and the recall and precision using CellSort are 82.61% and 72.15%, respectively. For NDSEP, the corresponding results (shown in Section 2.5.3) are: $\text{recall} = 72.46\%$, and $\text{precision} = 69.44\%$. From these results, we see that the accuracy of NDSEP neuron detection is comparable to other state-of-the-art approaches, such as CaImAn-CNMF and CellSort.

2.6 Discussion

In this chapter, we proposed a real-time neuron detection and neural activity extraction system called Neuron Detection and Signal Extraction Platform (NDSEP). NDSEP uses a novel integration of dataflow-based design architecture, and streamlined algorithms and software modules for real-time neural signal processing. The dataflow architecture of NDSEP provides flexibility to expand the system, experiment with design trade-offs, and manage complex constraints of real-time neuron detection and activity extraction (RNDAE) systems. Such constraints include those involving memory requirements and cost-effective deployment.

In an experiment based on simulated calcium imaging data, NDSEP effectively performed motion correction with the mean error of x and y displacement less than 1 pixel and the mean rotation error close to zero. NDSEP detected all active neurons and

achieved very high signal-to-noise ratio. For the Neurofinder database and for a real-world data, ALM, NDSEP gained comparable result for the detection and the detected neurons demonstrates typical calcium transient patterns. In all these experiments, the execution times were shorter than 25 milliseconds and NDSEP achieved real-time performance.

As presented in Section 2.4, the key subsystems in NDSEP for neural signal processing are system parameter optimization (represented by the SetParams actor), motion correction, neuron detection and neural signal extraction. We have developed and integrated initial versions of these subsystems through careful design, experimentation and optimization to achieve real-time performance with reasonable system accuracy. However, many alternative combinations of algorithms, algorithmic parameter settings, and design optimization techniques can be applied to achieve the same general functionality as the current version of NDSEP, which involves the mapping of neural image streams into sets of neurons and their associated signals. These combinations represent a complex, largely unexplored design space, which involves trade-offs among real-time performance, neuron detection and signal extraction accuracy, and computational resource costs.

In addition to providing a complete system prototype for RNDAE, NDSEP provides a useful framework for investigating this design space, and developing further innovations in algorithms and systems for RNDAE. Such innovations could, for example, help to further increase the accuracy of neural signal extraction while maintaining real-time performance. Alternatively, they could help to reduce system costs without significantly sacrificing accuracy, thereby contributing to more cost-effective technologies for scientists, clinicians, or patient-users. The model-based design architecture of NDSEP, based on our application of dataflow design methods, helps to precisely formulate the aforementioned

design space in terms of component subsystems (actors for RNDAE) and precise interfacing requirements between them. The modularity and abstract design of the NDSEP architecture greatly facilitates experimentation with alternative combinations of component algorithms, algorithm configurations and hardware/software realizations of the algorithms.

Four general directions for future work emerge naturally from the properties described above of the NDSEP architecture and its utility in defining and exploring important design spaces for RNDAE system design. The first direction is investigation into new algorithms and implementations for the four key component subsystems. Examples of concrete topics in this direction include applying downsampling strategically in parts of NDSEP outside of motion correction, where it is already applied (see Section 2.4.2.1). Another example is incorporating more sophisticated processes for parameter adaptation and optimization in the initialization mode of NDSEP, as motivated in Section 2.4.3.2.

A second direction for future work is in applying the NDSEP platform to develop novel systems for precise neuromodulation. Neuromodulation devices are becoming one of the most powerful tools for the treatment of brain disorders, enhancing neurocognitive performance, and demonstrating causality [57, 58]. A precise neuromodulation system integrates neural activity monitoring, real-time neural decoding, and neuromodulation. In precise neuromodulation, a decoding device predicts a behavioral variable based on neural data streams in real-time. Based on the decoding results, neuromodulation parameters such as timing, frequency, duration, and amplitude are changed. Precise neuromodulation systems with closed-loop real-time feedback are superior to the fixed (open-loop) neuromodulation paradigm [59, 60, 61]. For example, A recent direct brain stimulation study demonstrated significant advantages of precise neuromodulation over open-loop neuro-

modulation [61]. This study applied direct brain stimulation with decoding capability to patients with epilepsy to improve their memory. The study found that stimulation increased memory function only if delivered when the decoding device indicated low encoding efficiency, while stimulation decreased memory function if delivered when the decoding device indicated high encoding efficiency. An open-loop neuromodulation system with a fixed stimulation paradigm may not always facilitate memory function.

The current system will be part of a precise all optical closed-loop neuromodulation system which combines calcium image processing (the current system), prediction (predicting behavioral variables based on neural features), and neuromodulation (optogenetics). In our recent prior work, our team has developed pilot versions of prediction [62] and network-based feature extraction [63] for calcium imaging data. The primary design goal of NDSEP is real-time data processing. Existing optogenetics intervention permits millisecond precision manipulation of genetically-targeted neural populations [64]. In our future work, we will improve these pilot versions and integrate them with NDSEP. We expect that our future all optical closed-loop neuromodulation system can achieve real-time performance above 10Hz, providing neuroscience researchers an open-source, real-time neural decoding system that facilitates precise neuromodulation.

A third direction for future work is studying design optimization methods and trade-offs in NDSEP in the context of overall cost and performance in the enclosing neuromodulation systems.

A fourth future work direction is support for higher image acquisition rates. Results are unpredictable if the speed of the system is slower than the acquisition rate. The designer must therefore optimize and test the system carefully to ensure that constraints imposed

by the acquisition rate are satisfied. The dataflow-based system architecture facilitates these optimization and testing objectives. Our current system is designed for two-photon calcium imaging. The typical acquisition rate is 10-30Hz. Based on Table 4, the current implementation can handle such an acquisition rate. In the future, if we want to use NDSEP for high-speed calcium imaging with a 180–490 Hz sampling rate, hardware acceleration within the framework of dataflow-based design may be used.

On top of the four main directions described above, since NDSEP focuses on real-time computation using efficient detection algorithms, it may have difficulty to detect overlapping neurons. In addition to this, NDSEP can be extended to be enabled for one-photon calcium imaging with more noise. More comparisons to the state-of-the-art methods like OnACID are supposed to be made. Also, NDSEP does not include an actor for neuropil fluorescence contamination. We will address these limitations in our future work.

The NDSEP system developed in this study is an efficient, extensible system based on dataflow design for real-time neuron detection and neural activity extraction. We expect that the platform enables real-time calcium imaging based neural decoding, leading to precise neuromodulation.

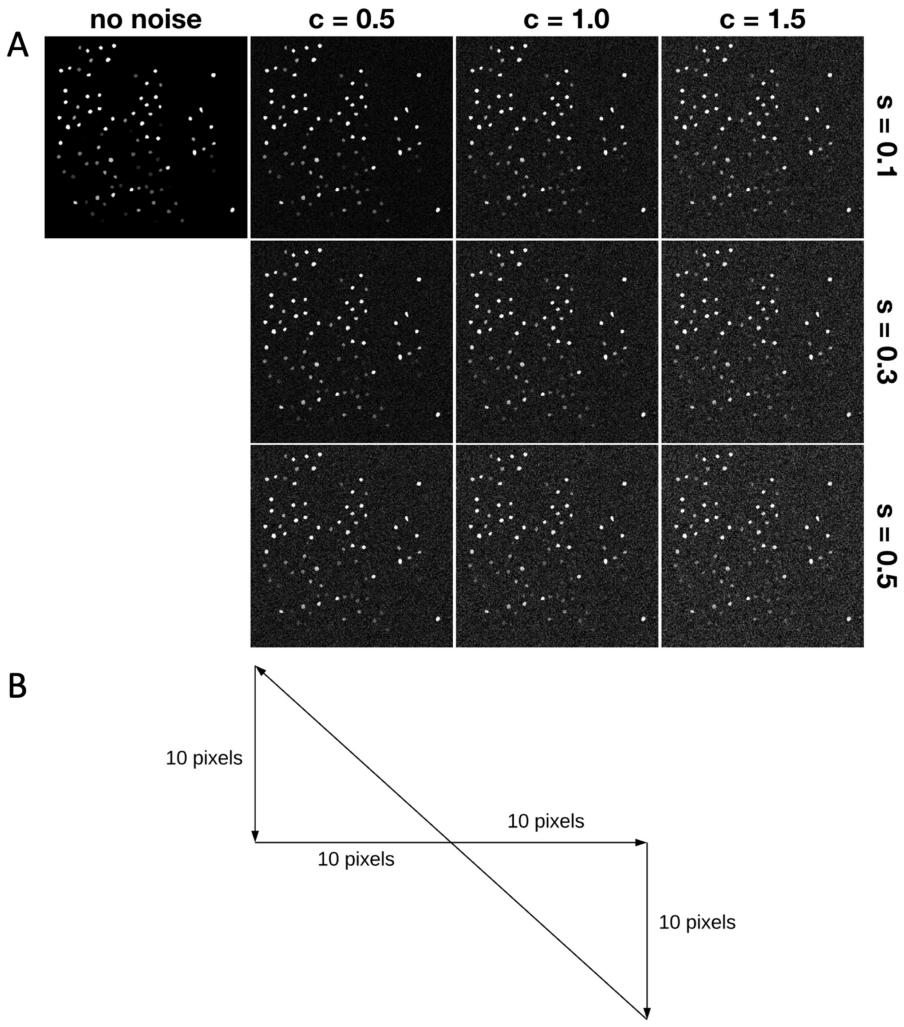

Figure 2.4: Simulation of motion and noise. Part (A) shows simulated noise. Here, "s" denotes the relative shot noise level, while "c" denotes the relative colored (red) noise level. For example, s05c15 denotes the noise case in the lower right corner. Part (B) shows a trajectory simulating slow and constant drift. Each frame moves 1 pixel along the path. For example, assume the first frame has x,y-position (0,0), then the second one is at (1,0), the third one is at (2,0), . . .

Table 2.4: Parameters of Other Approaches.

Parameter	Description	Value
CaImAn-NoRMCorre on Simulated data rigid mode		
bin_width	Width of each bin	50
max_shift	Maximum rigid shift in each direction	[30,30]
$init\_batch$	Length of initial batch	10
CaImAn-NoRMCorre on Simulated data nonrigid mode		
$grid\_size$	Size of non-overlapping regions	[16,16]
$overlap\_pre$	Size of overlapping region	[16,16]
$shifts\_method$	Method to apply shifts	cubic
$init\_batch$	Length of initial batch	50
CellSort on ALM data		
mu	Parameter (between 0 and 1) specifying weight of temporal information in spatio-temporal ICA	0.5
$maxrounds$	Maximum number of rounds of iterations	1000
CaImAn-CNMF on ALM data		
K	Number of components to be found	80
tau	Size of Gaussian kernel (half size of neuron)	8
$merge\_thr$	Merging threshold	0.2
min_SNR	Minimum SNR threshold	1
CaImAn-CNMF on Simulated data		
K	Number of components to be found	97
tau	Size of Gaussian kernel (half size of neuron)	3.4

Table 2.5: Neuron detection rate with different noise levels.

Noise level	no noise	s01c05	s01c10	s01c15	s03c05	s03c10	s03c15	s05c05	s05c10	s05c15
$CNMF_{det}(\%)$	100	97.9	89.7	74.2	92.8	89.7	74.2	85.6	74.2	68.0
$NDSEP_{det}(\%)$	100	100	100	100	100	100	100	100	100	100

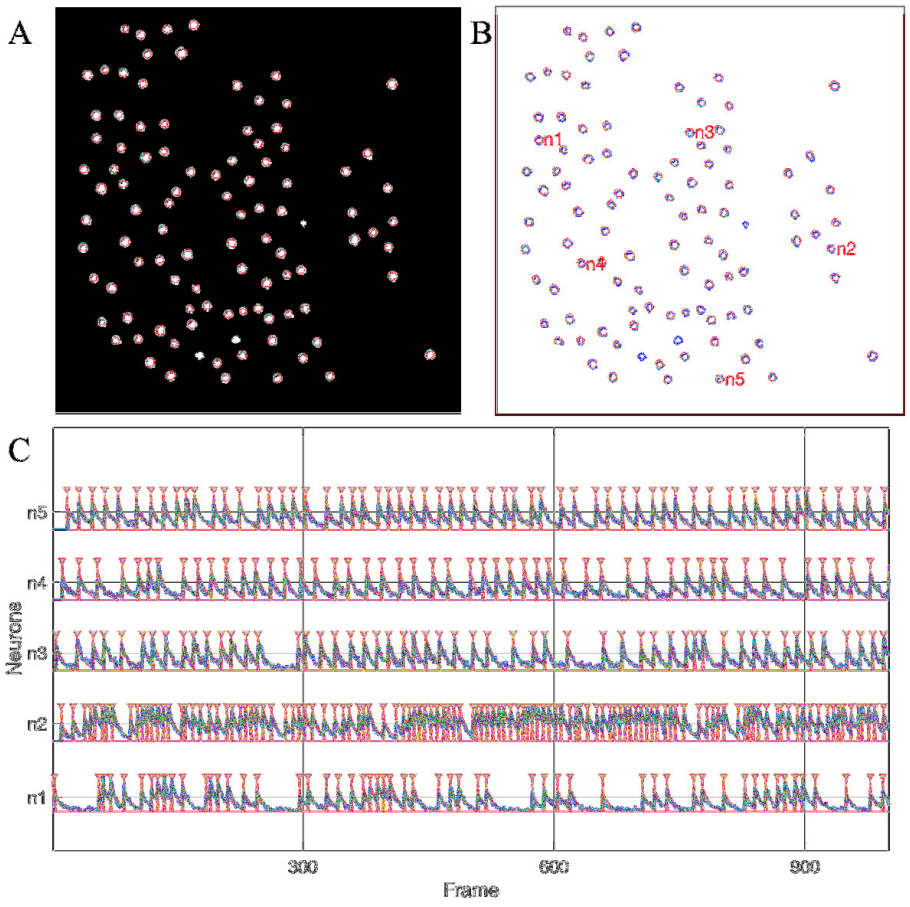

Figure 2.5: Neuron detection results from the simulated data. **(A)** shows detected neurons (red-perimeter circles) overlaid on the mean map of all frames in the simulation-derived dataset; **(B)** shows ground truth neurons with blue perimeters and detected neurons with red perimeters; **(C)** Representative neural signals (blue) extracted from the simulated data and ground truth spikes (red).

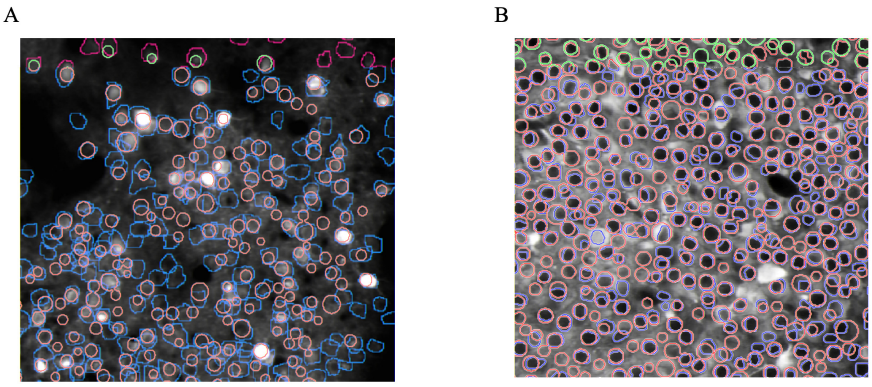

Figure 2.6: Results of experiments with the two Neurofinder datasets: **(A)** shows results from Dataset 01, and **(B)** shows results from Dataset 03. The ground truth regions are bounded with red perimeters, and the results from NDSEP are bounded with green perimeters.

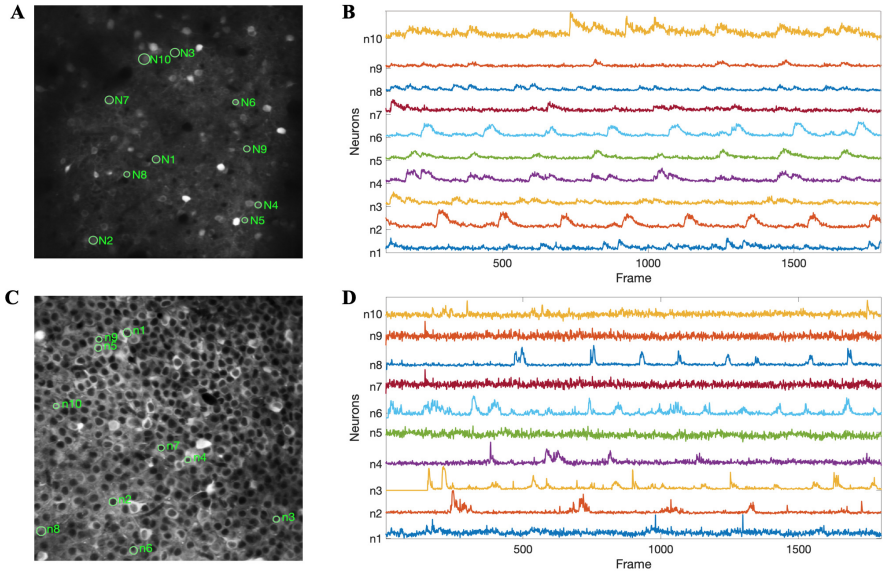

Figure 2.7: Results using the Neurofinder data for extracted signals from randomly-selected neurons: **(A)** and **(C)** show the locations of the randomly-selected neurons from Dataset 01 and Dataset 03, respectively; **(B)** and **(D)** show the signals that were extracted by NDSEP from each neuron. From each of the two datasets, 10 neurons were randomly selected.

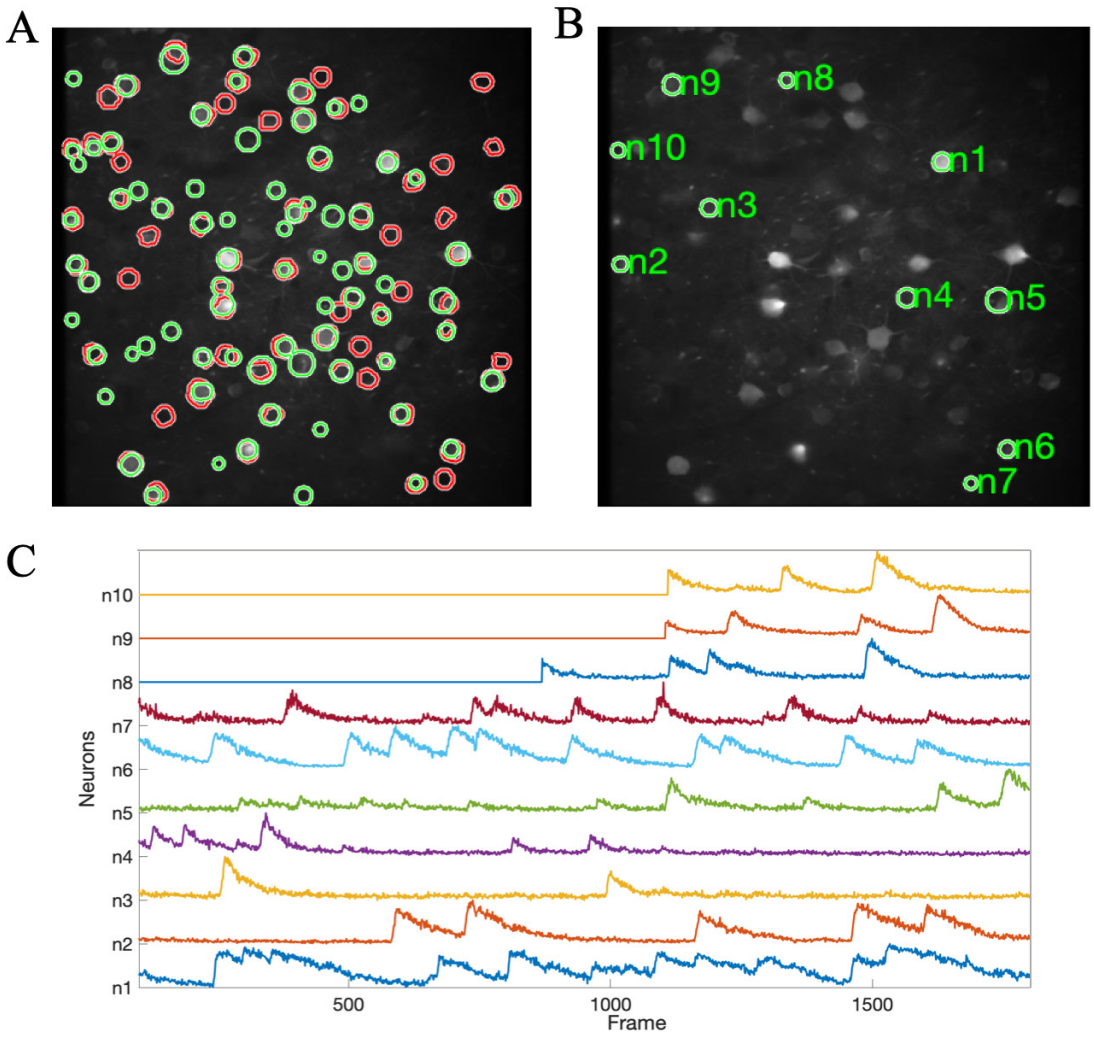

Figure 2.8: Results for ALM data: **(A)** shows neuron masks in regions with green perimeters with ground truth in red, overlaid on the mean map of all frames; **(B)** shows randomly selected neurons among the correctly detected result; **(C)** shows the signals that were extracted from selected neurons in B.

Table 2.6: Measured execution times for different actors in NDSEP

Data	ImgSrc (msec)	Motion Correction (msec)	Pre processing (msec)	Detection (msec)	Signal Extraction (msec)	Total Execution Time (msec)
Simulated (400×400) 97 neurons .png by NDSEP (Image acquisition rate: 10 Hz)	1.18 (std:0.04)	12.98 (std: 0.84)	0.66 (std: 0.02)	2.72 (std: 0.09)	4.49 (std: 0.14)	22.04 (std:0.87) (45.37Hz)
Simulated (400×400) 97 neurons .png by SIFT	-	131.11 (std: 0.9729)	-	-	-	-
Simulated (400×400) 97 neurons .png by NoRMCorre-Rigid	-	21.66 (std: 0.31)	-	-	-	-
Simulated (400×400) 97 neurons .png by NoRMCorre-Nonrigid	-	261.12 (std: 12.75)	-	-	-	-
Simulated (400×400) 97 neurons .png by CNMF	-	-	-	27.9382 (std: 0.1476)	-	-
Neurofinder 01 (512×512) 345 neurons .tiff by NDSEP (Image acquisition rate: 7.5Hz)	2.15 (std:0.03)	-	1.11 (std: 0.01)	4.56 (std: 0.02)	14.31 (std: 0.13)	22.13 (std:0.16) (45.19Hz)
Neurofinder 03 (498×490) 613 neurons .tiff by NDSEP (Image acquisition rate: 7.5Hz)	2.14 (std:0.11)	-	1.19 (std: 0.07)	12.76 (std: 0.68)	33.32 (std: 1.05)	49.42 (std:1.54) (20.24Hz)
ALM (512×512) 69 neurons .tif by NDSEP (Image acquisition rate: 15 Hz)	3.95 (std:0.04)	11.53 (std: 0.09)	0.71 (std: 0.01)	1.35 (std: 0.01)	3.13 (std: 0.01)	20.67 (std:0.15) (48.38Hz)

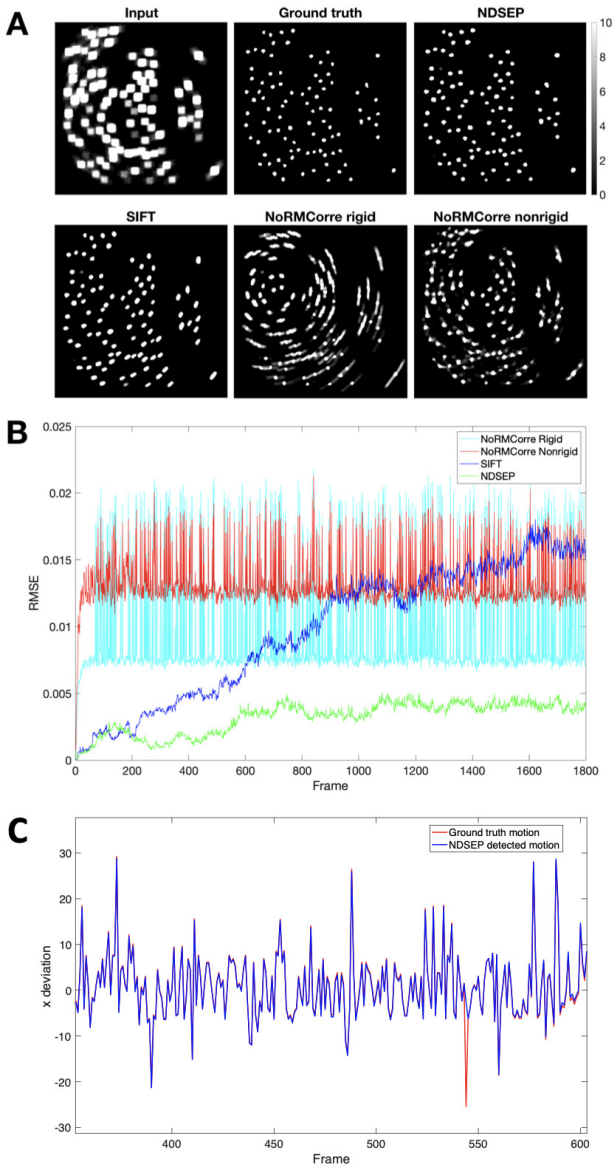

Figure 2.9: Motion correction comparison results on simulated dataset $P_{rot} = 25$, and $\alpha_{rot} = 6.3153$ without noise. **(A)** shows the uint8 [0, 255] meanmaps. The display range is [0, 10], where values greater than 10 are displayed as white. The input data is the meanmap of simulated data with motion but without noise. The ground truth is the meanmap of the no-motion simulated data. The remaining four images are the output meanmap of NDSEP, SIFT, the rigid mode of CaImAn-NoRMCorre, and the nonrigid mode of CaImAn-NoRMCorre. **(B)** shows the RMSE calculated by performing frame-by-frame comparison between the output of the four motion correction methods and the ground truth. (C) shows the x movement which is actual movement applied to simulated data in red line vs movement detected by NDSEP motion correction in blue line, including a program detected failure in frame 544. Please note the motions larger than 10 pixel upper bound is because of the rotations applied around the center of image.

Chapter 3

A Parameter-Optimization Framework for Neural Decoding Systems

3.1 Chapter Overview

Real-time neuron detection and neural activity extraction are critical components of real-time neural decoding. They are modeled effectively in dataflow graphs. However, these graphs and the components within them in general have many parameters, including hyper-parameters associated with machine learning subsystems. The dataflow graph parameters induce a complex design space, where alternative configurations (design points) provide different trade-offs involving key operational metrics including accuracy and time-efficiency. In this chapter, we propose a novel optimization framework that automatically configures the parameters in different neural decoders. The proposed optimization framework is evaluated in depth through two case studies. Significant performance improvement in terms of accuracy and efficiency is observed in both case studies compared to the manual parameter optimization that was associated with the published results of those case studies. Additionally, we investigate the application of efficient multi-threading strategies to speed-up the running time of our parameter optimization framework. Our proposed optimization framework enables efficient and effective estimation of parameters, which leads to more powerful neural decoding capabilities and allows researchers to experiment more easily with alternative decoding models.

Material in this chapter was published in partial, preliminary form in [65].

3.2 Introduction

Neural decoding based on neuroimaging signals is an important tool for understanding neural codes for studying and treating brain disorders, such as Alzheimer's disease and Parkinson's disease. Neural decoding systems in general have many parameters, including hyperparameters associated with machine learning subsystems. The parameters induce a complex design space, where alternative configurations (design points) provide different trade-offs involving key operational metrics, including accuracy and time-efficiency. Parameter optimization of neural decoding systems is important for achieving strategic trade-offs between accuracy and time-efficiency. For example, for off-line neural signal analysis, it is typically desirable to optimize parameters to favor high accuracy at the expense of relatively long running time. On the other hand, for real-time analysis, parameter optimization would typically be geared towards maximizing accuracy subject to strict execution time constraints. Real-time neural decoding is useful, for example, in precision neuromodulation systems, where stimulation to the brain must delivered in a timely manner in relation to the current state of brain activity. The diverse trade-offs that must be considered in neural decoding — depending on the application scenario — creates a need for flexible and effective parameter optimization that can be used to efficiently navigate the design spaces for configuring neural decoding systems.

Neural decoding systems may involve both continuous-valued parameters as well as discrete-valued parameters. In this work, we refer to *parameter optimization* as the optimization of parameter values for parameter sets that are in general hybrid combinations of continuous and discrete parameters. Our view of parameter optimization is therefore

more general than parameter "tuning", which is typically associated only with continuous-valued parameters.

Parameter optimization for neural decoding is challenging due to the complexity of the underlying design spaces, as described above. With manual approaches, which are conventionally used, system designers may be effective in selecting very high level parameters, such as the types of decoding or preprocessing algorithms to be used; however, it is extremely time consuming to study a wide range of alternative design points in a way that comprehensively takes into account the impact of and interactions between diverse sets of relevant parameters. Moreover, conventional parameter optimization approaches typically consider only algorithmic parameters, whereas dataflow parameters, which have significant impact of time-efficiency, are not considered. Here, by *dataflow* parameters, we mean parameters associated with executing the different signal processing modules in a neural decoding system on the targeted hardware platform.

In this chapter, we develop an optimization framework for automated and holistic parameter optimization of neural decoding systems. The framework considers both algorithmic and dataflow parameters while jointly taking into account both the neural decoding accuracy and execution time of the optimized solutions. The proposed framework applies a population-based search strategy to optimize the relevant algorithmic and dataflow parameters of the given neural decoding system. The framework is general in that a variety of search strategies can be plugged-in; it is not specific to a single type of search method. To demonstrate this generality, we apply two different search strategies in our experiments — Particle Swarm Optimization (PSO), which is a randomized search strategy that is effective for navigating nonlinear design spaces that are based on diverse types of parameters [66],

and Genetic Algorithms (GAs), which is a metaheuristic method that uses different kinds of biologically inspired operators, such as mutation, crossover and selection, for evolving successive generations of populations (sets of candidate solutions). We present a prototype implementation of the proposed framework in a novel software tool, which we refer to as the NEural DEcoding COnfiguration (NEDECO) package, since the objective of the package is to help experimental neuroscientists and neural decoding system designers to arrive at strategically-optimized configurations of neural decoding implementations.

NEDECO operates by iteratively executing alternative neural decoding configurations to assess their performance and feed back the assessment to help derive new candidate configurations to evaluate. This approach of feedback-driven design space exploration is carried out based on the PSO and GA methodology. To accelerate the evaluation of neural decoding configurations, we exploit their dataflow models to derive efficient multi-threaded executions of the configurations on commodity, off-the-shelf desktop or laptop computers that employ multicore processors. We perform an extensive experimental evaluation of NEDECO in which we compare its parameter optimization results to the manually-optimized parameter configurations for two previously published systems for neural decoding. Our results demonstrate the effectiveness of NEDECO in deriving neural decoding implementations that offer significantly improved trade-offs between decoding accuracy and the speed at which decoding is performed.

The remainder of this chapter is organized as follows. Section 3.3 presents related work in neural decoding and summarizes the contribution of this chapter in the context of the related work. In Section 3.4, we introduce the architecture of NEDECO, including the models and methods for automated parameter optimization and the methods for

accelerating the parameter optimization process through efficient use of parallel processing resources. In Section 3.5, we introduce the experimental methodology that we use to demonstrate and evaluate NEDECO, and we present the results of our experimental evaluation, which concretely demonstrate the effectiveness of NEDECO when it is applied for parameter optimization to multiple state-of-the-art neural decoding tools. Finally, in Section 3.6, we summarize the developments of the chapter, and discuss limitations of NEDECO and directions for future work.

3.3 Background and Related Work

A number of previous research efforts have studied parameter tuning for neural decoding. For example, Pnevmatikakis et al. proposed a method for automatically tuning a selected subset of parameters, including the calcium indicator dynamics and time-varying baseline concentration, but the remaining parameters still require manual tuning [67]. Moreover, there is no systematic integration between the human-tuned parameters and the optimization of parameter values for the parameter-subset that is automatically tuned. Giovannucci et al. eliminate the hyper-parameter tuning in the a convolutional neural network (CNN) classifier subsystem within their proposed neural decoding model [23]. However, like the work of Pnevmatikakis et al., the tuning process developed by Giovannucci et al. targets a subset of parameters and requires a separate human-driven tuning step for the remaining parameters. Additionally, and perhaps most significantly, the parameter tuning processes in [67] and [23] are based on the image analysis models and algorithms that they apply for their neural decoding systems. The automation strategy cannot be directly applied to other

analysis models for neural decoding.

Glaser et al. [68] present a tutorial and accompanying software package to assist neuroscientists in applying machine learning methods to neural decoding problems. While the machine learning methods integrated in the software package outperforms conventional neural decoding methods, the work of [68] does not address the problem of automated parameter tuning.

Liu et al. [69] proposed a real-time PSO method for power system optimization. In this method, a novel approach was developed to accelerate the particle evaluation (fitness evaluation) process. The approach of Liu is representative of various works that focus on fitness function acceleration. In contrast, NEDECO is designed as a higher-level framework that can be integrated with a variety of search strategies and fitness functions, and is developed and demonstrated based on specific constraints and requirements that are involved in the design and implementation of real-time neural decoding systems.

PSO methods have also been investigated for hyper-parameter optimization in deep neural networks [70, 71, 72, 73, 74]. In contrast to these methods, our proposed NEDECO framework can be applied to neural decoding systems that use any type of machine learning model (not limited to DNN models); NEDECO can apply other search strategies (not just PSO); and our work is the first to apply such a general and comprehensive parameter optimization framework to neural decoding, which is a useful contribution to computational neuroscience.

To the best of our knowledge, NEDECO is the first software tool for generalized, automated tuning of parameters in calcium-imaging-based neural decoding systems. By "generalized", we mean that the tool works comprehensively over all decoding algorithm

parameters, works across a wide variety of model types and associated information extraction algorithms rather than being restricted to a specific type of model, and takes into account both the neural decoding accuracy and run-time efficiency. In Section 3.5, we demonstrate the flexibility of NEDECO by applying it to two significantly different neural decoding tools from the literature — the Neuron Detection and Signal Extraction Platform (NDSEP) [75], and CellSort [56], using two different methods as the underlying search strategy — particle swarm optimization and genetic algorithms. Our experiments demonstrate the ability of NEDECO to derive parameter settings that lead to significantly improved neural decoding performance compared to the previous published results using NDSEP and CellSort, which are based on hand-tuned parameters.

3.4 Proposed Method

NEDECO applies PSO for optimizing heterogeneous collections of neural decoding system parameters, including continuous and discrete parameters. Additionally, the PSO techniques of NEDECO are implemented within a dataflow framework. This facilitates the retargetability of the framework to different neural decoding algorithms, and different platforms for optimization, and also facilitates the acceleration of the optimization process on multicore computing platforms. The latter feature — acceleration of the optimization process — is important because the the process is computationally intensive, and higher quality solutions can generally be produced within a given time period if more efficient execution of the optimization engine is enabled. In Section 3.4.1, Section 3.4.2, and Section 3.4.4, we review fundamentals of PSO, GAs, and dataflow modeling, which are

key foundations of NEDECO.

3.4.1 Particle Swarm Optimization

PSO is a form of population-based, randomized, iterative computation for optimization in the context of complex, multidimensional search spaces in which different dimensions of a search space may have very different characteristics and underlying data types [66]. Inspired by the social behavior of animal flocks, populations in PSOs are referred to as *swarms*, and each member of a population (swarm) is referred to as a *particle*. Each particle represents a candidate solution in the context of the optimization problem that the enclosing PSO is designed to solve. Operation of a PSO involves tracking the positions of particles in the current swarm, and iteratively generating a new swarm from the previous one, which leads to successive generations of swarms. Intuitively, as swarms evolve, the quality of solutions represented by the candidate solutions will tend to increase, which leads to capability of the method to derive optimized solutions.

In general, the position of a particle in a PSO is an encoding of the candidate solution. In NEDECO, particles move in an n-dimensional space, where n is the number of parameters to jointly optimize in the given neural decoding system. Each dimension corresponds to a distinct parameter. The component of the position vector in a given dimension gives the value of the corresponding parameter. This can be viewed as a direct way of mapping candidate solutions for a parameter optimization problem into particles (position vectors) for a PSO.

Particles move (transition from one position to another) based on velocity vectors

that are maintained along with the particles. The velocities are typically influenced by neighboring particles as well as by a current "best" particle within the population. A best particle is simply one whose associated design point maximizes the function that the PSO is designed to optimize among all elements of the current population. Here, we are assuming that the optimization objective is one of maximization; the approach described here can be adapted easily for minimization contexts.

Multiple particles may be tied to achieve the maximum function value. In the case of such a tie, one of the maximizing particles may be randomly selected as the best for the purpose of evolving the current PSO population (that is, for exerting influence on other particles' velocities). In NEDECO, this is the process by which multiple best particles are handled; although in general, there are other ways of handling ties.

As described in Section 3.4.1, the velocity of a particle p is used to update its position, and the velocity is determined both by neighboring particles of p in the swarm, as well as by the current best particle in the population. The concept of a neighboring particle is determined by the *topology*, which is a parameter of the PSO. The topology can be represented by a graph. In the *connectivity graph*, the vertices are in one-to-one correspondence with the particles, and two vertices are connected by an edge if the associated particles are neighbors in the given topology. For example, in a fully connected topology, all particles other than p are considered neighbors of p, while for a ring topology, the connectivity graph is ring-structured, and therefore, each particle has exactly two neighbors. A third commonly-used topology is the random topology, as defined by Clerc [76]. A random topology involves a dynamically evolving connectivity graph, which is initialized with randomly-placed connections (edges). The connectivity graph

for the random topology is modified (again using randomization techniques) whenever there is no improvement in the population's best particle after a given transition from one generation of particles to the next. Such a transition is known as an *iteration* of the PSO. Use of a random or ring topology can help a PSO to further avoid getting stuck in local optima.

3.4.2 Genetic Algorithms

Like a PSO particle, a candidate solution is an "individual" in a population within a GA. Individuals in GAs are also referred to as "chromosomes". When a GA is used the search strategy in NEDECO, each GA population contains a set of chromosomes, where each chromosome encapsulates a configuration of PNDS parameters. Initially, populations are randomly generated within the search space. To iteratively evolve toward better solutions, chromosomes within populations undergo random alterations (mutations). Additionally, pairs of chromosomes are randomly selected as "parents" from which new individuals are derived; this processes is referred to in GA terminology as "crossover". In each GA iteration (population generation) the PNDS is used to evaluate each individual of the current population. The fitness is used to determine the probability that a given individual is selected as a parent for crossover — that is, whether or not the individual will influence the next generation of the GA execution. For more details on GAs, we refer the reader to [77].

In our GA implementation for use with NEDECO, six selection methods are supported. These include proportional roulette wheel selection (RWS), stochastic universal

sampling (SUS), classic linear rank-based selection (RNK), linear rank-based selection with selective pressure (RSP), tournament selection (TNT), and transform ranking selection (TRS). Additionally, three crossover methods are supported, including one-point crossover (P1XO), two-point crossover (P2XO), uniform cross-over (UXO), and three mutation methods are supported, including boundary mutation (BDM), single-point mutation (SPM), and uniform mutation (UNM).

3.4.3 Multiobjective Optimization

NEDECO is designed to take into account two metrics for optimization of parameters in a given neural decoding system — both the accuracy of neural detection and the run-time efficiency of the process. In terms of calcium imaging, this joint consideration of both accuracy and efficiency is one of the key innovative aspects of NEDECO. NEDECO applies a linear aggregation approach [78] to weighting the objectives so that the user (neural decoding system designer) can adjust the relative importance levels given to the two metrics based on application requirements. For example, a neural decoding system that is deployed in a closed-loop neuromodulation system would typically have an increased weighting given to run-time efficiency, whereas an offline system for batch processing of neuroscience datasets may have much higher relative weighting for accuracy. The framework also gives the system designer a simple means to tune the final achieved trade-off. In particular, the NEDECO optimization process can be iteratively re-executed — with the relative weightings varied between iterations — in case the system designer would like to evaluate different implementation options in terms of the provided trade-offs.

We apply a linear aggregation approach to handling multiple objectives in NEDECO because a linear approach is intuitively easy for the system designer to understand, especially when a small number of different objective metrics is involved. A general issue that may be considered when extending PSOs and GAs to multiobjective design optimization contexts is that of extending the concept of best solutions to encompass solution subsets that are non-dominated in the sense of Pareto optimization [79]. However, this issue is avoided when *aggregating functions* are applied to map metric values in multiple dimensions into single-dimensional values (e.g., into real numbers). Among aggregating approaches, linear aggregation is perhaps easiest to understand and most commonly used. This approach defines the aggregating function as a linear combination $a_1 m_1 + a_2 m_2 + \ldots + a_q m_q$ of the (possibly normalized) metric values $m_1, m_2, \ldots, m_q$ for a given candidate solution, where there are q dimensions in the design evaluation space. Here $a_1, a_2, \ldots, a_q$ are coefficients of the linear aggregation; in NEDECO, $q = 2$, and the coefficients are defined as $a_1 = z$ and $a_2 = (1 - z)$, where z is a single parameter that the system designer uses to control the relative weighting given to the two metrics, accuracy and run-time efficiency.

Various alternative approaches have been developed to handle multiobjective optimization in PSOs and GAs, such as the hyper-volume indicator approaches, which are standard in multi-objective analysis using Pareto fronts. These approaches are more sophisticated compared to linear aggregation (e.g., see [80, 78, 81, 82, 83]). Investigation of such approaches in the context of NEDECO is an interesting direction for future work.

3.4.4 Dataflow Modeling

NEDECO is designed based on dataflow modeling concepts — particularly, on a form of dataflow modeling for signal and information processing systems that is called core functional dataflow [84]. Dataflow is a model of computation that is widely used in the design of signal and information processing systems, including, in recent years, in the design of systems for neural decoding [75].

In dataflow-based application modeling, applications are represented as directed graphs in which the vertices, called *actors* represent functional modules and each edge represents first-in, first-out (FIFO) communication of data from one actor to another [22]. Each unit of data that is communicated along a dataflow edge is referred to as a *token*. These tokens can have arbitrary data types associated with them, such as integers, floating point numbers, pointer types, and arbitrary classes in object-oriented programming languages.

Actors are executed in terms of discrete units of execution, which are referred to as *firings*, where the discrete units are typically defined in terms of the numbers of tokens that are produced and consumed from the edges. For example in synchronous dataflow (SDF), which is an important specialized form of dataflow, each actor produces and consumes a constant number of actors on each incident edge [85]. The numbers of tokens produced and consumed by an SDF actor can vary from one incident edge to another (and among different actors), but for each incident edge, the number must be constant for all firings of the actor.

Dataflow representations are useful for formally representing the high-level signal flow and computational organization of signal and information processing systems. A

wide variety of methods have been developed for analysis and design optimization of system implementations that are derived using dataflow representations [3]. Due to the precise manner in which components are interfaced — based on connectivity in the dataflow graph and characterizations of token production and consumption — dataflow-based representations also facilitate "plug-and-play" signal processing architectures, where different versions of an actor subsystem can be used at different times during execution or in different versions of the system. This enhanced modularity is exploited in NEDECO to make the tool easy to retarget for parameter optimization of different neural decoding systems.

For more background on dataflow modeling, analysis and design optimization for signal and information processing systems, we refer the reader to [22, 3], and for details on how dataflow methods can be applied to efficient neural decoding, we refer the reader to [75].

3.4.5 Core Functional Dataflow

As mentioned in Section 3.4.4, NEDECO applies a form of dataflow called core functional dataflow (CFDF), which is useful for designing and integrating dataflow actors that involve different modes of operation. A CFDF actor A has an associated set of modes $M(A) = \{\mu_1, \mu_2, \ldots, \mu_{c(A)}\}$, where $c(A)$ denotes the number of modes in A. Each mode μ_i corresponds to a distinct computational function that is to be performed by the actor. Each firing f of a CFDF actor A has associated with it a unique mode $\kappa(f) \in M(A)$, which governs the type of computation that is to be performed during the firing.

For each dataflow edge e that is incident to A, the dataflow rate associated with A and e is constant for a given mode. Here, by the *dataflow rate*, we mean the number of tokens consumed by A from e if e is an input edge of A, and the number of tokens produced by A onto e if e is an output edge. CFDF is therefore similar to SDF in its requirement of constant dataflow rates. However, CFDF is more flexible in that this requirement is imposed only at the finer-grained level of modes, rather than at the level of complete actors. Thus, different modes of A can have different dataflow rates associated with them. The requirement of constant mode-level dataflow rates and the potential for heterogeneous dataflow behavior between modes provides a useful combination of expressive power and analysis potential when applying CFDF (e.g., see [84]).

3.4.6 Architecture Design

Fig. 3.1 illustrates the overall architecture of the NEDECO platform. The platform is developed using the lightweight dataflow (LIDE) package, which is a software tool that facilitates design and implementation of dataflow-based software systems and tools for signal and information processing [36]. Dataflow modeling in LIDE is "lightweight" in the sense that it involves a compact set of application programming interfaces (APIs) that can easily be retargeted to different implementation languages. This makes it relatively easy to adapt LIDE for different design processes and apply it to different systems and tools, such as NEDECO.

The block in Fig. 3.1 labeled PNDS encapsulates the neural decoding system that is being applied to NEDECO so that optimized parameter configurations can be derived.

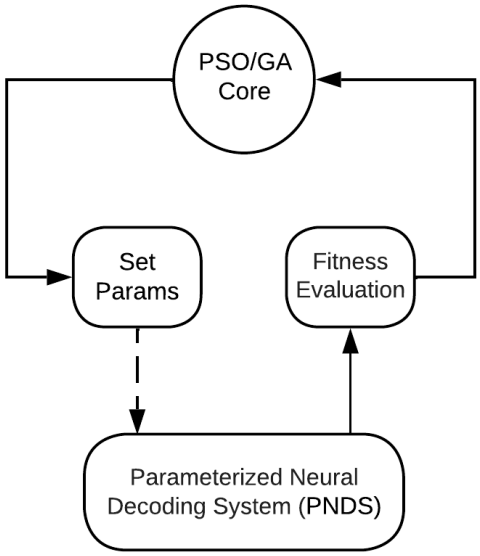

Figure 3.1: An illustration of the overall architecture of the NEDECO platform. The PSO/GA Core means the actor could be either PSO Core actor or GA Core actor.

Here, PNDS stands for Parameterized Neural Decoding System. Integrating a PNDS into NEDECO involves writing a compact software layer ("wrapper") for the PNDS so the overall decoding system can be executed as a CFDF actor within NEDECO, and so the parameters of the PNDS can be modified in a standard way, using the actor-parameter configuration mechanisms of the underlying LIDE tool. Here, we use a minor abuse of terminology where we use "PNDS" to describe both the neural decoding system whose parameters are being optimized and the actor in NEDECO, as shown in Fig. 3.1, that interfaces the neural decoding system to overall PSO-based or GA-based optimization process. A PNDS can be plugged into the NEDECO framework either based on an existing neural decoding algorithm or based on a newly-developed algorithm.

Each firing of the PNDS actor involves executing the given neural decoding system repeatedly on a pre-defined dataset of calcium imaging based neural images, and aggregat-

ing the resulting measurements on execution time and accuracy. The wrapper functionality is responsible for iterating across the given dataset, and producing the results, encapsulated in the form of dataflow tokens, so that they can be interpreted by the enclosing PSO or GA process. Each token produced by the PNDS actor encapsulates a pair of floating-point values — one that represents the accuracy and the other that represents the execution time — where both values are averaged across measurements taken for each image in the dataset.

In Section 3.5, we demonstrate and experiment with NEDECO using two alternative neural decoding systems as the PNDS — NDSEP and CellSort. However, NEDECO is not limited to NDSEP and CellSort; the process of defining wrappers for integration with NEDECO can be applied flexibly to other neural decoding systems, which significantly broadens the applicability of the platform.

The PNDS itself need not be implemented using CFDF nor any other kind of dataflow methods; only the CFDF-based wrapper is needed to ensure its proper integration into NEDECO. Between the two PNDSs that we experiment with in Section 3.5, NDSEP is implemented using LIDE and its associated CFDF support, while CellSort is implemented independently of LIDE, and without any explicit connection to dataflow modeling.

The PSO/GA Core block in Fig. 3.1 executes the core of the PSO optimization process. This is a PSO or a GA implementation that is encapsulated as a CFDF-based dataflow actor so that it can be connected with arbitrary PNDSs that have associated NEDECO wrappers. More details on the PSO Core actor and GA Core actor are discussed in Section 3.4.7, and Section 3.4.8, respectively. The Set Params block in Fig. 3.1 receives tokens that encapsulate PNDS parameters and associated values that should be used to change those parameters in the next execution of the PNDS. A dashed edge is drawn

in Fig. 3.1 to connect the Set Params block to the PNDS because this is not a dataflow connection. Instead, this is a connection that involves calling interface functions associated with the PNDS wrapper to set parameter values in the PNDS.

The Set Params block is not a pure dataflow actor since it communicates with another actor using mechanisms other than the passage of tokens through edges. The general approach of integrating non-dataflow parameter manipulation with parameterized dataflow subsystems is useful in many contexts of signal processing system design [3].

The Set Params block manages parameters in a general way so it just needs to be reconfigured (rather than re-implemented) when a new PNDS is applied to NEDECO. Reconfiguring the Set Params block involves providing a set of pointers to parameter values in the PNDS that can be varied by the PSO or GA, along with the sizes of the data types associated with the parameters.

The Fitness Evaluation block in Fig. 3.1 is a simple actor that applies the linear aggregation function discussed in Section 3.4.3 to the results produced by the PNDS actor. The actor can easily be replaced by alternative actors (or extended in a parameterized way within the same actor) to perform different aggregation functions. We anticipate that future developments in NEDECO will include incorporation of alternative aggregation functions to facilitate experimentation with this aspect of the PSO-based and GA-based approach.

Further workflow details for the overall NEDECO architecture are presented in the supplementary materials (Algorithm S1).

To summarize the flow of data in the optimization loop represented in Fig. 3.1, we start by observing that each token produced by the PSO Core actor or the GA Core actor corresponds to a single PSO particle or a single GA chromosome, which in turn

corresponds to a single candidate solution (parameter configuration) for the PNDS. Each token produced by the PNDS actor provides evaluation results — accuracy and execution time — for a given candidate solution. Each firing of the Fitness Evaluation actor simply converts the accuracy and execution results into a single floating point value through a linear aggregation of the two components of the input token. The `update-population` mode of the PSO Core actor and the `update-crossover` and `update-complete` modes of the GA Core actor, which are the most important modes of the core actors, each consume a block of P fitness evaluation results (for all candidate solutions in the current population), and use these results to generate the next generation of the population. The resulting new candidate solutions are output, as a block of P tokens upon completion of each of the `update-population`, `update-crossover`, and `update-complete` modes.

3.4.7 PSO Core Actor

In this section, we present design details for the PSO Core actor, and in the following section we provide an overview of the GA Core actor. These two actors are key components of NEDECO, and are also envisioned to be applicable to a wide variety of other parameter optimization contexts in computational neuroscience. The PSO and GA optimization algorithms share similar operating processes, including population initialization, iterative mutation of individuals within the population, and iterative fitness evaluation.

The PSO Core actor has six CFDF modes, which are referred to as the `initialize-write`, `initialize-read`, `update-population-write`, `update-population-read`, `stopping-evaluation`, and `write-output` modes.

Table 3.1: Dataflow table for the PSO Core actor.

Mode	Input	Output
initialize-write	0	P
initialize-read	-P	0
stopping-evaluation	0	0
update-population-write	0	P
update-population-read	-P	0
write-output	0	0

Fig. 3.2 illustrates the *mode transition diagram*, which is a graphical representation for a CFDF actor that shows how the current mode of the actor transitions from one firing to the next. A mode transition diagram is a general method for characterizing CFDF actors. As part of the general design rules for CFDF actors, the mode that a given firing transitions to is determined as part of the functionality that implements the mode [84].

Table 3.1 represents the dataflow table for the PSO Core actor. Like the mode transition diagram, the dataflow table is another general way for characterizing a CFDF actor. The rows of this table correspond to different modes of the actor and the columns correspond to different ports (incidences with input and output edges). For each output port of a CFDF actor, the corresponding column in the actor's dataflow table specifies the number of tokens produced in each mode, and similarly, for each input port, the corresponding column gives the *negative of* the number of tokens consumed from the port in each mode.

For the PSO Core actor, the parameter P gives the number of particles in the PSO population. Thus, for example, we see from the table that in the `update-population-write` mode, the actor produces P tokens onto the actor's output edge. In the `update-population-read` mode, the actor consumes the same number of tokens from the input edge.

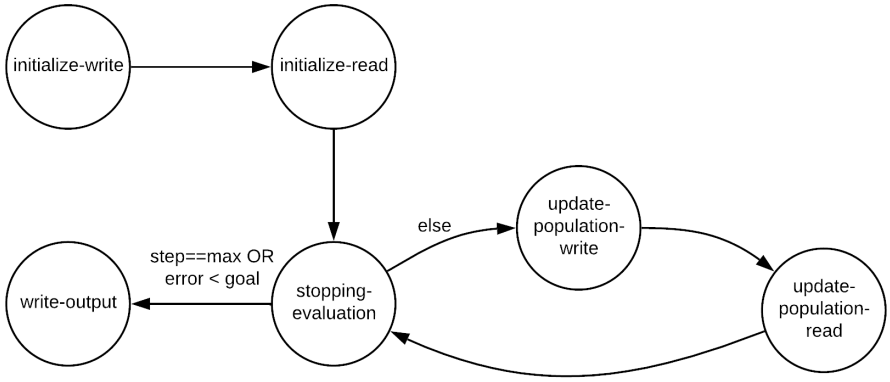

Figure 3.2: Mode transition graph for the PSO Core actor.

In a given execution of NEDECO, the PSO actor starts in the `initialize-write` mode. In this mode, the actor reads PSO parameters, such as the population size P and the number and types of PNDS parameters. The `initialize-write` mode then constructs the initial swarm population by randomly generating P particles. After generating the initial PSO population, the `initialize-write` mode outputs P tokens on the actor's output edge, where each token encapsulates a pointer to a distinct particle in the initial population. Then the actor mode is transitioned to `initialize-read`.

In the `initialize-read` mode, the PSO Core actor consumes a block of P fitness evaluation results, which correspond to computed fitness values for the different configurations of the PNDS, as defined by the different particles in the current PSO population. Each of the fitness values is consumed as a single token from the input edge to `update-population`. These fitness values are used to initialize the best fitness value of each particles, as well as the global best solution. The mode is then transitioned to `stopping-evaluation`.

The `update-population-write` mode updates the particle velocities and

positions according to the best particle fitness and positions (based on the best fitness achieved so far for a given particle) and global best fitness and positions. The PSO Core actor then produces the particles in the current population onto the actor's output edge (in a manner similar to the `initialize-write` mode). The actor transitions to the `update-population-read` mode afterwards.

In the `update-population-read` mode, the PSO Core actor consumes a block of P fitness evaluation results, similar to the `initialize-read` mode. The particles' best positions, best fitness and global best solutions are updated according to these fitness results. After that, the actor transitions to the `stopping-evaluation` mode.

In the `stopping-evaluation` mode, the PSO Core actor first checks whether the stopping criterion for PSO evolution has been reached. Stopping criteria can be configured by the user, and can include factors such as a maximum number of PSO iterations (generated PSO populations), and the maximum acceptable error, which defines a level of accuracy that, when reached, triggers termination of the PSO even if the maximum number iterations has not yet been reached. Details on the stopping criteria used in our experiments are discussed in Section 3.5. If the stopping criterion for the PSO Core actor is reached, then the actor transitions to the `write-output` mode. Otherwise, transitions back to `update-population-write` mode.

The `write-output` mode simply writes the final results of the PSO-based optimization process to a set of output files. The generated output includes the values of the optimized parameter settings — as determined by the best particle within the final PSO population — as well as diagnostic output that can be used to gain insight into how the optimization process evolved through different generations of the population. If the best

particle is not unique — that is, if multiple parameter settings achieve the maximum fitness value — then the the parameter settings associated with all of the tied-for-best particles are written in the output.

The overall time complexity per iteration of NEDECO using PSO optimization is $P \times F$ where F denotes the fitness evaluation time complexity when PNDS runs one time on the training dataset. No complexity expression for F is provided in PNDS, which are NDSEP [75] and CellSort [56] in our experiments, so it's difficult to derive because of third party functions involved.

A pseudocode sketch of NEDECO integrated with the PSO Core actor is given in Algorithm 1.

3.4.8 GA Core Actor

To use a GA as the search strategy instead of a PSO, we simply replace the PSO Core actor in NEDECO with the GA Core actor. The GA Core actor has eight modes, as illustrated in Table 3.2. This table shows the dataflow table of the actor. Here, P denotes the number of individuals in the GA population, and CP is the number of individuals that are selected for crossover in each GA iteration. The GA implementation in NEDECO supports elitism [77] and has an associated non-negative integer parameter EL. If $EL > 0$, then the top EL individuals, based on fitness evaluation, are unconditionally carried over from one generation to the next. The mode transition graph is illustrated in Fig. 3.3. The GA Core actor is designed in a manner similar to the PSO actor; further details on the GA Core actor design are omitted from the chapter for brevity; details can be found in the

Algorithm 1 A pseudocode sketch of NEDECO integrated with the PSO Core actor.

```
/* PSO actor initialize-write mode */
for each particle do
    Initialize particle position and velocity
    Evaluate fitness function
    Send particle parameters to evaluate
end for
PNDS actor fired to evaluate parameters
/* PSO actor initialize-read mode */
for each particle do
    Read fitness values
    Initialize Pbest, Gbest.
end for
while Not reach max iterations or error criteria do /* PSO actor
stopping-evaluation mode */
    /* PSO actor update-population-write mode */
    for each particle do
        Update particle velocity and position
        Send particle parameters to evaluate
    end for
    PNDS actor fired to evaluate parameters
    /* PSO actor update-population-read mode */
    for each particle do
        read fitness values
        Update Pbest
    end for
    update Gbest if necessary
end while
/* PSO actor write-output mode */
write results to output files
```

Table 3.2: Dataflow table for the GA Core actor.

Mode	Input	Output
initialize-write	0	P
initialize-read	-P	0
stopping-evaluation	0	0
update-population-crossover-write	0	CP-EL
update-population-crossover-read	-(CP-EL)	0
update-population-complete-write	0	P-CP
update-population-complete-read	-(P-CP)	0
write-output	0	0

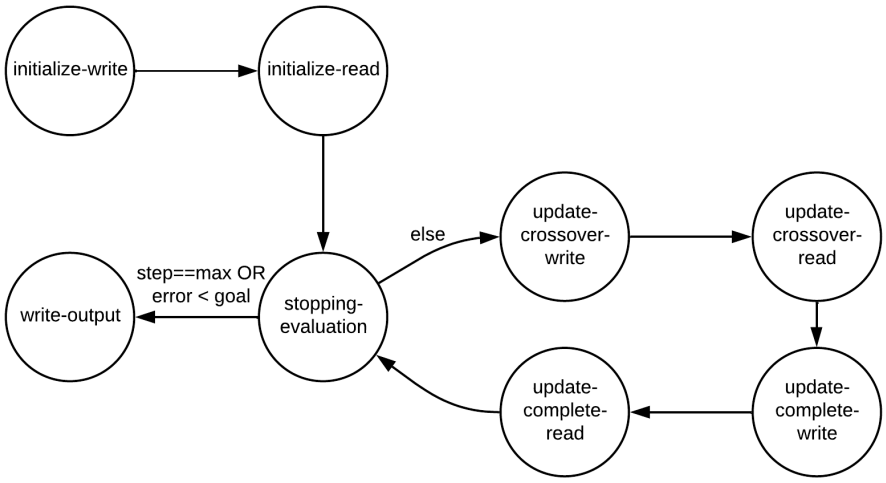

Figure 3.3: Mode transition graph for the GA Core actor.

supplementary materials (Algorithm S2).

The time complexity per iteration of NEDECO using GA optimization is $(P - EL) \times F$, where F denotes the fitness evaluation time complexity when PNDS operates once across the entire training dataset. The setting of EL that we use in our experiments is included in Table 3.3.

Our implementation of the GA Core actor utilizes modified versions of code from [86].

3.4.9 Accelerated Execution of the PNDS Actor

Parallel computing is commonly applied to PSO implementations to help reduce the time required for optimization. In NEDECO, the execution time is vastly dominated by that of the neural decoding system that is being optimized. Recall that the neural decoding system must be executed across the given dataset for every particle in every population generation until the PSO terminates. However, execution of multiple particles in the same population can be conducted in parallel if adequate computing resource are available. The dataflow-based model of NEDECO is utilized for efficient and reliable exploitation of parallelism for particle evaluation when NEDECO is operated on a multicore computing platform.

For efficient multithreaded execution on a multicore platform, each firing of the PNDS actor is encapsulated within a separate thread, and blocks of N_t firings of the PNDS actor are executed concurrently, where N_t is a user-defined parameter number that specifies the number of threads allocated to NEDECO. The default setting of N_t is simply $N_t = N_c$, where N_c is the number of cores or virtual cores (if hyperthreading is present) on the targeted processing platform. In Section 3.5, we include experimental results on the measured execution time improvement of NEDECO that is achieved through the incorporated features for accelerating particle evaluation.

When applying this multithreaded approach to PNDS acceleration, it should be noted that the measured execution time for each particle evaluation is the single-threaded performance of the associated neural decoding system configuration. If the optimized neural decoding system will be deployed as a multithreaded implementation, then the

single-threaded performance evaluated within NEDECO can be viewed as an estimate to help compare the relative processing complexity of alternative PNDS configurations. An interesting direction for extending NEDECO is to incorporate the ability to allocate multiple threads to the evaluation of individual particles, which can provide a better estimate of performance for a neural decoding system that will ultimately be deployed in multithreaded form.

A useful feature of the existing acceleration approach in NEDECO is that it can be applied uniformly to any neural decoding system that NEDECO is applied to — that is, regardless of whether or not a multithreaded implementation is available for the system.

3.5 Experiments

In this section, we demonstrate the utility of NEDECO through experiments using two different calcium-imaging based neural decoding systems. The systems that we apply in our experiments as alternative PNDSs are the Neuron Detection and Signal Extraction Platform (NDSEP) [75], and CellSort [56]. NDSEP is implemented in C++ so its integration with NEDECO, which is also implemented in C++, requires no need for interfacing between different programming languages. On the other hand, CellSort is developed in MATLAB. The integration of CellSort with NEDECO therefore helps to demonstrate the flexibility of NEDECO in being usable with neural decoding systems that are developed in languages other than the native language of NEDECO. To integrate CellSort with NEDECO, features available in MATLAB for interfacing with C++ code are utilized in the CFDF wrapper that is used to interface CellSort to the rest of NEDECO.

When integrating a specific neural decoding system X into NEDECO, we refer to the resulting setup for optimizing the configuration of X as *NEDECO-X*. Thus, our experiments in this section include experiments with NEDECO-NDSEP and NEDECO-CellSort.

By default NEDECO uses PSO as its underlying optimization strategy; however, NEDECO can be readily adapted to work with other population-based optimization strategies. If NEDECO is adapted to use a specific optimization strategy Y and integrated with a specific neural decoding system X, we refer to the resulting configuration as NEDECO-Y-X. If Y is not specified, then it is assumed to be the default strategy PSO. In addition to experiments with NEDECO-NDSEP and NEDECO-CellSort, this section also includes experiments with NEDECO-GA-CellSort an NEDECO-GA-NDSEP, where GA stands for "genetic algorithm".

Experiments are repeated 10 times on each training/testing group with the same configuration. Each of the 10 repetitions for a given group is based on a different seed for the random number generator used by NEDECO, which in turn generally leads to a different initial population for the PSO.

All of the experiments reported on in this section were performed using a 6-core (Intel i7-8700 @3.2GHz) desktop platform that supports up to two simultaneously-executing threads per core. The platform is equipped with 16 GB RAM.

3.5.1 Fitness Function

The fitness function of a PSO measures the quality of a given candidate solution with respect to the optimization objective of the PSO. As described in Section 3.4.6, the fitness

function is implemented in the Fitness Evaluation block of Fig. 3.1, and the function is configured as a linear aggregation of measurements on neuron detection accuracy and execution time.

In our experiments, the accuracy component of the fitness function is configured in terms of the Overall Dice Coefficient (ODC) for neuron detection, which intuitively captures the ratio of the overlap between detected and ground truth neurons to the area of the neuron mask, where the area is measured in terms of number of pixels. By definition, ODC values always lie within the interval $[0, 1]$. For details and motivation on using the Dice coefficient to measure accuracy for image segmentation problems, we refer the reader to [87]. For a given calcium imaging dataset, we typically have a single neuron mask, and the neuron detection process produces a single overall detection result for the dataset. Thus, the ODC provides a measure of neuron detection accuracy for an entire dataset in the context of NEDECO. It is for this reason that we use "overall" in the term ODC. When computing the ODC, if there are more than one detections that match a single neuron in the ground truth, then only the best match having the highest overlapping portion is kept. On the other hand, if a detection matches multiple neurons in ground truth, only the best match is taken into consideration.

The execution time component $\tau(p)$ of the fitness function of each particle p is normalized to be in the interval $[0, 1]$, which is the same interval that defines the range of ODC. In particular, $\tau(p)$ is defined by $\tau(p) = \frac{W(p)}{W_{max}(p)}$, where $W(p)$ is the execution time measured for the most recent evaluation of p (using the PNDS actor), and W_{max} is the maximum execution time observed so far, across all particles in all populations that have been generated and evaluated through the current PSO iteration.

The fitness function is a weighted harmonic mean of the accuracy and execution time assessments, as defined above; for background on fitness functions of this form, we refer the reader to [88]. The fitness function $F(p)$ value for a given particle p in NEDECO can be expressed as:

$$F(p) = 1 - \frac{1}{\frac{a_1}{D(p)} + \frac{a_2}{TimeScale}}; \tag{3.1}$$

$$TimeScale = 1 - \tau(p); \tag{3.2}$$

$$a_1 + a_2 = 1. \tag{3.3}$$

Here, a_1 and a_2 are both positive-valued weights for the component objectives, as discussed in Section 3.4.3. A higher value of a_1 corresponds to a higher relative importance weighting for accuracy compared to execution time. Unless stated otherwise, we use the setting $a_1 = 0.8, a_2 = 0.2$, which is representative of cases in which accuracy is the emphasized objective but some consideration to execution time is also important.

3.5.2 Dataset

In our experiments to evaluate NEDECO-NDSEP and NEDECO-CellSort, we employ the Anterior Lateral Motor Cortex (ALM) dataset [54]. ALM is a real-world two-photon calcium imaging dataset that is acquired from the anterior motor cortex in mice while the mice perform a tactile delay-response task. The dataset includes $11\,189$ frames of calcium imaging data. As a pre-processing step, we apply CaImAn-NoRMCorre [23] for motion correction. The objective of this preprocessing step is to eliminate the influence of motions on the model training and evaluation.

To evaluate NEDECO on the ALM dataset, we employ k-fold cross-validation with $k = 3$. The $11\,189$ calcium imaging frames are randomly split into 3 folds. In each of the three experiment groups, corresponding to our selection of $k = 3$, we select a different fold to be the testing dataset, while the other two datasets are used for training. Here, by "training", we mean executing NEDECO to optimize parameters, while by "testing", we mean evaluating the optimized system configuration that results from the training process. The temporal order of the frames in the training and testing sets is retained to keep temporal features intact.

3.5.3 PSO and GA actor Configuration

Table 3.3 lists PSO configuration settings and GA settings that are used in our experiments with both NDSEP and CellSort. These settings were derived empirically by experimenting with the PSO and GA under different configurations. Background on the first two PSO parameters listed in Table 3.3 is included in Section 3.4. Discussion of the other parameters

Table 3.3: PSO and GA configuration settings.

	Hyper-parameter	Value
PSO	Number of particles	24
	Neighborhood strategy	Ring
	Neighborhood size	10
	c_1	1.496
	c_2	1.496
	w_{min}	0.3
	w_{max}	0.7298
GA	Population size	24
	Selection method	Roulette wheel selection
	Crossover method	One-point crossover
	Mutation method	Single point mutation
	Crossover rate	0.5
	Mutation rate	0.05
	Selective pressure	1.5
	Elite population size	1
	Mating population size	24

in the table is beyond the scope of this chapter; for background on these parameters, we refer the reader to [80, 89]. The stopping criterion (see Section 3.4.7) for both the PSO and GA is set to 50 iterations in all experiments.

3.5.4 NDSEP

We first demonstrate NEDECO by using it to optimize parameters of NDSEP, which is designed to be a real-time neural decoding system for calcium imaging. NDSEP takes as input a video stream captured by a miniature calcium imaging device as input, and produces as output a set of detected neuron masks as well as extracted neuron signals corresponding to the masks. Details of NDSEP for neuron detection are in [75].

Table 3.4 lists parameters in NDSEP that need to be configured. For details on the meanings of these parameters, we refer the reader to [75]. The range of each parameter is

Table 3.4: Parameters in NDSEP.

Parameter	Type	Minimum value	Maximum value
`Threshold step`	continuous	10.0	100.0
`Minimum circularity`	continuous	0.0	1.0
`Minimum convexity`	continuous	0.0	1.0
`Minimum inertia ratio`	continuous	0.0	1.0
`Gaussian blur size`	integer	1	60
`Gaussian standard`	continuous	0.0	1.0
`Median blur size`	integer	1	60

given in Table 3.4 along with an indication of whether the parameter is continuous-valued or integer-valued. The admissible ranges of the parameters are imposed based on limits on the parameter values that defined by NDSEP. However, even with these limitations imposed on the parameter value ranges, the overall search space of parameter combinations is extremely large and prohibitively expensive for exhaustive evaluation (theoretically, the search space is infinite since some parameters are continuous-valued).

Table 3.5 shows training and testing results for NEDECO-PSO-NDSEP and NEDECO-GA-NDSEP using the aforementioned k-fold cross-validation approach, where $k = 3$. Experiments are repeated 10 times on each fold. The average ODC and frame rate values resulting from the training and testing datasets are shown in Table 3.5, where the average and standard deviation are taken over all 30 experimental runs. Here, the frame rate ρ, which is the average time to execute neuron detection on a single frame, is calculated as: $\rho = \frac{n_f}{t_{tot}}$, where n_f is the number of image frames in the given dataset, and t_{tot} is the total processing time for the dataset.

Table 3.5 also shows the average and standard deviation values for recall and precision results among the 30 repeated experiments. Here, precision and recall are calculated for the neuron detection results. A threshold T_{dice} is used to distinguish between true

Table 3.5: Training and testing results for NEDECO-PSO-NDSEP and NEDECO-GA-NDSEP.

Method	training				testing			
	dice coefficient	recall	precision	frame rate (fps)	dice coefficient	recall	precision	frame rate (fps)
PSO	0.5968 (±0.0069)	0.6853 (±0.0185)	0.9380 (±0.0429)	207.9 (±23.5000)	0.5549 (±0.0120)	0.6292 (±0.0298)	0.9167 (±0.0315)	206.9 (±23.8284)
GA	0.5781 (±0.0157)	0.6908 (±0.0798)	0.8894 (±0.0188)	186.4498 (±44.2536)	0.5310 (±0.0105)	0.6329 (±0.0443)	0.8904 (±0.0577)	186.8505 (±23.7246)

and false positives in neuron detection results — if the Dice coefficient $\lambda(\delta)$ for a given detection δ is greater than or equal to T_{dice}, then the detection is categorized as a true positive (TP) result; on the other hand, if $\lambda(\delta) < T_{dice}$, then the detection is categorized as a false positive (FP) result. For the precision and recall results in Table 3.5, we use $T_{dice} = 0.5$.

Fig. 3.4 shows the minimum, maximum, quartiles, and average of the ODC over all 30 experimental runs (across all 3 groups) for the training and testing datasets. The right side of Fig. 3.4 also shows the ODC achieved by the *base NDSEP configuration*, which is the original hand-optimized configuration of NDSEP [75].

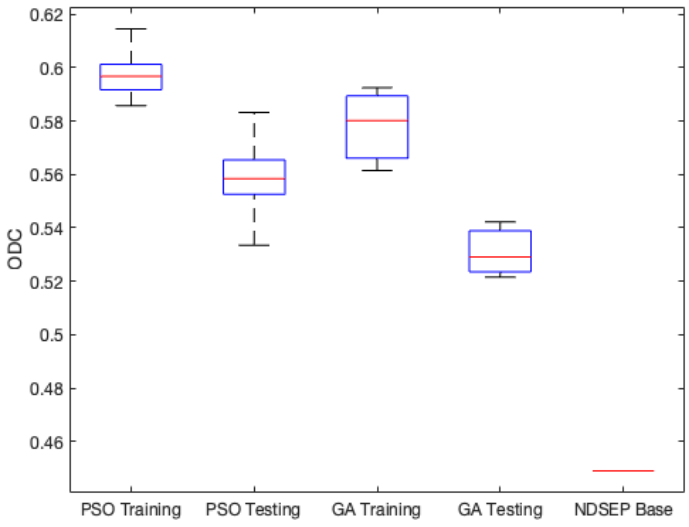

Figure 3.4: Minimum, maximum, quartiles, and average of the ODC over all $k = 3$ experimental groups for NEDECO-NDSEP.

Compared with the hand-optimized Base NDSEP configuration, NEDECO-PSO-NDSEP and NEDECO-GA-NDSEP demonstrate major advantages. In Fig. 3.4, we see that both NEDECO-PSO-NDSEP and NEDECO-GA-NDSEP consistently achieve significantly higher ODC values compared to Base NDSEP. The recall and precision levels achieved by Base NDSEP are 0.725 and 0.495, respectively. From Table 3.5, we see that the recall levels produced by NEDECO-NDSEP are marginally worse compared to Base NDSEP, however, the precision produced by NEDECO-NDSEP is much better. The slightly improved recall of NDSEP results because missed detections are intuitively easier to notice, whereas multiple detections on the same ground truth neuron are more difficult avoid, especially when the detections correspond to very small regions of interest. Moreover, the frame rates shown in Table 3.5 are also significantly higher compared to the frame rate provided by Base NDSEP, which is 149.9 fps. This demonstrates the potential for increased execution time performance provided by NEDECO.

Fig. 3.5 illustrates measured results from the application of multithreading to accelerate firings of the PNDS actor in NEDECO-NDSEP. Recall that a firing of the PNDS actor corresponds to execution of the neural decoding system (NDSEP in this case) on the entire given dataset. The horizontal axis in the figure corresponds to successive PSO iterations; recall that in each PSO iteration, the PNDS actor fires once to evaluate all of the particles in the current population. The vertical axis corresponds to the measured execution time for firings of the PNDS actor. The different curves in the figure correspond to different numbers of threads (i.e., different values of N_t). For each setting of N_t shown in Fig. 3.5, we executed NEDECO-NDSEP 10 times, and averaged the results at each PSO iteration number to derive curve associated with N_t.

From the results shown in Fig. 3.5 , we see significant improvements in execution time with increasing use of threads with the improvements saturating for each value of N_t for later PSO iterations. The only exception in terms of increasing values of N_t is for $N_t = 12$, which performs marginally worse than $N_t = 8$. It is anticipated that this trend for higher values of N_t is due to factors such as excessive contention for resources among the allocated threads or overheating in the processor.

Fig. 3.6 shows the measured speedups associated with the data illustrated in Fig. 3.5. Here, the speedup associated with $N_t = p$ is calculated as $\frac{T_1}{T_p}$, where T_1 and T_p are corresponding execution time measurements associated with single-threaded (sequential) execution and execution with $N_t = p$. T_1 and T_p are calculated by averaging the PNDS actor execution time over 50 iterations, as shown in Fig. 3.5.

These execution time improvements reflect not only faster operation of the NEDECO optimization process, but also improvements in the average frame rate associated with PSO particles (candidate NDSEP configurations).

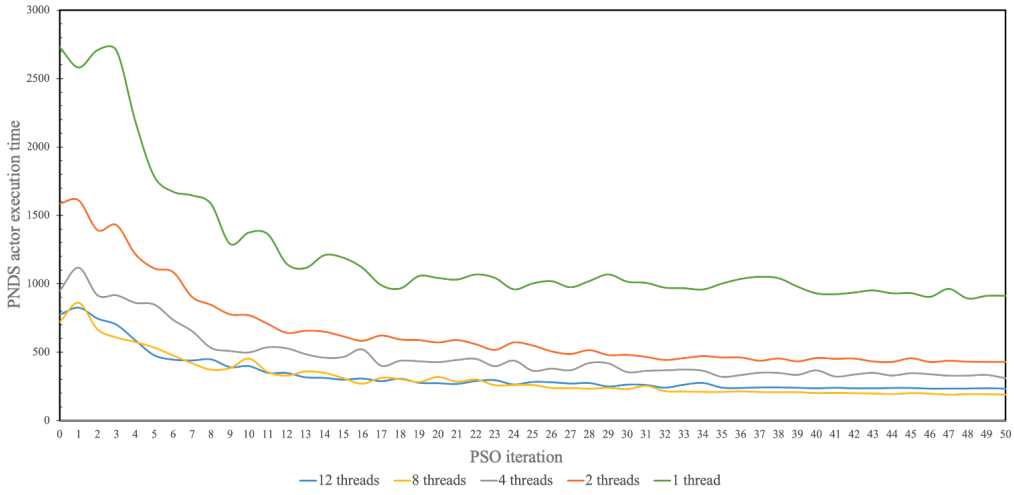

Figure 3.5: Measured results produced from use of multithreading to accelerate operation of NEDECO-NDSEP.

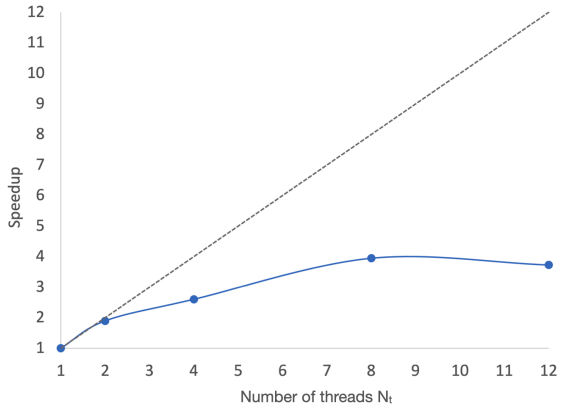

Figure 3.6: Speedups associated with the data illustrated in Fig. 3.5.

3.5.5 CellSort

CellSort is a widely used Matlab-based tool for neuron instance segmentation and signal analysis [56]. The CellSort algorithm for deriving neuron segmentation masks involves three main steps — Principle Component Analysis (PCA), Independent Component Analysis (ICA), and Image Segmentation. Like NDSEP, CellSort involves several parameters that need to be set by the user. These parameters have a major impact on the segmentation results.

Moreover, CellSort runs much more slowly compared to NDSEP, which leads to a corresponding slowdown for the PSO-based optimization process of NEDECO-CellSort. However, from our experiments, we found that a large portion of the time in CellSort is spent in the PCA step, which has no associated parameters that need to be configured by the user. Thus, the PCA step can be executed as a pre-processing step to the PSO optimization process, and there is no need to include PCA computation as part of the PNDS actor.

Table 3.6 summarizes the parameters involved in configuring CellSort. These

parameters are jointly optimized by NEDECO-PSO-CellSort and NEDECO-GA-CellSort. The parameters `PCl`, `PCf` and `mu` are associated with the ICA step of CellSort. These parameters respectively specify the first principal component to be kept for dimension reduction, the last principal component to be kept, and the weight to use for temporal information in the spatial-temporal ICA process. The other four parameters in Table 3.6 are associated with the image segmentation step. The `smwidth` parameter gives the standard deviation for the Gaussian smoothing kernel; the `areal` and `areah` parameters respectively give the minimum and maximum areas (in pixels) for segments that are to be retained; and the `thresh` parameter gives the threshold for spatial filters.

Table 3.6: Parameters in CellSort.

Parameter	Type	Minimum value	Maximum value
`PCl`	integer	1	60
`PCf`	integer	61	150
`mu`	continuous	0.0	1.0
`smwidth`	continuous	0.0	10.0
`areal`	integer	50	500
`areah`	integer	501	2000
`thresh`	continuous	1.5	10.0

We used the ALM dataset to demonstrate and experiment with NEDECO-CellSort. For NEDECO-CellSort, we used the same 3-fold cross-validation approach that is described in Section 3.5.2, and we averaged across 10 trials for each experiment setting. We ran experiments using NEDECO-CellSort that parallel the experiments for NEDECO-NDSEP reported in Table 3.5 and Fig. 3.4. The corresponding results for NEDECO-CellSort are shown in Table 3.7 and Fig. 3.7, respectively. To measure precision and recall, we used the same threshold setting, $T_{dice} = 0.5$, that we used for NEDECO-NDSEP (see Section 3.5.4). In Table 3.6, we list the execution time (over the whole dataset) instead of the frame rate

Table 3.7: Training and testing results for NEDECO-PSO-CellSort and NEDECO-GA-CellSort.

Method	training				testing			
	dice coefficient	recall	precision	time(sec)	dice coefficient	recall	precision	time(sec)
PSO	0.5169 (±0.0146)	0.5352 (±0.0365)	0.8112 (±0.0683)	20.4834 (±10.0690)	0.4138 (±0.0253)	0.3992 (±0.0410)	0.7425 (±0.0739)	21.38 (±13.5110)
GA	0.4833 (±0.0074)	0.4824 (±0.0527)	0.7784 (±0.1094)	40.9305 (±10.1727)	0.4106 (±0.0384)	0.3623 (±0.0586)	0.8259 (±0.1011)	25.9015 (±7.7650)

because CellSort is not designed to operate in a real-time fashion, with frame-by-frame input/output.

The Base CellSort configuration (see Fig. 3.7) used in our experiments was derived through a labor-intensive hand-optimization process using the ALM dataset. This optimization process took approximately three weeks. The parameter settings resulting from this hand-optimization process are discussed in [75]. NEDECO-CellSort eliminates the need for such labor-intensive hand-optimization, and at the same time, produces significantly more accurate and efficient configurations of CellSort.

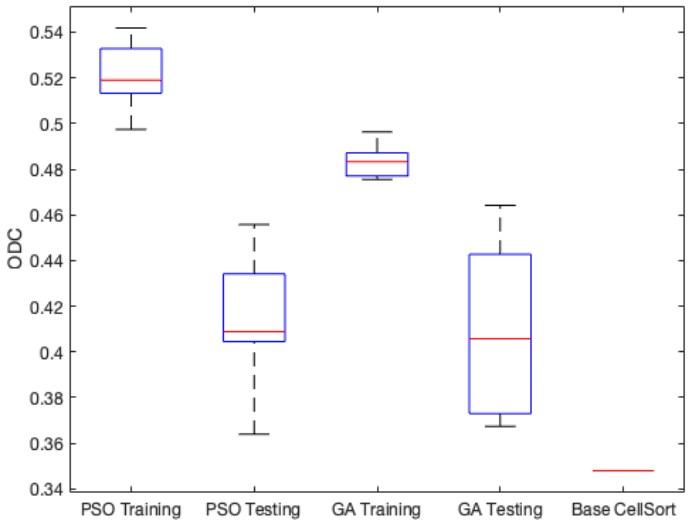

Figure 3.7: Minimum, maximum, quartiles and average of the ODC over all $k = 3$ experimental groups for NEDECO-CellSort.

From Table 3.7 and Fig. 3.7, we see that trends for accuracy metrics (ODC, recall and

precision) and execution time are similar for NEDECO-X-CellSort as those for NEDECO-X-NDSEP presented in Section 3.5.4. The recall and precision levels achieved by Base CellSort are 0.7101 and 0.3043, respectively. From Table 3.5, we see that the recall levels produced by NEDECO-X-CellSort are worse compared to Base CellSort, however, the precision produced by NEDECO-X-CellSort is better, and the degree of improvement in precision significantly exceeds the degree of reduction in recall. The F1 score, which can be viewed as an aggregation of the precision and recall metrics, for the NEDECO-PSO-CellSort and NEDECO-GA-CellSort results are 0.5132 and 0.5037, respectively, which represent improvements compared to the F1 score of 0.4260 for Base CellSort.

We measured the execution time of Base CellSort to be be 52.77 seconds (again averaged over 10 trials). This is significantly slower than the training- and testing-time averages of 20.48 seconds and 21.38 seconds for NEDECO-PSO-CellSort, and 40.93 seconds and 25.9015 seconds for NEDECO-GA-CellSort, as shown in Table 3.7.

In summary, our experiments demonstrate that NEDECO greatly reduces the effort involved in optimizing the parameters of CellSort while producing configurations of CellSort (for the different cross-validation groups) that are significantly more accurate in terms of F1 score and ODC, and also significantly more efficient compared to the baseline, hand-optimized version of CellSort.

3.5.6 Multi Objective Optimization

In this section, to evaluate the linear aggregation multiobjective approach described in Section 3.4.3, we perform a comparison with NSGA-II. NSGA-II is a widely used multi-

objective genetic algorithm. For details on NSGA-II, we refer readers to [90]. In our experiments, NEDECO-PSO-CellSort is compared with NSGA-II-CellSort, where we optimize CellSort using the MATLAB-based NSGA-II Package [91]. Unlike our previous experiments that fix the values of a_1 and a_2 (see Section 3.5.1), NEDECO-PSO-CellSort is trained with different values of (a_1, a_2) — in particular, a_1 is increased from 0.1 to 0.9, and a_2 is correspondingly decreased from 0.9 to 0.1. The increase/decrease is performed with a step size of 0.1. NEDECO-PSO-CellSort is trained 9 times with the different settings for (a_1, a_2) to derive 9 Pareto fronts.

In this experiment, we applied the same 3-fold cross-validation approach that is described in Section 3.5.2. We repeated the steps above to obtain 9 Pareto fronts in total, 3 on each fold. NSGA-II-CellSort was trained for 50 iterations as well with the same population size of 24. Training was repeated 9 times with different seeds (for random number generation), 3 on each fold.

Results on the test dataset are reported in Fig. 3.8. Here, we randomly pick two Pareto fronts from NEDECO-PSO-CellSort and three from NSGA-II-CellSort. From the results, we see that NEDECO-PSO-CellSort outperforms NSGA-II-CellSort with better Pareto fronts. We use the AUC (area under the curve) [92] metric to evaluate the Pareto fronts. The average AUC of NEDECO-PSO-CellSort is 0.3971 while the average AUC of NSGA-II-CellSort is 0.2121. The averaging is performed across all generated Pareto fronts; in contrast, only a proper subset of Pareto fronts is shown in Fig. 3.8 to avoid excessive clutter in the illustration. In summary, the linear aggregation multiobjective approach of NEDECO significantly outperforms NSGA-II in our experiments.

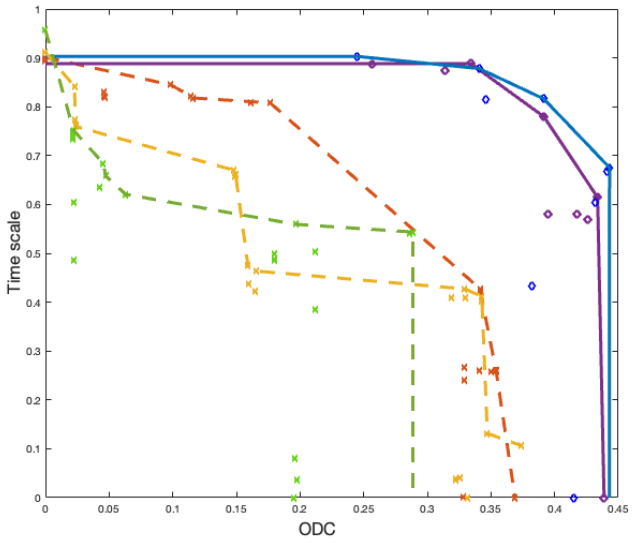

Figure 3.8: Pareto fronts derived from NEDECO-PSO-CellSort (solid lines) vs. NSGA-II-CellSort (dotted lines). Dots having the same color represent solutions derived from the same experiment. Pareto fronts are generated based on sets of dots having the same color.

3.6 Discussion

In this chapter, we have motivated the problem of parameter optimization for neural decoding systems, and we have presented a novel framework called the NEDECO package. NEDECO automatically performs parameter optimization to significantly improve the effectiveness of neural decoding systems. NEDECO applies methods from particle swarm optimization genetic algorithm, and dataflow modeling to develop a modular framework into which a wide variety of neural decoding systems can be integrated to help derive more effective configurations of the systems. NEDECO is also innovative in its consideration of multidimensional design evaluation spaces rather than being focused only on accuracy. In our current prototype of NEDECO, we have focused on the objectives of neuron segmentation accuracy and execution-time efficiency. This particular combination of design evaluation metrics is motivated by the important applications of neural decoding in

real-time contexts, especially in the context of neuromodulation systems.

We have demonstrated the adaptability of NEDECO to different optimization algorithms and systems by presenting case studies of its integration with two state-of-the-art neural decoding systems — NDSEP and CellSort. The experimental results indicate significant performance improvements for both NDSEP and CellSort when their system parameters are optimized using NEDECO. Moreover, it is demonstrated that the dataflow-based design of NEDECO leads to structured and efficient utilization of multicore computing technology to accelerate the optimization process of NEDECO.

Compared to the accuracy results demonstrated in the original work on NDSEP [75], the Base NDSEP and Base CellSort algorithms, which have manually-tuned parameters, demonstrate worse results in the experiments reported on in this chapter. This is because of the more strict evaluation of neuron segmentation performance that is employed in this chapter. For example, in [75] , when comparing with the ground truth mask, if the detected region of interest overlaps with any of the neurons in the mask, a match is counted, regardless of whether or not the ground truth neuron already has a match. However, in this chapter, true positives are counted based on a threshold on the percentage of overlapping pixels associated with the match. The multiple-to-one match situation is excluded as well. By making the evaluation more strict in this way, we are able to present better insight into the accuracy performance of the investigated neural decoding systems.

From Section 3.5, we see similar performance trends between NEDECO-PSO and NEDECO-GA when CellSort and NDSEP are applied, respectively, as the PNDS. However, we cannot conclude from our experiments that in general NEDECO-PSO and NEDECO-GA have similar effect to optimize neural decoding systems (regardless of the PNDS used).

Users benefit from the flexibility to try both search strategies — PSO and GA — for a given PNDS and operational scenario and to utilize the one that performs best. A useful direction for future work is to incorporate additional search strategies to further enhance this form of flexibility.

Due to the lack of suitable calcium imaging datasets for neuron detection, the experiments in this chapter are presented only on the ALM dataset. For most existing calcium imaging datasets, the absence of ground truth results prevents the evaluation of calcium imaging processing platforms. Additionally the ground truth in the ALM dataset is a mask representing the whole dataset, so when the dataset is split into training and testing subsets, some of the neuron activity can be lost, which introduces challenges to model optimization, and can make testing performance worse than expected in terms of accuracy. The smaller the subset is, the more activity is in general lost. In terms of the k-fold cross validation, with a greater k value, the test subset will be too small to keep from losing too many neuron activities. This is why we use a small k value of 3 in Section 3.5. To address this issue, an important direction for future work is development of and experimentation with datasets that have ground truth information available at the frame level [7].

We are among the first to apply automated parameter optimization to calcium-imaging-based neural decoding and provide a dataflow-based implementation, which helps to enhance the scalability, modularity, and efficiency of the framework. Theoretically, the proposed framework can handle any parameter optimization method. The current implementation includes two parameter optimization approaches, GA and PSO. We chose these approaches because they are representative of widely-used search techniques that are relevant to complex and irregular search spaces. The focus of this chapter is to

demonstrate the utility of automated parameter optimization for calcium imaging based neural decoding; the focus of the chapter is not to advocate for the superiority of any specific search technique.

Chapter 4

ShaderNN: A Lightweight and Efficient Inference Engine for Realtime Image Processing Applications on Mobile GPU

4.1 Chapter Overview

Inference using deep neural networks on mobile devices has been a very active area of research in recent years. The design of a deep learning inference framework targeted for mobile devices needs to consider various factors, such as the limited computational capacity of the devices, low power budget, varied memory access methods, and I/O bus bandwidth governed by the underlying processor's architecture. Furthermore, integrating an inference framework with time-sensitive applications — such as games and video-based software to perform tasks like ray tracing denoising and video processing — introduces the need to minimize data movement between processors and increase data locality in the target processor. In this chapter, we propose Shader Neural Network (ShaderNN), an OpenGL-based, fast, and power-efficient inference framework designed for mobile devices to address these challenges. Our contributions include the following: (1) the texture-based input/output provides an efficient, zero-copy integration with real-time graphics pipelines or image processing applications, thereby saving expensive data transfers between CPU and GPU, which are unavoidable in most existing inference engines; (2) we are the first to leverage fragment shaders based on the OpenGL backend in neural network

inference operators, which has an advantage in deploying parametrically small neural network models; (3) a hybrid implementation of the compute shader and fragment shader is proposed that enables layer-level shader selection to boost performance; and (4) we utilize OpenGL features — such as normalization, interpolation and texture padding — to improve performance. Experiments illustrate the favorable performance of ShaderNN over other popular on-device deep learning frameworks such as TensorFlow-Lite on the latest mobile devices powered by Qualcomm and MediaTek chips. A case study further demonstrates the usability and integration of the ShaderNN framework with a media processing Android application seamlessly. ShaderNN is available open source at Github (https://github.com/inferenceengine/shadernn).

4.2 Introduction

As a research hot spot, deep learning has made large leaps in the last decades. Nowadays, deep learning technology lies behind numerous everyday products and edge devices such as smart phones, TV remotes, and self-driving vehicles. On-device inference of neural networks brings many benefits, including low-latency, server-dependency removal, low communication overhead, and low system cost. With an efficient on-device inference engine, engineers are able to materialize a variety of real-time applications, like real-time denoising and image super-resolution, in a privacy-conscious way on mobile devices.

Energy consumption is a major challenge in this context. Usually, neural network models are computationally intensive. Mobile devices have very limited battery capacities and power supply capabilities. Increased energy consumption causes shorter battery life or

even thermal throttling. Energy and capability constraints must be taken into consideration. Some neural network models such as MobileNet [93] and SqueezeNet [94] have energy-related advantages on power constrained devices by reducing the number of parameters and compromising accuracy. Specific hardware chips such as FPGAs, and ASICs are designed to reduce power consumption when deploying deep learning networks [95]. However, the network and device generality of these approaches remains a challenge. A versatile and power-efficient inference framework supporting popular models and mobile devices is greatly needed.

On smartphones, most of the applications are actually graphics and image related, and the associated data processing and visualization heavily rely on GPUs. To improve graphics display capabilities and maximize the use of computing resources, a common strategy is to integrate neural network models into graphics applications, such as graphics rendering pipelines and game AI applications. However, existing frameworks don't operate directly on GPU textures. Few inference frameworks target graphics and image post-processing tasks on mobile devices. The input and output are also not GPU textures. I/O overhead introduced by data transfer between GPU textures and input/output buffers cannot be avoided, which compromises the real-time performance and energy efficiency of on-device graphic applications.

A certain speed for inference is required by most on-device applications. How to fully utilize the on-device computation resources, especially the GPU resources — which have significantly more compute power than CPU resources — is an important open question for researchers and software developers.

In this chapter, we present a new mobile inference framework named Shader Neural

Network (ShaderNN). The advantages of ShaderNN include:

1. ShaderNN is the first open-source neural network inference engine that exploits both graphics-centric abstractions (fragment shaders) and compute-centric abstractions (compute shaders). The integration of the fragment shader and compute shader makes significantly enhanced use of the parallel computing advantages of GPU shaders, and can provide optimized performance for a wide variety of neural networks.

2. ShaderNN is streamlined for real-time graphics and image post-processing. Based on Native OpenGL ES [96], ShaderNN directly operates on GPU texture data and image applications. It can be easily integrated with graphics rendering pipelines, saving I/O time on data transfer, which is critical for real-time applications on mobile platforms.

3. ShaderNN enables a hybrid implementation of compute and fragment shaders, with the ability to select layer-level shaders for performance optimization.

4. Performance is optimized using native OpenGL features, such as normalization, interpolation and texture padding.

5. We evaluate fragment shaders and compute shaders in terms of inference latency and energy efficiency on different types of neural network models, and we make suggestions on shader selection based on neural network features.

4.3 Background and Related Work

A number of research efforts have been made to accelerate on-device neural network inference.

TensorFlow Lite (TF-Lite) [97] is proposed by Google Research, providing a solution

to run inference of neural networks on various types of mobile devices. TF-Lite leverages on-device GPUs, and achieves a 2X–9X speed up compared to CPU-based inference. Most of the TensorFlow models are supported. Converting a TensorFlow model to TF-Lite is relatively simple. Conversion from other model formats, such as ONNX [98] and PyTorch [99], requires first converting to TensorFlow, and then converting the resulting TensorFlow model to TF-Lite. Model format conversion is prone to precision degradation, and can also result in parameter mismatching if loss and optimizer information is also involved.

PyTorch Mobile [100] provides an end-to-end workflow that helps to deploy trained models on mobile devices while the development process remains entirely within the PyTorch ecosystem. PyTorch Mobile does not require any model format conversion. Once a model is trained using PyTorch, it can be migrated to mobile devices and consumed by the PyTorch Mobile engine. A variety of devices are supported including IOS and Android. However, at the time of this writing, PyTorch Mobile is still in an experimental phase.

CoreML [101] is an on-device inference framework proposed by Apple, and targeted on iOS applications. Building on top of primitives such as Metal Performance Shaders and Basic Neural Network Subroutines (BNNS), the performance is optimized by utilization of CPU, GPU, and Neural Engine devices. However, the limited generality makes it unavailable on most Android mobile devices.

More lightweight inference frameworks have been proposed in recent years. Alibaba developed Mobile Neural Network (MNN) in 2020 [102], which is universal and optimized for mobile applications. Other works — such as Tencent NCNN [103], Tencent TNN [104], Huawei Bolt [105], Xiaomi MACE [106], and Baidu Anakin [107] — are also lightweight

with few third-party dependencies.

In this chapter, the inference engine ShaderNN that we propose exploits both compute shaders and fragment shaders. Compute shaders that enable general purpose computations are introduced by some of the inference engines mentioned above, such as TensorFlow Lite. But there are few inference engines that exploit fragment shaders in neural network computation. Early efforts have been made to map low-level neural network computations to fragment shaders. Cronin [108] demonstrated the possibility to implement matrix multiplication, which is the key computation associated with deep neural networks, through two programmable stages, the fragment shader and vertex shader, from the graphic pipeline. Lin [109] accelerated the web-based inference of their real-time video segmentation model with the help of the fragment shader using WebGL. This effort was motivated from the limitation that the compute shader is not supported by WebGL. However, there is no open-source inference engine that allows users to make operator/layer-level choice of fragment and compute shaders for arbitrary neural networks. The performance of fragment shaders versus compute shaders in this context remains unclear.

In this chapter, since our target involves graphics and image processing applications running on mobile GPUs, we only discuss the GPU backends of inference engines. On iPhone, the only option is Metal, which provides similar APIs as OpenGL compute shader. For most of Android phones, the options to run on GPUs are OpenGL, Vulkan [110] and OpenCL [111]. OpenGL is widely supported on almost all of the Android systems equipped with a GPU from any vendor. Vulkan is a low-overhead, cross-platform API and open standard for 3D graphics and computing [110]. The lower-level API of Vulkan involves more work for application developers. OpenCL gets support on some of the

latest mobile GPUs, and has not been widely supported yet. Among these GPU backends, OpenGL is the most ubiquitous in mobile devices.

The TF-Lite GPU mode supports both OpenGL compute shader and OpenCL, which is the most popular inference engine for deployment on mobile phones. MNN GPU mode supports all options — OpenGL compute shader, OpenCL and Vulkan. Its OpenGL implementation is not fully optimized and the supported operators are limited. PyTorch Mobile and NCNN only support Vulkan as GPU backend. Bolt, MACE and TNN only work with OpenCL on GPU backend. There is no mobile GPU backend support in Anakin.

Although there are several options to run deep learning models on mobile GPUs, there will be some issues while integrating graphics and image processing applications with these frameworks. The first one is the performance issue. The same model may have different latencies on different inference engines due to code optimizations that are involved. Another issue involves compatibility. Some models may not be able to run with GPU backend or may run slowly due to a fall-back to CPU mode. The last but not the least issue is that of data movement. All of the inference engines we have discussed provide core data structures — typically either called Tensor or Mat — but these data structures cannot handle OpenGL textures directly. This is a significant limitation because OpenGL textures are widely used in graphics and image processing applications.

As illustrated in the first part of Figure 4.3, although the input data already persists in the OpenGL texture, the texture data needs to be moved to CPU memory, and converted into input Tensor or Mat form for inference. Once the data movement and conversion is completed, the result needs to be retrieved and moved back to the OpenGL texture for the next stage of processing in the enclosing graphics or image pipeline. These overheads of

data movement and conversion are significant.

4.4 Methodology

Overall, our shader neural framework includes four basic steps — model conversion, model loading, computational graph generation, and operator execution. Figure 4.1 illustrates the entire structure of ShaderNN. In the model preparation step, static optimization is performed. The model converter reads deep neural network models either in TensorFlow [112], PyTorch [99], or ONNX [98] format, and then performs input operator checking, decouples weights and model structure, and fuses layers. Then the model architecture and weights are loaded to the inference engine. In the inference engine, a computational graph is generated by topological sorting. Operator execution is our core step. When executing operators in computation graphs, fragment shaders and compute shaders based on OpenGL backend are involved in ShaderNN operators, with compile-time and run-time optimizations. In terms of run-time optimizations, we performed convolution optimization; texture reuse; CPU and GPU memory reuse and data structure layout; and cache vectorization.

Typical applications of ShaderNN are illustrated in Figure 4.2.

In the remainder of this section, we present further details on the design of ShaderNN, and optimization techniques employed in ShaderNN.

4.4.1 Data Structure

For applications that integrate both graphics rendering and ML, unified buffer management can reduce data movement. In ShaderNN, layer-level input and output tensors are unified.

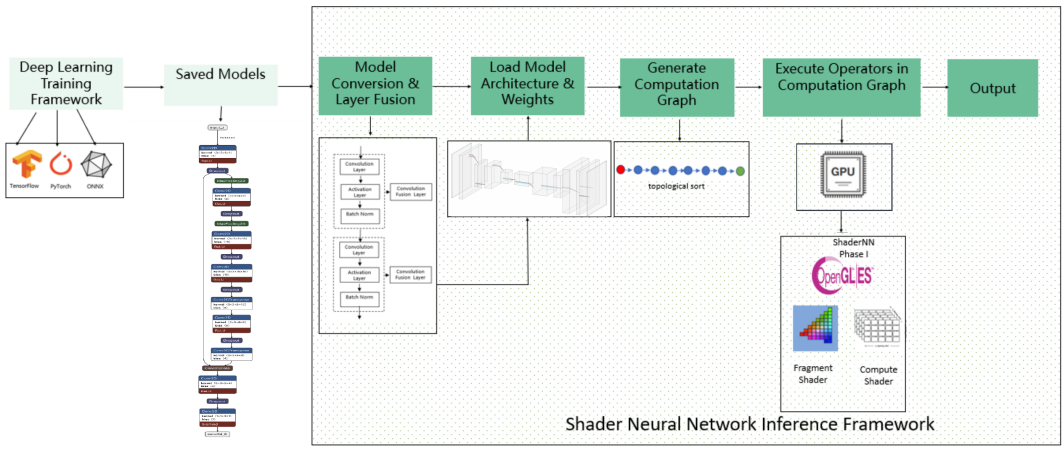

Figure 4.1: Structure of the proposed mobile inference framework, Shader Neural Network (ShaderNN).

Figure 4.3 illustrates the workflow of ShaderNN (A) versus pipelines of other inference engines, and (B) when integrating with graphics pipelines or applications. Compared with ShaderNN, the workflows of other inference engines require two more blocks to be copied between GPU texture and CPU memory or other GPU buffers. This need for copying arises because most graphics pipelines or applications operate on GPU textures, while network inference engines operate on their core data structures, such as Tensor or Mat, which cannot be handled by OpenGL textures directly. As an important feature, ShaderNN operates directly on GPU textures, providing streamlined integration with graphics pipelines, and saving I/O time. Unified data management in ShaderNN also enables integrated usage of fragment and compute shaders in a given neural network.

Figure 4.3 illustrates the integration of deep learning inference with a graphics or image processing pipeline. When we integrate with other inference engines such as TF-Lite or NCNN, GPU textures (as described in Section 4.3) have to be transferred

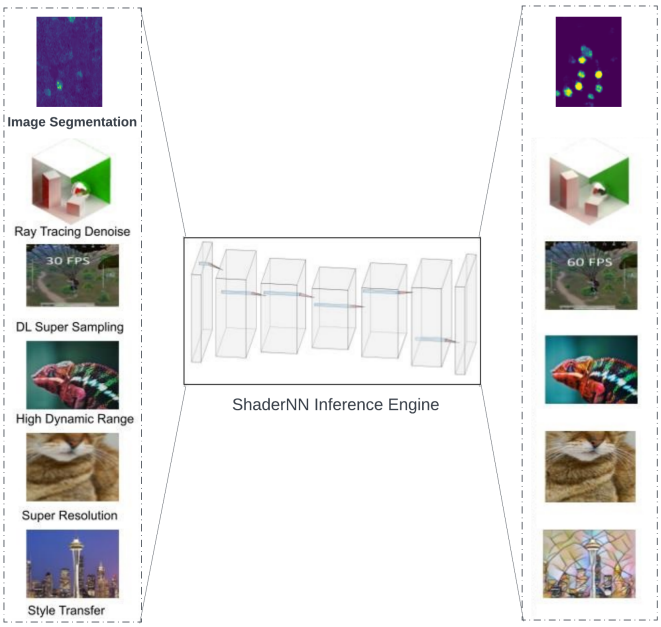

Figure 4.2: Typical use cases for ShaderNN.

to CPU memory as the input of the deep learning inference engine, and conversely, the output of the inference engine must be transferred from CPU memory back to GPU textures. ShaderNN directly operates on the texture data of graphics and image processing applications, and the I/O data transfer steps described above are avoided, which provides significant enhancement to run-time efficiency.

4.4.2 Layer Fusion

Layer fusion is an effective optimization on the computation graph. Layer fusion can significantly reduce the number of layers, and thus reduce the computation cost of inference. Theoretically, if one layer is an element-wise operation, or a Map function, it can be merged with its previous layer. A common example is the activation layer, which normally follows the convolution layer or batch normalization layer. For a batch normalization layer, it can also be merged with its previous layer, or it can be removed directly with batch

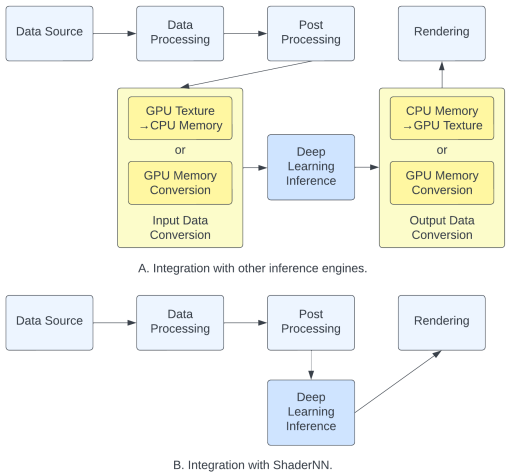

Figure 4.3: Graphics and image processing pipeline with deep learning model.

normalization folding.

Another case is the padding layer. If the padding mode is constant, repeat, or symmetric, a padding layer can be merged with its following layer. In the following layer, the OpenGL Sampler object binding to the input texture will provide the padding feature, and thus will reduce the overhead of boundary checking.

In ShaderNN, layer fusion is performed when building the static computation graph. Layers such as activation and batch normalization layers are fused to accelerate operator inference by reducing the overhead caused by calling operators. We also provide the independent operators of the fused layers to adapt to more customized network designs.

4.4.3 Fragment Shader

In the OpenGL graphics pipeline, the fragment shader (also known as the pixel shader) stage is the one that processes individual fragments to determine the color value of the corresponding pixels present in that fragment [96]. The output of a fragment shader is a depth value and color value, usually written to render target or a buffer in a framebuffer [96]. The fragment shader stage works in conjunction with and is dependent on the vertex shader stage, which is responsible for providing the vertex coordinates of the spatial (quad) area of interest for the fragment shader to process. Fragment shaders have found their application in tasks related to image processing and are a natural choice when the target underlying or accompanying the application pipeline is a graphics pipeline with some render target involved.

Unlike compute shaders (discussed in Section 4.4.4), which are more flexible and can

be used for a wider range of tasks, fragment shaders are optimized for rendering graphics to the screen. This type of rendering typically involves processing large numbers of pixels in parallel. Fragment shaders have shown computational and performance superiority compared to compute shaders due to better data locality and more efficient memory access patterns. For example, in context-sensitive algorithms, where each pixel value is calculated and dependent on its neighboring pixel values [113], fragment shaders operate on pixels that are adjacent to each other in screen space, which allows them to take advantage of data locality and potentially improve performance.

Additionally, fragment shaders access texture data, which is stored in a special type of memory called a texture cache. This can be faster than accessing data stored in global memory, especially when the texture data is accessed repeatedly.

In convolutional neural networks (CNNs), the operation having the highest occurrence and one of the highest levels of complexity is convolution. Convolution in two dimensions or on images is a context-sensitive algorithm, where the output value is calculated as the sum of the product of a kernel and sampled input pixels. Various other operators used in CNNs, such as Conv2DTranspose, Deconvolution, Pooling, and Upsampling, are also context-sensitive. This makes the fragment shader a viable option to be used for inference engine design to deploy CNNs.

Figure 4.4 illustrates a convolutional layer in a ShaderNN pipeline. In order to translate the operations involved in a CNN to the ShaderNN inference engine's graphics pipeline, a layer's input is stored into a texture and is processed via a vertex and fragment shader to ultimately produce an output, which is then written to a render target texture. Based on the number of channels of the input and output, it can either be stored as a texture

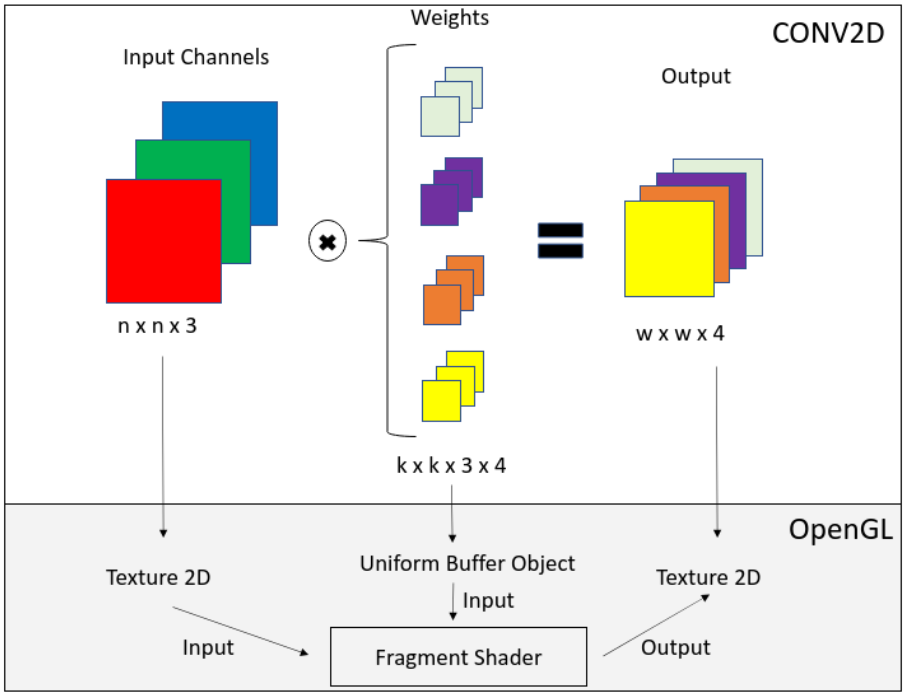

Figure 4.4: The structure and workflow of the Conv2D (2d convolution) operator in the fragment shader pipeline of ShaderNN.

Table 4.1: Breakdown of ESPCN [114] layers and render passes needed to generate the output of each layer. To generate 4 channels in its output, the ShaderNN pipeline needs a single render pass of the fragment shader.

ESPCN Model	ShaderNN Fragment Shader
Layer 1: Conv2D output: n x n x 16	Layer 1 output: 4 render passes
Layer 2: Conv2D output: n x n x 16	Layer 2 output: 4 render passes
Layer 3: Conv2D output: n x n x 4	Layer 3 output: 1 render pass
Layer 4: Subpixel output: 2n x 2n x 1	Layer 4 output: 1 render pass

2D, storing 4 channels or a texture 2D array, storing 4 more channels. For convolution, the weights are passed to the fragment shader via a uniform buffer object or as uniforms. The necessary shader glsl files are created on the mobile device in-situ after the model has been parsed by the inference engine, with each shader responsible for generating a 4-channel output in a render pass. In order to compute a layer having more than 4 outputs, there are multiple shader glsl files and render passes involved.

ShaderNN's fragment shader implementation avoids the overhead of pixel access from neighboring workgroups and thread management, which is unavoidable in compute shaders [113]. When executing architecturally and parametrically smaller models, such overheads become significant and adversely affect the performance. This allows the fragment shader pipeline to provide a considerable speedup over compute shaders. However, for models that have large numbers of parameters or layers generating large numbers of output channels, the ShaderNN fragment shader pipeline needs multiple render passes. Having multiple render passes affects the fragment shader pipeline's performance, thereby making them not a prime choice for inferencing parametrically large models. In order to alleviate the overhead of multiple render passes, an optimization involving multiple render targets (MRTs) is implemented in the fragment shader pipeline. This allows the pipeline to write to multiple target textures simultaneously.

4.4.4 Compute Shader

As we discussed before, fragment shader is good at handling Map and Gather or Stencil computation patterns. Map is an operation that transforms each input element into a corresponding output element. Typical Map operators in convolutional neural networks include Add, Activation, and Batch Normalization. Convolution, Pooling, Concat and Upsample operators are general Gather or Stencil operations, where multiple input elements are transformed into a single output element.

However, there are some operators that cannot be classified by these two categories, such as the Mean and Variance operators used in the Instance Normalization operator.

The Mean and Variance operators are examples of of a class of operators called Reduce operators. A Reduce operator examines all of the input elements, and outputs a single element.

Fragment shader usually renders multiple passes when the number of output channels of a layer is greater than 4 (greater than 8 with double plane MRT), as described in Section 4.4.3. In contrast, compute shader only renders a single pass for each layer regardless of how many output channels there are. In other words, in fragment shader, render passes are calculated sequentially, while in compute shader, since only one render pass is required, the channels in the output are calculated in parallel. This feature makes compute shader more suitable for implementing operators such as Reduce. Additionally, when a layer has a large number of output channels, compute shader is likely to outperform fragment shader by avoiding the excessive application of render passes.

To handle these different cases, we involved compute shader as well, which shares the exact same data structures with fragment shader, but provides more flexibility and better performance in some layers with large numbers of output channels.

As for the implementation details, compute shader is not limited by fixed output channels of a pixel as in fragment shader. As in OpenCL and Metal, almost all of the GPU-based parallel optimizations can be used in the ShaderNN compute shader.

In addition to these common algorithm-level optimizations, we emphasize three points that are important for achieving high performance inference using compute shader in ShaderNN:

1. The data layout of the weights. The Convolution operator is both computation-intensive and data-intensive, and weight access becomes the main bottleneck if not handled

correctly. We incorporated an approach of data layout in the Convolution and Depthwise Convolution operators from MNN [102] to reduce the overhead of weight access.

2. The work group size for Mobile GPUs. A work group is a combination of threads located on the same grid region in the 3D mesh problem space. The size of a work group has a large impact on performance, especially across different types of GPUs. Therefore, the work group size in compute shader operators has been extensively investigated and fine-tuned depending on the specific type of GPU that is targeted. The work group size is configured automatically at compile time according to the given GPU settings for the shader program.

3. Keeping the input and output data in textures. Generally, the input and output should be kept in textures as much as possible. Some operators may execute faster on a CPU when we do not take CPU-GPU data transfer overhead into account. However, when examining the overall execution time, including data transfer time, these operators often perform less efficiently on a CPU. Computing these operators with OpenGL shader and keeping the associated textures is more efficient.

4.4.5 Hybrid Implementation

Fragment shader has favorable performance over compute shader on neural networks having layers with relatively small numbers of output channels by reducing render pass overheads. Compute shader works better on more computationally-intensive layers (e.g., with more output channels) when the advantages of workgroups outweigh the workgroup creation overhead [113]. These two layer types may be present in a given neural network

model. The unified data structure described in Section 4.4.1 enables the mixed usage of fragment shader and compute shader in a single network. In ShaderNN, APIs are provided where users can choose either fragment shader or compute shader at the layer/operator level. Users are able to select either to use fragment shader or compute shader for each layer of an NN according to the layer output channels or the performance observed based on their experimentation.

Thanks to the unified texture data structure that ShaderNN operates on, users are able to freely assemble fragment shaders along with compute shader and use them in a hybrid fashion. For example, in the Shallow U-Net Neuron Segmentation (SUNS) model [4], it is possible to inference some middle layers on compute shader, while inferencing the other layers on fragment shader (details are discussed in Section 4.5.4.1). In other words, our framework provides layer-level shader backend selection, bringing great flexibility and potential for performance optimization.

4.4.6 Inference Core

Figuratively, all of the shaders and data structures in ShaderNN can be viewed as separate beads, and we need to string them together to get a workable inference engine. In ShaderNN, we construct an Inference Core class to complete this task. The two core functions associated with this class are listed in Algorithm 2 and Algorithm 3. After the given deep learning model is parsed and loaded, it is represented as a computation graph, which we refer to as InferenceGraph. Algorithm 2 takes this graph as input, and allocates output textures for the vertices of the graph. The vertices in InferenceGraph are mapped

into render stages in OpenGL. Each of these stages in general needs one or more render passes to complete the associated computation. The output texture of each vertex will be attached to the input texture of its following vertex to avoid resource re-allocation. After the resources are allocated in the initialization process, the run function is called iteratively.

Algorithm 2 Initialization of inference core.

```
Input: InferenceGraph
Output: RenderStage
function INIT( )
    layers ← InferenceGraph.layers
    M ← layers.size()
    for i ← 0 to M do
        stage[i] ← new RenderStage()
        stage[i].layer ← layers[i]
        N ← layers[i].inputs.size()
        for j ← 0 to N do
            input ← layers[i].inputs[j]
            if input.isStageOutput is true then
                texture ← input.stageOutputs[0].texture
            else
                texture ← modelInputs[j].texture
            end if
            stage[i].stageInputs[j].texture.attach(texture)
        end for
        stage[i].stageOutputs[0].texture.allocate()
        P ← layers[i].passes.size()
        for k ← 0 to P do
            stage[i].renderPasses[k].init()
        end for
    end for
end function
```

Algorithm 3 feeds an input texture into render passes at run-time, and sends the render passes to OpenGL. With this concise design, the inference engine is simple and efficient. New operators can be developed and registered dynamically without affecting the code of the inference engine.

Algorithm 3 Execution ("run function") of inference core

```
Input: RenderStages, InputTextures
Output: OutputTexture
function RUN( )
    L ← length(InputTextures)
    for i ← 0 to L do
        modelInputs[i].texture(0).attach(InputTextures[i])
    end for
    M ← RenderStages.size()
    for j ← 0 to M do
        renderPasses ← RenderStages[j].renderPasses
        N ← renderPasses.size()
        for k ← 0 to N do
            renderPasses[k].run()
        end for
    end for
end function
```

4.5 Results

In this section, we report on extensive experiments to evaluate the ShaderNN framework in terms of the time it takes to infer various benchmark CNN models on mobile devices, and in terms of the energy consumed by the framework during inference. The experiments are performed on four mobile devices, which are summarized in Table 4.2. The processors used in Devices 1 and 2 were released by MediaTek in 2022 to target the mid-range and high-end market. The processors used in Devices 3 and 4 were released by Qualcomm in 2020 and 2022, respectively.

For inference, we use two networks, Resnet18 [115] and YOLO v3 Tiny [116], which are used widely to benchmark various inference engines. We also use two additional networks, ESPCN [114] and Spatial Denoise, which are parametrically tiny and designed specifically to achieve real-time performance on mobile devices. For our experiments, we scaled ESPCN down by reducing channels to run on mobile devices efficiently. The

Spatial Denoise model is a modified version of U-Net [117] with fewer layers; again we performed this scaling-down of the model for efficient operation on mobile devices. We also demonstrate two application case studies — calcium imaging segmentation and camera style transfer. For each operator corresponding to a model layer, ShaderNN allows two primary underlying shader pipelines — fragment shader, and compute shader. The fragment shader pipeline has an option to execute as a no-MRT (single-plane MRT) or double-plane MRT.

In our experiments, all available shader pipelines are evaluated independently, illustrating the possible use-case of a pipeline mode based on the choice of model and intended target processor. The best results among the pipelines are compared with TensorFlow Lite GPU OpenGL as a baseline. As described in Section 4.5.4.1, hybrid implementation is offered to further minimize the inference latency. We draw a comparative study with TensorFlow Lite to evaluate the performance and efficiency of the ShaderNN inference framework. In addition to the inference evaluation, in Section 4.5.3, we also evaluate the reduction in I/O time that is mentioned in Section 4.4.1.

Table 4.2: Mobile devices use in our experiments.

Device	Name	Chipset	GPU
1	Find x5 Pro	Dimensity 1300 (MT6893)	Mali G77
2	Reno 8	Dimensity 9000 (MT6983)	Mali G710
3	One Plus 9 Pro	Snapdragon 888 (SM8350)	Adreno 660
4	One Plus 10 Pro	Snapdragon 8 Gen 1 (SM8450)	Adreno 730

4.5.1 Execution Time Evaluation

The four models described in Section 4.5 — Resnet18, YOLO v3 Tiny, ESPCN, and Spatial Denoise — are run on the four devices in Table 4.2 under a controlled test setup. When profiling running time, the mobile devices are charged to more than 95% of their battery capacity and plugged into a constant power supply. As a warm-up, the first 10 runs of the model are skipped; this allows for processor frequencies to stabilize. Then an average inference latency across 60 subsequent runs is reported. We evaluate ShaderNN for compute shader and fragment shader (MRT and no-MRT) pipeline modes and TensorFlow Lite on its OpenGL GPU backend mode.

The results are illustrated and summarized in Figure 4.5 and Figure 4.6. As shown in Figure 4.5, with a properly-selected shader pipeline, ShaderNN outperforms TensorFlow Lite in almost all of the use-cases. ShaderNN outperforms TensorFlow Lite generally by a 4X to 8X speedup on the Spatial Denoise and ESPCN models, and up to 2X speedup on the Resnet18 and YOLO v3 Tiny networks.

As seen in Figure 4.5, ShaderNN's compute shader pipeline mode performs best for inferencing Resnet18 and YOLO v3 Tiny on all target chipsets. Similarly, ShaderNN's fragment shader pipeline performs best for ESPCN and Spatial Denoise model inference on all target chipsets. It is further observed that MRT optimization in the fragment shader pipeline provides additional speedup on some Qualcomm target chipsets, such as the Snapdragon SM8450 and Qualcomm Snapdragon SM8350. The results also help justify the effectiveness of ShaderNN in deploying parametrically smaller models where layers have fewer output channels by using fragment shaders, thereby avoiding overhead induced

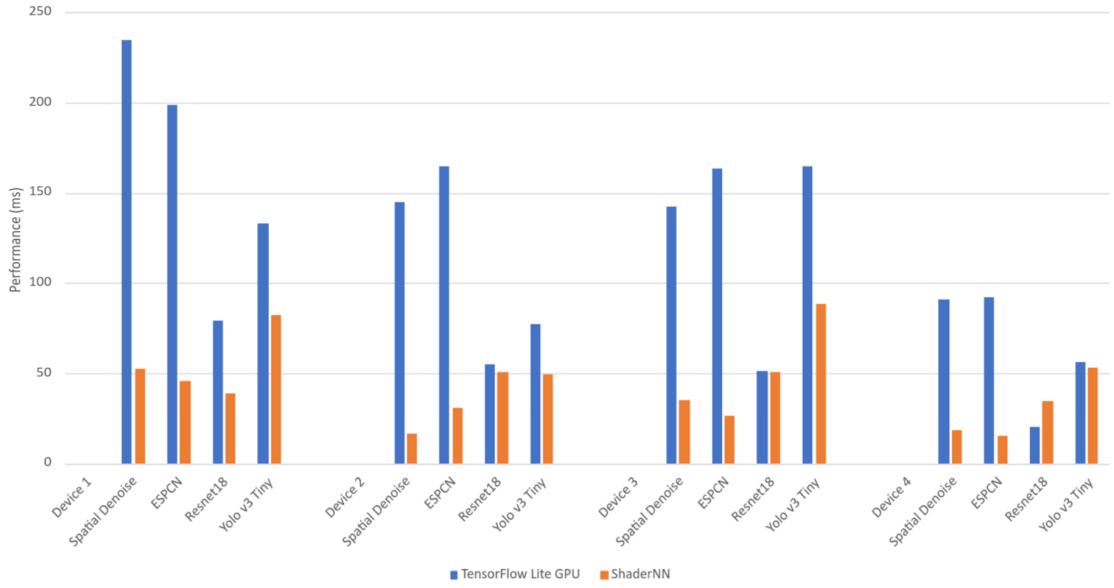

Figure 4.5: Inference latency (ms) comparison on (from left to right) Spatial Denoise, ESPCN, Resnet18, and YOLO v3 Tiny with target platforms (from left to right) Device 1, Device 2, Device 3, and Device 4.

by compute shaders, which is significant in such use-cases. In Section 4.5.4.1, we discuss shader selection in hybrid implementation of compute shader and fragment shader in a single neural network based on this feature. Meanwhile, it can be seen that compute shader outperforms fragment shader when we infer parametrically larger models where layers have more output channels, owing to an efficient implementation of the former and a large number of render passes needed for the latter's implementation.

4.5.2 Energy Consumption

To evaluate the energy consumption of ShaderNN and TensorFlow Lite when inferencing models, we follow a similar test setup as used in Section 4.5.1, but with some slight modifications. The mobile devices are unplugged from a power source with at least 90% battery remaining, following which, the current consumption and time are monitored

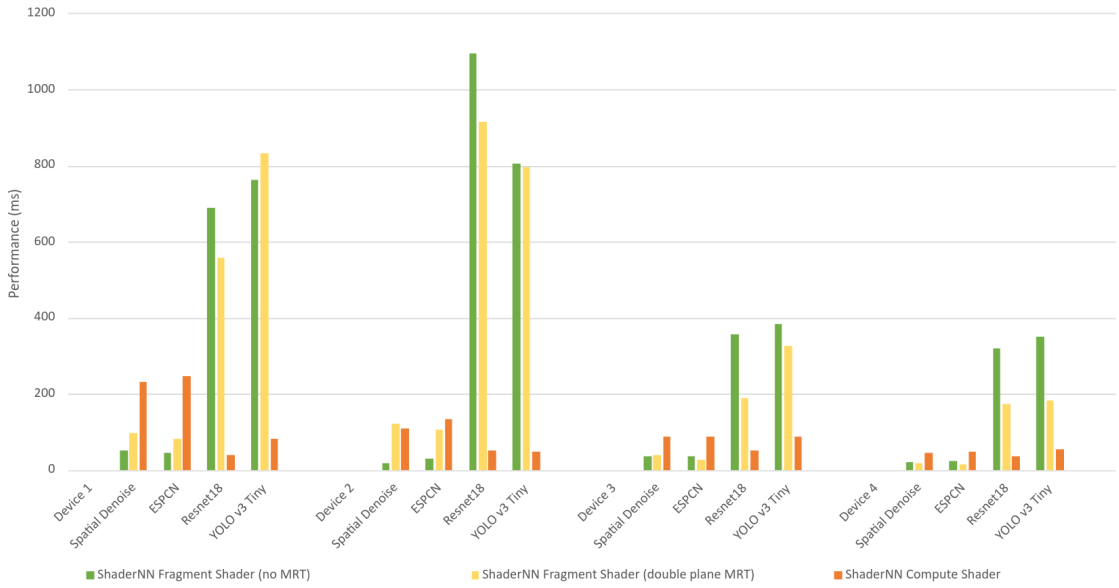

Figure 4.6: Inference latency (ms) of different models when choosing fragment shader without MRT, fragment shader with MRT, and compute shader. The comparison is performed on (from left to right) Spatial Denoise, ESPCN, Resnet18, and YOLO v3 Tiny with target platforms (from left to right) Device 1, Device 2, Device 3, and Device 4.

while the models are running. The first 10 runs are skipped in order to allow processor frequencies to stabilize. An average of 60 runs is then reported as Consumed Energy (E) to inference the network once. The consumed energy metric is derived by calculating the product of average voltage (U), average inference current (I), average baseline current (I_0) and average inference time (t), as seen in Equation 4.1.

$$E = U \cdot (I - I_0) \cdot t. \tag{4.1}$$

I_0 is monitored when there are no applications running on the devices. The values of U, I, and I_0 are recorded by an on-device application.

As illustrated in Figure 4.7, compared with TensorFlow Lite, ShaderNN saves energy

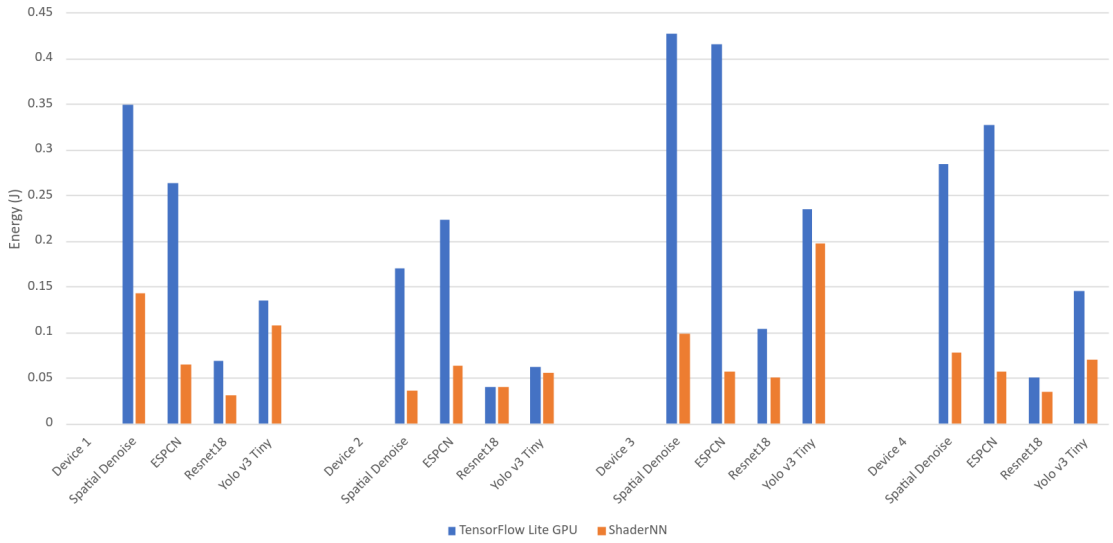

Figure 4.7: Energy consumption comparison for (from left to right) Spatial Denoise, ESPCN, Resnet18, YOLO v3 Tiny on (from left to right) Device 1, Device 2, Device 3, Device 4.

by up to 80%, 70%, 55% and 51% when inferencing Spatial Denoise, ESPCN, Resnet18 and YOLO v3 Tiny, respectively. The results are consistent with inference time data as the increase or decrease in current consumption of the device is not as significant as the difference in inference time observed. In summary, our evaluation of energy consumption shows that large reductions in energy consumption can be achieved with ShaderNN, which is of great importance for mobile applications.

4.5.3 I/O Time Savings in ShaderNN

In order to quantify the overhead of transferring data between GPU textures and CPU memory, a series of experiments was conducted on the previously-identified four mobile devices. The implementation entailed the use of the `glReadPixels()` function to facilitate the transfer of image data from GPU textures to CPU memory. Conversely, the `glTexSubImage2D()` function was employed to reverse the process, copying the

image data back to GPU textures. An assortment of 10 images was arbitrarily chosen, and this bi-directional transfer process was iteratively performed 100 times per image. We would like to emphasize that the textures and corresponding frame buffers were pre-established prior to the experiments, and as such, their initialization time was not factored into the resultant computations. Moreover, the smartphones were set to high-performance mode where applicable.

Table 4.3 presents the average duration required for the bi-directional transfer of data between GPU textures and CPU memory for each device, specifically for images with a resolution of 720p and 1080p and formatted as RGBA32F. The I/O expenditure from GPU to CPU oscillates between 2.523 ms to 8.559 ms per image in 720p image copy, and 6.851 ms to 10.662 ms in 1080p image copy, contingent on the specific device, whereas the I/O cost from CPU to GPU falls within the range of 1.973 ms to 5.512 ms, and 3.399 ms to 5.868 ms on 720p and 1080p image copy, respectively. These findings elucidate that the process of data transfer between GPU textures and CPU memory results in a considerable overhead. Given that ShaderNN does not require any data transfer between GPU and CPU, the results presented here underscore the significant advantages of employing ShaderNN.

Table 4.3: I/O costs of copying data between GPU textures and CPU memory.

Device	Time per image (ms)	GPU → CPU		CPU → GPU	
		720P	1080P	720P	1080P
Device 1	Average	**4.053**	**6.851**	**5.512**	**5.868**
	Standard deviation	1.799	2.113	3.169	3.718
Device 2	Average	**2.523**	**6.947**	**1.973**	**4.028**
	Standard deviation	0.173	0.980	0.147	0.342
Device 3	Average	**8.559**	**10.622**	**2.185**	**3.399**
	Standard deviation	0.788	1.353	0.262	0.715
Device 4	Average	**3.710**	**7.004**	**2.169**	**4.433**
	Standard deviation	0.336	0.825	0.226	0.348

4.5.4 Case Study

In this case study, we demonstrate that our implementation of ShaderNN achieves favorable performance on two relevant applications, calcium image segmentation and camera style transfer. In the image segmentation application, a compute shader/fragment shader hybrid implementation is deployed, and inference latency is measured on different input dimensions. In the style transfer application, we demonstrate the usage of ShaderNN through an application with a graphical interface running on Android phones and we illustrate the steps involved in using the ShaderNN inference engine.

4.5.4.1 Calcium Image Segmentation

In this case study, we demonstrate benefits brought about by ShaderNN in a neural imaging application. Neural imaging is a field of neuroscience where neural signals are extracted from in-vivo recordings, such as large-scale calcium imaging data. Fluorescence calcium imaging encodes the information of neuron activities. Neuron segmentation from calcium images extracts the spatial features of neurons, and helps neuroscientists understand brain functionality — e.g., to investigate treatments for brain disorders. SUNS, Shallow U-Net Neuron Segmentation [4], is a compact shallow U-Net that quickly (in real-time) and accurately segments active neurons from two-photon calcium imaging data. However, in previous work, real-time performance for SUNS is only demonstrated on PCs. On-device neuron segmentation brings important benefits, such as the potential for out-of-the-clinic treatments of neural disorders. With this motivation, we deploy the SUNS model through the ShaderNN pipeline on a mobile device. Specifically, we target Device 4 listed in

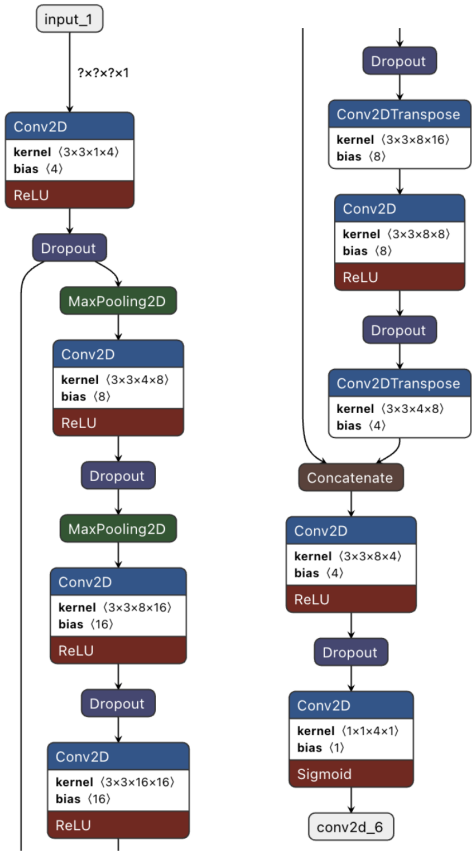

Figure 4.8: Model architecture of SUNS.

Table 4.2 — a One Plus 10 Pro device.

The original pre-trained SUNS model is in an ordinary Keras `.h5` format, including the model architecture and weights. Figure 4.8 shows the model architecture. The model is converted to `.json` format, which can be parsed for use by ShaderNN, by applying the ShaderNN conversion tool. In our case study, we use low to high resolution grayscale calcium images as model input, and compare the performance between ShaderNN and TensorFlow Lite using a converted `.tflite` model.

Hybrid shader selection is also demonstrated and compared with compute shader or fragment shader only (with single or double plane MRT) options. From Table 4.4, when

we select only one type of shader, double plane MRT performs almost the best. This is because SUNS is parametrically small and relatively shallow. Fragment shader works better in this scenario, as mentioned in Section 4.5.1. In the compute shader and fragment shader hybrid implementation, as shown in Table 4.5, we delegate layers with more output channels but smaller height and width to compute shader, and we keep the other layers on fragment shader with double plane MRT. In the 2048×2048 case, we keep all layers in fragment shader — in other words, we do not use hybrid implementation — because the height and width of each layer is large.

In terms of the inference latency, with proper shader selection, ShaderNN outperforms TensorFlow Lite in all input resolution cases by up to 2.1X speedup. In most cases, hybrid shader selection offers more favorable performance compared to using only one type of shader. In these experiments, ShaderNN provides real-time neural image segmentation on low to high resolution calcium image sequences. This case study concretely demonstrates the effectiveness of the ShaderNN inference engine in deep-learning-based calcium imaging processing applications.

4.5.4.2 Style Transfer

Neural Style Transfer is an application of CNNs in the field of image processing. Style transfer involves blending two images such that the output image looks like the content of one of the images that is painted with the style of the other image. Non photo-realistic rendering has been a long-established field in computer graphics, and the introduction of neural style transfer has led to many research efforts in the field. Johnson et al. [118]

Table 4.4: Inference latency of SUNS model on Device 4. Numbers in parentheses give standard deviations.

Inference Latency (ms)	Input/Output Image Resolution			
	120x88	512x512	1024x1024	2048x2048
TensorFlow Lite GPU (OpenGL)	1.207 (0.208)	11.100 (0.799)	25.117 (0.627)	93.377 (0.607)
ShaderNN Fragment Shader (no MRT)	2.3 (0.315)	7.7 (0.439)	20.7 (0.616)	49.7 (0.444)
ShaderNN Fragment Shader (double plane MRT)	2.4 (0.432)	7.1 (0.469)	16.8 (0.451)	**46.2** (0.611)
ShaderNN Computer Shader	1.5 (0.362)	7.2 (0.547)	20.4 (1.24)	62 (1.416)
ShaderNN Hybrid Shader	**1.1** (0.133)	**5.3** (0.337)	**14.9** (1.144)	

Table 4.5: Layer-level shader selection in hybrid implementation of compute and fragment shaders; CS: compute shader, FS: fragment shader.

Layer Name	Output Dims.	120 × 88, 512×512	1024×1024
Input	m×n×1		
Conv2D_1	m×n×4	FS	FS
MaxPooling2D_1	m/2×n/2×4	FS	FS
Conv2D_2	m/2×n/2×8	FS	FS
MaxPooling2D_2	m/4×n/4×8	FS	FS
Conv2D_3	m/4×n/4×16	CS	CS
Conv2D_4	m/4×n/4×16	CS	CS
Conv2DTranspose_1	m/2×n/2×16	CS	FS
Conv2D_5	m/2×n/2×8	FS	FS
Conv2DTranspose_2	m×n×8	FS	FS
Concatenate	m×n×2	FS	FS
Conv2D_6	m×n×4	FS	FS
Conv2D_7	m×n×1	FS	FS

proposed a feed-forward network that is pretrained to a specific style and generates a stylized image in a single forward pass in the testing phase. This algorithm was able to achieve real-time performance on larger form factor devices. Motivated by the aesthetic value this algorithm can bring on mobile devices, we use the algorithm as second case study for ShaderNN, and implement it using the ShaderNN pipeline.

The pretrained style-specific model, which is in ONNX format [119], is first converted to a decoupled weight format accepted by the ShaderNN inference engine using the ShaderNN convert tool. Then as shown in Figure 4.9, the model is parsed by ShaderNN with operators loaded to the inference engine. The shared library of the ShaderNN core is then linked to the Android application project, which encapsulates the function that fetches camera frames, processes them, and displays the processed frames. The ShaderNN core inference engine is initialized. The camera frames are then passed as textures to the initialized inference engine, and processed by the given style transfer model. The "run model" block will iteratively execute the model to transfer the style of the frame from the camera when the camera is on. The output texture from the model is then displayed as the output.

In our experiments, ShaderNN's pipeline was found to be efficient in processing the style transfer model of [118]. On Device 4, a performance of approximately 20 frames per second (fps) was achieved using compute shader on a real-time camera feed.

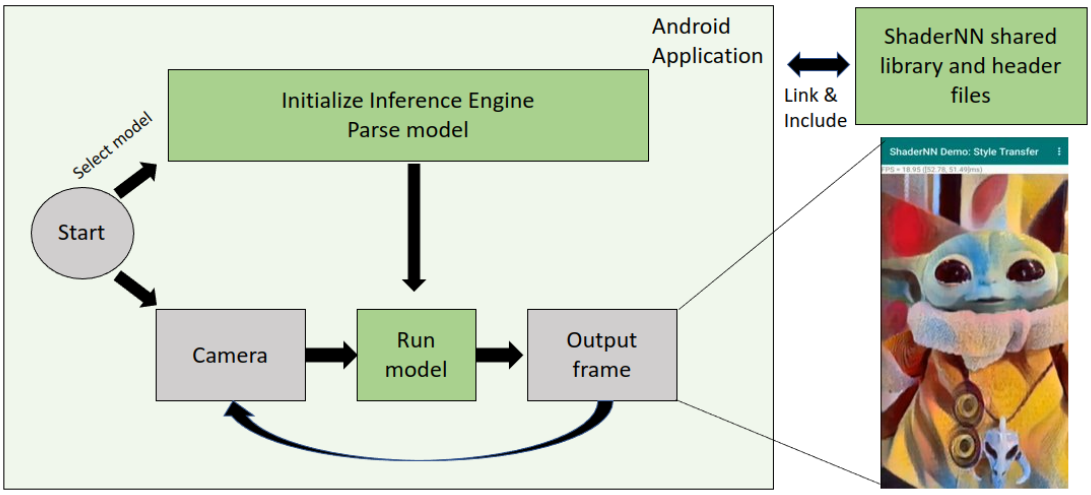

Figure 4.9: The workflow of the camera style transfer application for Android devices.

4.6 Discussion and Conclusion

In this chapter, we have presented ShaderNN, a lightweight deep learning inference framework optimized for neural networks. ShaderNN provides high-performance inference for deep learning applications in image processing and graphics on mobile devices. To the best of our knowledge, ShaderNN is the first implementation of fragment shader in a neural network inference engine. With ShaderNN, fragment shader provides favorable performance in parametrically small neural networks. The utility of ShaderNN to neural image processing and other mobile graphics applications has been demonstrated through extensive experiments with several relevant neural network models.

Many machine-learning-powered smartphone applications are graphics-heavy; such applications can benefit significantly from reduction of data movement across compute and graphics subsystems. ShaderNN offers superior performance, especially in graphics-based pipelines, due to its texture-based input and output, which reduces the volume of CPU-GPU data transfer. Operator-level compute or fragment shader selection is provided, enabling

hybrid shader implementation. OpenGL features are utilized to improve the performance of operators, including normalization, interpolation, and texture padding. When the shader is properly selected, ShaderNN significantly outperforms TF-Lite, which is one of the most popular off-the-shelf engines for on-device model inference.

In our experiments, we focused on comparing the performance and efficiency of ShaderNN with TensorFlow-Lite. Another popular inference platform for mobile devices is NCNN. However, ShaderNN is based on OpenGL, while NCNN does not provide an OpenGL backend. We did not include NCNN in our experimental comparison due to its lack of an OpenGL backend. Since ShaderNN and NCNN use different backend technologies, it would not be fair nor meaningful to compare their performance directly. We chose to compare ShaderNN with TensorFlow-Lite because both frameworks are based on OpenGL and can be used for mobile deep learning inference.

MediaPipe [120] is an open-source framework for building real-time, multimodal machine learning pipelines. MediaPipe differs from ShaderNN in that it provides a platform for developing end-to-end applications that incorporate both inference and other processing tasks, such as computer vision and multimedia processing. A direct experimental comparison of ShaderNN and MediaPipe is not appropriate for this chapter given their different scopes and purposes.

As our experiments show, ShaderNN has favorable performance on the SUNS, ESPCN and Spatial Denoise neural networks when delegating to fragment shader. Based on our observations, ShaderNN works better when the network has a high resolution input but a relatively small depth and small number of channels in each layer. For paramterically small layers, fragment shader works well, and for other types of layers, compute shader

typically provides better performance.

Useful directions for future work include investigating the potential advantages of ShaderNN for large-scale input models, and automatic shader pipeline selection based on the structure of the input model.

Chapter 5

Conclusions and Future Work

This thesis has developed novel methods and tools for development of real-time neural imaging processing systems. The contributions of the thesis include two dataflow based-design platforms, Neuron Detection and Signal Extraction Platform (NDSEP) and the NEural DEcoding COnfiguration Package (NEDECO). The thesis contributions also include an image-based neural network inference engine, called ShaderNN.

In Chapter 2, using a dataflow-based design architecture, we developed NDSEP, a real-time, high speed calcium imaging processing system, including object detection, motion correction, and signal extraction. NDSEP provides flexibility to expand the system, experiment with design trade-offs, and manage complex implementation constraints. In Chapter 3, we developed a general parameter optimization framework for neural decoding systems. NEDECO is designed to integrate population-based search strategies, including but not limited to particle swarm optimization and genetic algorithms, to find efficient solutions for neural decoding design spaces under customizable constraints. In Chapter 4, we presented a real-time NN inference engine, called ShaderNN. ShaderNN provides high-performance inference for image processing and graphics applications on mobile devices. We demonstrate the utility of ShaderNN to neural image processing through a case study involving real-time, resource-constrained calcium image segmentation.

In the remainder of this chapter, we summarize interesting directions for future work

to go beyond the developments of this thesis, and further address challenging aspects of system design for neural image processing systems. This future work has important applications to information extraction from neural signals.

As discussed in Chapter 2, the accuracy of neuron recognition and the quality of the output neural signal are highly sensitive to the performance of motion correction, which makes motion correction a crucial module in the NDSEP system, as well as in other neural image processing platforms. However, motion correction is the most time consuming step in the whole NDSEP system as demonstrated in Table 2.6. Motion correction takes approximately 50% of the system execution time both on simulated data and on real-world video streams.

Additionally, there are many types of motion, such as translation motion, Euclidean motion, and non-rigid motion, as the degree of freedom increases. In many neural image streams where motion correction is applied, motion is not purely randomly distributed, but follows certain kinds of patterns. In general, different motion correction algorithms are best suited for different kinds of motion patterns. An interesting direction for future work is the automatic selection of motion correction algorithms based on dynamic characteristics of a neural image stream. Such an automatic algorithm selection approach can help to improve the efficiency and accuracy of neural image processing pipelines, such as those provided in NDSEP.

In Chapter 2, we explored the hierarchical design of reconfigurable dataflow models, where an adapting subsystem and controlling subsystem are encapsulated hierarchically as single actors. Applying such a hierarchical design approach for adaptive motion correction is an interesting direction for future work. In such a motion correction framework, the

controlling subsystem could select different adapting subsystems, which execute different motion correction algorithms, according to the motion pattern that is learned from the input image data stream.

In such a hierarchical, adaptive motion correction approach, the controlling subsystem could also tune the parameters in each adapting subsystem depending on the past motion patterns acquired from the data. The production and consumption rates of the actors within the subsystems generally remain constant, which facilitates analysis of the system when the hierarchical actors are plugged into the higher-level dataflow graph.

Additionally, the proposed dataflow modeling methods that we apply facilitate automated parallelism exploration and optimization on multicore and hybrid CPU-GPU processing platforms through a high-level programming abstraction [121]. A possible approach to improved exploitation of parallelism is to decompose the motion correction actor into multiple actors, as shown in Figure 5.1. Such decomposition can provide more flexibility in exploiting parallelism from the enclosing dataflow graph. Moreover, the PSO-based optimization approach of Section 3.4.1 can be used to tune the searching strategy of motion correction algorithms such as “findECC”.

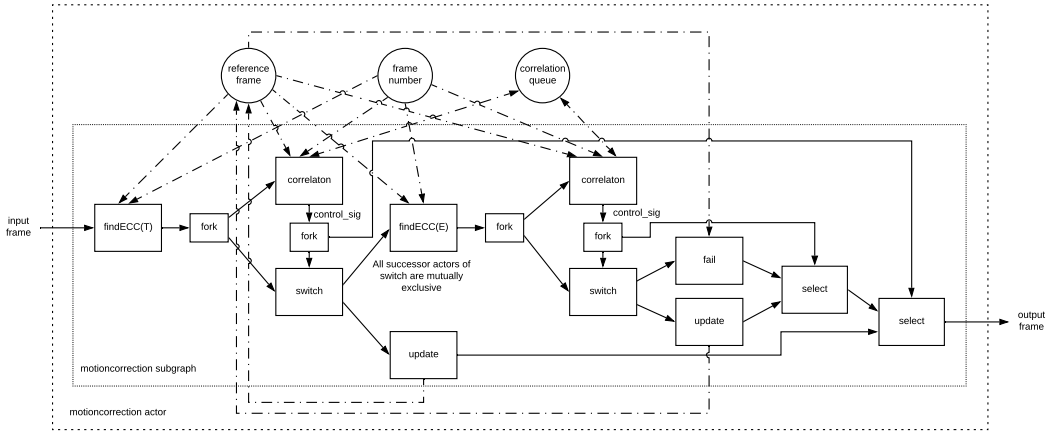

Figure 5.1: Decomposed design for improved motion correction in NDSEP.

Appendix A: Additional Information About NDSEP on Neurofinder Data and Summary of Variables

In this appendix, we provide additional information about the neurofinder data from Section 2.5.1, and a summary of variables and symbols from Chapter 2.

Table 5.1: Information about the neurofinder data that we used [45].

	Neurofinder 01.00	Neurofinder 03.00
Lab	Hausser Lab	Losonczy Lab
contributors	[“Adam Packer”, “Lloyd Russell”]	[“Jeff Zaremba”]
animal	Mouse	Mouse
animal-state	Awake head-fixed	Awake head-fixed
experiment	Drifting grating visual stimuli	Hidden reward spatial navigation
method	Two-photon raster	Two-photon raster
indicator	GCaMP6s	GCaMP6f
region	V1	dHPC CA1
pixels-per-micron	0.8	1.7007
rate-hz	7.5	7.5
dimensions	[512, 512, 2250]	[498, 490, 2250]

Table 5.2: Summary of variables and symbols.

Variable	Section	Description and Value
$\{n_1, n_2, \ldots, n_m\}$	1	Neuron masks set
G	2.2	Dataflow graph
X	2.2	Set of actors
E	2.2	Set of edges
e	2.2	An edge in E
p	3.1	Output port of the subinit graph
$n(p)$	3.1	The number of ordered pairs associated with p
$A_i(p)$	3.1	An actor in the body graph
$P_i(p)$	3.1	A parameter of actor $A_i(p)$
F_c	3.2.1	The current frame
F_s	3.2.1	The shifted frame
F_r	3.2.1	The reference frame
C_1, C_2	3.2.1	Computed correlation in Translation Only (C_1)/Euclidean (C_2) mode
$\tau = \{\tau_1, \tau_1\}$	3.2.1	Threshold in Translation Only (τ_1)/Euclidean (τ_2) mode
$p(\tau)$	3.2.1	Empirically defined parameter for τ_1 and τ_2, $p(\tau_1) = 2$, $p(\tau_2) = 10$
I	3.2.2	The current pixel's intensity
n_d	3.2.3	The number of detected neurons
δ	3.2.3	Neuron detection matrix
`thresholdStep`	3.2.3	The minimum intensity difference between the inside and outside of a blob
A_{min}, A_{max}	3.2.3	The minimum/maximum sizes of the blobs to detect
η	3.2.3	A set of neurons identified by neuron detection
F_{mc}	3.2.4	A motion-corrected image frame
β	3.2.4	Firing output vector
L	3.2.4, 3.3.3	Total number of image frames in the input video sequence
T_n	3.3.2	Pre-determined target number of neurons, $T_n = 5$
V	4.1	Membrane potential
V_{rest}	4.1	Rest potential
λ	4.1	Membrane time constant
P_{rot}	4.1	Rotation occurrence probability
α_{rot}	4.1	Rotation range
ϕ	4.1	Detected neuron
r	4.1	Some region in the frame
F	4.1	Average pixel intensity in I
R	4.1	Average pixel intensity in r
R_s	4.1	Average correlation coefficient across all neurons
$\rho(\phi)$	4.1	Correlation between $\Delta F/F$ of r and the ground-truth spike train of ϕ
R_n	4.1	The average value of $\rho(\phi)$
M_x, M_y	4.1	x-displacement, y-displacement
M_{rot}	4.1	angle error
$Rate_{fail}$	4.1	Failure rate of motion correction
$mean(\alpha_{rot})$	4.1	mean rotation error

Appendix B: Pseudocode Representations of Selected Algorithms of NEDECO

In this appendix, we provide a pseudocode representation of the overall workflow of the NEDECO architecture, as well as a pseudocode representation of the GA Core actor in NEDECO.

Algorithm 4 A pseudocode sketch of the NEDECO architecture. Here, by the Search Core actor, we mean the PSO Core or GA Core actor, depending on which search strategy is being employed.

```
while There is at least one enabled actor in the dataflow graph do
    if The Search Core actor is enabled then

        /* Fire the Search Core actor (see Algorithm 1 and Algorithm 5 for details) */

    else if The Set Params actor is enabled then

        /* Fire the Set Params actor */
        Read tokens that encapsulate new PNDS parameter values from the input FIFO
of the Set Params
                actor
        Reconfigure the PNDS (wrapper) actor by updating its parameters

    else if The PNDS actor is enabled then

        /* Fire the PNDS actor */
        Execute the given neural decoding system repeatedly on a pre-defined dataset of
              calcium-imaging-based neural images
        Aggregate the resulting measurements on execution time and accuracy
        Construct a token that encapsulates the measured accuracy and execution time
        Write this token to the output FIFO of the actor

    else if The Fitness Evaluation actor is enabled then

        /* Fire the Fitness Evaluation actor */
        Read a token that encapsulates accuracy and execution time data from the input
FIFO
        Apply a linear aggregation function to calculate the fitness value
        Write the calculated fitness value to the output FIFO

    end if
end while
```

Algorithm 5 A pseudocode sketch of NEDECO integrated with the GA Core actor.

```
/* GA actor / initialize-write mode */
for each chromosome in the current population do
    Initialize the chromosome
    Send parameters in chromosome to evaluate
end for

/* Fire the Set Params, PNDS, and Fitness Evaluation actors to derive the new parameter
settings for the
      PNDS (see Algorithm 4 for details) */

/* GA actor / initialize-read mode */
for each chromosome do
    Read fitness values
    Initialize best result and previous best result
end for

/* Stopping-evaluation mode */
while not (max iterations reached or error criteria met) do

    /* GA actor / update-crossover-write mode */
    Apply elitism
    Select mating population
    Start crossover on mating population
    for i ← 1 to CP-EL do
        Send the parameters for the i^th individual in the mating population
    end for

    /* Fire the Set Params, PNDS, and Fitness Evaluation actors to derive the new
parameter settings for
          the PNDS (see Algorithm 4 for details) */

    /* GA actor / update-crossover-read mode */
    for i ← 1 to CP-EL do
        Read fitness values
    end for
    Finish crossing-over mating population

    /* GA actor / update-complete-write mode */
    Start mutation
    for i ← 1 to P-CP do
        Select a chromosome randomly from mating population
```

Algorithm 5 A pseudocode sketch of NEDECO integrated with the GA Core actor (continued).

Send the parameters for the i^{th} individual in the mutation population
end for

/* Fire the Set Params, PNDS, and Fitness Evaluation actors to derive the new parameter settings for
the PNDS (see Algorithm 4 for details) */

/* GA actor / `update-complete-read` mode */
for $i \leftarrow 1$ to $P\text{-}CP$ **do**
Read fitness values
end for
Finish mutation

end while

/* GA actor / `write-output` mode */
Write results to output file

Bibliography

[1] Youssef Ezzyat, James E Kragel, John F Burke, Deborah F Levy, Anastasia Lyalenko, Paul Wanda, Logan O'Sullivan, Katherine B Hurley, Stanislav Busygin, Isaac Pedisich, et al., "Direct brain stimulation modulates encoding states and memory performance in humans," *Current Biology*, vol. 27, no. 9, pp. 1251–1258, 2017.

[2] Edward A Lee and Thomas M Parks, "Dataflow process networks," *Proceedings of the IEEE*, vol. 83, no. 5, pp. 773–801, 1995.

[3] S. S. Bhattacharyya, E. Deprettere, R. Leupers, and J. Takala, Eds., *Handbook of Signal Processing Systems*, Springer, third edition, 2019.

[4] Yijun Bao, Somayyeh Soltanian-Zadeh, Sina Farsiu, and Yiyang Gong, "Segmentation of neurons from fluorescence calcium recordings beyond real time," *Nature machine intelligence*, vol. 3, no. 7, pp. 590–600, 2021.

[5] Ling Tong, Rachel Langton, Joseph Glykys, and Stephen Baek, "Anmaf: an automated neuronal morphology analysis framework using convolutional neural networks," *Scientific reports*, vol. 11, no. 1, pp. 8179, 2021.

[6] Zhehao Xu, Yukun Wu, Jiangheng Guan, Shanshan Liang, Junxia Pan, Meng Wang, Qianshuo Hu, Hongbo Jia, Xiaowei Chen, and Xiang Liao, "Neuroseg-ii: A deep learning approach for generalized neuron segmentation in two-photon ca2+ imaging," *Frontiers in Cellular Neuroscience*, vol. 17, pp. 1127847, 2023.

[7] Luca Sità, Marco Brondi, Pedro Lagomarsino de Leon Roig, Sebastiano Curreli, Mariangela Panniello, Dania Vecchia, and Tommaso Fellin, "A deep-learning approach for online cell identification and trace extraction in functional two-photon calcium imaging," *Nature communications*, vol. 13, no. 1, pp. 1–22, 2022.

[8] Yuanlong Zhang, Guoxun Zhang, Xiaofei Han, Jiamin Wu, Ziwei Li, Xinyang Li, Guihua Xiao, Hao Xie, Lu Fang, and Qionghai Dai, "Rapid detection of neurons in widefield calcium imaging datasets after training with synthetic data," *Nature Methods*, vol. 20, no. 5, pp. 747–754, 2023.

[9] Dongsheng Xiao, Brandon J Forys, Matthieu P Vanni, and Timothy H Murphy, "Mesonet allows automated scaling and segmentation of mouse mesoscale cortical maps using machine learning," *Nature communications*, vol. 12, no. 1, pp. 5992, 2021.

[10] Tsubasa Ito, Keisuke Ota, Kanako Ueno, Yasuhiro Oisi, Chie Matsubara, Kenta Kobayashi, Masamichi Ohkura, Junichi Nakai, Masanori Murayama, and Toru Aonishi, "Low computational-cost cell detection method for calcium imaging data," *Neuroscience Research*, vol. 179, pp. 39–50, 2022.

[11] Yagna J Pathak, Walter Greenleaf, Leo Verhagen Metman, Pieter Kubben, Sridevi Sarma, Brian Pepin, Douglas Lautner, Scott DeBates, Alex M Benison, Binesh Balasingh, et al., "Digital health integration with neuromodulation therapies: The future of patient-centric innovation in neuromodulation," *Frontiers in Digital Health*, p. 43, 2021.

[12] Jochen Klucken, Rejko Krüger, Peter Schmidt, and Bastiaan R Bloem, "Management of parkinson's disease 20 years from now: towards digital health pathways," *Journal of Parkinson's disease*, vol. 8, no. s1, pp. S85–S94, 2018.

[13] Martina Chirra, Luca Marsili, Linsdey Wattley, Leonard L Sokol, Elizabeth Keeling, Simona Maule, Gabriele Sobrero, Carlo Alberto Artusi, Alberto Romagnolo, Maurizio Zibetti, et al., "Telemedicine in neurological disorders: opportunities and challenges," *Telemedicine and e-Health*, vol. 25, no. 7, pp. 541–550, 2019.

[14] Yaesop Lee, Jing Xie, Eungjoo Lee, Srijesh Sudarsanan, Da-Ting Lin, Rong Chen, and Shuvra S Bhattacharyya, "Real-time neuron detection and neural signal extraction platform for miniature calcium imaging," *Frontiers in computational neuroscience*, vol. 14, pp. 43, 2020.

[15] Til Ole Bergmann, Anke Karabanov, Gesa Hartwigsen, Axel Thielscher, and Hartwig Roman Siebner, "Combining non-invasive transcranial brain stimulation with neuroimaging and electrophysiology: current approaches and future perspectives," *Neuroimage*, vol. 140, pp. 4–19, 2016.

[16] Helena Knotkova and Dirk Rasche, *Textbook of neuromodulation*, Springer, 2016.

[17] David T Brocker, Brandon D Swan, Rosa Q So, Dennis A Turner, Robert E Gross, and Warren M Grill, "Optimized temporal pattern of brain stimulation designed by computational evolution," *Science translational medicine*, vol. 9, no. 371, pp. eaah3532, 2017.

[18] T deBettencourt Megan, Jonathan D Cohen, Ray F Lee, Kenneth A Norman, and Nicholas B Turk-Browne, "Closed-loop training of attention with real-time brain imaging," *Nature neuroscience*, vol. 18, no. 3, pp. 470, 2015.

[19] K. K. Ghosh, L. D. Burns, E. D. Cocker, A. Nimmerjahn, Y. Ziv, A. El Gamal, and M. J. Schnitzer, "Miniaturized integration of a fluorescence microscope," *Nature Methods*, vol. 8, pp. 871–878, 2011.

[20] J. Kerr and A. Nimmerjahn, "Functional imaging in freely moving animals," *Current Opinion in Neurobiology*, vol. 22, no. 1, pp. 45–53, 2012.

[21] B. B. Scott, C. D. Brody, and D. W. Tank, "Cellular resolution functional imaging in behaving rats using voluntary head restraint," *Neuron*, vol. 80, no. 2, pp. 371–384, 2013.

[22] E. A. Lee and T. M. Parks, "Dataflow process networks," *Proceedings of the IEEE*, pp. 773–799, May 1995.

[23] Andrea Giovannucci, Johannes Friedrich, Pat Gunn, Jérémie Kalfon, Brandon L Brown, Sue Ann Koay, Jiannis Taxidis, Farzaneh Najafi, Jeffrey L Gauthier, Pengcheng Zhou, et al., "Caiman an open source tool for scalable calcium imaging data analysis," *Elife*, vol. 8, pp. e38173, 2019.

[24] Somayyeh Soltanian-Zadeh, Kaan Sahingur, Sarah Blau, Yiyang Gong, and Sina Farsiu, "Fast and robust active neuron segmentation in two-photon calcium imaging using spatiotemporal deep learning," *Proceedings of the National Academy of Sciences*, vol. 116, no. 17, pp. 8554–8563, 2019.

[25] E. A. Mukamel, A. Nimmerjahn, and M. J. Schnitzer, "Automated analysis of cellular signals from large-scale calcium imaging data," *Neuron*, vol. 63, no. 6, pp. 747–760, 2009.

[26] E. A. Pnevmatikakis et al., "Simultaneous denoising, deconvolution, and demixing of calcium imaging data," *Neuron*, vol. 89, no. 2, pp. 285–299, 2016.

[27] P. Zhou et al., "Efficient and accurate extraction of in vivo calcium signals from microendoscopic video data," *eLife*, vol. 7, pp. e28728, 2018.

[28] N. J. Apthorpe, A. J. Riordan, R. E. Aguilar, J. Homann, Y. Gu, D. W. Tank, and H. S. Seung, "Automatic neuron detection in calcium imaging data using convolutional networks," in *Advances in Neural Information Processing Systems*, 2016, pp. 3278–3286.

[29] S. L Resendez, J. H Jennings, R. L. Ung, V. M. K. Namboodiri, Z. C. Zhou, J. M. Otis, H. Nomura, J. A. McHenry, O. Kosyk, and G. D Stuber, "Visualization of cortical, subcortical and deep brain neural circuit dynamics during naturalistic mammalian behavior with head-mounted microscopes and chronically implanted lenses," *Nature Protocols*, vol. 11, pp. 566–597, 2016.

[30] Tom Vercauteren, X. Pennec, A. Perchant, and N. Ayache, "Diffeomorphic demons: Efficient non-parametric image registration," *NeuroImage*, vol. 45, no. 1, pp. S61–S72, 2009.

[31] S. Lin, J. Wu, and S. S. Bhattacharyya, "Memory-constrained vectorization and scheduling of dataflow graphs for hybrid CPU-GPU platforms," *ACM Transactions on Embedded Computing Systems*, vol. 17, no. 2, pp. 50:1–50:25, January 2018.

[32] J. B. Dennis, "First version of a data flow procedure language," in *Programming Symposium*, B. Robinet, Ed., vol. 19 of *Lecture Notes in Computer Science*, pp. 362–376. Springer Berlin Heidelberg, 1974.

[33] KAHN Gilles, "The semantics of a simple language for parallel programming," *Information processing*, vol. 74, pp. 471–475, 1974.

[34] B. Bhattacharya and S. S. Bhattacharyya, "Parameterized dataflow modeling for DSP systems," *IEEE Transactions on Signal Processing*, vol. 49, no. 10, pp. 2408–2421, October 2001.

[35] K. Desnos and F. Palumbo, "Dataflow modeling for reconfigurable signal processing systems," in *Handbook of Signal Processing Systems*, S. S. Bhattacharyya, E. F. Deprettere, R. Leupers, and J. Takala, Eds., pp. 787–824. Springer, third edition, 2019.

[36] S. Lin, Y. Liu, K. Lee, L. Li, W. Plishker, and S. S. Bhattacharyya, "The DSPCAD framework for modeling and synthesis of signal processing systems," in *Handbook of Hardware/Software Codesign*, S. Ha and J. Teich, Eds., pp. 1–35. Springer, 2017.

[37] G. D. Evangelidis and E. Z. Psarakis, "Parametric image alignment using enhanced correlation coefficient maximization," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 30, no. 10, pp. 1858–1865, 2008.

[38] Barıs Evrim Demiröz, *The OpenCV Reference Manual, Release 2.4.13.7*, February 2019.

[39] Carsen Stringer and Marius Pachitariu, "Computational processing of neural recordings from calcium imaging data," *Current opinion in neurobiology*, vol. 55, pp. 22–31, 2019.

[40] Philippe Thevenaz, Urs E Ruttimann, and Michael Unser, "A pyramid approach to subpixel registration based on intensity," *IEEE transactions on image processing*, vol. 7, no. 1, pp. 27–41, 1998.

[41] Z. Zhou, C. Shen, W. Plishker, and S. S. Bhattacharyya, "Dataflow-based, cross-platform design flow for DSP applications," in *Embedded Systems Development: From Functional Models to Implementations*, A. Sangiovanni-Vincentelli, H. Zeng, M. Di Natale, and P. Marwedel, Eds., pp. 41–65. Springer, 2014.

[42] Sebastián A Romano, Verónica Pérez-Schuster, Adrien Jouary, Jonathan Boulanger-Weill, Alessia Candeo, Thomas Pietri, and Germán Sumbre, "An integrated calcium imaging processing toolbox for the analysis of neuronal population dynamics," *PLoS computational biology*, vol. 13, no. 6, pp. e1005526, 2017.

[43] Robert Gütig and Haim Sompolinsky, "The tempotron: a neuron that learns spike timing–based decisions," *Nature neuroscience*, vol. 9, no. 3, pp. 420, 2006.

[44] Romain Brette, Michelle Rudolph, Ted Carnevale, Michael Hines, David Beeman, James M Bower, Markus Diesmann, Abigail Morrison, Philip H Goodman, Frederick C Harris, et al., "Simulation of networks of spiking neurons: a review of tools and strategies," *Journal of computational neuroscience*, vol. 23, no. 3, pp. 349–398, 2007.

[45] Simon Peron et al., "The neurofinder chanllenge," 2016.

[46] Hod Dana, Yi Sun, Boaz Mohar, Brad K Hulse, Aaron M Kerlin, Jeremy P Hasseman, Getahun Tsegaye, Arthur Tsang, Allan Wong, Ronak Patel, et al., "High-performance calcium sensors for imaging activity in neuronal populations and microcompartments.," *Nature methods*, vol. 16, no. 7, pp. 649–657, 2019.

[47] Ivan Svetunkov, *smooth: Forecasting Using State Space Models, Release 2.5.1*, June 2019.

[48] Julia Pilowsky, *colorednoise: Simulate Temporally Autocorrelated Populations, Release 1.0.4*, January 2019.

[49] Aleksander Klibisz, Derek Rose, Matthew Eicholtz, Jay Blundon, and Stanislav Zakharenko, "Fast, simple calcium imaging segmentation with fully convolutional networks," in *Deep Learning in Medical Image Analysis and Multimodal Learning for Clinical Decision Support*, pp. 285–293. Springer, 2017.

[50] Quico Spaen, Dorit S Hochbaum, and Roberto Asín-Achá, "Hnccorr: A novel combinatorial approach for cell identification in calcium-imaging movies," *arXiv preprint arXiv:1703.01999*, 2017.

[51] Marius Pachitariu, Carsen Stringer, Mario Dipoppa, Sylvia Schröder, L Federico Rossi, Henry Dalgleish, Matteo Carandini, and Kenneth D Harris, "Suite2p: beyond 10,000 neurons with standard two-photon microscopy," *Biorxiv*, 2017.

[52] Marius Pachitariu, Adam M Packer, Noah Pettit, Henry Dalgleish, Michael Hausser, and Maneesh Sahani, "Extracting regions of interest from biological images with convolutional sparse block coding," in *Advances in Neural Information Processing Systems 26*, C. J. C. Burges, L. Bottou, M. Welling, Z. Ghahramani, and K. Q. Weinberger, Eds., pp. 1745–1753. Curran Associates, Inc., 2013.

[53] Elke Kirschbaum, Alberto Bailoni, and Fred A Hamprecht, "Disco for the cia: Deep learning, instance segmentation, and correlations for calcium imaging analysis," *arXiv preprint arXiv:1908.07957*, 2019.

[54] Nuo Li, Tsai-Wen Chen, Zengcai V Guo, Charles R Gerfen, and Karel Svoboda, "A motor cortex circuit for motor planning and movement," *Nature*, vol. 519, no. 7541, pp. 51, 2015.

[55] David G Lowe, "Distinctive image features from scale-invariant keypoints," *International journal of computer vision*, vol. 60, no. 2, pp. 91–110, 2004.

[56] Eran A Mukamel, Axel Nimmerjahn, and Mark J Schnitzer, "Automated analysis of cellular signals from large-scale calcium imaging data," *Neuron*, vol. 63, no. 6, pp. 747–760, 2009.

[57] T. O. Bergmann, A. Karabanov, G. Hartwigsen, A. Thielscher, and H. R. Siebner, "Combining non-invasive transcranial brain stimulation with neuroimaging and electrophysiology: Current approaches and future perspectives," *NeuroImage*, vol. 140, pp. 4–19, 2016.

[58] H. Knotkova and D. Rasche, Eds., *Textbook of Neuromodulation*, Springer, 2015.

[59] D. T. Brocker, B. D. Swan, R. Q. So, D. A. Turner, R. E. Gross, and W. M. Grill, "Optimized temporal pattern of brain stimulation designed by computational evolution," *Science Translational Medicine*, vol. 9, no. 371, 2017.

[60] M. T. deBettencourt, J. D. Cohen, R. F. Lee, K. A. Norman, and N. B. Turk-Browne, "Closed-loop training of attention with real-time brain imaging," *Nature Neuroscience*, vol. 18, no. 3, pp. 470–475, 2015.

[61] Y. Ezzyat et al., "Direct brain stimulation modulates encoding states and memory performance in humans," *Current Biology*, vol. 27, no. 9, pp. 1251–1258, 2017.

[62] Yaesop Lee, Sreenuj Chellath Madayambath, Yanzhou Liu, Da-Ting Lin, Rong Chen, and Shuvra S Bhattacharyya, "Online learning in neural decoding using incremental linear discriminant analysis," in *2017 IEEE International Conference on Cyborg and Bionic Systems (CBS)*. IEEE, 2017, pp. 173–177.

[63] Rong Chen and Da-Ting Lin, "Decoding brain states based on microcircuits," in *2018 IEEE International Conference on Cyborg and Bionic Systems (CBS)*. IEEE, 2018, pp. 397–400.

[64] Michael Häusser, "Optogenetics: the age of light," *Nature methods*, vol. 11, no. 10, pp. 1012, 2014.

[65] Jing Xie, Shuvra Bhattacharyya, and Rong Chen, "A parameter-optimization framework for neural decoding systems," *Frontiers in Neuroinformatics*, vol. 17, pp. 2, 2023.

[66] James Kennedy and Russell Eberhart, "Particle swarm optimization," in *Proceedings of ICNN'95-International Conference on Neural Networks*. IEEE, 1995, vol. 4, pp. 1942–1948.

[67] Eftychios A Pnevmatikakis, Daniel Soudry, Yuanjun Gao, Timothy A Machado, Josh Merel, David Pfau, Thomas Reardon, Yu Mu, Clay Lacefield, Weijian Yang, et al., "Simultaneous denoising, deconvolution, and demixing of calcium imaging data," *Neuron*, vol. 89, no. 2, pp. 285–299, 2016.

[68] J. I. Glaser, A. S. Benjamin, R. H. Chowdhury, M. G. Perich, L. E. Miller, and K. P. Kording, "Machine learning for neural decoding," *eNeuro*, vol. 7, no. 4, pp. 1–16, 2020.

[69] Wenxin Liu, Il-Yop Chung, Li Liu, Siyu Leng, and David A Cartes, "Real-time particle swarm optimization based current harmonic cancellation," *Engineering Applications of Artificial Intelligence*, vol. 24, no. 1, pp. 132–141, 2011.

[70] Pablo Ribalta Lorenzo, Jakub Nalepa, Michal Kawulok, Luciano Sanchez Ramos, and José Ranilla Pastor, "Particle swarm optimization for hyper-parameter selection in deep neural networks," in *Proceedings of the genetic and evolutionary computation conference*, 2017, pp. 481–488.

[71] Pratibha Singh, Santanu Chaudhury, and Bijaya Ketan Panigrahi, "Hybrid mpso-cnn: Multi-level particle swarm optimized hyperparameters of convolutional neural network," *Swarm and Evolutionary Computation*, vol. 63, pp. 100863, 2021.

[72] Yaru Li and Yulai Zhang, "Hyper-parameter estimation method with particle swarm optimization," *arXiv preprint arXiv:2011.11944*, 2020.

[73] Yu Guo, Jian-Yu Li, and Zhi-Hui Zhan, "Efficient hyperparameter optimization for convolution neural networks in deep learning: A distributed particle swarm optimization approach," *Cybernetics and Systems*, vol. 52, no. 1, pp. 36–57, 2020.

[74] Toshihiko Yamasaki, Takuto Honma, and Kiyoharu Aizawa, "Efficient optimization of convolutional neural networks using particle swarm optimization," in *2017 IEEE third international conference on multimedia big data (BigMM)*. IEEE, 2017, pp. 70–73.

[75] Y. Lee, J. Xie, E. Lee, S. Sudarsanan, D.-T. Lin, R. Chen, and S. Bhattacharyya, "Real-time neuron detection and neural signal extraction platform for miniature calcium imaging," *Frontiers in Computational Neuroscience*, vol. 14, pp. 1–20, 2020.

[76] Maurice Clerc, "Back to random topology," *Online at http://clerc. maurice. free. fr/pso*, 2007, (accessed July 12, 2020).

[77] T. Back, U. Hammel, and H-P Schwefel, "Evolutionary computation: Comments on the history and current state," *IEEE Transactions on Evolutionary Computation*, vol. 1, no. 1, pp. 3–17, April 1997.

[78] Konstantinos E Parsopoulos and Michael N Vrahatis, "Particle swarm optimization method in multiobjective problems," in *Proceedings of the 2002 ACM symposium on Applied computing*, 2002, pp. 603–607.

[79] A. Zhou, B.-Y. Qu, H. Li, S.-Z. Zhao, P. N. Suganthan, and Q. Zhang, "Multi-objective evolutionary algorithms: A survey of the state of the art," *Swarm and Evolutionary Computation*, vol. 1, no. 1, pp. 32–49, 2011.

[80] Maurice Clerc and James Kennedy, "The particle swarm-explosion, stability, and convergence in a multidimensional complex space," *IEEE transactions on Evolutionary Computation*, vol. 6, no. 1, pp. 58–73, 2002.

[81] Yaochu Jin, Markus Olhofer, and Bernhard Sendhoff, "Dynamic weighted aggregation for evolutionary multi-objective optimization: Why does it work and how?," in *Proceedings of the 3rd Annual Conference on Genetic and Evolutionary Computation*, 2001, pp. 1042–1049.

[82] K. Miettinen, *Nonlinear Multiobjective Optimization*, Kluwer Academic Publishers, 1999.

[83] Daniel Dimanov, Emili Balaguer-Ballester, Colin Singleton, and Shahin Rostami, "Moncae: Multi-objective neuroevolution of convolutional autoencoders," *arXiv preprint arXiv:2106.11914*, 2021.

[84] W. Plishker, N. Sane, and S. S. Bhattacharyya, "A generalized scheduling approach for dynamic dataflow applications," in *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition*, Nice, France, April 2009, pp. 111–116.

[85] E. A. Lee and D. G. Messerschmitt, "Synchronous dataflow," *Proceedings of the IEEE*, vol. 75, no. 9, pp. 1235–1245, September 1987.

[86] Olivier Mallet, "Galgo-2.0," `https://github.com/olmallet81/GALGO-2.0`, 2022, Retrieved August 25, 2022.

[87] Kelly H Zou, Simon K Warfield, Aditya Bharatha, Clare MC Tempany, Michael R Kaus, Steven J Haker, William M Wells III, Ferenc A Jolesz, and Ron Kikinis, "Statistical validation of image segmentation quality based on a spatial overlap index1: scientific reports," *Academic radiology*, vol. 11, no. 2, pp. 178–189, 2004.

[88] Margarita Reyes-Sierra, CA Coello Coello, et al., "Multi-objective particle swarm optimizers: A survey of the state-of-the-art," *International journal of computational intelligence research*, vol. 2, no. 3, pp. 287–308, 2006.

[89] Adrian A Hopgood and A Mierzejewska, "Transform ranking: a new method of fitness scaling in genetic algorithms," in *International Conference on Innovative Techniques and Applications of Artificial Intelligence*. Springer, 2008, pp. 349–354.

[90] Kalyanmoy Deb, Samir Agrawal, Amrit Pratap, and Tanaka Meyarivan, "A fast elitist non-dominated sorting genetic algorithm for multi-objective optimization: Nsga-ii," in *International conference on parallel problem solving from nature*. Springer, 2000, pp. 849–858.

[91] Aravind Seshadri, "Nsga - ii: A multi-objective optimization algorithm," `https://www.mathworks.com/matlabcentral/fileexchange/10429-nsga-ii-a-multi-objective-optimization-algorithm`, 2022, Retrieved August 25, 2022.

[92] Urvesh Bhowan, Mengjie Zhang, and Mark Johnston, "Auc analysis of the pareto-front using multi-objective gp for classification with unbalanced data," in *Proceedings of the 12th annual conference on Genetic and evolutionary computation*, 2010, pp. 845–852.

[93] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen, "MobileNetV2: Inverted residuals and linear bottlenecks," in *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2018.

[94] Forrest N Iandola, Song Han, Matthew W Moskewicz, Khalid Ashraf, William J Dally, and Kurt Keutzer, "Squeezenet: Alexnet-level accuracy with 50x fewer parameters and¡ 0.5 mb model size," *arXiv preprint arXiv:1602.07360*, 2016.

[95] Tianming Zhao, Yucheng Xie, Yan Wang, Jerry Cheng, Xiaonan Guo, Bin Hu, and Yingying Chen, "A survey of deep learning on mobile devices: Applications, optimizations, challenges, and research opportunities," *Proceedings of the IEEE*, vol. 110, no. 3, pp. 334–354, 2022.

[96] "Fragment shader — OpenGL Wiki,," 2020, [Online; accessed 18-August-2022].

[97] Juhyun Lee, Nikolay Chirkov, Ekaterina Ignasheva, Yury Pisarchyk, Mogan Shieh, Fabio Riccardi, Raman Sarokin, Andrei Kulik, and Matthias Grundmann, "On-device neural net inference with mobile GPUs," *arXiv preprint arXiv:1907.01989*, 2019.

[98] Junjie Bai, Fang Lu, Ke Zhang, et al., "ONNX: Open neural network exchange," `https://github.com/onnx/onnx`, 2019.

[99] Adam Paszke et al., "PyTorch: An imperative style, high-performance deep learning library," *CoRR*, vol. abs/1912.01703, 2019.

[100] "PyTorch mobile," `https://pytorch.org/mobile/home/`, 2020.

[101] "Metal: Render advanced 3D graphics and compute data in parallel with graphics processors.," `https://developer.apple.com/documentation/metal/`, 2022.

[102] Xiaotang Jiang, Huan Wang, Yiliu Chen, Ziqi Wu, Lichuan Wang, Bin Zou, Yafeng Yang, Zongyang Cui, Yu Cai, Tianhang Yu, Chengfei Lv, and Zhihua Wu, "MNN: A universal and efficient inference engine," in *MLSys*, 2020.

[103] "NCNN: a high-performance neural network inference computing framework optimized for mobile platforms," `https://github.com/Tencent/ncnn`, 2022.

[104] "TNN: A high-performance, lightweight neural network inference framework," `https://github.com/Tencent/TNN`, 2022.

[105] "Bolt: A light-weight library for deep learning," `https://github.com/huawei-noah/bolt`, 2022.

[106] "Mobile AI compute engine," `https://github.com/XiaoMi/mace`, 2022.

[107] "Anakin," `https://github.com/PaddlePaddle/Anakin/`, 2020.

[108] David Cronin, *Deep Neural Network Algorithms on Graphics Processors for Embedded Systems*, Ph.D. thesis, School of Computer Science and Statistics, 2018.

[109] Jamie Menjay Lin, Siargey Pisarchyk, Juhyun Lee, David Tian, Tingbo Hou, Karthik Raveendran, Raman Sarokin, George Sung, Trent Tolley, and Matthias Grundmann, "Efficient heterogeneous video segmentation at the edge," *arXiv preprint arXiv:2208.11666*, 2022.

[110] The Khronos Group, "Vulkan tutorials," 2022, [Online; accessed 18-August-2022].

[111] Khronos OpenCL Working Group, *The OpenCL C Specification, Version 2.0*, 2015.

[112] Martín Abadi et al., "TensorFlow: A system for large-scale machine learning," in *Operating Systems Design and Implementation (OSDI)*, 2016.

[113] Robert Tornai and Péter Fürjes-Benke, "Compute shader in image processing development," 2021.

[114] Wenzhe Shi, Jose Caballero, Ferenc Huszár, Johannes Totz, Andrew P Aitken, Rob Bishop, Daniel Rueckert, and Zehan Wang, "Real-time single image and video super-resolution using an efficient sub-pixel convolutional neural network," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 1874–1883.

[115] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun, "Deep residual learning for image recognition," *arXiv preprint arXiv:1512.03385*, 2015.

[116] Joseph Redmon and Ali Farhadi, "Yolov3: An incremental improvement," *CoRR*, vol. abs/1804.02767, 2018.

[117] Olaf Ronneberger, Philipp Fischer, and Thomas Brox, "U-Net: Convolutional networks for biomedical image segmentation," *CoRR*, vol. abs/1505.04597, 2015.

[118] Justin Johnson, Alexandre Alahi, and Li Fei-Fei, "Perceptual losses for real-time style transfer and super-resolution," in *European conference on computer vision*. Springer, 2016, pp. 694–711.

[119] "ONNX model zoo," `https://github.com/onnx/models/tree/main/vision/style_transfer/fast_neural_style/model`, 2022.

[120] Camillo Lugaresi, Jiuqiang Tang, Hadon Nash, Chris McClanahan, Esha Uboweja, Michael Hays, Fan Zhang, Chuo-Ling Chang, Ming Guang Yong, Juhyun Lee, et al., "Mediapipe: A framework for building perception pipelines," *arXiv preprint arXiv:1906.08172*, 2019.

[121] Jiahao Wu, Jing Xie, Alexandre Bardakoff, Timothy Blattner, Walid Keyrouz, and Shuvra S Bhattacharyya, "Cgmbe: a model-based tool for the design and implementation of real-time image processing applications on cpu–gpu platforms," *Journal of Real-Time Image Processing*, vol. 18, pp. 561–583, 2021.