HIGH-PERFORMANCE POWER-EFFICIENT SOLUTIONS FOR EMBEDDED VISION COMPUTING

A Dissertation Presented
by

Hamed Tabkhivayghan

to

The Department of Electrical and Computer Engineering

in partial fulfillment of the requirements
for the degree of

Doctor of Philosophy

in

Electrical and Computer Engineering

Northeastern University
Boston, Massachusetts

August 2014
Dissertation Title: HIGH-PERFORMANCE POWER-EFFICIENT SOLUTIONS FOR EMBEDDED VISION COMPUTING.

Author: Hamed Tabkhivayghan.

Department: Electrical and Computer Engineering.
To my family.
# Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>List of Figures</td>
<td>v</td>
</tr>
<tr>
<td>List of Tables</td>
<td>vii</td>
</tr>
<tr>
<td>Abstract</td>
<td>viii</td>
</tr>
<tr>
<td><strong>1 Introduction</strong></td>
<td>1</td>
</tr>
<tr>
<td>1.1 Embedded Vision Computing</td>
<td>1</td>
</tr>
<tr>
<td>1.1.1 Embedded Vision Pipeline</td>
<td>4</td>
</tr>
<tr>
<td>1.2 Problem Definition</td>
<td>5</td>
</tr>
<tr>
<td>1.3 Dissertation Contributions</td>
<td>6</td>
</tr>
<tr>
<td>1.4 Dissertation Overview</td>
<td>7</td>
</tr>
<tr>
<td><strong>2 Related Work</strong></td>
<td>9</td>
</tr>
<tr>
<td>2.1 Embedded Vision SoCs</td>
<td>9</td>
</tr>
<tr>
<td>2.1.1 Adaptive Vision Algorithms</td>
<td>10</td>
</tr>
<tr>
<td>2.1.2 Bandwidth / Quality Trade-off</td>
<td>11</td>
</tr>
<tr>
<td>2.2 Architecture Alternatives for Flexibility / Efficiency trade-off</td>
<td>12</td>
</tr>
<tr>
<td>2.2.1 Coarse-Grain Reconfigurable Architectures (CGRAs)</td>
<td>12</td>
</tr>
<tr>
<td>2.2.2 Application-Specific Instruction Processors (ASIPs)</td>
<td>13</td>
</tr>
<tr>
<td>2.3 System Specification Development</td>
<td>14</td>
</tr>
<tr>
<td>2.3.1 System Requirement Capturing</td>
<td>15</td>
</tr>
<tr>
<td>2.3.2 Abstractions Above Specification</td>
<td>16</td>
</tr>
<tr>
<td>2.4 Chapter Summary</td>
<td>17</td>
</tr>
<tr>
<td><strong>3 Traffic Separation in Adaptive Vision Algorithms</strong></td>
<td>18</td>
</tr>
<tr>
<td>3.1 Demand for Communication-Centric Design</td>
<td>18</td>
</tr>
<tr>
<td>3.2 Traffic Separation in Adaptive Vision Algorithms</td>
<td>19</td>
</tr>
<tr>
<td>3.2.1 Applications Examples</td>
<td>21</td>
</tr>
<tr>
<td>3.2.2 Quality / Bandwidth Trade-off</td>
<td>23</td>
</tr>
<tr>
<td>3.2.3 Communication-Centric Architecture Template</td>
<td>26</td>
</tr>
<tr>
<td>3.2.4 System Integration</td>
<td>28</td>
</tr>
<tr>
<td>3.3 Case Study on MoG Background Subtraction</td>
<td>29</td>
</tr>
<tr>
<td>3.4 Experimental Results</td>
<td>32</td>
</tr>
</tbody>
</table>
3.4.1 Quality / Bandwidth Exploration ................................................. 32
3.4.2 Zynq prototyping ........................................................................ 36
3.4.3 System Integration ..................................................................... 40
3.5 Chapter Summary .......................................................................... 40

4 Function-Level Processor (FLP) .................................................. 42
4.1 Demands for Flexibility and Efficiency ........................................ 42
4.1.1 Efficiency/Flexibility Trade-off ............................................... 44
4.2 Function-Level Processor (FLP) .................................................. 46
4.2.1 Function-Block (FB) ................................................................. 47
4.2.2 FLP Architecture Overview .................................................. 48
4.2.3 Function-Set Architecture (FSA) ........................................... 53
4.2.4 FLP Example: Pipeline-Vision-Processor (PVP) ................... 54
4.3 Experimental Results .................................................................. 56
4.3.1 Mapping Flexibility ................................................................. 56
4.3.2 Computation / Communication Demands ............................. 58
4.3.3 Power Overhead ..................................................................... 60
4.4 Challenges and Opportunities ..................................................... 62
4.4.1 Research Opportunities in FLP Design ................................. 62
4.4.2 Research Opportunities Simplifying FLP Use ....................... 63
4.5 Chapter Summary ........................................................................ 64

5 Conceptual Abstraction Levels (CALs) ........................................ 65
5.1 Architecting Complexity of Heterogeneous MPSoCs .................. 65
5.2 Conceptual Abstraction Levels (CALs) ........................................ 67
5.2.1 Market Requirements .............................................................. 70
5.2.2 Conceptual Application-Block Flow .................................... 71
5.2.3 Algorithm Selection ................................................................. 74
5.2.4 Executable Application-Block .............................................. 76
5.2.5 System Specification .............................................................. 84
5.3 CALs Automation Demands ......................................................... 90
5.4 Chapter Summary ........................................................................ 91

6 Conclusions and Future Research Opportunities ..................... 93

7 List of Publications ....................................................................... 96

Bibliography ...................................................................................... 98
List of Figures

1.1 Pervasive embedded vision computing ................................................. 2
1.2 Heterogeneous Multiprocessor System-on-Chip (MPSoC) ......................... 2
1.3 Design gap between algorithm demands and SoC capabilities .................... 3
1.4 Coarse-level vision computing pipeline ............................................ 4
1.5 Streaming vs algorithm-intrinsic traffic in adaptive vision algorithms ........ 7
1.6 Function-Level Processor (FLP) .................................................. 8

2.1 Accelerator-Rich MPSoC [30] ................................................................ 12
2.2 Coarse-Grain Reconfigurable Architectures (CGRAs) [46] ...................... 13
2.3 Application-Specific Instruction Processors (ASIPs) ............................... 14
2.4 Abstract pyramid model [58] .......................................................... 17

3.1 Streaming vs algorithm-intrinsic traffic in adaptive vision algorithms .......... 20
3.2 Example of traffic separation in adaptive vision algorithms .................... 21
3.3 Memory Access per Pixel in MoG algorithm ....................................... 22
3.4 Orthogonal impact factors to power in quality ..................................... 24
3.5 Bandwidth quality exploration for Gaussian parameters ......................... 25
3.6 Communication-centric architecture template for adaptive vision algorithms 26
3.7 Precision adjustment ..................................................................... 27
3.8 System integration options .................................................................. 28
3.9 MoG design flow ............................................................................. 30
3.10 MoG specification model including coarse-grained parallelism ............... 30
3.11 MoG computation micro-pipeline stages .......................................... 31
3.12 MoG communication overview ...................................................... 32
3.13 Exploration for complex scene ......................................................... 34
3.14 Exploration for the medium scene ..................................................... 35
3.15 Optical flow quality / bandwidth trade-off .......................................... 35
3.16 Experimental setup ......................................................................... 36
3.17 Power consumption of proposed architecture template ........................ 39
3.18 Abstract algorithm overview of object tracking flow ............................. 40
3.19 Architecture overview of embedded vision SoC solution ....................... 41

4.1 High-level ILP+HWACC architecture .................................................. 45
4.2 General FLP Architecture Composition ............................................. 46
4.3 Function Block (FB) ........................................ 48
4.4 Chaining Function Blocks (FBs) .................................. 49
4.5 FB-to-FB MUX-based communication .................................. 50
4.6 Streaming data vs operational data ................................ 51
4.7 Autonomous fetch of control and configuration data from system memory. 52
4.8 2MB on-chip memory ............................................. 52
4.9 FLP system integration ............................................ 53
4.10 Pipeline Vision Processor (PVP) .................................. 55
4.11 The structures of PMA and THC functions .......................... 55
4.12 Transferring Application FLow Chart to Function Graph ........................ 57
4.13 HB/LB and Canny concurrent execution. ......................... 57
4.14 Architecture alternatives for result comparison ..................... 58
4.15 Overall power consumption of FLP-PVP, ILP-BFDSP and ILP-BFDSP+ACCS .... 61
4.16 Conceptual flow of realizing FSA for a set of application markets. ............ 63

5.1 Conceptual Abstraction Levels (CALs) above system specification. ............ 67
5.2 Transition between two abstraction levels ................................ 69
5.3 Identified application-blocks at Conceptual Application-Block Flow level ........ 73
5.4 Identified reuse application-blocks after IP-reuse selection ...................... 73
5.5 Specified algorithms at Algorithm Selection level ........................... 76
5.6 Estimated computation / memory Access demand for MoG, Blob Detection and Blob Tracking application blocks .................................................. 78
5.7 Specified PE-classes at Executable Application-Block level ....................... 80
5.8 compression/Decompression of Gaussian Parameters. ......................... 82
5.9 MoG Quality /bandwidth exploration ................................ 82
5.10 Generalization of memory bandwidth benefit in MoG .......................... 83
5.11 Identified support blocks after Application-Block Tuning ...................... 83
5.12 Specification model of vision flow captured in SpecC SLDL ...................... 85
5.13 Expanded Specification model of vision flow with Application-Blocks decomposition and hierarchy .......................................................... 86
5.14 Performance estimation of system specification model ......................... 87
5.15 Memory access estimation of system specification model ...................... 87
5.16 Transaction-Level-Model for Fine-Grain Architecture Exploration ............ 89
5.17 Fine-grain 6-level pipeline MoG ................................... 89
5.18 Abstraction Characteristics Throughout the Design Process .................. 91
# List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Properties of vision computing pipeline stages</td>
<td>4</td>
</tr>
<tr>
<td>3.1</td>
<td>Scenes under the test</td>
<td>33</td>
</tr>
<tr>
<td>3.2</td>
<td>Estimated resource utilization of Mixture of Gaussians (MOG) algorithm with the frame rate of 30 Hz.</td>
<td>33</td>
</tr>
<tr>
<td>3.3</td>
<td>MoG execution on different scene complexity.</td>
<td>37</td>
</tr>
<tr>
<td>3.4</td>
<td>System Resource Utilization</td>
<td>38</td>
</tr>
<tr>
<td>4.1</td>
<td>Computation Granularity: efficiency/flexibility trade-off</td>
<td>44</td>
</tr>
<tr>
<td>4.2</td>
<td>Application Mapping to PVP</td>
<td>58</td>
</tr>
<tr>
<td>4.3</td>
<td>Computation demand</td>
<td>59</td>
</tr>
<tr>
<td>4.4</td>
<td>Off-chip communication demand</td>
<td>61</td>
</tr>
<tr>
<td>5.1</td>
<td>Characteristics / Comparison of background elimination algorithms.</td>
<td>75</td>
</tr>
<tr>
<td>5.2</td>
<td>Estimated computation resource for Mixture of Gaussians (MOG) algorithm with the frame rate of 30 Hz.</td>
<td>78</td>
</tr>
<tr>
<td>5.3</td>
<td>Estimated power consumption of MOG</td>
<td>79</td>
</tr>
</tbody>
</table>
Abstract

HIGH-PERFORMANCE POWER-EFFICIENT SOLUTIONS FOR EMBEDDED VISION COMPUTING

by

Hamed Tabkhivayghan

Doctor of Philosophy in Computer Engineering

Northeastern University, August 2014

Prof. Gunar Schirner, Adviser

Embedded vision is a top tier, rapidly growing market with a host of challenges and conflicting requirements. Complex algorithms can pose immense computation (>50 GOPs) and communication (>8 GB/s) demands, especially considering adaptive vision algorithms (which employ machine learning techniques and concurrently update a scene model). Yet at the same time, their embedded deployment demands low power operation consuming only a few Watts. Multiprocessor System-on-Chips (MPSoCs) have emerged as one main architecture approach to improve power efficiency while increasing computational performance. Still, current vision MPSoCs either shy away from supporting adaptive vision algorithms or operate at very limited resolutions due to the immense communication and computation demands.

This dissertation identifies major challenges that hinder embedded realizations of advanced vision algorithms. (1) immense communication demands (>8 GB/s) render efficient embedded implementations infeasible. (2) constructing larger vision applications out of independent vision processing elements (even if (1) would be solved) is challenging due to the combined communication demand. (3) to recover design cost, sales quantities need to be increased which can be achieved through targeting a domain of applications. This, however, requires novel architectures to simultaneously provide efficiency (performance and power) and flexibility (to execute multiple applications). Finally, (4) system architects often start from system specification model and rely on their evolving knowledge to architect vision platforms. Abstraction levels and automation tools need to be identified to guide system architects on the path from market requirements to a system specification.

This dissertation makes three major contributions to address the above identified challenges. First, this dissertation outlines how to reduce the communication demand of adaptive vision algorithms which removes a tremendous hurdle for their embedded realization. For this, we have
identified two communication types commonly present in these algorithms, namely streaming and algorithm-intrinsic traffic. Separating these traffic types enables application-specific management of algorithm-intrinsic data (e.g. through compression, prioritization). We have demonstrated the benefits using Mixture of Gaussian (MoG) background subtraction. Through compression, we reduced the memory bandwidth by 60% without impacting quality. Through an architecture template, we demonstrate how the traffic separation can be realized in a platform. We furthermore demonstrate the benefits of traffic separation when constructing complete vision applications. Our complete object tracking vision flow (image smoothing, MoG background subtraction, morphology, component labeling, histogram checking) realized on a Zynq-based architecture processes 1080p at 30Hz. It executes 40 GOPs at only 1.7 Watts of on-chip power.

Second, this dissertation introduces a novel processor class to efficiently support a set of vision applications within a market. In particular, we introduce Function-level Processor (FLP) which offers efficiency similar to custom hardware and yet is sufficient flexible to execute different applications (of the same market). An FLP achieves efficiency by coarsening architecture programmability from instructions (as in an ILP) to functions. We demonstrate the benefits using Analog Devices’ Pipeline Vision Processor (PVP). We show how 10 different Automotive Driver Assistance System (ADAS) applications can be entirely mapped to the PVP. The PVP processes to 22.4 GOPs while consuming 314 mW – 14x-18x less than a compared ILP-based solutions.

Third, this dissertation provides guidance for system architects in early stages of the design, i.e. from market requirements to a system specification model. For this, we introduce Conceptual Abstraction Levels (CALs). CALs identify a sequence of critical areas for early architecture exploration and resolve interdependent challenges and dependencies through iteration. CALs help system architects to identify the potential application taking benefits of traffic separation or application blocks for function-level processing at early stages of design.

Overall, this dissertation tackles complexities associated with architecting embedded vision MPSoCs from three different angles: (1) in abstract design phase, (2) realization of individual algorithms in hardware, and (3) embedded realization of a complete flow even when simultaneously targeting multiple applications. The dissertation contributions can also provide guidance considering other challenging streaming domains, such as radar processing, wireless base-band processing or software-defined radio.
Chapter 1

Introduction

1.1 Embedded Vision Computing

Among digital signal processing markets, embedded vision is considered one top-tier, fast-growing area \[14,17\]. Embedded vision refers to the deployment of visual capabilities to embedded systems for better understanding of 2D/3D visual scenes \[14\]. It covers a variety of rapidly growing markets and applications. Examples are Advanced Driver Assistance System (ADAS), industrial vision, video surveillance and robotics (illustrated in Figure 1.1). With the industry movement toward ubiquitous vision computing, vision capabilities and visual analysis will become an inherent part of many embedded platforms. Similar to the way that wireless communication has dominated digital industry over the past decade, embedded vision technology would be a pioneer market in the following decade. As one indication, the vision market share seems to be skyrocketing. For video surveillance, the estimation shows more than 3-fold growth from $11.5 billion in 2008 to $37.5 billion in 2015 \[36\]. An even stronger growth is predicted for the ADAS market with a 14-fold growth over eight years ($16.6 billion in 2012 to $261 billion in 2020).

While vision computing is already a strong research focus, embedded deployment of vision algorithms is still in a fairly early stage. The embedded realization of vision algorithms is notoriously difficult. Vision computing increases the demand for extremely high performance, coupled with very low power and desire for low cost. Often in vision applications, many concurrent operations drive the pixel compute complexity well into the many billions of operations per second (GOPs) range. Embedded vision platforms must offer extreme compute performance while consuming very little power (often less than 1 W) and still be sufficiently programmable. At the same time, vision platforms must be offered at remarkably low price points. The conflicting goals (high performance,
CHAPTER 1. INTRODUCTION

Figure 1.1: Pervasive embedded vision computing

low power and low cost) imposes massive challenges in architecting embedded vision platforms.

To reconcile the conflicting requirements, Multi-Processor System-on-Chips (MPSoCs) have emerged as one main approach. MPSoCs are often heterogeneous to offer flexibility and performance/power efficiency at the same time. Figure 1.2a illustrates an example of heterogeneous MPSoC including Instruction-Level Processors (ILPs) and custom-HW engines. ILPs comprise of different processor classes such as application/control processors, ASIPs, DSPs and GPUs. They provide high flexibility but have a comparatively low efficiency. Conversely, custom-HWs offer ultimate power efficiency with high performance but very limited flexibility. The general aim is to map computation-intense kernels (often vision filters) into custom-HWs and leave the higher-level processing and control for ILPs (SW execution).

Figure 1.2: Heterogeneous Multiprocessor System-on-Chip (MPSoC)

Rather than targeting a narrow set of vision applications, vision MPSoCs are often designed...
for a group of evolving markets supporting many applications \cite{2,94}. Architecting for a group of markets/applications increases design productivity and reduces overall design and development costs including the very expensive fabrication and Non-Recurring Engineering (NRE) costs. Examples of MPSoCs targeting multiple vision markets are ADI ADSP-BF60x \cite{50}, TI DaVinci \cite{5} and Toshiba Visconti (illustrated in Figure 1.2b). They all share a general trend which is efficient implementation of vision filters in custom-HWs (such as edge / corner detection filters) and executing the remaining application parts on ILPs.

We observe a gap between market demands (e.g. performance and complexity) and what current vision SoCs can deliver. In algorithm design environments (e.g. Matlab and OpenCV), vision experts focus on the development of advanced vision algorithms for complex scene analysis and multiple objects tracking. The advanced vision algorithms often employ machine-learning principles. They keep a model of the scene and continuously update the model with respect to the changes in the scene under analysis. We also call the learning-based vision algorithms “adaptive vision algorithms”. At high resolutions, adaptive vision algorithms result in high computation and more importantly immense communication demands due to updating the scene models. One grand challenge of vision SoCs is how to manage the immense communication of adaptive vision algorithms. The state-of-the-art vision SoCs are not able to efficiently tackle this challenge. In the result, they either shy away from adaptive vision algorithms or perform very limited resolutions (mainly due to not managing the algorithm’s communication demands). Hence, new solutions are needed to open a path toward advanced embedded vision computing by tackling the associated challenges of adaptive vision algorithms.
CHAPTER 1. INTRODUCTION

1.1.1 Embedded Vision Pipeline

For the purpose of this dissertation and in order to put the dissertation contributions into context, we roughly divide vision algorithms into three major classes. Figure 1.4 highlights the coarse-grained vision processing pipeline: pre-processing, mid-processing and post-processing.

![Coarse-level vision computing pipeline](image)

Figure 1.4: Coarse-level vision computing pipeline

<table>
<thead>
<tr>
<th>Stage</th>
<th>Algorithm Class</th>
<th>Computation</th>
<th>Communication</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pre-processing</td>
<td>Filters</td>
<td>High (regular)</td>
<td>Low / Medium</td>
<td>Convolution</td>
</tr>
<tr>
<td>Mid-processing</td>
<td>Adaptive</td>
<td>High (irregular)</td>
<td>Very High</td>
<td>MoG, Optical flow</td>
</tr>
<tr>
<td>Post-processing</td>
<td>Control / Intelligence</td>
<td>Low</td>
<td>Low</td>
<td>Trajectory Analysis</td>
</tr>
</tbody>
</table>

Table 1.1 outlines the architecture properties per individual processing class. (1) Pre-processing mainly contains vision filter algorithms (e.g. convolution; color extraction) used to extract scene basic features such as objects’ edges or corners. They often operate on an isolated video frame with a very limited data interaction across the frames. Their regular execution patterns with small communication demands, limited to pixel streams, make pre-processing algorithms ideal candidates for HW implementation. (2) Mid-processing contains adaptive vision algorithms for advanced visual analysis and object tracking. Mid-processing algorithms are often adaptive vision algorithms relying on machine-learning principles. In the results, the algorithms often pose immense communication demands to the system just for updating data models of the scene. The immense communication traffic is inherently caused by the algorithms themselves independent from implementation. For example, the communication traffic for accessing Gaussian parameters in MoG background subtraction is about 8 GB/s, or KLT optical flow demands for 2 GB/s memory access to update features’ histograms. (3) Post-processing algorithms are typically control-dominated algorithms focusing on intelligent decision making and high-level object analysis. The algorithms have fairly limited computation demands and are well suited for CPU execution.

Overall, the embedded realization of pre-processing and post-processing is mostly settled (pre-processing in HW and post-processing in SW). Innovations are highly needed for mid-processing...
CHAPTER 1. INTRODUCTION

with immerse communication traffic. Mid-processing algorithms pose tremendous challenges for
their embedded implementation: (a) a huge computation demand including conditional execution;
(b) large storage volume (and bandwidth) for keeping and updating an internal adaptive model; and
(c) simultaneous access to the internal model aligned with the incoming pixel stream. New solutions
are needed to make possible the embedded realization of mid-processing; simultaneously offering
high performance and low power.

1.2 Problem Definition

We observe a gap between market demands for advanced vision (adaptive vision algorithms)
and SoC realization capabilities. Among the tremendous challenges, the immerse communication of
adaptive vision algorithms is a major bottleneck. Other challenges appear when integrating multiple
vision algorithms in a single platform. Overall, we have identified four major challenges hindering
the embedded realization of advanced vision processing:

• 1- How to realize an individual adaptive vision algorithm in an embedded platform? The
  first challenge is how to efficiently manage and reduce the immense communication traffic per
  adaptive vision algorithm (mid-processing).

• 2- How to construct single larger vision flow in an embedded platform? Once the prob-
  lem 1 is resolved, the second challenge is how to efficiently compose a complete vision
  flow out of multiple adaptive (mid-processing) and non-adaptive (pre/post-processing) vision
  algorithms.

• 3- How to support many vision flows on same embedded platform? To cope with design
  and development cost, the third challenge is how to efficiently support many vision flows in a
  single platform.

• 4- How to manage architectural complexity as early as possible? Architecting a vision
  SoC for many applications is notoriously difficult and complex task. New design principles
  and methodologies are in high demand to manage the design/architecting complexity.
CHAPTER 1. INTRODUCTION

1.3 Dissertation Contributions

To properly address the identified challenges and open a path toward embedded realization of advanced vision processing, this dissertation:

- **Identifies and separates two classes of traffic (streaming and algorithm-intrinsic traffic):** The traffic separation provides opportunity to reduce and manage the tremendous communication traffic per adaptive vision algorithm. The traffic separation allows customizing algorithm-intrinsic traffic for a quality / bandwidth tradeoff and reducing the algorithm-intrinsic traffic with respect to algorithm quality (addresses problem 1). The traffic separation also opens a path toward embedded realization of a complete vision flow by efficient integration of multiple vision algorithms (addresses problem 2).

- **Introduces a novel processor class: Function Level Processor (FLP).** FLP is a novel processing class offering an architecture template for efficient execution of many vision flows simultaneously. FLP raises architecture programming abstraction from instruction-level (as in an ILP) to function-level. By coarsening the programmability, FLP achieves a comparable power/performance efficiency to custom-HWs. At the same time, FLP offers much higher flexibility than individual custom-HWs by providing a Function-Set-Architecture (FSA) and programmability to support many vision flows (addresses problem 3).

- **Proposes a set of higher abstraction levels called Conceptual Abstraction Levels (CALs).** CALs offer a systematic design methodology to tackle the architecting complexity of vision SoCs. CALs main focus is on identifying abstraction above system specification and resolving the system bottlenecks. CALs help system architects to identify the benefits of traffic separation or application candidates for function-level processing at early stages of design (addresses problem 4).

Overall, the aim of this dissertation is to open a new path toward embedded realization of advanced vision capabilities. This dissertation brings a new areas of research and exploration to both academia and industry by unifying algorithm exploration and architecture realization. Tremendous opportunities arise when jointly considering algorithm and architecture. Traffic separation and function-level processing already bring very positive benefits toward the realization of embedded vision. In a broader perspective, the principles of this dissertation is applicable to other challenging markets demand for high-performance low-power computing.
CHAPTER 1. INTRODUCTION

1.4 Dissertation Overview

The remaining of this dissertation is organized as following. Chapter 2 overviews the state-of-the-art work in embedded vision computing and highlights the differences of this dissertation from the previous work.

Chapter 3 introduces traffic separation in adaptive vision algorithms. Looking at the communication of adaptive vision algorithms (mid-processing), Chapter 3 identifies two types of communication traffics: streaming and algorithm-intrinsic traffic. Figure 1.5 illustrates these two traffic types. Streaming traffic is communication for accessing data under processing (e.g. image pixels, signal data samples) – inputs/outputs of the algorithm. Conversely, algorithm-intrinsic traffic, highlighted by red arrow, is a communication for accessing the data required by algorithm itself (algorithm-intrinsic data). To reduce the traffic demand of adaptive vision algorithms, Chapter 3 also proposes a separation between streaming traffic and algorithm-intrinsic traffic. The traffic separation allows customizing algorithm-intrinsic traffic for a quality/bandwidth tradeoff and reducing the traffic with respect to algorithm quality. It also simplifies synchronizes between streaming and algorithm-intrinsic data. Based on the traffic separation, Chapter 3 proposes a communication-centric architecture template to offer a practical solution supporting traffic separation and bandwidth/quality trade-off in the underlying architecture. Furthermore, the traffic separation is a major step toward efficient chaining of multiple vision algorithms constructing a complete vision flow. Chapter 3 demonstrates the efficiency of our solution by constructing a complete object detection/tracking vision flow (consisting of six nodes) processing 1080p 30Hz on Zynq platform. Our Zynq-prototyped solution performs 40GOPs at 1.7Watts of on-chip power.

Figure 1.5: Streaming vs algorithm-intrinsic traffic in adaptive vision algorithms

Chapter 4 introduces Function-Level Processor (FLP) which is a novel architecture template to efficiently support many vision flows in a single platform. FLP is the result of a joint academia and industry effort to bridge the flexibility / efficiency gap between ILPs and custom-HWIs. In comparison to ILPs, an FLP raises architecture and programming abstraction from instruction-level to a coarser
CHAPTER 1. INTRODUCTION

function-level, aiming for increased efficiency. Compared to an HWACC, an FLP increases flexibility by offering a finer grained programmability being constructed out of smaller, composable function blocks (FBs). The FBs then are efficiently chained together to construct the larger application. FLP (highlighted in Figure 1.6) provides an architecture template to efficiently manage FB composition for constructing bigger applications. By coarsening the programmability of architecture, FLPs nearly achieve the same power/performance efficiency as custom-HWs as well as enough flexibility to efficiently execute many vision flows. Chapter 4 demonstrates FLP benefits using an industry example of the Pipeline Video Processor (PVP) part of the ADI BF609. Chapter 4 highlights the flexibility by mapping ten embedded vision applications to the PVP. Overall, the FLP-PVP has a much lower power consumption over comparable ILP and ILP+custom-HW solutions (on average 14x-18x and 5x reduction respectively).

Figure 1.6: Function-Level Processor (FLP)

Chapter 5 proposes a set of higher abstraction levels called Conceptual Abstraction Levels (CALs) to manage the design / architecting complexity of embedded vision SoCs. CALs are a result of a joint industry / academia effort aiming to fill the gap between vision market requirements and the system specification. By identifying abstraction levels, their characteristics and necessary transition decisions, CALs expose essential system properties at higher levels of abstraction. With the guidance offered by CALs more efficient specifications become possible, ultimately yielding to a more flexible products with lower cost parameters such as power, bandwidth, and/or die area. Following CALs abstractions, system architects are able to define a more expressive specification model exposing algorithm choices, parallelism, and hierarchy more effectively. The resulting specification model serves as an input for a top-down ESL flow for further fine grained exploration, synthesis and implementation.
Chapter 2

Related Work

In this chapter, we review the state-of-the-art work in embedded vision computing, architectures for flexibility / power-efficiency and Electronic System Level (ESL) design flows. This chapter also highlights the differences of this dissertation from the previous work.

2.1 Embedded Vision SoCs

The limited power budget starkly restricts deployability of embedded vision applications, especially when considering Full-HD resolutions. Therefore, system architects move toward heterogeneous solutions combining embedded processors with specialized hardware accelerators. Developing customized solution for common frequently used vision algorithms is a predominant approach in both industry and academia [50,5]. Examples include ADI ADSP-BF60x [50] and TI DaVinci [5]. They offload compute-intense kernels into specialized hardware accelerators while control and high-level analytic execute on embedded processors. However, current solutions mainly focus on basic vision filters, e.g., Canny edge detection, with regular computation and communication patterns [118,92]. More advance vision algorithms such as KLT optical flow or MoG background subtraction with much higher complexity and irregularity of execution, have been left behind.

The existing solutions all share a general trend which is custom-HW implementations of frequently used pre-processing vision kernels and executing the remaining application parts on ILPs. In the current heterogeneous ILP+Custom-HW implementation, custom-HW IPs are integrated either as a co-processor or pre-processor into systems. In both models, custom-HW IPs are synchronized and scheduled by the host ILP. In the co-processor model, the just ILP has a direct access to system I/Os. Consequently, the custom-HW IP accesses the streaming data from system memory. Conversely,
CHAPTER 2. RELATED WORK

in the pre-processor model, custom-HW IP can directly access the system I/Os result in lower system-level traffic. To prototype a custom-HW realization, FPGA-based solutions are highly popular due to their comparable performance power efficiency. Compared to custom ASICs, FPGAs have slightly higher power consumption, 5x-12x \([60]\), but they are very efficient for prototyping as well as real deployment due to their much lower design and development cost.

2.1.1 Adaptive Vision Algorithms

Few researchers \([120][65][23][101][120]\) have targeted adaptive vision algorithms (e.g., MoG, KLT, optical flow) for embedded HW accelerators. All approaches do not separate traffic types, emitting everything into a common infrastructure (e.g. Network-on-chip (NoC) \([101]\) or customized \([120]\)). Thus, they either ignore the algorithm-intrinsic traffic, or assume it hidden in the hierarchy. In result, these HW accelerators are limited to very low resolutions \((300\times200 [120])\), while the market demands Full-HD or higher. Furthermore, existing HW vision accelerators are often organized as co-processors relying on frequent host ILP interaction for scheduling, synchronization and data transfers. This burdens the ILP with significant synchronization overhead and incurs unnecessary traffic for moving both streaming and operational data throughout system memory hierarchy. This overhead is exaggerated with adaptive vision algorithms due to alignment of intrinsic and streaming traffic.

2.1.1.1 MoG Background Subtraction

Only few MoG embedded realization have been proposed, all targeting FPGAs \([9][91]\). \([91]\) proposes a single stage MoG implementation on Xilinx Virtex-II Pro V2P30 FPGA. The architecture in \([91]\) avoids using the costly square root while accepting a significant loss in quality. Overall, by targeting frame size of \(320\times240\) at \(30\) frames per second, \([91]\) operates at \(27.65\)MHz with \(265\)mW power consumption. It seems however, that the reported result are estimations from the synthesis tool and it is unclear if the design has been executed on FPGA with functional validation and quality assessment. Using synthesis estimation for delays are not always reliable as routing delays are not considered yet.

In a different approach, \([9]\) demonstrates deployment MoG onto Xilinx Virtex-II XC2v6000. The proposed architecture includes 6 pipeline stages and support \(640\times480\) resolution. \([9]\) starts from MoG with single Gaussian and then extend the approach to two Gaussian components. \([9]\) also enhances MoG algorithm to improve quality in scenes light variations. However, similar to
CHAPTER 2. RELATED WORK

[91], [9] avoids the square root to reduce computation complexity. Furthermore, [9] does not report quality measurement to compare the quality of proposed implementation against the original MoG algorithm.

2.1.2 Bandwidth / Quality Trade-off

We look at the current approaches from two different perspectives: architecture view, algorithm view.

From the architecture perspective, many architecture solutions have been proposed for optimizing the communication traffic of streaming applications both in the context of Network-on-chip or customized communication-centric solutions. However, almost all of these approaches have a unified view of communication traffic and do not distinguish operational traffic from streaming traffic. In the result, their focus is mainly on optimizing streaming traffic such as exploring the communication properties (bandwidth, throughput, bridge buffers) of streaming channels or even compression of streaming traffic for the purpose of power saving [120] [101]. In particular, efficient streaming traffic management between multiple streaming nodes have been widely studied in context of Globally-Asynchronous Locally-Synchronous (GALS). However, they either ignore the operational data traffic or assume that operational traffic is hidden behind communication infrastructure. Consequently, they do not address challenges lie in Adaptive algorithms and limit their applicability to non-adaptive streaming applications where streaming data dominates system traffics.

Form the algorithm perspective, much research focuses on optimizing communication aspect of streaming algorithms with much lower demand for operational data access. A well-studied class of streaming applications are video coding algorithms (e.g., H.264) in which most focus lies on optimizing the computation and bandwidth of streaming traffic, encoding / decoding of streaming data [65][23]. Similarly, in embedded vision domain, current focus is on non-adaptive algorithms (e.g., Canny or Soble edge detection) with no demand for data exchange between frames [118][92] and thus minimize operational data access. In the result, current solutions cannot address ADAs algorithms (e.g., MoG, KLT, optical flow) are not addressable at high resolutions (e.g., Full-HD 1080*1920) because they do not provide a solution for managing and synchronizing the excessive access to operational data. The adaptive streaming algorithms are still implemented at a much lower resolution (300*200) [120] while market demand is already there for HD resolution.
2.2 Architecture Alternatives for Flexibility / Efficiency trade-off

Much research has been invested into utilizing hardware accelerators for building heterogeneous ILP+HWACC architectures offering design points along the efficiency/flexibility trade-off. Loop-Accelerator [57], Platform 2012 [71], VEAL [27], Streaming Accelerator [54], SARC Architecture [85], and BERET [44] are only few examples. Hybrid ILP+HWACCs have been designed addressing market demands, such as the IBM power EN for cloud computer systems [59], or Texas Instruments (TI) DaVinci platform [6] for vision/video processing. In general, these approaches demonstrate significant performance improvement and energy reduction over pure ILP solutions. However, in contrast to the FLP concept, the HW ACCs are considered as co-processors instead of autonomous processing elements. As such, they always depend on the main-ILP for operating on data streams. Furthermore, HW ACCs are mainly designed at application-level granularity where they can only perform a predefined kernel with limited configurability.

Few approaches touch on the idea of composing bigger applications out of many accelerators to further increase efficiency (basic architecture is highlighted in Figure 2.1). Accelerator-Rich CMPs [30] and Accelerator-Store [66] hint on the potential of accelerator chaining, but do not demonstrate how a composition can be performed to compose HW ACCs to an application. CHARM [31], which extends over [30], proposes that accelerators are distributed across the chip and exchange data through DMAs. However, such mechanism imposes a considerable traffic as well as requires on-chip buffers to realize accelerator data exchange.

2.2.1 Coarse-Grain Reconfigurable Architectures (CGRAs)

One approach classes that aim for both efficiency and flexibility are Coarse-Grain Reconfigurable Architectures (CGRAs) [46]. Figure 2.2 illustrates the basic architecture elements of Coarse-Grain Reconfigurable Architectures. CGRAs interconnect homogeneous compute units.
CHAPTER 2. RELATED WORK

(CUs) which are instruction-level programmable via a mesh or crossbar network. They offer spatially decomposed fine-grained parallelism. CGRAs aim to accelerate compute intense-loops, but are not intended to execute whole applications. Polymorphic Pipeline Array (PPA) [77] proposes hierarchical decomposition of CGRAs to expand the scope. [73], expands by offering both SIMD and MIMD parallelism on a 1D array of homogeneous CUs. In [78], a similar idea is explored with a 2D homogenous array of CUs targeting Algebra matrix operations. Overall, CGRAs and its extended architectures are promising approaches showing considerable performance improvements over ILPs. However, they have higher power consumption in comparison to HWACCs since they offer more flexibility. Their fine-grained interconnects also contribute to power consumption and cost limiting their scalability. Furthermore, these approaches are still dependent on a host ILP for management, scheduling and realization of a complete application.

Figure 2.2: Coarse-Grain Reconfigurable Architectures (CGRAs) [46]

2.2.2 Application-Specific Instruction Processors (ASIPs)

Application-Specific Instruction Processors (ASIPs) also aim for efficiency and flexibility. Examples include Tensilica [48] and the Software-Configurable Processor (SCP) proposed by Stretch [112]. ASIPs offer special instructions for mapping dedicated C-functions to specific custom-HW units inside the processor pipeline at a level from single operations to complex functions [112]. Figure 2.2 presents an Application-Specific Instruction Processors (ASIPs) architecture.

An efficient integration of ASIPs and CGRAs has been presented in DYSER [42]. DYSER [42] proposes a hybrid CGRA/ASIP approach proposing a CGRA-like structure integrated into an ILP’s pipeline. It allows to customize the data-path according to application needs. With a different angle, QSCORE tiles HWACCs to an ILP core [112], for delivering power-efficiency for most recurring traces of general-purpose programs including OS kernel. Similarly, the approach in [111] tiles a coarse grain programmable pipelined co-processor with an ILP offering power efficiency and flexibility for Software Defined Radio (SDR) protocols. In general, ASIPs can provide higher efficiency.
CHAPTER 2. RELATED WORK

Figure 2.3: Application-Specific Instruction Processors (ASIPs)

with fairly good flexibility. They can work on limited code size while they still depend on instructions and suffer from the limitation and in-efficiency of ILPs.

2.3 System Specification Development

A system specification is often considered the entry of Electronic System Level (ESL) design flows. A system specification can be seen as a set of behavioral models, captured by Model of Computations (MoCs) describing the system functionality and abstract communications. System Level Design Languages (SLDLs) such as SystemC and SpecC have been proposed to capture system specification under a certain Model of Computation (MoC) as well as expressing virtual platforms. In order to address specification challenges of heterogeneous platforms, SLDLs have been also extended to capture multiple MoCs at same time. Two examples are HetSC and HetMoC which are extensions over SystemC to capture different MoCs in a unified model.

Current ESL design flows mainly focus on system-level exploration starting from a given specification model (captured by an SLDL). ESL methodologies can be roughly divided into three different categories: free design, Platform-Based Design (PBD), and hybrid design. Free design flows cover three major steps: platform architecture allocation, application mapping and also scheduling following by HW and SW synthesis flows. In contrast, in PBD, HW components and software APIs have been already identified. In result, the exploration mainly focuses on application mapping and system scheduling rather than identifying desired architecture platform. The third, hybrid category, can handle cases where a portion of the solution is based on a given platform while the remaining is considered free design. Also industry approaches aim to improve visualization and raising the abstractions of design. Tool-chains such as Synopsys Visualizer, Cadence Virtual System Platform, and COMPLEX enable designers
CHAPTER 2. RELATED WORK

to develop extensible virtual platforms at multiple abstraction-levels (TLM and cycle accurate). These tool-chains offer great potential for early virtual platform development and SW validation. However, they rely on an executable system specification, and are bound by the decisions implicitly buried in the executable system specification (including algorithm selection/exploration, behavior decomposition and parallelism definition). The work in this article aims to identify the necessary steps hot to obtain an executable system specification.

2.3.1 System Requirement Capturing

With the increasing complexity of heterogeneous MPSoCs and the large diversity of requirements, capturing all system properties is a challenging process. Modeling languages have been utilized to facilitate the process of capturing system properties. Unified Modeling Language (UML), as an example, can capture system structure including decomposition, hierarchy and data types at early design stages. Standard profiles over UML, such as SysML [15] and MARTE profile [113] also provide primitives to capture high-level system requirements and resource constraints. However, rather than proposing a design methodology for system-level design, UML profiles can be applied for describing an engineering model.

UML-based notations are widely used to capture system properties in Model-Based System Engineering (MBSE) [38]. For example, the System Engineering Toolbox helps system engineers to capture a Holistic Requirements Model (HRM) and Functional Model (FM) of the planned system [13]. In general, system engineering tools and methodologies can be applied to capture the design cycles of any engineering process. They inherently share some of the features of our proposed CALs approach (e.g., requirement capturing, functional and technical requirements, representing system functions as a stat flowchart) and can be used to capture some of the CALs abstraction levels. Missing are however, SoC-oriented design flow and methodologies to explicitly identify and expose the essential design decisions and characteristics for architecting market-oriented MPSoCs.

To further expand the breadth of captured system properties, recent work combines multiple UML profiles [75, 86, 87]. For example, [86] proposes an integration between SysML and SystemC UML to realize a transitioning from a platform independent model to a platform dependent model. Although, abstracting system behavior and requirements by UML profiles is a step to simplifying the design space exploration, still [75, 86, 87] do not touch the main challenges above system specification. In these approaches, the main focus is automatic generation of platform dependent model out of a given available architecture resources, requirements, and system functionality captured.
CHAPTER 2. RELATED WORK

by different UML profiles. Thereby, areas such as algorithm exploration, algorithm /architecture co-design and tuning are still insufficiently supported. Nonetheless, UML and dedicated profiles could be employed to capture individual CALs abstractions.

2.3.2 Abstractions Above Specification

Few system design approaches investigate abstractions above the system specification. Some mainly focus on exploring parallelism and composition opportunities within an existing system specification [21][89][95]. [21] realizes a user-guided automated translation from a sequential specification code in C to a flexible parallel specification. [89][95] also starts with pure functional specification model followed by separate explorations for decomposition and parallelism leading to more structural specification captured by a SLDL. These approaches are orthogonal to our abstraction definition, were CALs also focus more on the definition of levels and identification of decision making processes.

A group of approaches [12][81][70] more purely focus on the algorithm development and exploration. Matlab/Simulink offers a model-based design environment for development, exploration and analysis of algorithms [70]. Simulink models are highly algorithm-oriented with less focus on architecture implications such as performance / memory, and heterogeneous components [61]. To include heterogeneous concepts, [12][81] link algorithm development environment and system-level design in which a Simulink model serves as an input. However, in the system-level context the challenge of algorithm selection [82] has to increasingly take the architecture suitability into account.

The focus on algorithm selection and evaluation becomes more pronounced in markets (such as embedded vision computing) where a wide diversity of algorithms exist for individual applications.

The work presented in [58] has conceptual similarities to CALs. The flow in [58] starts with a paper architecture containing initial design ideas, Back-of-the envelop and estimation model, exploring design decision above specification (Figure 2.4 highlights the flow in [58]). The outlined approach is simplified, as architecture exploration limits itself to evaluating instances of a defined architecture template. In addition, [58] only focuses on early performance evaluation of architecture instances, while many other decision aspects such as algorithm exploration, computation granularity and architecture flexibility are not considered. Furthermore, the transitions between abstractions, as well as their implications and potentials are not as clearly described.
2.4 Chapter Summary

Altogether, current approaches share a problem in common that is almost ignoring operational data traffic. In fact, optimizing the algorithm-intrinsic data has received less attention despite being crucial for real-time low-power implementations. In contrast, one of the main focus of this dissertation is on optimizing operational traffic by target adaptive vision algorithms. Our FLP architecture differs from current approaches in multiple aspects. First, rather than accelerating individual kernel applications by dedicated HWACCs, we propose set of composable Function-Blocks that can be chained to realize a set of applications within market. Secondly, we propose an architecture solution for cascading FBs to avoid unnecessary data movements, on-chip buffers and DMA transfers. Third, our proposed FLP can operate autonomously, meaning that an FLP configured to execute and schedule applications independently from ILPs. Looking at ESL design flows, many academic and industrial approaches, system level exploration starts from an executable specification model where many system properties including requirements, functionality, algorithms, behavior decomposition and parallelisms have been already identified. In contrast, our aim with CALs is to explore areas above system specification model and explicitly expose steps toward obtaining an executable system specification. As such, the output of CALs could feed as input to an existing ESL design flow.
Chapter 3

Traffic Separation in Adaptive Vision Algorithms

This chapter at first focuses on managing the massive communication traffic of adaptive vision algorithms. The proposed solution is primarily based on separation between streaming and algorithm-intrinsic communication traffic. The chapter then expands the principle to construct a complete vision flow out of multiple vision algorithms.

3.1 Demand for Communication-Centric Design

With the progress in High-Level Synthesis (HLS) many options exist for realizing custom computation in hardware streamlining the process over a handcrafted hardware design [56]. Conversely, the communication aspect has been left as a challenge in particular with high-volume traffic hitting the memory hierarchy under bandwidth and latency constraints, as well as considering its significant contribution to the system power. Hence, more attention is needed for optimizing communication, shifting the attention toward communication-centric design principles.

Embedded streaming applications targeted to hardware have been limited to simpler streaming kernels (e.g., convolution, FIR filter). For these kernels, streaming traffic dominates the communication. Furthermore, algorithm-intrinsic data is small (e.g., gradients in convolution) which can be kept in on-chip buffers. With the computational advances in embedded platforms, realizing more complex streaming applications such as adaptive vision algorithms becomes more of a focus. Adaptive vision algorithms are based on machine-learning algorithms (e.g., Mixture of Gaussian (MoG), Support Vector Machines (SVM)) which track the stream data through a continuously updated
CHAPTER 3. TRAFFIC SEPARATION IN ADAPTIVE VISION ALGORITHMS

model for realizing the functionality (e.g. background classification). Because of the adaptive nature and can tackle more complex tasks (e.g., object detection, tracking, classification) at high quality and therefore have a great potential for many applications.

Adaptive vision algorithms have been studied and realized at an abstract level (e.g. Matlab). However, their embedded realization is very challenging due to challenges induced by the frame model (large traffic, synchronization) and more complex computation. Adaptive vision algorithms pose significant challenges to an embedded realization: (a) larger volume of algorithm-intrinsic data and (b) strict synchronization requirements. In many cases, algorithm-intrinsic data is many times larger than streaming traffic. This results in a high bandwidth demand and power consumption; making their embedded realization at high resolution almost impossible. Secondly, algorithm-intrinsic data (e.g. the stream model) needs to be synchronized with the streaming data. As stream samples pass through the node, their corresponding model data has to be delivered and updated. Existing approaches do not address these challenges and in result do not realize adaptive vision algorithms in hardware, or are limited to very small resolutions (contrary to market desire). Therefore, novel solutions are required enabling efficient embedded realizations of adaptive vision algorithms.

In this chapter, we show an approach to tackle the challenges of adaptive vision algorithms based on the example of Mixture of Gaussians (MoG) background subtraction. To cope with the significant communication demands, this chapter proposes a separation between algorithm-intrinsic and operational traffic. Following the traffic separation, this chapter also proposes a communication-centric architecture template with two key features: (1) algorithm-intrinsic traffic from streaming traffic, and (2) autonomous control and synchronization. The traffic separation (1) allows customizing algorithm-intrinsic traffic for a quality/bandwidth tradeoff, and in addition simplifies interconnecting multiple heterogeneous nodes. The autonomous control (2) synchronizes between streaming and algorithm-intrinsic data independent from the host ILP, and thus minimizes the system-level synchronization overhead. In effect, our proposed solution operates independently (as a peer processing element) on streaming pixels in parallel to a host processor.

3.2 Traffic Separation in Adaptive Vision Algorithms

Vision algorithms can be roughly divided into two classes: (1) filter-based and (2) adaptive. (1) filter-based algorithms (e.g. convolution, Canny or Sobel edge detection, Harris corner detection) keep limited algorithm-intrinsic data (e.g. few hundred bytes for convolution) and mostly focus on one frame at a time with very limited interaction across frames. In contrast, adaptive algorithms
mainly work across frames, often based on machine-learning algorithms (e.g. Mixture of Gaussian (MoG), Support Vector Machines (SVM)) which track frames through a continuously updated model (e.g. background classification, motion detection). The frame model is often very large (e.g. 248 MB for MoG) causing significant update traffic across frames, but nonetheless allows for more complex tasks (e.g., object detection, tracking, classification) at high quality.

We observe two type of traffic: streaming traffic and algorithm-intrinsic traffic. Figure 3.1 generalizes these types for adaptive vision algorithms, separation between streaming data and algorithm-intrinsic traffic. **Streaming traffic** is data under processing (pixels in case of vision) and is typically read from input ports and written to output ports or system memory. Streaming traffic deals with input/output of a module independent of the algorithm selected for realizing the functionality. Conversely, **algorithm-intrinsic traffic** is due to data used for realizing the algorithm (e.g. kernel density histogram or Gaussian parameters). While different algorithms may achieve the same functionality, they may use vastly different internal data structures causing different algorithm-intrinsic traffic. For background subtraction, the streaming data consists of gray pixel and BG/FG mask. The algorithm-intrinsic data for MoG are Gaussian parameters. A different algorithm, e.g. Mean Shift [28] requiring a complete history of N-frames, would produce different algorithm-intrinsic traffic.

Figure 3.1: Streaming vs algorithm-intrinsic traffic in adaptive vision algorithms

The algorithm-intrinsic traffic volume of many vision applications, particularly in adaptive vision algorithms, is dominant and dwarfs the streaming traffic (e.g. 60x in MoG or 8x in component labeling). This poses significant bandwidth and consequently power challenges, sometimes even rendering an implementation infeasible. In MoG, 65% of system on-chip power and more than 90% of overall power (combined on-chip and off-chip memory access) is consumed by algorithm-intrinsic data accesses. Separating traffic types enables trading off quality for bandwidth independent of the streaming pixels. This requires identifying the traffic in the specification, as well as architecture
support. Therefore, the underlying hardware architecture has to offer traffic separation optimizing
the memory bandwidth as well as managing system-level traffic based on the role and nature of data
on computation.

3.2.1 Applications Examples

We observe that the concept of traffic separation is valid for different mid-processing vision
algorithms. To demonstrate the generalization of traffic separation, we have picked four adaptive
vision algorithms: (1) Mixture-of-Gaussians (MoG) Background subtraction, (2) component labeling,
(3) Mean-shift object tracking and (4) Kanade-Lucas-Tomasi (KLT) optical flow.

Figure 3.2 highlights the traffic separation for the different adaptive vision algorithms. For MoG
background subtraction, the algorithm-intrinsic traffic dedicated to Gaussian parameters
keeping the track of the background model per individual pixels (highlighted in Figure 3.2a). For
component labeling, the algorithm-intrinsic traffic belongs to component label for keeping the
connection information among the FG pixels identified by ForeGround (FG) mask (highlighted in
Figure 3.2b). For mean-shift object tracking, the algorithm-intrinsic traffic is caused by modeling
and updating histogram data for individual objects (highlighted in Figure 3.2c). For KLT optical
flow, the algorithm-intrinsic traffic is the feature signature models which kept updated per individual
features across the frames under processing (highlighted in Figure 3.2d).

Figure 3.2: Example of traffic separation in adaptive vision algorithms
3.2.1.1 Mixture of Gaussian (MoG) Background Subtraction

Among the introduced adaptive vision algorithms, we use Mixture of Gaussian (MoG) background subtraction for the further explorations. Mixture of Gaussian (MoG) is widely used for background subtraction in fixed camera position. It offers a very good quality and efficiency in capturing multi-modal background scenes \[25\] \[96\]. Each pixel’s background value is modeled by three to five Gaussian Components (distribution). Each Gaussian component has its own set of Gaussian parameters: weight $\omega_{i,t}$, an intensity mean $\mu_{i,t}$ and a standard deviation $\sigma_{i,t}$. With a new pixel coming in, all Gaussian parameters are updated at frame basis to track BG changes of the pixel. Each Gaussian has an individual learning factor, adjusting the speed at which the background model is updated. If none of the Gaussians sufficiently matches the new pixel’s value, that pixel is declared as a foreground. Otherwise, the pixel is a background.

![Memory Access per Pixel in MoG algorithm](image)

Figure 3.3: Memory Access per Pixel in MoG algorithm

To further guide understanding of the MoG background subtraction algorithm, Algorithm 1 outlines the reference MoG algorithm (see further detail in \[24\]). The algorithm loops through all pixels in the frame (lines 2). For each pixel, the algorithm first classifies the pixel’s Gaussian components into match or non-match components (line 6). A component matches, if the component’s mean is within a match threshold $\Gamma_{FG}$ of the current pixel value. Gaussian parameters are updated based on match classification. In case that no match exists (line 16), algorithm creates a new Gaussian component, called virtual component replacing the Gaussian component with smallest weight value (line 17). Then, the components are ranked and sorted based on their weight over standard deviation ratio (line 19). Starting from the highest rank component, the algorithm declares a pixel to be background if its weight is less than the FG threshold $\Gamma_{FG}$ and the updated mean over updated sd is less than match threshold $\Gamma_{match}$ (line 23). When finding the first match, the comparison stops and the algorithm continues with the next pixel. MoG demands significant computation with many conditional statement (if-then-else) for calculating the foreground state of an individual pixel.
Algorithm 1 MoG Algorithm

1: function MoG (in Frame, inout Gaussian, out Foreground)
2: for $i = 0$ to numPixel do
3:     match = 0
4:     for all gaussian do
5:         $diff[k] = abs(m_k - pixel)$
6:         if $diff[k] < \Gamma_{match}$ then \(\triangleright\) Match
7:             update $\{w_k, m_k, sd_k\}$ based on $\alpha$
8:             match = 1
9:         else \(\triangleright\) non-Match
10:             update $\{w_k\}$
11:         end if
12:     end for
13:     for all gaussian do
14:         normalization $w_k$
15:     end for
16:     if !match then
17:         Create virtual component for the smallest $w_k$
18:     end if
19:     for all gaussian do
20:         Rank and Sort gaussian based on $w[k]/sd_k$
21:     end for
22:     Foreground = 1
23:     for $k = \text{HighestRank}$ to 0 do
24:         if $w_k \geq \Gamma_{FG}$ && $diff[k]/sd_k < \Gamma_{match}$ then
25:             $\text{Foreground} = 0$
26:             break
27:         end if
28:     end for
29: end for
30: end function

3.2.2 Quality / Bandwidth Trade-off

We can roughly divide the algorithm quality contributors into two orthogonal axes: computation precision, and communication precision. A lot of research has been already published exploring the quality for computation aspect of design, optimizing width size of arithmetic operations [22]. In contrast to computation, communication and memory access has much higher contribution in power consumption (more than 70%) and may less contribution in quality. Therefore, one feasible possibility is to reduce the precision (bit-width) of communication data while keeping the computation accuracy at highest precision. Such an approach potentially lead to much more power saving with some degree of quality loss, introduces a trade-off between the quality and memory access.
CHAPTER 3. TRAFFIC SEPARATION IN ADAPTIVE VISION ALGORITHMS

demand (bandwidth).

![Figure 3.4: Orthogonal impact factors to power in quality.](image)

The exploration between quality and memory access demand is a significant step toward architecture realization of adaptive vision algorithms. First of all, it helps us to better understand the role and nature of traffic (streaming vs. operational) at higher level of abstraction before actual architecture realization. Secondly, it helps us to translate the application-level quality measurement to an architecture realizable term. In following, we explain these two aspects.

To reduce the memory bandwidth demands for accessing operational data, we propose a shift in quality exploration form computation properties of the design to communication aspect. The traffic separation between streaming data and operational (algorithm-intrinsic data) enables the exploration of operational data demand on quality independent from streaming traffic. However, even algorithm developer unifies in two sorts of traffic. Therefore, at the first step we need to separate these traffics at application-level.

Figure 3.5a highlights the conceptual separation between the streaming and operational data, capture in a High-Level System Design Language (SLDL). As highlighted, streaming and operational data have their own dedicated channels. To explore the effect of operational data on quality, we propose to add support blocks, we call them precision adjustment, upon operational data read / write access (highlighted in Figure 3.5a). Assuming that computation is optimized for 32 bits operational data precision can reduced to any range between N bits to 32 bits before memory write. Upon read, the operational data are transformed back to 32 bit for processing. For simplicity at this stage, we compress through discretization focusing on the most significant bits. The precision adjustment for operational data introduces a trade-off between the quality of algorithm and operational data bandwidth.

We have realized precision adjustment for the example of MoG algorithm. Figure 3.5b presents the result of exploration as a trade-off bandwidth per frame (1080*1920 resolution) on
CHAPTER 3. TRAFFIC SEPARATION IN ADAPTIVE VISION ALGORITHMS

(a) MoG Specification with precision adjustment components

(b) Quality over bandwidth trade-off

Figure 3.5: Bandwidth quality exploration for Gaussian parameters

the x-axis and quality (MS-SSIM) on y-axis. For quality assessment and quantifying the quality degradation due to the data compression we choose MS-SSIM \[^{115}\]. MS-SSIM is a very good indicator for visual quality focusing on structural similarity rather than a direct pixel comparison approach. Based on MS-SSIM, we have exhaustively explored quality and required bandwidth for the MoG algorithm. We evaluate quality by comparing against a ground truth obtained from the reference algorithm (32-bit fixed point operation and 480-bit operational data storage per frame pixel). As highlighted by Figure 3.5b, the quality is improved by increasing the (operational data) bandwidth per frame. The exploration has been done across different bit widths as well as possible discretionary per operational data access (Gaussian parameters in MoG).

Figure 3.5b reveals two important points. First, there is a possibility to transfer operational-data at lower precision and still maintain 100% quality (e.g., for MoG 244-bits per pixel is enough for the maximum possible quality). Secondly, identical bit-width sizes with different bit discretizations lead to different quality. The Pareto curve (red-line) in Figure 3.5b demonstrates the best achievable quality for an individual quality / bandwidth point (e.g., 70-bits per pixel can maximally reach 0.64 quality).

With quality/bit-width exploration in place, we introduce the term of Operation-bit per Streaming-sample (Ob/Ss) as an impact factor to quality. The term of Ob/Ss helps us to translate the quality number which presents in percentages at application level to realizable architecture terms (bits). This is a step toward realizing an architecture solution for adaptive vision algorithms, as architecture can manage the operational traffic based on their corresponding Ob/Ss metric. In the example of MoG, 74 [Ob/Ss] and 244 [Ob/Ss] refers to qualities of 0.64 and 1, respectively.
CHAPTER 3. TRAFFIC SEPARATION IN ADAPTIVE VISION ALGORITHMS

3.2.3 Communication-Centric Architecture Template

Current related work lacks support of adaptive vision algorithms with significant algorithm-intrinsic traffic. They either avoid this class of applications, or intermix the traffic types and consequently being limited to very small resolutions. To overcome this gap, we propose a communication-centric architecture template that (a) provides a framework for algorithm-intrinsic data access, (b) resolves data alignment between streaming and algorithm-intrinsic traffic, and (c) offers design options for trading bandwidth against quality.

Figure 3.6 outlines the essential components of our architecture template. It consists of two clock domains: computation domain and communication domain. The computation clock is driven by streaming data (pixels) clocking the adaptive vision kernel. The communication clock is set by the bus/interconnect for accessing operational data. A different design choice is to unify both clock domains, and re-time the input stream. However, the separation between computation and communication clock aids in efficiently managing the traffic of operational data access independently.

Streaming and algorithm-intrinsic data access are separated by using individual ports. Streaming data most efficiently enters the system through a system interface (e.g. HDMI from a camera). This avoids CPU interaction with the traffic. This also simplifies chaining across multiple instances, as streaming traffic can be directly forwarded without hitting the system memory. Our design uses this direct connection for receiving gray pixels from RGB2Gray model (see later introduced Figure 3.16a).

Using the system memory for algorithm-intrinsic data is unavoidable due to its volume (up to 248 MB for MoG). Dedicated DMA channels continuously read/write back algorithm-intrinsic data.

Figure 3.6: Communication-centric architecture template for adaptive vision algorithms
CHAPTER 3. TRAFFIC SEPARATION IN ADAPTIVE VISION ALGORITHMS

data at a data frame. DMA channels operate in parallel but synchronized to preserve correct read after update sequence. DMA channels connect through system interconnect to the memory interface. The DMA channels operate in circular mode (auto repeat) to restart with each frame independent of the host. This eliminates unnecessary synchronization with the processor, freeing up computation cycles, e.g. for a downstream application.

By using separate ports, streaming and algorithm-intrinsic data are transferred in parallel, but need a tight alignment. With each incoming pixel, the according pixel’s model data (i.e., algorithm-intrinsic data) is required at the same time. Any misalignment will make the algorithm fail due operating on the wrong model. Initial alignment is achieved by correctly configuring the DMA’s start address (pointing to the model in memory). However, to guarantee continues alignment, with each new pixel its model data needs to be delivered without any interruption. Our architecture template maintains alignment using Async. FIFOs that (1) bridge the clock domains and (2) compensate for burstiness of bus traffic.

![Figure 3.7: Precision adjustment](image)

In result of the findings in Section 3.7, precision adjustment blocks reduce the bandwidth requirement by re-size algorithm-intrinsic data (Gaussian parameters) before being delivered to/from the vision algorithm. With the simple focus on MSB, their implementation is straight forward, yet has a profound effect on system performance.

In current heterogeneous systems, the host processor is responsible for controlling of vision accelerators imposing a considerable load to the host processor while potentially leading to a low accelerator utilization. To avoid this overhead, the architecture template employs a Control Unit (CU) to minimize or even eliminate the need for host processor interaction. The CU offers set of Memory-Mapped Registers (MMRs) to the outside for initializing and configuring the vision processing. In our architecture template, software is only responsible to initialize the DMAs and CU’s MMRs. After configuration-initialization, the architecture template executes independently.
CHAPTER 3. TRAFFIC SEPARATION IN ADAPTIVE VISION ALGORITHMS

from the host processor on many number of frames. The CU responsibility includes controlling of
data alignment between the streaming and algorithm-intrinsic, quality adjustment in precision units,
keeping read/write DMAs synchronized.

Our architecture template offers a set of configurable knobs (design choices) to the design-
ers. The design choices are Async. FIFO depth, DMA inline buffers as well as DMA channels and
communication bus/interconnect width and frequency. The template has some limitation, as well.
We assume same width size between bus, inline buffers and FIFO width. Given a desired quality
and bandwidth, as well as the interconnect parameters, the architecture template can be properly
dimensioned.

3.2.4 System Integration

We consider pairing of our architecture template with other streaming cores (either in SW
or HW) to realize larger applications such as object tracking vision flow. This utilizes combined
strengths of high throughput low-power execution of compute-intense adaptive vision processing
(e.g., MoG background subtraction) in hardware, while a processor offers top-level adaptive control
and intelligence (e.g. for tracking objects across frames).

Traditionally, vision kernels have been implemented in HW as accelerators, i.e. a co-
processor which is called from and synchronized by a host processor. Figure 3.8a illustrates an MoG
coprocessor similar to [9] (assuming that adaptive vision algorithm is MoG background subtraction).
In the co-processor arrangement both streaming traffic (green) and algorithm-intrinsic traffic (red)
occupy the interconnect and system memory. Pixels are received by the processor and forwarded to
memory (In⇒Mem (1)). After being triggered by the processor, MoG reads the frame from memory
(Mem⇒MoG (2)), starts the background subtraction processing. By finishing the background
subtraction, and writes back the FG mask to memory (MoG⇒Mem (3)). Finally, the FG mask is read

![Figure 3.8: System integration options.](image-url)
CHAPTER 3. TRAFFIC SEPARATION IN ADAPTIVE VISION ALGORITHMS

by the processor for post-processing, or directly forwarded to the output port (Mem⇒Out (4)). In total, 4 transfers are necessary (In⇒Mem⇒MoG⇒Mem⇒Out) buffered through the system memory. The cycle repeats for each frame. All transactions are scheduled by the processor, leading to high overhead and consequently an inefficient solution. Concurrently to streaming, algorithm-intrinsic data (Gaussian parameters) hits the system memory as well, creating contention and increasing bandwidth demands.

A more efficient solution is a peer-processor arrangement enabled by our architecture template (see Figure 3.8b). The architecture template has direct access to system I/O interfaces for input and output of data without requiring constant processor interaction. In addition, streaming nodes can be chained as illustrated by RGB2Gray. Separating streaming and algorithm-intrinsic data allows keeping the streaming data on-chip without the costly memory interaction (and enables streaming node chaining). Only algorithm-intrinsic traffic (after precision adjustment based on quality constraints) hits the system memory. The host processor only performs first initialization, after which RGB2Gray and MoG operate completely independently, eliminating the synchronization overheads. Hence, more cycles are available on the processor when receiving FG mask for higher level processing.

3.3 Case Study on MoG Background Subtraction

For the MoG computation realization, we focus on a manual RTL implementation guided by the system-level specification. Alternatively, High-Level Synthesis (HLS) tools, e.g., Xilinx Vivado, could be employed. They are promising especially when a system specification model is available. However, compared to hand-crafted design, HLS are typically less efficient. Furthermore, for the case of MoG targeted at FULL-HD resolution, HLS tools are unable to meet the timing requirements for 148 MHz clock frequency. In the result, we chose the hand-crafted approach using Verilog HDL to capture our RTL model.

In order to realize our proposed hand-crafted MoG solution, we follow a system-level design flow throughout the entire design process. The ESL flow to tackle the design complexity of MoG background subtraction dividing it into manageable steps. Figure 3.9 captures the essence of our design flow starting from MoG specification down to actual execution. Individual MoG models (green background color) include: Specification, RTL, Implementation and Execution models. In between, the refinement/synthesis tools (or manual design) realize MoG in the next lower abstraction
level. For each level, validation tools (yellow background color) validate design properties including functionality, quality and timing.

The Specification model is the reference design model reflects all system-level design decisions and properties including output quality. The RTL model captures our hand-crafted MoG RTL design which we derived from the specification model. Following that, Implementation is the result of synthesis and mapping tool-chain exposing the all resource allocation and timing properties of the design. Finally, MoG execution reflect the properties of real MoG execution on Zynq platform. The transition from Specification to RTL occurs through implementation (an automated approach utilizing High-level Synthesis remains future work). The Xilinx ISE 14.4 tool-chain is used for FPGA implementation synthesis. Quality validation is based on MS-SIM. For RTL functionality validation and timing validation we use ISE ISIM and Timing Analyzer respectively. For validation of FPGA execution, we have developed our own tool operates on top of collected run-time data from chip-scope. The next subsections describe the individual levels in more detail.

Figure 3.10: MoG specification model including coarse-grained parallelism.

Figure 3.9: MoG design flow
CHAPTER 3. TRAFFIC SEPARATION IN ADAPTIVE VISION ALGORITHMS

Deriving from our specification model which captured the coarse-level parallelism (Figure 3.10), we iteratively refine the MoG RTL model to meet the timing properties of the design. The first stage of RTL design is a direct translation of specification model into a behavioral RTL model. The initial RTL has only three pipeline stages with parallel execution of Gaussian updates and operates at only 9 MHz (far away from the target of 148 MHz). To realize Full-HD MoG processing, we explore three different optimizations: Algorithm Tuning; (2) Operation Width Sizing; (3) Deep Pipelining. At Algorithm Tuning stage (1), we modify the algorithm to match it better to parallel HW execution. As an example, we replace ranking and sorting with parallel checking of Gaussian components. At Operation Width Sizing stage (2), we identify the optimal quality / width point for high-latency arithmetic (SQRT, divide and multiply). At Deep Pipelining stage (3), we further break down individual Coarse-level pipeline stages into the finer micro-pipeline stages to meet the timing requirements.

Figure 3.11: MoG computation micro-pipeline stages.

Figure 3.11 shows the extended macro pipeline stages for MoG RTL model including: (1) Gaussian Match Detection; (2) Weight Update; (3) Mean Update; (4) SD Update; (5) Weight Normalization; (6) Virtual Component; (7) Foreground Detection. In the first four pipeline stages, all three Gaussian components execute in parallel.

Figure 3.12 outlines the essential components of MoG realization on our architecture template. It consists of two clock domains: computation domain and communication domain. The computation clock is driven by streaming data (pixels) clocking the adaptive vision kernel. The communication clock is set by the bus/interconnect for accessing operational data. A different design choice is to unify both clock domains, and re-time the input stream. Streaming and algorithm-intrinsic data access are separated by using individual ports. Streaming data most efficiently enters the system through a system interface (e.g. HDMI from a camera). This avoids CPU interaction with the traffic. This also simplifies chaining across multiple instances, as streaming traffic can be directly forwarded without hitting the system memory. Our design uses this direct connection for receiving gray pixels from RGB2Gray model.
CHAPTER 3. TRAFFIC SEPARATION IN ADAPTIVE VISION ALGORITHMS

3.4 Experimental Results

To demonstrate the benefits of our approach, this section describes an instance of our architecture template running MoG on Xilinx Zynq platform. It evaluates performance, power consumption, and as resource utilization, as well as highlights advances over the closest related work.

3.4.1 Quality / Bandwidth Exploration

3.4.1.1 MoG Parameters exploration

For quality assessment, we use MS-SSIM as a good indicator for visual quality focusing on structural similarity rather than direct pixel comparison approach. We evaluate quality by comparing against a ground truth obtained from the reference algorithm (32-bit fixed point operation and 32-bit storage for each Gaussian parameters). We investigate three scenes: simple an indoor shopping center scene with limited motion, medium a subway station and complex a busy outdoor scene of a street corner. Table 3.1 shows a sample frame of each scene. We characterize scene complexity based on the number of moving objects, illumination, shadows, and many other details. Table 3.2 summarizes the features of each scene. The most challenging scene is San Francisco Street with variations from multi-modal changes in background to different kind of moving object at foreground as well as frequent changes in illumination.

Figure 3.12: MoG communication overview.
CHAPTER 3. TRAFFIC SEPARATION IN ADAPTIVE VISION ALGORITHMS

Table 3.1: Scenes under the test

<table>
<thead>
<tr>
<th>Scene Type</th>
<th>Scene Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Simple</td>
<td>(Shopping Mall)</td>
</tr>
<tr>
<td>Medium</td>
<td>(Subway Station)</td>
</tr>
<tr>
<td>Complex</td>
<td>(SanFrancisco Street)</td>
</tr>
</tbody>
</table>

Table 3.2: Estimated resource utilization of Mixture of Gaussians (MOG) algorithm with the frame rate of 30 Hz.

<table>
<thead>
<tr>
<th>Data Set</th>
<th>BG Changes</th>
<th>FG Changes</th>
<th>Variation Changes</th>
<th>Object Changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Simple</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>Medium</td>
<td>no</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>Complex</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
</tr>
</tbody>
</table>

We systematically analyze and explore the impact of parameter bit-width (individually for mean, weight, and deviation) as well as number of Gaussian to the quality of background subtraction for all scenes. In this subsection, we only present part of the exploration focusing on the complex scene.

Figure 3.13a and Figure 3.13b highlight two exploration examples for the complex scene. Figure 3.13a shows the quality correlation between # of Gaussian components and length of Deviation. The other parameters (Mean and Weight) are kept at maximum length (32-bit).

Increasing # of Gaussian components from 3 to 5 linearly improves quality due to the high scene complexity. This confirms that 5 Gaussian components are required in the complex scene. They track the multi-modal changes in background including variations in illumination and waving leaves caused by wind. In order to reduce memory bandwidth, we tune the width of individual Gaussian parameters. Figure 3.13a also shows that storing Mean with 16-bits is sufficient. No quality improvement is achieved by storing the Mean with more than 16-bits. With 5 Gaussian components and 16-bit Mean in place, Figure 3.13b presents the effect of Weight and Deviation width on output quality. It is characteristic that a plateau is reached early. 16bit Deviation as well as 14bit Weight
already achieve the best quality.

Figure 3.14a and Figure 3.14b explore the same dependencies for the medium scene (subway station). Figure 3.14a shows that a considerable quality improvement is achieved by increasing number of Gaussian components from 3 to 4. However no further quality improvement occurs when increasing to 5 Gaussian component. Similarly, the quality improvement saturates at 14bit Deviation and 12bit Weight as shown in Figure 3.14b).

By reducing parameter bit-widths we significantly reduce memory bandwidth without visible quality loss and retaining ≥96% MS-SSIM. The memory bandwidth is significantly reduced by up to 63% for the simple scene; from 4.3 MBytes/s to 1.6 MByte/s (63%). Bandwidths of medium and complex scenes are also reduced by 59% and 56% respectively. These huge reduction in communication demand come at a slight cost of increased computation. Compression/decompression increases overall computation by 6%. Since compression is not on a critical path, the latency of design does not necessarily increase.

3.4.1.2 KLT Optical Flow Exploration

We have also explored the quantified the quality / bandwidth trade-off for KLT optical flow algorithm. As outlined in Section 3.2.1 for KLT optical flow, the algorithm-intrinsic traffic is the feature signature models which kept updated per individual features across the frames under
CHAPTER 3. TRAFFIC SEPARATION IN ADAPTIVE VISION ALGORITHMS

Figure 3.14: Exploration for the medium scene

processing (highlighted in Figure 3.2d). We have explored algorithm-intrinsic bandwidth for five
different size of feature window (3*3, 5*5, 7*7, 9*9, 11*11). In this exploration, signature window
with size of 11*11 is considered as the ground-through condition.

Figure 3.15: Optical flow quality / bandwidth trade-off

Figure 3.15 highlights the quality / bandwidth trade-off for KLT optical flow. The range
of algorithm-intrinsic bandwidth is from 100 MB to 1.1 GB depending on the signature window
size. The quality results have been reported for two different quality assessment metrics: PSNR
(highlighted in Figure 3.15a) and MS-SIM (highlighted in Figure 3.15b). While PSNR just con-
sidered direct feature motions comparison for assessing the quality, MS-SIM focus on the structural
CHAPTER 3. TRAFFIC SEPARATION IN ADAPTIVE VISION ALGORITHMS

similarity between the motion patterns. In the results, the quality results presented for PSNR assessment (Figure 3.15a) illustrates much higher quality than the quality results for MS-SIM metric (Figure 3.15b). However, both metrics demonstrate a pronounced increase over quality by increasing the algorithm-intrinsic bandwidths. Overall the presented results in Figure 3.2d demonstrates the validity of algorithm quality / algorithm-intrinsic bandwidth trad-off for the adaptive vision algorithms beyond MoG background subtraction.

3.4.2 Zynq prototyping

Our realization targets the Zynq-7000 XC7Z020-CLG484-1 SoC [119], which combines two ARM Cortex A9 processor cores with programmable logic (Artix-7 FPGA). Processor cores and logic are interconnected through AXI and share I/Os, as well as off-chip memory interfaces (DDR3, LPDDR2) with a peak bandwidth of 4.2 GBs.

3.4.2.1 MoG Relization

Figure 3.16a outlines our communication-centric architecture on Zynq platform running MoG background subtraction. Following our architecture template, communication (125 MHz) and computation clock (148 MHz) domain are separated. MoG receives and updates Gaussian parameters throughout DMA Read and Write channels operating in circular modes. The DMA channels connect to the DDR3 memory interface through AXI-Stream bus. For initialization and runtime configuration, AXI-lite exposes the MMRs of a Control Unit (CU), configuring and controlling MoG, to the ARM cores. Software implementation is minimal. It only needs to initialize Gaussian parameters, configure the DMAs and the MoG MMRs, which all happens once at startup.

Figure 3.16: Experimental setup

(a) Communication-centric MoG realization on Zynq platform
(b) Real-time Background subtraction
Table 3.3: MoG execution on different scene complexity.

<table>
<thead>
<tr>
<th>properties</th>
<th>input scene</th>
<th>FG mask</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\alpha = 0.01$ $\Gamma_{match} = 1.5$ $\Gamma_{FG} = 0.24$</td>
<td><img src="image1.png" alt="Input Scene 1" /></td>
<td><img src="image2.png" alt="FG Mask 1" /></td>
</tr>
<tr>
<td>$\alpha = 0.5$ $\Gamma_{match} = 2.5$ $\Gamma_{FG} = 0.24$</td>
<td><img src="image3.png" alt="Input Scene 2" /></td>
<td><img src="image4.png" alt="FG Mask 2" /></td>
</tr>
<tr>
<td>$\alpha = 0.25$ $\Gamma_{match} = 1$ $\Gamma_{FG} = 0.18$</td>
<td><img src="image5.png" alt="Input Scene 3" /></td>
<td><img src="image6.png" alt="FG Mask 3" /></td>
</tr>
</tbody>
</table>

Figure 3.16b exemplifies real-time execution of MoG on Zynq platform at 1080p 30Hz (1080*1920 at 30 frame per second). The MoG operates on pixel stream receiving video input from one HDMI input interface and outputting foreground pixels to another HDMI output connected to the monitor. Our system also operates at 1080p 60Hz. However, due to DDR3 bandwidth limitations, operation is limited to a window of interest (ie. subset the frame) of 50%.

To presenting the actual execution result, a rich data-set representing real scenes at Full-HD resolution is required. For this purpose, we have recorded and collected our own input data-set representing real scenes including: indoor corridor, crowded highway and street corner. In a nutshell, indoor corridor represents a simple scene with stable background with slow moving objects. Crowded highway represents an outdoor scene with very fast moving objects. And street corner shows a complex scene with multi-model changes in background including illumination, and different FG moving objects with different speeds (pedestrian and cars).

Table 3.3 demonstrates one input frame per each scene as well as its corresponding FG mask output. For all three senses, we observe that moving foreground (FG) objects correctly captured and exposed in FG mask output. Table 3.3 also highlights the algorithm knobs we have set for individual scenes. As an example, indoor corridor has the lower learning rate ($\alpha$ equal to 0.01) due to slow speed of moving objects. Conversely, $\alpha$ is much higher, 0.25 and 0.5 for street corner and crowded highway respectively, as moving cars are much faster than humans. Other algorithm
parameters (FG Threshold \( \Gamma_{FG} \) and Match threshold) are adjusted properly to deliver highest background subtraction quality.

Table 3.4 and Figure 3.17a highlight resource utilization and on-chip power consumption for one configuration instance of our architecture template for achieving 100% quality (8K FIFO depth, 256 bits Bus Width, 125 MHz communication and 148 MHz computation frequency). The design occupies 38% slices, 8% DSP blocks and 26% 36-bit Block-RAMs of the Zynq programmable logic fabric. This leaves sufficient room for other applications.

Table 3.4: System Resource Utilization

<table>
<thead>
<tr>
<th>Resource</th>
<th>BUFG</th>
<th>DSP48E1</th>
<th>RAM Block</th>
<th>Slice</th>
<th>LUT</th>
<th>Register</th>
</tr>
</thead>
<tbody>
<tr>
<td>Utilization%</td>
<td>9</td>
<td>8</td>
<td>26</td>
<td>38</td>
<td>20</td>
<td>12</td>
</tr>
</tbody>
</table>

On-chip power shown in Figure 3.17a is estimated using the Xilinx X-Power Analyzer after post synthesis and mapping of the entire design to the Zynq. More than 67% of on-chip power is consumed by the communication subsystem. Even at a high video clock frequency, MoG only consumes 18%. Within communication components (right side of Figure 3.17a), DMA and FIFO have the highest contribution with 28% and 26%, respectively. The added precision adjustment block consume minimally with less than 1%.

To give an indication of the impact of quality requirements, Figure 3.17b plots the on-chip power consumption for across quality levels. Lowering the quality demand results in smaller algorithm-intrinsic data volume, thus reducing on-chip power consumption. Achieving 100% quality (as expressed by MS-SSIM) requires 380mW. Relaxing the quality requirements to 90% already reduces to 340mW. In addition to on-chip power consumption, the off-chip memory accesses for algorithm-intrinsic data significantly contribute to systems overall power demand. It attributes to around 65% of entire system power in our implementation. The estimated system power consumption (including on-chip and off-chip memory power) is about 1.2 Watt. By Gaussian parameter sizing and thus cutting volume almost into half, the off-chip power halved as well. There, precision adjustment has a profound effect of making an HW implementation feasible (given the memory interface), but also significantly reduces power consumption.

\(^1\)Many other configuration configuration points are possible, as will be explored later.
CHAPTER 3. TRAFFIC SEPARATION IN ADAPTIVE VISION ALGORITHMS

(a) Contribution of architecture components to on-chip power

Figure 3.17: Power consumption of proposed architecture template

3.4.2.2 Comparison to Alternative Approaches

To contrast against a MoG Software (SW) attempt, we have also realized MoG in C and compiled it for ARM Cortex A9 using gcc with O3 optimizations and enabled NEON optimizations. One Zynq’s Cortex A9 takes 610s (360s) to process 60 frames in full-HD (half-HD). In other words, SW execution is 610x (360x) slower than real-time. Even if real-time execution could be achieved through massive parallel execution on 610 (360) A9 cores, power consumption would be unacceptably high. Assuming each core (with cache) consumes 0.6 W at 666 MHz \cite{84}, 360 W (216 W) would be required. Even hand-crafted NEON SIMD optimizations cannot push this approach into a solution envelope. Even with an unrealistic 4x SIMD speedup, still (90) A9 cores would be required, exceeding the power constraints.

Comparing against the closest related hardware approach \cite{9}, reveals tremendous performance differences operating at 640*480 \cite{9} vs. 1920x1080. Unfortunately, \cite{9} does not report power or quality, making it difficult to directly compare. Nonetheless, the main approach of \cite{9} is to assume a constant standard deviation to reduce computation complexity and bandwidth. For the purpose of comparison, we implemented this approach at specification level and observed an output quality about 40\%\textsuperscript{2} (MS-SSIM evaluation) and at 33\% bandwidth reduction. In contrast, adjusting the precision Gaussian parameter storage in our approach reduces bandwidth to half (50\% bandwidth reduction) while still maintaining 100\% quality. On top of the quality advances of our approach, our solution is also more power efficient and scalable than \cite{9} which uses a co-processor arrangement (similar to Figure 3.8a). In contrast our architecture template thanks to its peer-processor mode (Figure 3.8b) completely eliminates the memory access for streaming data, as well as simplifies

\textsuperscript{2}The quality results vary based on the complexity of the scene under evaluation
CHAPTER 3. TRAFFIC SEPARATION IN ADAPTIVE VISION ALGORITHMS

node-to-node streaming (see *RGB2Gray*) leading to a better scalability.

3.4.3 System Integration

We have demonstrated that traffic separation enable system architect to tune the traffic of individual vision kernels depending on their quality requirements. The traffic separation principles also provide an opportunity to manage system level traffic across multiple streaming kernels by simplifying pipelining of adaptive vision processing kernels. Separating streaming and algorithm-intrinsic data allows keeping the streaming data on-chip without the costly memory interaction (and enables streaming node chaining). Only algorithm-intrinsic traffic (after precision adjustment based on quality constraints) hits the system memory.

![Figure 3.18: Abstract algorithm overview of object tracking flow](image)

To demonstrate the system integration principles, we have constructed an object tracking vision flow out of three adaptive vision algorithms including MoG background subtraction, component labeling and mean-shift object tracking (highlighted in Figure 3.18). We have also integrated image smoothing as and morphology kernel representing the vision filters (pre-processing) to construct. Figure 3.19 demonstrates the system architecture view of object tracking vision flow. We have implemented the whole system in Zynq platform. The estimated power performance results demonstrate that the proposed vision architecture solution performs 50GOPs/s at 1.5Watts of on-chip power consumption.

3.5 Chapter Summary

In this chapter, we identified two classes of traffic in adaptive vision algorithms (streaming and algorithm-intrinsic traffic). The traffic separation enables to reduce the immense communication demand and removes a tremendous hurdle of embedded realization of adaptive vision algorithms. It provides an opportunity to explore bandwidth / quality tradeoff for adaptive vision algorithms enabled by traffic separation. We quantified the tradeoff on the example of Mixture of Gaussians. Compressing the algorithm-intrinsic traffic of MoG, reduces traffic by 63% with less than 5%
CHAPTER 3. TRAFFIC SEPARATION IN ADAPTIVE VISION ALGORITHMS

loss in quality (based on MS-SSIM). To realize the traffic separation, this chapter also proposed a communication-centric architecture template supporting streaming and algorithm-intrinsic traffic. Our template enables application-specific management of algorithm-intrinsic data (compression, prioritization), and simplifies pipelining of stream processing nodes. The chapter demonstrated the versatility and power of our architecture template by constructing a complete object tracking vision flow (consisting of 6 nodes) processing 1080p 60Hz. Our Zynq-based architecture performs 40GOPs/s at 1.7Watts of on-chip power.

Figure 3.19: Architecture overview of embedded vision SoC solution.
Chapter 4

Function-Level Processor (FLP)

This chapter introduces Function-Level Processor (FLP) to fill the flexibility / efficiency gap between ILPs and custom-HWs. FLP is a novel processing class offering an architecture template for efficient execution of many vision flows on a single platform.

4.1 Demands for Flexibility and Efficiency

Traditionally, Instruction-Level Processors (ILPs) have been the predominant choice for computationally complex digital signal processing due to their high flexibility and cost-efficiency. As computational complexity increases, ILPs mainly suffer from high energy per operation ratio often leading to exorbitantly high power consumption to compute only a few billion operations per second \[45\]. On average, only 3% of the energy per instruction is consumed for the actual computation \[55\] while the remaining 97% are spent for data and instruction movement through the memory/cache hierarchy and complex instruction/data fetch. In result, the demand for performance and power drive system architects to heterogeneous platforms combining ILPs and custom HW Accelerators (ILP-HWACCs) \[6, 59, 71, 88\]. HWACCs have the potential for ultimate power efficiency; however, typically target a dedicated application with limited flexibility. This forces designers to navigate a tradeoff between flexibility and performance/power efficiency. ILP provides ultimate flexibility; however, this results in high power consumption and thermal cost. Conversely, HWACC has the potential for ultimate power efficiency; however, HWACC supports a dedicated application with limited flexibility. This forces designers to navigate a tradeoff between flexibility and performance/power efficiency.
In current heterogeneous ILP+HWACCs platform, HWACCs are used as the offload engines for compute-intense frequently occurring application kernels (e.g. H.264, JPEG encoder). Designing dedicated HWACCs and ASICs for every application is prohibitively expensive due to resources, tools and high NRE costs. Conversely, the opportunities of utilizing smaller HWACCs and pipelining them to construct larger applications are very little explored due to many challenges. Current designs are bandwidth hungry as accelerators exchange data through Direct Memory Access (DMAs) which creates significant traffic as well as requires large on-chip buffers. In addition, with increasing HWACCs, ILPs are increasingly burdened with synchronizing, scheduling and coordinating data exchange between HWACCs, not being able to efficiently utilize the HWACCs [88, 59, 66]. In result, while individual large HWACCs offer computation efficiency opportunities, composing application out of multiple accelerators is currently not efficient. To improve performance and power efficiency, novel architecture solutions are needed offering a flexible and efficient composition of accelerators. Such architectures should offer comparable power/performance efficiency to HWACCs while retaining enough flexibility to meet functional requirements across many applications of a targeted market.

In this chapter, we introduce Function-Level Processors (FLPs) aiming to close the flexibility and efficiency gap between ILPs and HWACCs. In comparison to ILPs, an FLP raises architecture and programming abstraction from instruction-level to a coarser function-level, aiming for increased efficiency. Compared to a HWACC, an FLP increases flexiblity by offering a finer grained programmability being constructed out of smaller, recomposable function blocks (FBs). Most importantly, our FLP serves as an architecture template to efficiently manage FB composition for constructing bigger applications. It offers an efficient MUX-based interconnection between FBs and reduces ILP synchronization effort through FB-to-FB synchronization. FLPs further increase autonomy with an own control unit for run-time FB synchronization and these are concept benefits. FLPs offer a new class of autonomous processing elements which can be paired with ILPs or other application-level accelerators for constructing high-performance MPSoCs. By coarsening the programmability of architecture, FLPs nearly achieve the same power/performance efficiency as HWACCs as well as enough flexibility to execute applications of a designated market. This configurability and efficiency make FLPs ideal candidates for market-oriented MPSoCs. Markets of interest include embedded vision computing, software define radio, and bio-medical analytics.
CHAPTER 4. FUNCTION-LEVEL PROCESSOR (FLP)

4.1.1 Efficiency/Flexibility Trade-off

To provide background and motivate our design choice in granularity for the FLP concept, we first categorize existing approaches. A trade-off exists between flexibility (i.e. ability to execute different applications) and efficiency in terms of performance and power. Table 4.1 conceptualizes the trade-off, comparing architectures with varying programmability granularity from the finest granularity (bit-level in FPGAs) to the coarsest granularity (application-level in Custom-HWs).

Table 4.1: Computation Granularity: efficiency/flexibility trade-off

<table>
<thead>
<tr>
<th>Granularity of Computation</th>
<th>Custom-HW</th>
<th>Function-Level</th>
<th>Instruction-Level</th>
<th>FPGA</th>
</tr>
</thead>
<tbody>
<tr>
<td>Application</td>
<td>Function0</td>
<td>Function3</td>
<td>Instruction</td>
<td>Bit/Binate</td>
</tr>
<tr>
<td>Function1</td>
<td>Function4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Function2</td>
<td>Function5</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Function3</td>
<td>ALU</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RF</td>
<td>Bit_0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bit_1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bit_2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bit_3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

FPGAs are configurable at bit-level [63] and thus offer the finest granularity. Operations (e.g. logic, arithmetic) at arbitrary bit width can be composed with an arbitrary connection. FPGAs have a high area & power cost in comparison to ASIC design (5x-12x higher power [60]). In result, FPGAs are favored for prototyping and reconfigurable computing.

Instruction-Level Processors (ILPs) are highly flexible being programmable through sequentially composing instructions from well-defined Instruction-Set Architecture (ISA). Communication between operations occurs through the register file or memory. However, the instruction-level flexibility results in a considerable power and performance overhead. Only 3% of the energy per instruction is consumed on average for the actual computation [55] while the remaining 97% are spent for data and instruction movement through the memory hierarchy and the ILP micro-architecture. We also consider VLIWs and SIMD architecture ranging from vector processors to Graphic Processor Units (GPUs) [77] as ILPs, offering options to increase throughput and power efficiency.

Custom-HW, on the other extreme, have the least flexibility. Their selection defines the
application being executed. Composition of multiple HW ACCs is not much explored. On the other hand, HW ACCs allow the most efficient implementation with a fixed, application-specific data path yielding much higher performance and lower power consumption in comparison to ILPs. As an example, [45] reports an ASIC H.264 implementation which is 500x and 25x more energy efficient than ILP RISC-like and optimized VLIW-SIMD implementations, respectively. Nonetheless, this efficiency comes at the cost of limited flexibility. With the high NRE costs, developing a dedicated Custom-HW for every application is not practical.

To still construct flexible solutions, Custom-HW components are utilized as HW Accelerators (HW ACCs) coupled to an ILP such as shown in Figure 4.1. Here, the ILP executes the top-level application (often still quite compute intense) providing flexibility and offloads predefined application kernels to HW ACCs. The ILP is responsible for HW ACC configuration, managing data transfers and synchronization. On-chip scratchpad memory is required to avoid additional data movement and inefficient accesses to off-chip memory which would otherwise diminish the accelerator’s efficiency [99]. However, large scratchpads are often under-utilized ([66] reports 30% utilization). Overall, the efficiency of a heterogeneous ILP+HWACCs architecture is bounded by the remaining ILP computation and overhead of managing HW ACCs.

Analyzing the solution space of existing efficiency/flexibility design choices reveals a gap to be filled between Custom-HW and ILP approaches. New solutions are needed that increase the flexibility (i.e. programmability) over existing Custom-HW, while at the same time improving the efficiency over ILP approaches. In this context, focusing on selected markets (or an application domain) allows to restrict the required flexibility to a set of applications, opening up opportunities to increase efficiency through specialization.

To fill this gap, we envision a programming granularity at a function level (coarser than instructions and finer than applications) formalized as an Function Set Architecture (FSA). To translate the coarser granularity into efficiency, an architecture could be composed of smaller custom-HWs that implement a small kernel function each (instead of applications). They then
have to be efficiently chained together to construct the larger application. Challenges on how to efficiently compose custom-HW blocks, avoiding costly external memory traffic, and avoiding high synchronization overhead need to be overcome. Section 4.2 describes our approach to fill the gap with the Function-Level Processor (FLP).

The set of configuration options (including # and type of FBs) defines the Function-Set Architecture. Programming an FLP involves FB allocation, FB configuration as well as defining the FB interconnection all in a concerted effort to realize the application. At the same time, By raising the programming abstraction to a function granularity, FLP dramatically simplifies the application composition. The rest of this chapter will focus on introducing the architectural aspects of FLPs. loosely coupled accelerator (LCA), which acts independently of individual cores and is more amenable to sharing, indicating that these LCAs can be built using a limited number of smaller, more general LCAs that we call accelerator building blocks (ABBs).

4.2 Function-Level Processor (FLP)

To fill the gap between ILP flexibility and custom-HW efficiency, we propose a new architecture template: Function-level Processor (FLP). An FLP is targeted for a selected market, to constrict the flexibility needs allowing for specialization. It aims to combine efficiency of dedicated custom HW with a sufficient level of flexibility to execute a set of applications of a selected market. Instead of monolithic dedicated HW ACCs for individual application kernels, an FLP offers a platform containing composable function primitives with a flexible interconnect that allows to construct market applications. To minimize synchronization effort (and overhead) when paired with an ILP, an FLP includes autonomous control, scheduling and reconfiguration capabilities.

Figure 4.2: General FLP Architecture Composition.
CHAPTER 4. FUNCTION-LEVEL PROCESSOR (FLP)

Figure 4.2 shows the overall architecture of an FLP as a composition of Function Blocks (FBs) with configurable custom interconnects (MUXs) and hierarchical memory buffers/caches. Each FB represents a function primitive of the targeted market. To achieve efficient communication and also to minimize power hungry memory subsystem accesses, the FLP distinguishes (as illustrated by colors in Figure 4.2) between streaming data (white), operational data (black), and control / configuration (blue). In this section, we discuss the main architectural features of the FLP starting from individual function-Blocks (FB), FB composition to construction within an FLP, and finally the FLP system integration into heterogeneous MPSoCs.

4.2.1 Function-Block (FB)

At the core of an FLP are its FBs. Each FB implements a single (or small set of) well-defined functions providing a meaningful API that enables interconnecting with adjacent FBs. One can consider a FB as a small configurable IP component with C-function granularity (e.g. Convolution, Cartesian to Polar conversion, Gaussian equation). FBs are not meant to realize application-level algorithms, but rather are function primitives of designated markets; interconnected/fused together to construct multiple algorithms / applications.

Figure 4.3 outlines the general structure of an FB with five major components: (a) FB datapath, (b) stream-in / stream-out, (c) stream buffer (d) configuration/control registers, (e) operational data. The FB’s datapath (a) is optimized to efficiently realize the FB’s functionality \( y = f(x) \). The datapath offers some flexibility through operation modes, such as configuring the matrix size (5x5 or 3x3) of a convolution FB or switching between different range of data types or data structures. Stream-in / stream-out (b) carries the main data for direct FB-to-FB communication. An FB can have in/out stream ports each with unique data types and lengths. Depending on the operation mode the FB may use primitive (uint32), aggregated (array) or custom data types (e.g. structures). An FB may contain a stream buffer (c) for temporary storage of streaming data (e.g. an integral function would utilize a local buffer). Control/configuration (d): each FB implements a slave interface exposing control and configuration registers. The configuration registers alter the computation performed and parametrize the functionality. Examples include threshold values, data dimensions, or even specific operations. Control registers disable / enable a FB. Finally, an (e) operational data buffer allows the FB to access the memory hierarchy (e.g. coefficient parameters for Convolution or distribution parameters in a Guassian equation).

Identifying and selecting suitable FBs is a research challenge similar to the existing problem
of accelerator selection in a HW/SW co-design. It opens a research opportunity to identify suitable function blocks that can be composed meaningfully. In general, compute intense kernels that are frequently occurring in the selected application domain are good FB candidates. Additionally, FBs need to be selected such that they allow macro pipeline construction (building applications). This may necessitate FBs that act as glue logic with little own computation (e.g. basic arithmetic and logic operations), which in an accelerator context would not be selected for HW implementation. Conversely, in the FLP concept such glue logic FBs allow keeping data within the FLP and avoid costly data movement and synchronization with an ILP. They help to increase the FLPs efficiency and autonomy.

4.2.2 FLP Architecture Overview

4.2.2.1 FBs Interconnection

Irrespective of the granularity of custom HW components (e.g. FB in FLP or accelerators in other architectures), efficiently chaining them to construct larger applications is a sought after research topic. Initial accelerator chaining approaches are inefficient as they rely on DMA-based communication, imposing significant traffic to memory hierarchy and require large on-chip buffers. To obtain an efficient overall solution, the aim is to avoid synchronization overhead with an ILP, as well as hide most of the traffic from the memory hierarchy. To avoid memory traffic, a dedicated point-to-point connection for streaming data between FBs is desirable. At the same time however, some interconnect flexibility is needed to allow constructing varying datapaths of FBs for different applications.

To achieve both efficiency and flexibility, we propose a configurable MUX-based interconnect between FBs. As shown in Figure 4.2, direct FB-to-FB communication is realized by a set of configurable multiplexers (MUXs). Theoretically, an n:n communication would offer the most
flexibility. However, in practice a sparser connectivity is desirable that only allows connections between (a) compatible ports that use the same data type and (b) feasible connection which allows a functionally meaningful composition. Compatibility (a) is determined by the datatype, data length and semantic (Polar or Cartesian representation). The feasibility (b) of connections is defined by the application flows envisioned for the targeted market as not all combinations necessarily lead to meaningful applications. Determining FB connection requires analyzing the application domain and determining meaningful FB compositions.

Figure 4.4: Chaining Function Blocks (FBs).

At run-time, FBs are interconnected by MUX configuration. After establishing the connection, no further control is required and FBs directly communicate independently; synchronized by the availability of data or control signals. For organizational purposes the FLP groups FBs into stages following the main flow of the analyzed applications (highlighted in Figure 4.2). FBs in different stages can be interconnected together in scatter, gather, and self-feedback modes. Additional MUXs allow for- and back-warding data between FLP stages increasing the flexibility in macro-pipeline construction. The distributed MUXed interconnect system empowers concurrent execution.

Figure 4.4 exemplifies a possible chaining across multiple FBs, focusing on FLP customized interconnection. In Figure 4.4, \( f(x) \), \( h(x) \), \( q(x) \) and \( p(x) \) are chained together through the MUXs building a bigger application, while \( g(x) \) and \( p(x) \) are not used, thus power-gated. Each MUX can receive a limited set of inputs from a predefined set of FBs, thus determining the chaining flexibility and limitation. For instance, MUX0 has only two inputs from \( f(x) \) and \( g(x) \), or MUX2 can get input from \( f(x) \), \( g(x) \) and \( h(x) \). Different arrows widths visualize different data types that can be passed across FBs.

To further illustrate how FB-to-FB MUX-based communication is realized, Figure 4.5 zooms into \( f(x) \) to \( h(x) \) connection. MUXes are agnostic of the type of data passed. Hence, FBs are
CHAPTER 4. FUNCTION-LEVEL PROCESSOR (FLP)

responsible to perform data marshaling and de-marshaling in their input/output ports Figure 4.5. As individual ports connect to different types of FBs, control registers within the FB determine the type of marshaling/de-marshaling with respect to source and destination.

![Figure 4.5: FB-to-FB MUX-based communication](image)

Since FBs in an FLP operate on streaming data, we consider all FBs and MUXes operating on the same streaming clock. Streaming data can vary in data complexity (from primitive to aggregate data type) and in frequency of data availability (at most one sample per clock). This necessitates side-channel signals provide run-time control over communication modes and availability of data (enable signal in Figure 4.5). The source FB is responsible for signaling availability. As an example, an FB with an integral operation over 10 pixels generates single valid data every 10 cycles to the next FB.

In summary, the FLP’s MUX-based interconnect offers configurable point-to-point connections between compatible FBs avoiding costly data movement in the memory hierarchy.

4.2.2.2 Memory Organization

One of the main sources of inefficiency in ILPs is the lack of separation between streaming data and operational data. Figure 4.6 illustrates the difference. Streaming data is the data under processing (pixels in case of vision). Streaming data often enters or exits the system through dedicated system I/O ports (e.g. video interfaces), bypassing the memory hierarchy. Conversely, operational data is local data needed for realizing the computation (e.g. kernel density histogram or Gaussian parameters). If the amount of local data exceeds local storage (e.g. scratchpad), it is emitted to the memory hierarchy. ILPs waste significant power and processing cycles as streaming and operational data (which have different access patterns) are intertwined and compete for the same memory hierarchy.

In order to avoid unnecessary memory traffic, the FLP separates between streaming data (white), operational data (black) highlighted in Figure 4.2. This separation offers prioritizing traffic
based on nature and role of data in execution. Most traffic occurs with the streaming data, hidden by using point-to-point connections between FBs. In fact, FBs are fused together to remove the unnecessary memory access for streaming data. The stream data may hit the memory system only at the end of entire application through streaming DMAs. In addition, the FLP offers access to memory hierarchy which is however reserved for operational data. Operational data (e.g. Gaussian parameters) is frequently accessed by the FBs thus should be cached. Two stages of buffer / caches are provided one per each FLP stage and one global data buffer/cache which is shared between all stages (highlighted by pink color in Figure 4.2).

The operational memory can be configured to be either scratchpad buffers or local cache memories. In scratchpad mode, with a Direct Memory Access (DMA) interface, these buffers can directly write and read operational data to/from the main memory. In cache mode, the memory in the FLP stages, combined with FLP shared cache construct a memory hierarchy. One research question is to determine the suitability of coherency across stages and levels. By design, FB-to-FB communication occurs through the stream interfaces, thus relieving strain on the memory subsystem and reducing the need of coherency.

4.2.2.3 Autonomous Runtime Control/Configuration

In current heterogeneous systems, the host ILP is responsible for controlling HWACCs imposing a considerable load to the ILP while potentially leading to a low HWACCs utilization \[66\]. To avoid this overhead, the FLP employs a Control Unit (CU) to minimize or even eliminate the need for ILP-centric control. The CU governs allocation of FBs (e.g. power gates unused FBs), forwards configuration to the individual FBs and configures the MUX-based interconnection. The FLP has a dedicated bus (highlighted by blue color in Figure 4.2) for dissemination of control data into to the FB’s control/configure registers at configuration points.

To receive control data, two different operation modes are conceivable for the CU: (a)
CHAPTER 4. FUNCTION-LEVEL PROCESSOR (FLP)

Figure 4.7: Autonomous fetch of control and configuration data from system memory.

ILP controlled mode and (b) independent mode. In ILP control mode (a), the ILP is responsible to fill the FLP control buffers with proper information either directly through memory mapped registers or indirectly through a configuration descriptor to be fetched via DMA (similar to DMA descriptor-based programming). The FLP controller then independently fetches new configurations and transitions between configuration sets as triggered by events (either FLP internal or external from ILP). Conversely, in an independent mode (b), the FLP would be programmed similarly to a coarse-grained ILP. The FLP configuration data is placed in predefined memory addresses and fetched autonomously through FLP configurations. In result, the FLP can operate as an independent processor unit.

The CU is also responsible for governing spatial and temporal parallelism. In our current FLP design, we consider spatial decomposition on FBs across applications. Multiple applications with disjoint FB instances can run in parallel on individual data streams. No FBs are shared between spatially parallel applications. Temporal parallelism, where FBs are shared across concurrent streams, is considered as future work.

Figure 4.8: 2MB on-chip memory

To quantify the benefits of FLP autonomous control, we have modeled both FLP and
CHAPTER 4. FUNCTION-LEVEL PROCESSOR (FLP)

ILP+ACC solutions in SpecC SLDL. The models offer a configurable number of accelerators (FBs in case of FLP) coupled with ARM9 TDMI. Figure 4.8 presents the ILP utilization for different numbers of FBs when the available on-chip scratchpad memory is 2MB. In an ILP+ACC solution, as the number of accelerators increases, the ILP utilization grows exponentially. In contrast, FLP solutions need minimal ILP utilization independent from the number of FBs (ACCs) in the system.

4.2.2.4 System Integration

Figure 4.9 shows a possible system integration of an FLP to a ILP-based system. FLPs do offer more independence than individual custom hardware IP blocks as they can independently operate on input streams. However, we anticipate pairing FLP(s) with ILP cores to increase flexibility. While the FLP performs compute intense stream processing, ILPs can offer top-level adaptive control and intelligence. Here, FLP and ILP exchange data through DMAs operating on the common memory hierarchy. Hybrid solutions are common for video applications, in which the FLP performs pre-processing and an ILP makes higher-level decisions on the preprocessed data.

Figure 4.9: FLP system integration

In order to allow efficient implementation of streaming applications, the FLP has direct access to system I/O interfaces for input and output of data without requiring constant processor interaction. Examples for system I/O ports include streaming interface for cameras, general acquisition boards, or outputs like video. With access to the I/O interfaces, it is conceivable to build a complete application only out of FBs in an FLP. Overall, FLP adds additional degree of heterogeneity to the current MPSoCs to offer low-power high-throughput computation along with enough flexibility to support targeted markets.

4.2.3 Function-Set Architecture (FSA)

FSA offers a coarse-grained programmability of the FLP at the function-level granularity; significantly more abstract than the ISA of an ILP. In contrast to an ISA which offers an arbitrary sequence of instructions, an FLP reduces flexibility (governed by FB selection, FB configurability,
CHAPTER 4. FUNCTION-LEVEL PROCESSOR (FLP)

and feasible FB connection). As a result, the FSA exposes the type and number of FBs, their configurations and all potential ways that these FBs can be interconnected to the FLP programmers.

FSA summarizes the set of configuration and control options for one FLP instance. The FSA forms the basis for programming the FLP, which involves allocation of FBs, configuring each FB, and determining the interconnection between FBs to meet applications requirements. One aspect of developing an FSA for a given market is identifying efficient number of FBs to deliver maximum autonomous support for applications within the targeted market.

The FSA can be considered with similar granularity to common SW libraries (e.g. OpenCV for vision, OpenAL for audio or Transceiver for Software Define Radio (SDR)) which offer base function primitives for application development and simplify programming. As an example, a complex vision application can be developed by a few lines using OpenCV API. Nonetheless, tools to guide FLP users to develop their applications on a given FLP are desired.

4.2.4 FLP Example: Pipeline-Vision-Processor (PVP)

To demonstrate the benefits of an FLP, we discuss and analyze an early example: the Pipeline Vision Processor (PVP) integrated with two Blackfin DSP/RISC cores into the Analog Devices BF60x SoC [18]. The PVP is an industry product that has been published with user-level documentation. However, only limited architecture-level detail of the PVP is revealed as this is internal to the company. Our FLP is the result of joint work with the PVP chief architect to generalize the architecture and improve upon it for scalability and efficiency. We use the results from the manufactured PVP as good indicators for the FLP potentials.

Figure 4.10 highlights a simplified architecture of the PVP offering eleven FBs, including: (a) four 2-D Convolution blocks (CNV), (b) one General-purpose Arithmetic Unit (ACU), (c) one Cartesian to polar converter (PMA), (d) one 2-D derivations Pixel-Edge Classifier (PEC), (e) two range reduction Threshold-Histogram-Compression (THC), two 2-D Integral Image (IIT). FBs are grouped into four stages, simplifying their connectivity. PVP’s FBs selection was driven by direct customer feedback and system architects based on sufficient generality across the Advanced Driver Assistance Systems (ADAS) and general vision markets combined with feasibility of their implementation. As shown, the PVP includes one Arithmetic Unit (ACU) to increase flexibility of PVP by supporting basic arithmetic and logic operations.

PVP utilizes the main FLP architecture principles including MUX-based FB-to-FB communication, Autonomous Control and scheduling, parallel application execution and runtime on-the-fly...
re-configuration. However, in contrast to FLP, the PVP focuses on streaming data only and does not include operational data and its associated memory hierarchy. The selected FBs operate with local storage only. PVP fetches configuration data via separate DMA channels similar to descriptor-based programming for the DMA controllers. Stream data enters/exits the SoC through system interfaces and interfaces through DMAs with the cores. Internal traffic between FBs is independent from system communication fabric avoiding the need for DMAs and its associated buffers. On-the-fly re-configuration may occur at granularity of a new image frame.

To have better insight about the PVP, Figure 4.11 illustrates two function blocks PMA and THC. PMA only supports single operation mode which is translation from Cartesian input \((x, y)\) to polar representation \((\phi, \text{Magnitude})\). It has two input ports for receiving the Cartesian values \(x\) and \(y\) (both with length of 16 signed bits) and three output ports including \(\text{Magnitude}\) (16 bits), \(\phi\) (5 bits) and \(\text{Magnitude} + \phi\) (21 bits). The PMA input/output ports are connect to a
limited subset of FBs based on occurring flows of the considered vision applications. As an example, \textit{Magnitude} output port is only connected to \textit{THC} and \textit{PEC}). The \textit{PMA} control register determine the targeted input/output ports.

\textit{THC} (highlighted in Figure 4.11b) offers three operation modes: clipping, quantification and hysteresis. The operation mode is selectable though configuration registers. \textit{THC} also uses threshold buffers (programmable at runtime) for calculating histogram values. The \textit{THC} realizes a fairly simple statistical and control flow (if-then-else conditions) operations with low computational complexity. Usually, such a block would rarely be chosen for acceleration in HW. However, by adding \textit{THC} blocks to PVP, it allows to keep the computation/communication on the PVP. This avoids unnecessary interactions with an ILP and thus significantly contributes to efficiency.

4.3 Experimental Results

We first demonstrate the increased flexibility by mapping a set of vision applications. Following that, we compare the PVP against ILP and ILP+HWACCs solutions with respect to both computation/communication demands and power overhead.

4.3.1 Mapping Flexibility

To demonstrate flexibility and efficiency of the PVP, we selected ten applications from embedded vision computing which are frequently used in ADAS and video surveillance: (1) High/Low Bean Adjustment (HB/LB); (2) Canny Edge Detection (Canny); (3) Sobel Edge Detection (Sobel); (4) Laplacian on Gaussian (LoG); (5) Threshold Binary Edge Map (TBEM); (6) Level Histogram (LH); (7) Integral of Binary Edge Map (IBEM); (8) Integral of Variance (IoV); (9) Difference of Gaussian (DoG); (10) Histogram of Gradients (HoG).

In this part, we focus on the FLP-PVP flexibility in supporting many applications within targeted markets. To illustrate how an application can be composed out of a set of FBs, Figure 4.12 exemplifies the mapping of two applications: \textit{Canny} and \textit{HB/LB}. For \textit{Canny} (Figure 4.12a), one \textit{CNV} block is required for image smoothing and two \textit{CNV} blocks for finding the \textit{x} and \textit{y} gradients. Following that, \textit{PMA} computes the edge strength (\textit{Magnitude(M)}) and the direction (\textit{ϕ}) as an indicator of edges for gradients values. Finally, \textit{PEC} converts blurred edges to sharp edges. The second example, \textit{HB/LB} is shown in Figure 4.12b. The application is implemented by mapping down-sampling to an \textit{CNV} block and histogram to \textit{THC} block.
CHAPTER 4. FUNCTION-LEVEL PROCESSOR (FLP)

Figure 4.12: Transferring Application Flow Chart to Function Graph

(a) Canny

(b) HB/LB

Figure 4.13: HB/LB and Canny concurrent execution.

Figure 4.13 shows a graphical view of the PVP, executing High/Low Bean Adjustment (HB/LB) (red) and Traffic Sign Recognition (TSR) (green) at the same time. Both applications TSR and HB/LB can be scheduled in parallel on the PVP as highlighted in Figure 4.13. For TSR, the output of CNV1 is fed back to CNV2 and CNV3, and their joint output feeds PMA. For HB/LB the second stages is bypassed and CNV0 directly outputs to THC0. Overall, since multiple instances of same FB have identical functionality, there is no preference in allocation of FBs from same type. However, one particular FB allocation can be more desirable since it allows the higher number concurrent application execution on a single FLP. In our examples, one can use THC1 instead of THC0 for realizing HB/LB on PVP, or CNV1 and CNV0 can be replaced for implementing TSR and HB/LB.

To show further flexibility, Table 4.2 lists the mapping of 10 embedded vision applications to the PVP. For each application, we identified the FB allocation and mapping based on the PVP FSA. Table 4.2 also shows the order in which the FBs operate to give a sense of pipeline construction. Consider the mapping example of IBEM: CONV0 and CONV1 run in parallel in Stage1 followed by PMA in Stage2, THC0 in Stage3 and finally II0 in Stage4. Applications in Table 4.2 with the same
CHAPTER 4. FUNCTION-LEVEL PROCESSOR (FLP)

Table 4.2: Application Mapping to PVP.

<table>
<thead>
<tr>
<th>Stage 1</th>
<th>Stage 2</th>
<th>Stage 3</th>
<th>Stage 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>CNV0</td>
<td>CNV1</td>
<td>CNV2</td>
<td>CNV3</td>
</tr>
<tr>
<td>ACU</td>
<td>PMA</td>
<td>PEC</td>
<td>THC0</td>
</tr>
<tr>
<td>THC1</td>
<td>IIM0</td>
<td>IIM1</td>
<td></td>
</tr>
</tbody>
</table>

- HB/LB: Step 0
- Canny: Step 0, Step 1, Step 1, Step 2, Step 3
- Sobel: Step 0, Step 0
- LoG: Step 0, Step 1
- TBEM: Step 0, Step 0
- LH: Step 0
- IBEM: Step 0, Step 0
- IoV: Step 0, Step 0, Step 2, Step 3
- DoG: Step 0, Step 1, Step 1, Step 2, Step 3
- HoG: Step 0, Step 0, Step 1, Step 2, Step 3

-background color can run in parallel (HB/LB & Canny; Sobel & LoG, IBEM & LH, IBEM & IoV).

4.3.2 Computation / Communication Demands

To evaluate computation and communication complexity, based on the flexibility/efficiency tradeoff we compare three architecture solutions: a) FLP-PVP solution (highlighted in Figure 4.14a), b) a heterogeneous ILP-BFDSP+HWACCs solution (Figure 4.14b) and c) a ILP-BFDSP solution (Figure 4.14c). For ILP-BFDSP, we consider a fully ILP-based solution, meaning the entire application executes on Blackfin DSP cores at 600MHz. A comparison with ILP+HWACCs is difficult, because FLP offers a different flexibility/efficiency trade-off as other heterogeneous architectures. A

Figure 4.14: Architecture alternatives for result comparison
dedicated HW ACC for each individual application would yield optimal efficiency. However, such an SoC does not exist. In order to find some middle ground, we choose the most computation-intense as well as the most frequently appearing kernel to be implemented in HW, while the remaining computation occurs on the ILPs. In result, we consider HW ACCs for a low-pass filter (based on convolution) and color/illumination extraction in the heterogeneous ILP-BFDSP+HWACC. This selection is also supported by the fact that few existing MPSoCs already provide convolution filter as a dedicated HW ACCs [6].

Table 4.3: Computation demand

<table>
<thead>
<tr>
<th>Applications</th>
<th>FLP-PVP</th>
<th></th>
<th></th>
<th>ILP-BFDSP+ACCs</th>
<th></th>
<th></th>
<th>ILP-BFDSP</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>FLP Oper. [GOPs/s]</td>
<td>ILP Util. [%]</td>
<td>ACC Oper. [GOPs/s]</td>
<td>ILP Util. [%]</td>
<td>ACC Oper. [GOPs/s]</td>
<td>ILP Util. [%]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>HB/LB</td>
<td>7.387</td>
<td>0</td>
<td>4.316</td>
<td>159</td>
<td>0</td>
<td>370</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TSR(Canny)</td>
<td>15.106</td>
<td>0</td>
<td>12.616</td>
<td>135</td>
<td>0</td>
<td>756</td>
<td></td>
<td></td>
</tr>
<tr>
<td>HB/LB+TSR</td>
<td>22.493</td>
<td>0</td>
<td>16.932</td>
<td>294</td>
<td>0</td>
<td>1125</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sobel</td>
<td>9.628</td>
<td>0</td>
<td>8.466</td>
<td>66</td>
<td>0</td>
<td>482</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LoG</td>
<td>10.126</td>
<td>0</td>
<td>8.466</td>
<td>91</td>
<td>0</td>
<td>507</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sobel+LoG</td>
<td>19.754</td>
<td>0</td>
<td>16.932</td>
<td>157</td>
<td>0</td>
<td>988</td>
<td></td>
<td></td>
</tr>
<tr>
<td>IBEM</td>
<td>12.699</td>
<td>0</td>
<td>8.466</td>
<td>220</td>
<td>0</td>
<td>635</td>
<td></td>
<td></td>
</tr>
<tr>
<td>IoV</td>
<td>1.162</td>
<td>0</td>
<td>0.166</td>
<td>53</td>
<td>0</td>
<td>59</td>
<td></td>
<td></td>
</tr>
<tr>
<td>IBEM+IoV</td>
<td>13.861</td>
<td>0</td>
<td>8.632</td>
<td>272</td>
<td>0</td>
<td>694</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TBEM</td>
<td>12.367</td>
<td>0</td>
<td>8.466</td>
<td>203</td>
<td>0</td>
<td>619</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LH</td>
<td>7.387</td>
<td>0</td>
<td>4.316</td>
<td>159</td>
<td>0</td>
<td>370</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TBEM+LH</td>
<td>19.754</td>
<td>0</td>
<td>12.782</td>
<td>362</td>
<td>0</td>
<td>988</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DoG</td>
<td>14.608</td>
<td>0</td>
<td>12.616</td>
<td>110</td>
<td>0</td>
<td>731</td>
<td></td>
<td></td>
</tr>
<tr>
<td>HG</td>
<td>11.288</td>
<td>0</td>
<td>8.466</td>
<td>149</td>
<td>0</td>
<td>565</td>
<td></td>
<td></td>
</tr>
<tr>
<td>AVG</td>
<td>12.687</td>
<td>0</td>
<td>9.402</td>
<td>173</td>
<td>0</td>
<td>634</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 4.3 compares the computation demand for the 10 applications when operating on a 1280x960x30Hz video input over the three target architectures. The table lists the GOPs/s achieved by the FLP-PVP when executing individual applications. It also presents the estimated required ILP utilization for both ILP-BFDSP+HWACCs and ILP-BFDSP solutions when executing the same application. The FLP-PVP executes 12.6 GOPs/s on average and 22.4 GOPs/s on peak performance with no need for an ILP. In contrast, 634% ILP utilization is required on average for ILP-BFDSP
CHAPTER 4. FUNCTION-LEVEL PROCESSOR (FLP)

solution to offer same amount of computation (requiring 7 Blackfin cores at least). Our estimations make many favorable assumptions for the ILPs including optimal pipeline utilization without any stalls and also exclude data fetches from memory - rendering an ILP implementation unfeasible in a realistic setting. For ILP-BFDSP+HWACCs, the ILP utilization is lower – thanks to acceleration on HWACCs. However, the ILPs still run the remaining application, as well as schedule and synchronize the HWACCs. On average the ILP is 173% utilized (requiring 2 Blackfin cores). One example, up 4 cores are still needed for parallel (TBEM + LH) applications in addition to HW ACCs, while they can completely run on a single FLP.

Comparing communication demands in Table 4.4, we focus on off-chip traffic, since it significantly contributes to energy consumption in comparison to on-chip traffic (on-chip traffic consumes about 100x less power consumption than off-chip traffic). FLP-PVP shows constant communication (148 MB/s) across applications - only the transmission from PVP to ILP hits the memory interface. For ILP-BFDSP, we assume two implementations: (a) non-fused in which each FB is implemented as component and full frames are transferred, (b) fused which combines all function blocks into one top-level loop around all pixels. We optimistically assume that 90% of data traffic is hidden from off-chip memory. Even with our optimistic assumptions, non-fused and fused ILP-BFDSP solutions demand for 2.8 GB/s and 0.96 GB/s memory bandwidth on average which is at least 5x more than the FLP-PVP solution. For ILP-BFDSP+HWACCs case, 0.45 GB/s off-chip memory access is required on average (2.3x more than the FLP-PVP solution). For ILP-BFDSP+HWACCs, we also make optimistic assumptions including 128KB as the HW ACCs job size which allows keeping streaming data passed between HW ACCs and ILPs in on-chip scratchpads. To give an idea, on-chip fabric needs to support more than 1 GB/s on average and up to 1.7 GB/s to move data between HW ACCs and Blackfin cores.

4.3.3 Power Overhead

For power comparison, we measured the PVP compute-related power consumption on the TSMC 65nm Low Power (LP) process at 83MHz \[49\]. For Blackfin core, we measured 280mW in full-on mode (no memory traffic) on a comparable single core BF527 in 65nm process \[51\]. We assume LPDDR2 memory interface, consuming 40pJ per bit transfer \[68\] with 15pJ per bit for SDRAM off-chip memory access \[55\]. For the selected HW accelerators, we assume an equal power efficiency as measured for the corresponding PVP FB.

Figure 4.15 shows the power consumption for the architecture alternatives: ILP-BFDSP
Table 4.4: Off-chip communication demand

<table>
<thead>
<tr>
<th>Applications</th>
<th>FLP-PVP [GB/s]</th>
<th>ILP-BFDSP+ACCs [GB/s]</th>
<th>ILP-BFDSP Non-Fused [GB/s]</th>
<th>ILP-BFDSP Fused [GB/s]</th>
</tr>
</thead>
<tbody>
<tr>
<td>HB/LB</td>
<td>0.148</td>
<td>0.325</td>
<td>1.77</td>
<td>0.708</td>
</tr>
<tr>
<td>TSR(Canny)</td>
<td>0.148</td>
<td>0.413</td>
<td>2.655</td>
<td>0.797</td>
</tr>
<tr>
<td>HB/LB+TSR</td>
<td>0.295</td>
<td>0.59</td>
<td>4.424</td>
<td>1.505</td>
</tr>
<tr>
<td>Sobel</td>
<td>0.148</td>
<td>0.325</td>
<td>2.065</td>
<td>0.738</td>
</tr>
<tr>
<td>LoG</td>
<td>0.148</td>
<td>0.325</td>
<td>2.065</td>
<td>0.738</td>
</tr>
<tr>
<td>Sobel+LoG</td>
<td>0.295</td>
<td>0.502</td>
<td>4.129</td>
<td>1.475</td>
</tr>
<tr>
<td>IBEM</td>
<td>0.148</td>
<td>0.502</td>
<td>2.655</td>
<td>0.797</td>
</tr>
<tr>
<td>IoV</td>
<td>0.148</td>
<td>0.413</td>
<td>1.77</td>
<td>0.708</td>
</tr>
<tr>
<td>IBEM+IoV</td>
<td>0.295</td>
<td>0.767</td>
<td>4.424</td>
<td>1.505</td>
</tr>
<tr>
<td>TBEM</td>
<td>0.148</td>
<td>0.413</td>
<td>2.36</td>
<td>0.767</td>
</tr>
<tr>
<td>LH</td>
<td>0.148</td>
<td>0.325</td>
<td>1.77</td>
<td>0.708</td>
</tr>
<tr>
<td>TBEM+LH</td>
<td>0.295</td>
<td>0.59</td>
<td>4.129</td>
<td>1.475</td>
</tr>
<tr>
<td>DoG</td>
<td>0.148</td>
<td>0.413</td>
<td>2.655</td>
<td>0.797</td>
</tr>
<tr>
<td>HG</td>
<td>0.148</td>
<td>0.502</td>
<td>2.655</td>
<td>0.797</td>
</tr>
<tr>
<td>AVG</td>
<td>0.19</td>
<td>0.458</td>
<td>2.823</td>
<td>0.965</td>
</tr>
</tbody>
</table>

NonFused (ILP-NF), ILP-BFDSP Fused (ILP-F), ILP-BFDSP+HWACCs (ILP-ACC) and FLP-PVP (FLP). We distinguish between compute (blue) and communication (red) power. For both ILP and ILP+ACC we also consider an idealistic assumption of 0 power consumption for the sleep

![Figure 4.15: Overall power consumption of FLP-PVP, ILP-BFDSP and ILP-BFDSP+ACCs](image-url)
mode of ILPs that are not fully utilized. The Figure 4.15 demonstrates that FLP-PVP considerably reduces computation power consumption by 18x over both ILP-NF and ILP-F and close to 6x over ILP-BFDSP+HWACCs. The significant power reduction stems from FLP eliminating instruction overhead and general-purpose micro-architecture and data path, redundant data/instruction memory accesses, leading to a considerable power reduction on FLP-PVP solution. The FLP-PVP also shows a significantly lower communication power (red) in Figure 4.15 FLP achieves 18x, 7x and more than 2x power reduction in off-chip memory access in comparison to ILP-Fused, ILP-NF and ILP+ACC respectively, excluding the power of on-chip communication and SRAM buffers required for realizing ILP+ACC.

On average, our FLP-based solution consumes 14x-18x less power than ILP-based solutions and 5x less power than an hybrid ILP+ACC. Our results clearly demonstrate the excellent performance and power efficiency of the PVP even when making many unrealistically optimistic assumptions in favor of the compared ILP solutions.

### 4.4 Challenges and Opportunities

An FLP architecture offers new research opportunities/challenges when considering both FLP designers and users.

#### 4.4.1 Research Opportunities in FLP Design

The success of an FLP depends on the selection of FBs and their potential composition making it a crucial aspect defining flexibility and usability. New research is needed that shifts from optimizing individual applications to identifying common functions that are present in many applications of a market. The challenge becomes to define a small enough set of sufficiently composable functions that provide meaningful computation for a given market. A promising approach is to analyze existing programming frameworks (such as OpenCV for vision) for frequently used function primitives (as candidates to be an FB) and their composition. In addition to focusing only on compute intense functions, a more holistic view is advisable as the goal is to compute as much as possible within the FLP to avoid costly data movement. Conversely, a missing FB even if not compute intense may force expensive transfers to and from ILP. Therefore, a minimal but sufficiently contiguous set of FBs is desired. Similar to selecting a common functionality, common data representations have to be defined to enable FB to FB interaction.
CHAPTER 4. FUNCTION-LEVEL PROCESSOR (FLP)

Figure 4.16 outlines a conceptual design flow for realizing an FSA. The basic idea is identifying the required function per individual applications by extracting their Function Call Graph (CFG) out of the applications’ source codes. The next step is translating CFGs to Synchronous Data Flow (SDF) graphs. SDFs presentation is more suitable format for Function-Set exploration, as it exposes all possible orders and connectivity among the function within individual applications. After SDF extraction, an FSA explorer realizes the desired Function-Set based on function demands and connectivity of individual applications under exploration as well as design constraints (e.g., number of affordable FBs). For FSA exploration, possible solutions are graph matching algorithms or Function-Level synthesis and scheduling algorithms such as the proposed approach in [62].

4.4.2 Research Opportunities Simplifying FLP Use

With the higher abstraction of the FSA, opportunities arise to construct frameworks for simplified access and programming for the end-user. Points of interest include the allocation of FBs, i.e. selecting from a user specification and the simplified configuration of FBs. While computation complexity is hidden, some challenges resurface as composition and communication challenges. As such, bandwidth management, traffic policing is needed to facilitate real-time operation. An IDE is desired that for allocation and visualization of traffic patterns to simplify configuration. Scheduling
aspects can be studied when multiple applications concurrently execute on an FLP. At the beginning only a static scheduling is assumed. However FBs could dynamically join different data streams, which demand a dynamic scheduling, offering challenge about context switching and policies.

As part of our ongoing research we are working on tools to guide FLP designers in identifying an FSA for a market and FLP users to develop/compose their applications based on an existing FLP.

4.5 Chapter Summary

This chapter has introduced Function-Level Processors (FLPs) as a new processing element architecture that combines efficient execution (performance and power) with sufficient flexibility that allows executing different applications within the same market. An FLPs consist of a set of configurable Function Blocks (FBs), each implementing a kernel function. FLPs offer a flexible interconnect network that allows composing FBs into a custom pipeline. We demonstrated FLP benefits using an industry example of the Pipeline Video Processor (PVP) part of the ADI BF609. We highlight the flexibility by mapping ten embedded vision applications to the PVP. This configurability and efficiency make FLPs ideal candidates for market-oriented MPSoCs. Markets of interest include embedded vision computing, software defined radio, and bio-medical analytic. This chapter demonstrates FLP’s benefits by using an industry example: the Pipeline Vision Processor (PVP) [49]. The FLP is a generalization of the lessons learned from the PVP design toward a more general architecture template for realizing function-level processing. We highlight the flexibility by mapping ten embedded vision applications to the PVP, realizing 1280x960 30Hz real-time embedded vision processing solutions. The FLP-PVP has a much lower power consumption over comparable ILP and ILP+HWACC solutions (on average 14x-18x and 5x reduction respectively) while at the same time avoiding most of the memory traffic.
Chapter 5

Conceptual Abstraction Levels (CALs)

This chapter proposes a set of higher abstraction levels called Conceptual Abstraction Levels (CALs). CALs guide architects to develop a more expressive architecture specification model by identifying architecture challenges and potentials at early stages of design. As an example, CALs help to identify applications taking benefits of traffic separation or are suitable for function-level processing.

5.1 Architecting Complexity of Heterogeneous MPSoCs

Increasing abstraction levels is a common approach to manage design complexity in Electronic System Level (ESL) design \cite{32,43,90,98}. Abstraction allows focusing on a global system scope rather than dealing with implementation details. Top-down ESL flows aim to mitigate system-level design challenges, start with executable system specification as an input model and yield a system implementation \cite{32,33,39,43,56,64,74,93}. They generate multiple Transaction-Level-Models (TLMs) in the process for detailed analysis of modeled platforms and performance estimation of architecture alternatives. By initially starting with an executable specification, the platform mapping opportunities highly depend on the quality and flexibility of the entry specification model. With the architecture refinements and TLM generation being increasingly automated, 90% of the design effort is now spent for developing the efficient system specification \cite{21}. We have noted that many essential architectural decisions with significant impact on final product costs are already implicitly locked down in the system specification, such as algorithm selection, parallelism type, communication / computation granularity, and data types. At the same time, Intellectual Property (IP) considerations are often either implied or declared as part of the system specification model. As
CHAPTER 5. CONCEPTUAL ABSTRACTION LEVELS (CALS)

a result, many decisions are seemingly made at once, often based on prior considerations, leading to tightly intertwined and highly challenging decision process. To improve the potential for the ESL design process, more attention, focus and effort is needed toward developing efficient system specifications.

Developing a system specification is a complex and time consuming process; considering market-oriented MPSoCs that span a multitude of demanding markets and applications with often conflicting requirements yielding highly heterogeneous platforms. System architects currently often have to rely on their evolving knowledge and experiences from the previous products to tackle complexity of multi-dimensional design spaces. However, apriori-based SoC design cannot address emerging challenges. Abstraction definitions and supporting automation is missing, which defines the abstraction gap between requirements and system specification. New solutions are required that can fill the abstraction gap and improve the design process. To guide system architects when creating a system specification, it is paramount to identify individual areas of exploration and coordinate the decision making process. Implicit decisions buried in the specification model need to be exposed in a design flow to be explicitly managed individually. Example decisions include algorithms selection, identification of possible bottlenecks for computation, memory and other interfaces, early consideration of IP reuse, definition of quality and functional flexibility / configurability.

In this chapter, we propose a set of higher abstraction levels called Conceptual Abstraction Levels (CALs) to aid the development of an efficient system specification for market-oriented MPSoCs. CALs are a result of a joint industry / academia effort aiming to fill the gap between market requirements and the system specification. By identifying abstraction levels, their characteristics and necessary transition decisions, CALs expose essential system properties at higher levels of abstraction. With the guidance offered by CALs more efficient specifications become possible, ultimately yielding to a more flexible products with lower cost parameters such as power, bandwidth, and/or die area. This captor mostly focuses on identifying abstractions above executable system specification, characterizing individual levels and the decisions necessary to transition across levels. This leads way to highlighting the potential for associated EDA tools for refinement or aiding the decision making process.

To demonstrate the efficiency and effectiveness of our proposed design flow, we applied CALs to the demanding market of embedded vision computing. Embedded vision computing refers to deploying visual analysis and computer vision capabilities to embedded systems [14], covering a variety of markets/applications with a high diversity in algorithms and conflicting requirements [14, 17]. In this chapter, we focus on primary application kernels within embedded vision markets.
CHAPTER 5. CONCEPTUAL ABSTRACTION LEVELS (CALS)

Application kernels include Background Subtraction, Object detection and Object tracking. We specifically concentrate on Background Subtraction as one of the most challenging and computationally complex applications within this area. Following our CALs abstractions, we develop an efficient specification model which exposes algorithm choices, identifies processing element classes, suitable parallelism, and hierarchy more effectively. As one aspect, we explore the tradeoff between image quality and external memory bandwidth for Background Subtraction. The resulting specification model serves as an input for a top-down downstream ESL flow for further fine grained exploration, synthesis and implementation.

5.2 Conceptual Abstraction Levels (CALs)

This section first overviews the proposed CALs approach providing the context. It then introduces key aspects to be considered when defining abstraction levels, their properties and the transition between levels. This will guide the later discussions.

Figure 5.1: Conceptual Abstraction Levels (CALs) above system specification.

Figure 5.1 captures the essence of our proposed CALs flow starting from exploring market
requirements and yielding the system specification captured by an SLDL. CALs abstractions are: (1) Market Requirements, (2) Application-Block Flow, (3) Algorithm Specified, (4) Executable Application Block, (5) System Specification. Each abstraction level explores one aspect of the system design space. Individual abstractions can leverage a database or profiled information for decision making and refinement process. The system specification in result of the CALs is then the basis for system-level explorations using Transaction-Level Models (TLMs).

Looking at Figure 5.1, we distinguish CALs internally into two phases: (1) CALs-analytical; (2) CALs-executable. This distinction signifies whether CALs models are analytically-based (only reason about properties), or are executable (ie. have the algorithm captured in a language). CALs-analytical abstractions are independent of algorithm captured in a language and not necessarily need to be executable. CALs-executable models can be captured in an SLDL, but still aim for an iterative refinement and improvement to reach the desired system specification model. In one aspect, an executable model also helps to explore the algorithm / architecture co-tuning possibilities, parallelism, decomposition and data-type vs quality. However, a CALs abstraction captured by an SLDL is not necessarily a system specification model, since many design decisions (parallelism, decomposition, data type) still need to be made.

In a glance, (1) Market Requirements encapsulate functional and non-functional requirements for a set of markets with similar functionality. Then, at (2) Application-Block Flow, the functionality of identified markets is split into smaller and chain-able application block with independent meaningful functionality. At this level, we also propose IP-reuse selection as an internal refinement step within the Application-Block Flow to facilitate early consideration of IP reuse. (3) Algorithm Specified explores and selects a suitable algorithm with respect to market demands for individual application-block. Following that, (4) Executable Application-Block captures each application-block in an executable language for profiling and analysis of computation / communication demands. This provides an opportunity to identify a specific PE-class type (general-purpose processor, DSP, GPU, custom-HW) with respect to profiled demands. At this level, we propose Application-Block tuning as a refinement step within Executable Application-Block for tuning blocks based on selected algorithm and targeted PE-class. This enables algorithm / architecture co-optimization to satisfy the market requirements. Finally, (5) System Specification captures an integrated model of the system functionality in an SLDL. At system specification the effort shift toward integration challenges between the application-blocks and their composition, including system-level parallelism / pipeline and communication mechanism (channels, shared memory) between application blocks. The resulting system specification is the input for a ESL design flow for detailed architectural exploration.
CHAPTER 5. CONCEPTUAL ABSTRACTION LEVELS (CALS)

through TLMs and implementation exploration.

Figure 5.2: Transition between two abstraction levels.

To structure our CALs semantics, we use semi-formal semantics capturing the essential elements involved in the transition between CALs abstractions. In this article, we mainly focus on explicitly exposing and organizing the implicit design decisions above system specification. For this, semi-formal notions with straightforward semantics are suitable to guide our CALs descriptions and avoid possible confusions. More formal notations annotations will be developed as a step toward automating higher abstractions.

Figure 5.2 illustrates the transition from a higher abstraction \((N - 1)\) to a lower abstraction \((N)\). The transition consists of four components: Rationales, Decisions, Refinement and Database. Decision are determined either manually or automatically feeding into the refinement. Rationales capture the reasoning (e.g. by human) that have led to the design decisions. Explicitly captured rationales are useful for later iterations when revisiting (potentially revising) an earlier decision. Refinement is the process of realizing desired design decisions in a new abstraction \(N - 1\). The result is a new abstraction which exhibits new system characteristics and observable features. For the purpose of this article, refinement can either be realized as a tool, or through manual modifications. A refinement tool realizes the design decisions utilizing the associated databases. The database typically contains profiled components to aid the decision making, and the necessary structural definitions for the refinement to the next abstraction.

For the organization of this chapter, we analyze each level in four aspects:

- **Decisions**: describe changes needed to transition from previous \((N-1)\) to current abstraction \(N\).
- **Characteristics**: show the observable features of the refined abstraction.
- **Example**: highlights the reasoning, decisions and transitions based on the case study embedded vision focusing on *Background Subtraction*.  

69
5.2.1 Market Requirements

We consider the Market Requirements to be the highest abstraction level. The requirements are often identified by key technologists/architects after surveying market potentials and customer needs. They identify the need for a new or next generation solution.

Decisions: At this level, decisions cover both functional and non-functional requirements as demanded by the identified markets. On the functional side, the key is to identify a group of similar applications with closely related functionality as this defines the scope of work, as well as determining the quality requirements. Typical non-functional requirements include design constraints such as power, performance, quality and time-to-market.

Key technologists/architects collaborate with customers directly to gather and validate requirements. Already at this level, the requirements are shaped and grouped to better match the targeted market. At a more in-depth stage, the requirements are refined by a cross-functional team including: product and test engineering, applications, software and tools, manufacturing, design, and architecture. A balance is desired across these functional areas with marketing and technologists leading the process to define market requirements. As such, decisions also include feasibility and NRE cost estimation, taking into account targeted marked volume and feature set coverage.

Characteristics: The characteristics include identified markets and their requirements. The targeted functionality can be observed as a set of coarse-grained input/output system interfaces and their correlation. Boundaries for performance, quality, energy and cost are defined. Based on these characteristics, the suitability of existing technology can be evaluated and the demands for new technology derived.

Example: Our case study looks at the embedded vision related markets. Embedded vision comprises a few, fast growing markets involving high-performance embedded computing such as Advanced Driver Assistance System (ADAS), industrial vision and video surveillance. The surveillance market alone is estimated to more than triple from $11.5 billion in 2008 to $37.5 billion in 2015 [36]. The global value of ADAS market has an even a 13-fold projected growth from $10 billion in 2011 to $130 billion in 2016 [4]. These markets share the same top-level functionality of real-time video processing, including modeling of captured scenes and object analysis. For
the surveillance market, analysis examples may include identification of an object left behind and detecting when a secured zone is entered.

Common to all is a system input of a video stream with resolutions of 1920*1080 pixels (Full-HD) or 1280*960 pixels (Half-HD), and frame rates of 60Hz or 30Hz, depending on application. This drives the need for fast, reliable camera interfaces, high-performance computation, and significant memory bandwidth. Nonetheless, latency requirements may largely differ. While an ADAS application may require decisions within a few milliseconds, a surveillance application may tolerate sub-second decision latency. Even different deployments of the same application may have different demands. A surveillance deployment may differ in demands based on the complexity of the scene under observation: tracking people on a busy street with rapidly changing surroundings is more demanding than a subway station scene with minimal environmental changes. We will focus on embedded surveillance for our continued example.

Potential: At this level, requirement management tools can be expanded to aid in the process of exploring and tracking functional and non-functional requirements of similar markets. Example tools include IBM Rational DOORS [11] and Rational Jazz [83]. Requirement management should be expanded to provide facilities to investigate and list potential new markets with their matching and conflicting requirements as a set of meta-data information. However, the main challenge remains to extract technical requirements out of matching and conflicting requirements in a set of similar markets.

5.2.2 Conceptual Application-Block Flow

At the Conceptual Application-Block Flow level, system functionality is broken down into a sequence of behavioral or functional blocks that are connected through input / output interactions. The functionality for an application-block is defined as an input / output correlation rather than in terms of an algorithm. The aim of application-block flow is to close the gap between market and algorithm selection by identifying what has to be done while leaving the how open. In the result, in application-block flow the functionality is captured in an abstract way as an input to output correlation while no particular algorithm is specified. In addition, having functionality distributed to application-blocks, as well as identifying their connection enables analysis of commonalities (functionality distribution and connectivity) across applications within the targeted market.

The application-block flow can appear as a coarse-level system pipeline. Its main purpose is to breakdown the entire system functionality (captured in market requirement) into individual
blocks with the aim to be independently explored and optimized. Note that we do not study at this how the decision for one block impacts another block. This is part of later abstractions when resource sharing and synchronization mechanisms are identified to some extent.

**Decisions:** The system architect refines the market functional requirements into one or more conceptual application-block flows, each composed of a set of application-blocks describing part of entire system functionality. Identifying the application-blocks of each application-block flow enables analyzing common aspects across multiple applications / markets.

At this level, the decomposition of desired market functionality (see market requirements) should be realized. To achieve this, the decisions involve dividing the desired functionality into a set of meaningful blocks, which we call application-blocks, and also identifying application-blocks that can be shared across targeted markets. To achieve this, system architect identifies the desired functionality of individual application-blocks, as well as their input / output relations. Each application-block needs to have a distinguishable meaningful functionality can be composed with other application-blocks to construct the targeted system functionality. At this abstract stage, the desired functionality of application-blocks is independent of an exact algorithm selection. A sorting block, for example, would have an unsorted array as an input and sorted array as output; the exact sorting algorithm, however, is unspecified. In addition to functionality, the decision include connection and interaction between application-blocks. This enables early discussion about system-level connectivity.

**Characteristics:** With dividing markets functional requirements into individual conceptual application-block flows, common application-blocks across targeted markets become observable. In addition, interactions of application-blocks are observable as a initial indication for connectivity between application-blocks. This enables judging possible application-block compositions across markets.

**Example:** Figure 5.3 highlights a generic application block flow to support both video surveillance and ADAS. We mostly focus on the surveillance market for our discussion. Nonetheless, the flow still partially supports ADAS (it does not support a moving camera). Figure 5.3 consist of **Pre-processing** for frame smoothing and RGB to gray translation, **Background Subtraction** for separating the foreground (FG) - which are moving objects - from the background (BG), **Object Detection** for identifying objects in FG pixels, **Object tracking** for tracking of identified objects across multiple frames, and finally **Trajectory Analysis** for analysis of the objects’ trajectory within the scenes. All five application blocks are common in the area of embedded vision computing with more emphasis in video surveillance market.

**Potential:** Opportunities for tools and automation lie in capturing conceptual flows with
CHAPTER 5. CONCEPTUAL ABSTRACTION LEVELS (CALS)

5.2.2.1 IP-Reuse Selection

IP-Reuse is considered as a refinement step within the Conceptual Application-Block Flow level. The main goal is to identify the IP blocks that are already available either in company internal or as a 3rd party IP.

Decisions: This is the most abstract level at which IP-reuse can be considered. IP-reuse is an essential means to reduce design and development cost as highlighted by ITRS [2] which predicts reuse rates to increase from 54% in 2011 to 98% in 2026, as well as emphasized in the component-based design [61]. In component-based design the focus lies on integrating already developed and well-understood blocks while also designing new application-blocks as per demand. At this level, the designer identifies the application-blocks that exist in previous product generations while matching the new requirements. With integration of 3rd party IP, additional application-blocks for supporting protocol or interface translation may also be considered.

Characteristics: At this level the IP-reuse potential is evaluated which includes identifying reuse of existing components and application-blocks. This in turn allows system architects to focus efforts on the new application-blocks.

Example: Figure 5.4 exemplifies IP-reuse for our case study. An IP-block already exists
CHAPTER 5. CONCEPTUAL ABSTRACTION LEVELS (CALS)

for Pre-processing with gray conversion and convolution filters for noise reduction. Similarly, a classification algorithm based on trajectory analysis was already implemented in SW in a previous generation already meeting market requirements and constraints. The architect relies on existing IP thus inherits the HW, and SW mapping respectively. In contrast, Background Subtraction, Object Detection and Object Tracking are new application-blocks for this design requiring further investigation.

Potential: Significant automation potential exists for realizing IP-reuse possibilities. This will require databases to categorize functionality and constraints of IP blocs. Current databases, such as IP-XACT\[52\], mainly focus on interface compatibility without capturing behavioral characteristics of IPs.

5.2.3 Algorithm Selection

The exploration at Algorithm Selection focuses on identifying algorithms for individual application-blocks. Selecting a suitable algorithm for each application-block can be seen as an algorithm space exploration traversing a multidimensional space of functionality, quality, platform demands, parallelization potential and algorithm adaptability to name a few. As such, considerations include functional and quality characteristics which define the algorithm’s suitability for an application and execution environment. Algorithms differ in their demand to the execution platform, as well as in their potential for parallelization (e.g. divide and conquer vs. iterative vs. extensively parallelizable).

Decision: Rather than developing new algorithms, the decisions at this level involve identifying the most efficient algorithm from a set of algorithm candidates. For each application-block, the designer selects the most suitable algorithm through exploring the multidimensional space outlined above. Guidelines can be given by estimating order-of-magnitude bounds for computational demands, memory complexity (size, traffic), and by analyzing the parallelism / flexibility potential of each available algorithm. In the context of market solutions, the algorithm flexibility / configurability is additionally important to allow adaptation to market demands with unique performance / quality tradeoff.

Characteristics: At this stage, algorithm candidates with their coarse-grain platform requirements and features are identified. These allow for estimating computational, memory and bandwidth demands. Without an algorithm representation in a concrete language however, only coarse order-of-magnitude estimations are possible.
Table 5.1: Characteristics / Comparison of background elimination algorithms.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Type</th>
<th>Computation</th>
<th>Memory</th>
<th>Quality</th>
</tr>
</thead>
<tbody>
<tr>
<td>Average Median</td>
<td>History-Based</td>
<td>Low O(N)</td>
<td>High O(M*N)</td>
<td>Low</td>
</tr>
<tr>
<td>Mean Shift</td>
<td>History-Based</td>
<td>High O(M*N)</td>
<td>High O(M*N)</td>
<td>Highest</td>
</tr>
<tr>
<td>Eigenbackground</td>
<td>Adaptive</td>
<td>Medium O(M)</td>
<td>Medium O(M)</td>
<td>Medium</td>
</tr>
<tr>
<td>MoG</td>
<td>Adaptive</td>
<td>Medium O(M)</td>
<td>Medium O(M)</td>
<td>Good</td>
</tr>
</tbody>
</table>

**Example:** We mainly focus on Background Subtraction for further analysis. Background subtraction extracts FG pixels out of a video input stream isolating moving objects. Multiple algorithm alternatives are available with some listed in Table 5.1. Following the survey of Background Subtraction algorithms in [26], Table 5.1 lists computation and memory complexities with relative comparison as well as order of magnitudes where M is number of pixels and N is number of history frames (e.g. around 100 frames). The relative quality refers to the ability to identify FG pixels. The algorithms can be divided into: history-based and adaptive algorithms. History-based algorithms operate on a history of video frames leading to high memory capacity requirements for storing past frames and well as large bandwidth requirement of accessing the frames. Conversely, adaptive algorithms track the background with a model. They operate only on the current video frame to iteratively update background parameters. They handle gradual variations better and have lower memory requirements.

For our work, we select Mixture of Gaussian (MoG) [79] as it provides sufficient quality with acceptable memory and computation requirements. MoG uses multiple Gaussian distributions, also called Gaussian components, to model each pixel’s background. Each component uses three parameters: Mean (BG gray scale intensity), Weight (as indicator how accurately it tracks current BG) and Deviation (distribution over Mean). The Gaussian parameters are updated with each frame to track BG changes. Gaussian components are varied in learning and sensitivity rates enabling them to track a range of BG changes (e.g. light changes vs. rustling leaves). MoG offers configurability [79] (e.g. # of components) to adjust to scene complexity or desired output quality. One advantage of MoG is its systematic configurability. By changing the number of Gaussian components, the quality can be adjusted at cost of additional computation and memory bandwidth.
CHAPTER 5. CONCEPTUAL ABSTRACTION LEVELS (CALS)

For object detection we selected Blob Detection algorithm to find the new objects in the scene [100]. The Bob Detection is a contour-based algorithm to identify the border of objects through topological structural analysis of the foreground mask (further discussion omitted for brevity). For Object Tracking, many algorithms aim to offer robust techniques in various conditions (such as partial or complete occlusions). Here, we choose Blob Tracking based on Mean-Shift algorithm to find a most-likely template or possibility in a local area, and set the center of the blob to that point [29]. Mean-shift is a non-parametric (i.e. kernel) density estimator that exploits similarity in Probability Density Function (PDF) of pixels to find the direction of the target’s movement. Figure 5.5 shows the flow with the selected algorithms, representing the refined abstraction.

Potential: There is great automation demand for algorithm space exploration aiding the quest to select the most suitable algorithm based on functionality and platform demand. At the same time however, challenges are significant, e.g. on how to categorize algorithms for selection and how to evaluate a chains of algorithms. Current tools, such as Matlab/Simulink [70] and LabView [69] offer large databases, and subject-specific collections, such as OpenCV [16], exist. However selection and evaluation are completely manual. Automation is needed to extract platform requirements, and potentially evaluate quality considerations. Characterized algorithm databases offer a good initial interface between algorithm development teams and system architects.

5.2.4 Executable Application-Block

Application-blocks with their selected algorithms need to be captured in a programming language to obtain an executable model. With an executable model, run-time properties can be more accurately extracted such as computation demand and memory traffic. Still at this level, the aim is a high abstraction with a main focus on computation / communication demands (i.e., the amount of computation and communication). Since the actual PE type (i.e., a particular processor type) is not selected, the aim is to quantify the amount of computation and communication (which is PE type independent) rather than identifying absolute time. This enables evaluating the suitability for different PE-classes. In other words, we aim for an early comparison exploring across different
CHAPTER 5. CONCEPTUAL ABSTRACTION LEVELS (CALS)

PE-classes (e.g. SW, HW, GPU) to retain the high level overview. Selecting a concrete PE-type within a PE-class occurs later in the envisioned flow.

**Decisions:** Selecting a language representation for each algorithm determines much of the available parallelism and suitability for execution on a processing element class (e.g. CPU vs. GPU vs. Custom HW). Potentially largely different algorithm implementations may exist. Nonetheless, at this level a general purpose language, such as C/C++, or a System-Level Design Language (SLDL), such as SpecC [40] and SystemC [72] can be employed in the consideration that final implementation language may change again. The main goal at this stage is to obtain early estimates for system metrics, such as computation demand, communication volume and power, for individual application-blocks. With the executable model, the system metrics can be estimated independent of a specific mapping or targeted toward a particular processing element class. As part of the performance / power / quality trade-off, the designer can select suitable data types for processing.

**Characteristics:** The first executable model can serve as a golden reference model for functional validation along the design process and possibly again later in the development lifecycle. As such, the overall functional quality of the designed system is observable (e.g. as influenced by data type selection). With the application-blocks and selected algorithms captured in a programming language, target-specific performance estimation is possible. Furthermore, memory traffic (implicit and explicit) is observable which allows early identification of interconnect / memory interface bottlenecks. Overall, this allows identifying potential system bottlenecks and motivates for application-block tuning.

**Example:** We captured the *Mixture of Gaussian (MoG), Blob Detection* and *Blob Tracking* in SpecC SLDL for ease and convenience of analysis. We used the profiling capabilities of System-on-Chip Environment (SCE) [34] to capture computation and communication of each application-block. Figure 5.6 shows the computation demands (Figure 5.6a in number of operations per second [GOPs/Sec]) and memory access demands (Figure 5.6b in number of bytes transferred per second [GBs/Sec]) for individual application blocks. The profiling is based on Full-HD (1920*1080 resolution) and half-HD (1280*960 resolution) formats, with the frame rate of 30 Hz.

As highlighted in Figure 5.6a, while all three application-blocks have a fairly high computation demands, *Mixture of Gaussians (MoG)* is most demanding with around 12 GOPs/Sec for Full-HD. MoG dominates with computation for each individual pixel to remove the background and generate meta-data for Blob tracking and detection. MOG’s computational complexity makes it inefficient and too costly to implement in software, and conversely an excellent candidate for a hardware realization. Similarly, Figure 5.6b shows that the communication demand of MoG is much
higher than Blob Detection and Blob Tracking application-blocks. The high communication demand of MoG stems from fetch and write back of Gaussian components. For each pixel the background model (consisting of 3-5 Gaussians with 3 parameters each) is updated with each incoming image frame. For each frame, depending on resolution, 248MB (146MB) are accessed when reading and writing Gaussian parameters. This results in 7464MB (4360MB) of traffic per second (assuming 32bit parameter length). The high computation and communication demands prompt us to further investigate MoG’s resource utilization and the associated power consumption.

Table 5.2: Estimated computation resource for Mixture of Gaussians (MOG) algorithm with the frame rate of 30 Hz.

<table>
<thead>
<tr>
<th>Image size</th>
<th># of Blackfin Cores</th>
<th># of ARM Cortex-A9 Cores</th>
<th>Util. of Memory Bandwidth [%]</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Float</td>
<td>Fixed</td>
<td>Float</td>
</tr>
<tr>
<td>1920*1080</td>
<td>11</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>1280*960</td>
<td>7</td>
<td>4</td>
<td>5</td>
</tr>
</tbody>
</table>

For the purpose of illustrating SW resource utilization and giving concrete examples, we have estimated resource utilization for a SW realization on two embedded processors: (1) Blackfin
CHAPTER 5. CONCEPTUAL ABSTRACTION LEVELS (CALS)

52x core with 600MHz [50], and ARM Cortex-A9 with Neon-SIMD extension with 1GHz [10]. These are two examples of frequently used processors. A more detailed comparison (with more PE-types) will occur later in the flow (TLM architecture).

For estimating the resource utilization shown in Table 5.2, we make many favorable assumptions for the processor cores including optimal pipeline utilization without any stalls and also exclude data fetches from memory. Looking at Table 5.2, 11 cores and 7 cores are respectively required for executing MoG in Full-HD and Half-HD resolutions on Blackfin cores. For ARM-Cortex A9 (including Neon-SIMD co-processor) the estimations slightly show lower number of cores with 8 and 5 cores for executing MoG in Full-HD and Half-HD resolutions.

To reduce the resource demands of MoG in SW, we consider shifting from floating-point to fixed-point execution. This potentially reduces utilization as fixed-point operations have lower computation overheads. Our computation profiling shows that approximately 76% of ALU operations (and 60% of overall operations) of MoG application-block are floating-point operations. In the SW implementation, we have converted all floating-point operations to fixed-point. The fixed-point version is 1.5x to 1.7x times faster. The speedup is limited since only 60% of the total computation was modified. Still, 7 and 4 Blackfin DSPs, and 5 and 3 ARM Cortex-A9 cores are required for executing the fixed-point MoG.

Looking at communication utilization, Table 5.2 shows a significant memory traffic for SDRAM memory interfaces. We have estimated the MoG communication resource utilization for three different off-chip memory controller interfaces: DDR3-1066 and LPDDR2-800. MoG would over-saturate DDR2-800 for Full-HD resolution (with the giving the same results for its lower power variant LVDDR3-800). Even the more powerful DDR3-1066 memory interface would operate at 88% of its theoretical peak performance making such an implementation unrealistic.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Float</td>
<td>Fixed</td>
<td>Float</td>
</tr>
<tr>
<td>1920*1080</td>
<td>3.2</td>
<td>1.8</td>
<td>3.3</td>
</tr>
<tr>
<td>1280*960</td>
<td>1.9</td>
<td>1.1</td>
<td>2.4</td>
</tr>
</tbody>
</table>

Table 5.3 shows estimated power consumption of MoG based on the determined com-
CHAPTER 5. CONCEPTUAL ABSTRACTION LEVELS (CALS)

Computation and memory bandwidth resources. Our power estimation is based on assumptions that one Blackfin core consumes about 280 mW [7] and one ARM-Cortex A9 core with Neon-SIMD consumes 500 mW[14]. Processing MoG in Full-HD on Blackfin cores would consume 3.2 W with floating-point and 1.8 W with fixed-point operations. The estimated power is slightly higher for Cortex-A9 with 3.3 W (floating-point) and 2.1 W power (fixed-point). The power demands of the memory interface is also very high. When operating in Full-HD, the memory interface consumes 4.3 W for DDR3 and 2.1 W for LPDDR2 (based on the results in [67]).

Combining resource and power estimations, we conclude that a SW implementation of MoG is not cost efficient and outside the power budget (considering that entire SoC should consume less than 2 W). One alternative is to implement MoG as a Custom-HW. Assuming one operation in HW requires 3.8 pJ in 65 nm TSMC fast process [8], the estimated power consumption only for the operations would be 46 mW (Full-HD) and 27 mW (Half-HD) – far below the SW implementation. However, the challenge arises to maintain enough flexibility in the custom-HW implementation to support all targeted markets. In addition, the high memory bandwidth remains an critical issue in HW implementation needing further exploration.

While a HW implementation of MoG seems necessary to realize the power goals, the estimations in Figure 5.6a suggest that a SW solution could be sufficient for Blob Detection and Blob Tracking application-blocks. Their computation demands are within boundaries of embedded processors (e.g DSPs or ARM with Neon SIMD within acceptable power budget) and furthermore have fairly low memory bandwidth requirements. Blob Detection and Blob Tracking can be explored at finer level of granularity after finalizing the system specification. Figure 5.7 summarizes executable model and also shows the intended mapping to PE-classes.

![Figure 5.7: Specified PE-classes at Executable Application-Block level.](image)

**Potential:** Automation potential exists for identifying computation, memory and communication demands targeted for each class of PE. Much research effort has already been investigated into the estimation for reasons of worst case or average execution time analysis [117]. Such tools can help in identifying bottlenecks. At this level, estimating the exact execution time is not necessary. The main point is identifying computation / communication demands with respect to the required resources and also their associated power consumptives. Early examples that illustrate the tool poen-
CHAPTER 5. CONCEPTUAL ABSTRACTION LEVELS (CALS)

... are estimating HW demands, and for SW demand estimation. Additionally, toolboxes, such as floating to fixed point conversion (e.g. in SUIF compiler suite) help transformation to investigate the effect of standard data types on resource utilization.

5.2.4.1 Application-Block Tuning

Application-Block Tuning is not considered a new abstraction level itself but rather a gradual refinement part of the earlier described abstraction, the Executable Application-Block (Section 5.2.4). Here, the algorithm implementation is adjusted for an anticipated PE class in order to reduce architectural demands, such as computation, communication, memory and power. This refinement offers potential for algorithm / architecture co-design for tuning both algorithm and architecture to meet the requirements.

**Decisions:** At Application-Block Tuning level, the designer adjusts the application-blocks to better fit the anticipated PE class (CPU, GPU, HW). This can involve modifying existing blocks, such as changing data types, adjusting the exposed parallelism, or adding support blocks that aid the functional blocks (e.g. compression or preprocessing). The overall goal is to meet functional / non-functional requirements (performance, bandwidth, quality, power), as well as offering sufficient configurability for a range of markets. Decisions may include modifications to the algorithm to deliver more efficient execution on the targeted PE-class. During tuning, the system designer may identify the need for additional support blocks based on the already identified bottlenecks to meet system requirements.

**Characteristics:** The same characteristics as previously identified are observable here. In addition however, new support blocks may be presented to aid the main application-blocks. Overall, the characteristics reflect the tuning improvements such as finer-grain parameters type, higher degree of freedom and configurability.

**Example:** For a MoG implementation in HW, the computation demand is not problematic as sufficient parallelism (across Gaussian components) and pipelining opportunities (within components) exist. Our main goal for tuning is to reduce the memory bandwidth. We therefore explore the bandwidth / quality trade-off to identify the lowest possible bandwidth with acceptable quality for each market. Bandwidth reduction is possible by reducing the number of Gaussian components or reducing the bit-width of parameters when stored in memory.

As shown in Figure 5.8, we add a support block which compresses / de-compress parameters interfacing with memory. It is configurable to compress the three Gaussian parameters: mean,
deviation and weight. While internal processing is still performed at 32 bits, before storing back to memory, they are compressed to 8 and 24 bits each. To analyze the tradeoff, we exhaustively explore across lengths of Gaussian parameters (8 to 24 bits) and number of Gaussian components. We evaluate quality by comparing against a ground truth obtained from the reference algorithm (floating point). For quality assessment, we use MS-SSIM as a good indicator for visual quality focusing on structural similarity rather than direct pixel comparison.

Figure 5.9a plots quality (MS-SIM) on y-axis over memory bandwidth in result of the different configurations (# Gaussians, length). By increasing the number of Gaussian components quality improves at the cost of additional memory bandwidth. However, qualities between different # of Gaussians overlap. Furthermore, different configurations (e.g. in parameter length) yield the same bandwidth, however show a varying quality. To identify which configurations yield the best quality, we analyze the impact of parameter lengths and number of Gaussians components on quality. Figure 5.9b is just one exploration example showing the quality correlation between # of Gaussian components and length of deviation parameter (assuming 32-bit length for Mean and Weight). Figure 5.9b shows that deviation stored with 16 bit already achieves best possible quality for
integer operation.

We systematically analyzed and explored the impact of parameter bitwidths (individually for mean, weight, and deviation) as well as number of Gaussian into the quality of subtracting the background. Figure 5.10a generalizes our findings for scenes of different complexity (e.g. varying in number of moving objects, illumination, shadows, and many other details). For instance, for a medium-complex scene 14 bits for mean and deviation, and 12 bits for weight can achieve maximum quality of fixed point implementation. We observed that equal # of bits should be allocated for deviation and mean, while weight may require fewer bits.

![Figure 5.10: Generalization of memory bandwidth benefit in MoG](image)

By reducing parameter bit-widths we significantly reduce memory bandwidth without visible or quantitative loss in quality. Figure 5.10b shows the memory bandwidth reduction for different scene complexity in comparison to a 3 bits implementation. For the simple scene, memory bandwidth reduces from 4.3 MBytes/s to 1.6 MByte/s (63%), while bandwidth in medium and complex scenes is reduced by 59% and 56% respectively. This saving will come at cost of an increase in computation due to parameter compression/decompression. Since compression is not on a critical path, the latency does not necessarily increase.

![Figure 5.11: Identified support blocks after Application-Block Tuning](image)

Figure 5.11 shows our updated vision flow. In order to realize MoG in an embedded archi-
tecture the parameter compression/de-compression is required to reduce the bandwidth requirement. For this a parameter compression/de-compression block is added to the main flow.

**Potential:** Tools are needed to aid analysis of proper data types and bit widths. In addition, estimation and analysis of memory traffic and computation demand are essential. More generally, tools for exploration and analysis multidimensional trade-offs are needed. One example of potential tools at this level is the simulation-based tuning approach presented in [122], focusing on mixed analog/digital design.

### 5.2.5 System Specification

The higher abstractions in CALs have focused on identifying design aspects of the individual application-blocks. Now, the specification model is the first executable model of entire system, capturing all application-blocks computation and abstract communication between and within application-block in one common language. The specification model, typically captured in a System-Level Design language (SLDL), is the bridge between CALs and a current ESL flow. The system specification allows for system evaluation and feeds back into other levels as key discoveries are made.

**Decisions:** The designer captures the system functionality in an SLDL with clean separation between system computation and communication. While at previous CALs abstractions, we mainly focus on individual application-blocks – including realizing functionality, specifying algorithm, capturing an executable model, identifying PE-class and tuning for potential bottlenecks – at system specification we mainly focus on task- and data-level parallelism between and within application blocks.

Between application-blocks, the decisions mainly include determining the order of execution (serial, parallel, pipeline), identifying required channels and shared memories, selecting synchronization mechanisms, as well as exploring granularity of computation and communication (e.g. pixel vs. row vs. segment vs. frame). These decisions can considerably influence the final product. For example, channel sizes influence memory placement decisions (on-chip vs. off-chip) and consequently off-chip memory bandwidth demands. Within the application-blocks the decisions mainly involve realizing and exploring decomposition and hierarchy of individual application-blocks. This is with respect to the targeted PE-class.

**Characteristics:** The specification model captures all functional aspects regardless of their later implementation on a heterogeneous platform. Sufficiently fine-grained behavioral hierarchy exposes parallelism and structural hierarchy allows flexible mapping opportunities. In addition
application-level communication mechanism and suitable channel sizes between application-blocks also already realized. The resulting specification model can be used as an input model of ESL flow for automatic TLM exploration, or feed back to the internal engineering teams for developing handcrafted designs.

**Example:** Figure 5.12 outlines the specification model for the vision flow captured using a System Level Design Language (SLDL), in our case SpecC \[\text{\cite{40}}\]. The specification reflects all decisions that have been made traversing the CALs abstractions. The initial system specification is an almost one-to-one translations of application-blocks into functional behaviors. It includes the five functional behaviors: RGB2Gray, MoG, Blob Detection, Blob Tracking, and Trajectory Analysis.

In addition to capturing application blocks as SLDL behaviors, the specification captures more explicitly the communication and synchronization between the SLDL behaviors. In our case, communication channels (asynchronous) and shared memory regions are used depending on the type of interaction between the behaviors. For streaming pixels (e.g. Gray pixels, FG pixels), we choose the FIFO queue where one behavior is stream pixel producer and one is stream pixel consumer. In result, behaviors can run in parallel and operating on their own streaming pixels. We also employ shared memory regions in cases that multiple behaviors need access to the same data avoiding unnecessary data copies. One example is Blob Lists memory sharing the lists of updated blobs between Blob Detection (read) and Blob Tracking (read/write). Synchronization to shared memory in this example occurs through the New Blobs channel.

Figure 5.12 highlights the overall decomposition. RGB2Gray receives the RGB pixels from stimulus and sends gray values to MoG. After identifying FG pixels, MoG forwards FG pixels though its channel to Blob Detection. In addition, MoG stores the foreground image into the FG Frame memory for further processing in Block tracking. As Block tracking employs mean-shift

![Figure 5.12: Specification model of vision flow captured in SpecC SLDL.](image-url)
algorithm, it requires an FG-mask of entire frame. In result, we use double buffering techniques for synchronization between MoG and Blob Tracking. While MoG is calculating FG pixels of frame number N+1 and writing FG pixels to FG Frame memory, Blob Tracking reads the FG pixels of frame number N and calculating the new blobs positions. Blob Detection identifies new blobs signaling them through New Blobs channel to Blob tracking.

Till now, we realized abstract communication / synchronizations mechanisms between application-blocks which individually explored and tuned throughout the CALs abstraction levels. One additional step is further decomposition of blocks with high computation demands, since coarse-grain mapping of computation-intensive application-blocks (SLDL behaviors) would be either infeasible or lead to an inefficient solutions. In result, these SLDL behaviors can be decomposed into multiple less intense blocks to increase mapping flexibility for finer-grained design space exploration. CALs’ output can direct the focus of system architect by identifying the Blocks’ computation / communication demands at higher abstractions. Earlier in Figure 5.6a and Figure 5.6b we showed an estimation of computation and communication demands of individual application-blocks. One approach is to identify and isolate kernel functions within each application-block. An example decomposition is shown in Figure 5.13 focusing on compute-intense behaviors: MoG, Blob Detection and Blob Tracking. MoG is split into three main behaviors: Gaussian Components, BG Model Update and FG Detection executing in a pipeline on individual streaming pixels. Similarly,
Blob Detection, is decomposed into Component Labeling (Contour algorithm) and Find Blob. Blob Tracking has been also split into three main kernel functions: Mean-Shift, Kernel Histogram and Bhatt. Coefficient, all communicating through double handshake channels.

The resulting system specification gives further insight into application-block properties, composition of behavior, system-level traffic, system pipeline, and enables for example channel capacity sizing. Figure 5.14 presents the performance requirements of the five application-blocks in dark blue color and their sub-blocks in light blue. The results are based on the captured design in SpecC SLDL presented in Figure 5.13 operating on Full-HD resolution at 30 frames per second for the complex scene. Five Gaussian components are employed (16 bit mean, 16 bit standard deviation, and 14 bits weight). MoG overall requires around 14 GOPs/s where G-Comp. is its most compute intense sub block with 8.2 GOPs/s. Compression of Gaussian parameters increases computation demand minimally by 0.7%. Nonetheless, the benefit of reduced MoG memory bandwidth much outweighs slight increase in computation (especially considering that MoG is targeted for pipelined...
CHAPTER 5. CONCEPTUAL ABSTRACTION LEVELS (CALS)

HW implementation).

Figure 5.15 illustrates the system’s memory traffic. Memory traffic includes both accesses to the system channels and shared memory between behaviors highlighted in Figure 5.13. For example in MoG, the overall traffic is around 3.5 GBs/s mostly generated by Parameter Compress/De-compress behavior executing both read & de-compression and write-back & compression of Gaussian parameters.

Automation: Tools that aid in the analysis and that support restructuring the specification model will significantly improve design efficiency at this level. Examples include the recoding \[21\] which aids translating a sequential specification to a flexible parallel specification. The recently developed tool in \[123\] also provides an opportunity to explore the granularity of parallelism and efficiency of execution at specification level.

5.2.5.1 Transaction-Level Model (TLM)

With the system-level specification in place, top-down design methodologies can be employed for design space exploration evaluating system-level architecture alternatives, module mapping and scheduling.

By utilizing CALs, we have identified the PE-class or IP-type of each application-block, parallelism, and potential system bottlenecks prior to arriving at the specification. This simplifies the design space exploration, and greatly aids the narrowing down of candidate architectures. Conversely, now more time can be spent for finer grained architecture exploration comparing PE-types within PE-classes. For the surveillance market example, the designer still needs to select processor type and mapping for Object Detection and Object Tracking evaluating multi- or single core mapping alternatives, as well as perform communication design and synchronization. Significant research work has been invested into TLM abstraction levels, TLM-tradeoff and exploration methodologies. Our work focuses on abstractions above the system specification, with the goal to interface with TLM-based methodologies and tool-chains for further architecture exploration.

Figure 5.16 shows one possible TLM architecture for the embedded for our case study vision flow. MoG and RGB2Gray are mapped to Custom-HW IPs. Both Blob Detection and Blob Tracking are mapped to Blackfin DSP cores. Additionally an ARM core can be integrated for other possible top-level control applications – considered as application processor.

At TLM, channel and shared memory are mapped to architectural elements. The RGB Pixels channel represents the camera input, streaming the pixels to RGB2Gray IP. A dedicated
memory is required to keep an entire FG Pixels. One option is to dedicate 2 MBs on-chip scratch pad for FG Pixels channel, which would eliminate off-chip traffic. Due to size constraints, MoG parameters (minimal 50MB) need to be stored in external memory. In contrast, blob list can be easily mapped to an on-chip scratchpad memory. Many other alternative system architectures are feasible for the developed specification model. Their exploration and analysis is out of scope of this article. Instead, we show here a small insight into our example, the MoG.

The performance results obtained at TLM, may prompt for further decomposition and even coarse-grain micro-architecture exploration, e.g. for hand-over to manual RTL design or as an improved input to high-level synthesis. In our example, the MoG demands this attention, e.g. to parallelize the high computation demand of G-Comp. Throughout the CALs flow, MoG has been identified for HW implementation. Therefore, finer grained spatial and temporal parallelism are desired. The result of TLM exploration are illustrated in Figure 5.17 with a 6-stage pipelined implementation operating on a pixel stream. The Gaussian-components, including Gaussian Calculation and Parameter Update, run in parallel, followed by Background Model Update and Foreground Detection.  

![Figure 5.17: Fine-grain 6-level pipeline MoG.](image-url)
CHAPTER 5. CONCEPTUAL ABSTRACTION LEVELS (CALS)

Detection. This finer grained model provides additional detail for optimization before actual RTL design. Simulation results show a pipeline delay of 30 ns implemented on a standard HW. This performance indication is helpful for system-level exploration and analysis. TLMs, e.g. in result of our CALs, lend themselves for fine-grained explorations of the remaining application-blocks (e.g. Blob Detection and Blob Tracking), which however exceeds the scope for this article.

5.3 CALs Automation Demands

The proposed conceptual abstractions reflect our work on reducing design complexity above the system specification level. While previously many concurrent design considerations have been lumped into the specification-level, our work exposes essential decisions in the form of higher abstraction levels that reside above the system specification. The goal is to articulate the demand for higher levels of abstraction and to provide guidelines for early exploration. For example, by selecting PE-class (HW, SW, GPU ...) even before the system specification model, the algorithm can be tuned and captured specifically for that PE-class leads to a more expressive specification model allowing a more efficient design space exploration. An earlier and richer design space exploration has the potential to yield an improved system specification model and associated solutions, and will reduce overall development time, and ultimately require fewer resources.

To summarize, Figure 5.18 shows the design characteristics that are expressed at each abstraction level - including both our proposed CALs flow and ESL design flow. System specification is considered as the joint abstraction level between CALs and ESL - Specification is considered as CALs output driving ESL design flow. Figure 5.18 illustrates what previously was bundled in the system specification model is now expanded and realized through CALs abstractions. Design decisions such as technical requirements, functionality decomposition, and algorithm selection and tuning are realized in individual abstractions above system specification model.

Our main focus is to systematically and explicitly expose design decisions, as well as to identify an incremental ordering of the previously implicit design decisions on the path to a system specification. We see much potential in the earlier phase (CALS-analytical: before the algorithm is tied down in a language). By defining abstraction levels (semantics of individual levels), and by defining the transformations / decision across abstraction steps, our CALs is an initial step toward a formalization and automation. Identifying the abstraction levels is the base for constructing a methodology. This allows to then identify suitable existing tools for each transition, or highlight the need for novel automation tools.
### 5.4 Chapter Summary

In this chapter, we proposed a set of higher abstraction levels – Conceptual Abstraction Levels (CALs) – to bridge the abstraction gap between the market requirements and the system specification model. Our CALs offer a systematic and simplified method for developing a system.
specification exposing essential design decisions when traversing along the abstraction levels: Market Requirements, Conceptual Application-Block Flow, Algorithm Selection, Executable Application-Blocks and Application-Block Tuning. Following proposed CALs methodology will yield a more expressive specification allowing for a more efficient design space exploration. Further, CALs identify opportunities for automation and thus offer the potential to further speed-up the specification development process through tools. CALs are part of our larger effort to create a methodology adding structure to the process of traversing from market concepts to an efficient system-level technical solution definition. Following our CALs-based methodology allowed us to develop an efficient, more optimized system specification model, and enhanced the effectiveness and efficiency of the overall top-down ESL flow.
Chapter 6

Conclusions and Future Research Opportunities

With the significantly growth of processing capabilities, vision algorithms are now more often pushed toward embedded platforms. Embedded vision computing refers to deploying visual analysis and computer vision capabilities to embedded systems, covering a variety of markets/applications with a high diversity in algorithms and conflicting requirements (high performance, low power and cost). The conflicting requirements result in a gap between market demands for advanced vision processing and SoC realization capabilities. While algorithm developers focus on advanced vision analysis (mainly utilize adaptive vision algorithms), embedded vision SoCs either shy away from adaptive algorithms or perform very limited resolutions (mainly due to not managing the algorithm’s communication demands). A significant challenge is to architect an efficient solution that (1) satisfies immense communication demands of an adaptive vision algorithm, (2) offers an integration solution to construct a complete vision flow out of multiple vision algorithms, and (3) can efficiently support many vision flows on a single platform. To properly address the embedded vision challenges and open a path toward embedded realization of advanced vision processing, this dissertation:

- Identified and separated two types of traffic (streaming and algorithm-intrinsic traffic) to remove a tremendous hurdle for embedded realizations of adaptive vision algorithms. The dissertation also proposed a communication-centric architecture template to support traffic separation in underlying hardware. The traffic separation also simplified managing of system-level traffic when constructing a complete vision flow out of multiple vision algorithms. We demonstrate the efficiency of our solution by constructing a complete object detection/tracking...
vision flow (consisting of six vision algorithms) processing 1080p 30Hz on Zynq platform. Our Zynq-prototyped solution performs 40GOPs at 1.7Watts of on-chip power.

- **Introduced a novel processor class: Function Level Processor (FLP)** which is a new processing architecture combining execution efficiency (performance and power) with sufficient flexibility. An FLP can execute multiple vision flows with an efficiency comparable to custom-HWs. It consists of configurable Function Blocks (FBs), each implementing a kernel function interconnected throughout a flexible network to construct vision data-path. We demonstrated FLP benefits using an industry example of the Pipeline Video Processor (PVP) part of the ADI BF609. We highlighted the flexibility by mapping ten embedded vision applications to the PVP. The FLP-PVP has a much lower power consumption over comparable ILP and ILP+HWACC solutions (on average 14x-18x and 5x reduction respectively).

- **Proposed Conceptual Abstraction Levels (CALs)** which define new abstractions from market requirements to system specification model. CALs offer a systematic method to identify system bottlenecks at early stage of design and iteratively resolve them yielding a more expressive architecture specification model. CALs also identify opportunities for automation and thus offer the potential to further speed-up the specification development process and reduce time to market. CALs are part of our larger effort to create a methodology adding structure to the process of traversing from market concepts to an efficient system-level technical solution definition. Following our CALs-based methodology allowed us to develop an efficient, more optimized system specification model, and enhanced the effectiveness and efficiency of the overall top-down ESL flow.

This dissertation opens new areas of research and exploration to both academia and industry in particular in context of FLPs. Tremendous opportunities arise for architecting and programming at a function-level granularity – shifting from optimizing individual applications to identifying common functions that are present in many applications of the market. The challenge becomes to define a small enough set of sufficiently composable functions that provide meaningful computation for a given market. Points of interest include the allocation of FBs, i.e. selecting from a user specification, and enhancement from FB’s spatial multi-streaming to FB’s timed multi-streaming. We also see much potential in exploring and automating the refinements of abstractions above system specification models accordingly. This allows us to define a methodology and step-by-step select and integrate with
existing tools and realize novel new support tools. The realization of the methodology in a complete tool flow is part of our future work.
Chapter 7

List of Publications

Here, we provide the list of publications related to this dissertation. Among the publication, seven conference papers ([107, 106, 108, 110, 19, 105, 103]) and two journal articles ([109, 104]) have been already accepted.

CHAPTER 7. LIST OF PUBLICATIONS


- H. Tabkhi, M. Sabbagh, G. Schirner, Real-time Power-efficient implementation of Mixture of Gaussian for FPGA platforms, (draft under submission).
Bibliography

BIBLIOGRAPHY


BIBLIOGRAPHY


101
BIBLIOGRAPHY


BIBLIOGRAPHY


103
BIBLIOGRAPHY


BIBLIOGRAPHY


