Chapter 1
TRP: A Foundational Platform for High-Performance Low-Power Embedded Image Processing

Magnus Jahre and Philippe Millet

Abstract Embedded image processing systems face stringent and conflicting constraints which commonly result in developers overly specialising systems to the problem-at-hand. In other words, they give priority to efficiency, which is an immediate concern, over the longer term development cost reduction benefits of building reusable components. In this paper, we present the foundational TULIPP Reference Platform (TRP) which enables making domain-specific generality versus specificity trade-offs through the definition of TRP instances. Each TRP instance includes the key software and hardware components for a given domain as well as productivity-enhancing components if these can be accommodated within the typical constraints of the domain. While TRP instances primarily enable intra-domain reuse, they also enable inter-domain reuse as collections of components used in one instance may be straightforwardly reused in other instances. At present, TRP instances are defined for the space, medical, automotive, robotics, and Unmanned Aerial Vehicle (UAV) domains.

1.1 Introduction

Image processing embedded systems are ubiquitous and a critical component in future technologies such as self-driving cars and autonomous robots. Essentially, image processing enables these systems to see and thereby assess their surroundings. To fulfil this function, the systems commonly need to be fast so that the car or robot has sufficient time to react to events. Unfortunately, performance is only one requirement. Depending on the system, it may be constrained by energy (e.g., because

Magnus Jahre
Norwegian University of Science and Technology, Trondheim, Norway,
e-mail: magnus.jahre@ntnu.no

Philippe Millet
Thales, Palaiseau, France, e-mail: philippe.millet@thalesgroup.com

battery capacity is limited) or power consumption (e.g., because it is unacceptable to increase the temperature of other components), and nearly all embedded image processing systems are cost-sensitive. These constraints are commonly conflicting. One example is that power consumption is fundamentally related to the hardware clock frequency and hence performance [4].

This causes a challenging situation in which image processing applications need to be carefully tuned to satisfy all constraints. This is typically achieved by *specialising* the system to the problem-at-hand by selecting or developing a set of well-suited software and hardware components and removing all superfluous functionality. Although such systems are typically very efficient, they incur a fair amount of system-specific implementation work. This work is typically not reusable and leads to similar features being repeatedly implemented across engineering teams and companies — resulting in unnecessarily high development costs. The alternative approach is *generalisation* in which substantial resources are devoted to preserving a one-size-fits-all solution. This hurts efficiency and typically results in image processing systems that cannot satisfy all constraints. Thus, the key challenge is to appropriately balance specificity – to satisfy all constraints – with generality – to reduce development costs by enabling substantial reuse.

We took on this challenge in the recently completed EU-funded TULIPP project [8], and our proposed solution is the TULIPP Reference Platform (TRP). The TRP is a *foundational platform* for high-performance low-power embedded image processing systems. A foundational platform is a collection of components with clearly defined interfaces as well as a record of the components' compatibility. Thus, the TRP enables developers to trade-off generality against specificity to minimise development costs while satisfying the system's specific set of constraints.

To use the TRP, developers define TRP *instances* that contain only the components that are required within a particular domain. Two components are compatible if they are used together in a TRP instance. In this way, the TRP instances combine intra-domain reuse with specialisation by tailoring the platform to meet the typical constraints of the domain. We defined TRP instances for the medical, automotive, and Unmanned Aerial Vehicle (UAV) domains within the TULIPP project. Afterwards, Thales and Sundance have defined TRP instances for the space and robotics domains, respectively. The TRP is dynamic and new components (and their compatibility) are added as they become supported in TRP instances. The TRP also enables inter-domain reuse as new instances can build upon components that are already known to be compatible (i.e., they are used in combination in other instances).

Figure 1.1 illustrates that the current TRP instances are placed at very different design points regarding the specificity versus generality trade-off. For completeness, Figure 1.1 further compares the TULIPP instances to Application Specific Integrated Circuits (ASICs), Graphics Processing Units (GPUs), and general-purpose Central Processing Units (CPUs). The Space instance and the Medical instance have stringent constraints which leads to specialised instances with relatively few components. Conversely, the Robotics instance has much less stringent constraints as current robots are relatively large and slow which relaxes performance, power, and energy constraints. However, even the Robotics instance includes a Field-Programmable

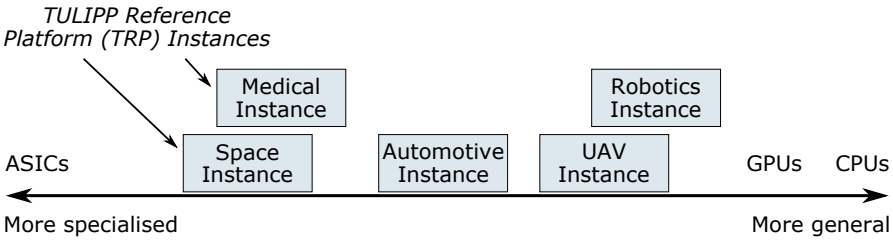


Fig. 1.1: The TULIPP Reference Platform (TRP) enables specificity versus generality trade-offs. Depending on severity of constraints, TRP instances tend towards specificity (for efficiency) or generality (to save development costs through reuse).

Gate Array (FPGA) to enable application-specific acceleration. Thus, we argue that it is more specialised than a GPU. A key advantage of the TRP is that the resource-constrained instances (e.g., Space and Medical) can leverage the rich component compatibility achieved within the reuse-oriented instances (e.g., Robotics).

Defining new TRP instances is an non-trivial task since it is not straight-forward to establish which components should be supported. The core issue is that supporting more components adds features or simplifies application development which is advantageous as long as the typical domain-specific constraints are satisfied. In TULIPP, we proposed the *guidelines* concept to aid designers in making this choice. A guideline encapsulates an expert insight in a precise, context-based formulation which orients the follower towards a goal by recommending an implementation method. In this way, the guidelines help designers select components based on prior TRP-relevant experience rather than a pure trial-and-error approach.

1.2 The Constraints of Embedded Image Processing

Embedded image processing systems attempt seek the sweet-spot which provides sufficient performance and low-enough-power (typically somewhere in the range between 1 W and 25 W). This is in contrast to server or desktop systems – where a power consumption of over 100 W can be acceptable – or the Internet of Things (IoT) – where power consumption is typically much less than 1 W. We expect the area of moderate power consumption systems will develop with the ever growing needs of for instance Advanced Driver Assistance Systems (ADAS), but a challenge is that not all technologies are accessible to limited-volume applications. One example is mobile System on Chips (SoCs) which are typically only sold in large quantities.

For smaller series of products, developers need to select platforms that meet processing, cost, power, and energy consumption requirements. Ideally, embedded image processing systems should have high performance, dissipate minimal power, and cost as little as possible – a classic case of conflicting objectives. Thus, im-

plementing an efficient image processing application requires carefully trading off different alternatives.

More specifically, the following constraints commonly need to be considered:

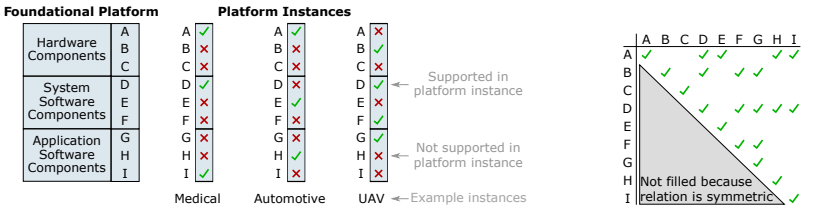
- **Power and energy requirements:** Embedded systems are often limited by battery life, but even when the product has access to the electrical grid it can be limited by the thermal dissipation within the heat sink, the cabinet, or the packaging. Therefore, energy consumption, power dissipation, and thermal issues commonly place restrictions on the implementation of image processing systems.
- **Performance:** Image processing applications tend to require more and more performance to deal with the large data sets provided by newer sensors, and many systems require real-time frame rates (typically at least 30 Hz). This creates a push for more powerful compute platforms as they make it easier for software developers to meet performance requirements.
- **Non-Recurrent Costs (NRCs):** NRCs are costs incurred during the development of the product. Higher NRCs might not be a problem when developing a high-volume product, but for low-volume products care must be taken to reduce NRCs since they may significantly increase the price of the product.
- **Recurrent Costs (RC):** RCs are costs incurred during the production phase of the product and strongly depends on the choices made at design time (e.g., more expensive components result in higher costs). Higher RCs will always impact the final product price, but unlike NRCs they require more attention in high-volume products as each cent saved can lead to millions gained in revenue.

A trade-off does not always mean that the system developer has to find a solution that matches all constraints. Commonly, the developer can analyse the constraints and possibly move the thresholds. If all functionalities cannot be implemented, the designer can see if it is possible to remove some of them or reduce the effectiveness or the accuracy of key functions within certain margins. Reducing the accuracy by 1% to 2% might in some case reduce the compute load by several tens of percentage points and allow for using smaller and cheaper components.

1.3 Foundational Platforms

We have now established that high-performance low-power embedded image processing platforms needs to combine specialisation – to meet stringent performance and power constraints – with generality – to save development costs by enabling reuse. In this paper, we advocate foundational platforms as a mechanism for balancing these conflicting requirements.

Figure 1.2 illustrates the *foundational platform* concept. A foundational platform enables reuse by listing key reusable components (see Figure 1.2a) and a compatibility matrix illustrating the component combinations that are currently supported (see Figure 1.2b). In this context, components can be hardware (e.g., a Xilinx Zynq MPSoC), system software (e.g., Linux), or application software (e.g., the OpenCV



(a) Foundational platform with instances.

(b) Compatibility matrix.

Fig. 1.2: The foundational platform concept. The foundational platform captures the key components and their compatibility to enable cross-domain reuse and serves as foundation for creating a number of domain-specific platform instances.

image processing library). Specialisation is achieved by selecting a minimal subset of compatible components for a particular application domain. Thereby, the foundational platform provides a generic substrate for which domain-specialised – and therefore efficient – *platform instances* can be created.

A specialised platform instance is more efficient than a generic platform primarily because it supports fewer components. In the hardware domain, including fewer components results in lower component costs. In addition, verification and integration costs are reduced. In the software domain, including fewer components simplifies the system; thereby lowering development time through less integration work and making testing and bug-fixing easier. For these reasons, it is rarely a good idea to create a physical implementation of the foundational platform. The exception is when you only have the resources to implement a single platform, but are required to support diverse applications, which was the case in the TULIPP project.

The main utility of the compatibility matrix is to aid designers when specifying new platform instances. The effort necessary to implement a new platform depends on the degree to which the platform instance can be created from components that are known to be compatible. Thus, the compatibility matrix encourages reuse by (i) incentivising developers to choose compatible components wherever possible, and (ii) motivating companies to invest effort into becoming more compatible to make their components more attractive to include in platform instances.

We illustrate the relationship between the foundational platform and the compatibility matrix with a simple example where Figure 1.2b shows the compatibility matrix of the foundational platform in Figure 1.2a. For simplicity, we assume that the platform instances completely define which components are compatible with each other (i.e., two components i and j are only compatible if i and j are part of a single platform instance). For example, component A is compatible with component D because they are both used in the platform instance for the medical domain. Similarly, component A is not compatible with component B because they are not both used in any platform instance. More specifically, A is used in the medical and automotive instances while B is used in the Unmanned Aerial Vehicle (UAV) instance. All boxes on the diagonal are ticked because a component must be compatible with itself. Further, only the upper triangle is shown because the compatibility matrix must be

symmetric (i.e., if component i is compatible with component j, component j is also compatible with component i).

A key objective of the TULIPP project was to create a path towards enabling more standardisation within high-performance low-power embedded image processing, and we believe the foundational platform concept can serve as an enabler of standardisation along multiple fronts. The most obvious opportunity is perhaps to standardise platform instances. In this way, standard compute platforms can be defined for key domains (e.g., automotive). Another option is to standardise key interfaces and thereby simplify the process of making components compatible. Finally, aspects of the foundational platform itself can be standardised. In this case, likely options are (i) standard procedures for approving compatibility between components, and (ii) procedures for defining platform instances that match the requirements of key players in the target domain.

1.4 The TULIPP Reference Platform and Its Instances

In this section, we describe the TULIPP Reference Platform (TRP) and its instances. The TRP is a foundational platform for high-performance low-power embedded image processing that was developed in the TULIPP project. During the project, TRP instances were defined for key applications in the UAV, automotive, and medical domains. After the project, Thales has defined a TRP for use in space applications while Sundance has proposed a TRP instance for robotics applications. Thus, there are currently five domain-specific TRP instances.

1.4.1 The TULIPP Reference Platform (TRP)

Table 1.1 lists the components of the TRP and the instances in which each component is supported. The criteria for including a component in the TRP is that it is required in at least one instance. A key take-away is that no instance implements all TRP components. In particular, the instances only support hardware components that are critical for the targeted domain as adding more hardware components increases costs and may increase power consumption. This effect is not as visible for the software components where the difference between the instances is mainly if they target running bare-metal applications or the application on top of an OS (i.e., Linux). Interestingly, only the least resource constrained instances (i.e., the UAV and Robotics instances) support OpenCV. For software components, overheads manifest themselves as larger memory and storage requirements as well as increased implementation and validation effort.

Table 1.2 shows the compatibility matrix of the TRP components. Basically, components either have rich or limited compatibility with other components. Rich connectivity occurs in two main cases. First, the component can be the de facto

Table 1.1: The components of the TULIPP Reference Platform (TRP).

Category	ID	Component	Medical	Space	Automotive	UAV	Robotics
Hardware Components	A	GigE Vision	X				
	B	CameraLink				X	X
	C	HDMI	X		X	X	X
	D	USB		X	X		X
	E	GigE					X
	F	JTAG			X	X	X
	G	MAVLink				X	
	H	RS422		X			
	I	DDR memory		X	X		X
	J	SD Card	X				X
System Software Components	K	Linux			X	X	X
	L	Fat32	X				X
	M	TCP/IP					X
	N	U-Boot			X	X	X
Application Software Components	O	C/C++	X	X	X	X	X
	P	OpenCV				X	X
	Q	GCC	X	X	X	X	X
	R	Xilinx Vivado HLS	X	X	X	X	X
	S	ROS					X

Table 1.2: The TRP compatiblity matrix.

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S
A	X		X							X		X			X		X	X	
B		X	X			X	X				X			X	X	X	X	X	
C			X	X		X	X		X	X	X	X		X	X	X	X	X	
D				X	X	X		X	X	X	X	X	X	X	X	X	X	X	X
E					X	X			X	X	X	X	X	X	X	X	X	X	X
F						X	X		X	X	X	X	X	X	X	X	X	X	X
G							X				X			X	X	X	X	X	
H								X	X						X		X	X	
I									X	X	X	X	X	X	X	X	X	X	X
J										X	X	X	X	X	X	X	X	X	X
K											X	X	X	X	X	X	X	X	X
L												X	X	X	X	X	X	X	X
M													X	X	X	X	X	X	X
N														X	X	X	X	X	X
O															X	X	X	X	X
P																X	X	X	X
Q																	X	X	X
R																		X	X
S																			X

standard for the domain (e.g., component *O* which is C/C++) or necessary to access a critical feature of the platform (e.g., component *R* which is the Xilinx Vivado HLS tool [22]). Second, the component can be infrequently used but included in a TRP instance that supports a variety of components. A good example is GigE (component *E*) which is only supported in the Robotics instance but still has rich compatibility. A counterexample is RS422 (component *H*) which is only needed in the Space instance and therefore has limited compatibility. The reason is that the Space instance is severely resource constrained and therefore only supports a limited number of TRP components.

The rich compatibility of Xilinx Vivado HLS is a consequence of the hardware platform developed in the TULIPP project. More specifically, we built the hardware platform around a Zynq UltraScale+ MPSoC which integrates a multi-core processor and an FPGA fabric on a single chip. Although adopting a multi-core platform is not without challenges (see e.g., [5, 6, 7]), these are outweighed by its performance and energy-efficiency advantages. Further, we only had the resources to develop a single hardware platform within the project which forced us to chose a platform that was acceptable for all target domains. Unfortunately, this also means that it was not necessarily optimal for any of them.

The TULIPP hardware platform contains an FPGA for application-specific acceleration since FPGAs have been shown to provide high performance at low power consumption for image classification tasks [20] and other compute-intensive kernels (see e.g., [9, 21]). However, it is also well known that developing highly efficient FPGA-solutions is challenging [1]. In response to this situation, there has been a significant research effort towards developing High-Level Synthesis (HLS) tools. HLS tools are able to transform an implementation in a high-level language (i.e., C or C++) into an accelerator circuit. Thus, they offer significantly improved productivity compared to traditional RTL design approaches which typically require developers to describe a plethora of low-level implementation details. Although we used Xilinx Vivado HLS [22], a number of other HLS tools are available (e.g., [3, 18]).

1.4.2 The TULIPP Reference Platform (TRP) Instances

We now delve into the details of the component selection for the TRP instances in Table 1.1. The objective is to provide insight on how the specific constrains of a particular domain influences component selection. This adds further depth to the more general discussion of constraints which we provided in Section 1.2.

For each TRP instance, we focus on the particular application(s) that we have implemented within the target domain. We foresee that more components may be added to the TRP instance as more applications are implemented. That said, we will only add components that are commonly used in the domain to retain efficiency. Eventually, clusters of applications which prefer different component sets may emerge within domains. In this case, it could be beneficial to define different TRP instances that specifically cater to the needs of each component cluster.

All currently defined TRP instances rely on FPGAs for acceleration due to the low-power focus of the TULIPP project and that FPGAs are typically more energy-efficient than CPUs and GPUs for more complicated image processing pipelines [15]. To offset the programmability challenges of FPGA-based acceleration, all TRP instances rely on Vivado HLS. This may change if future TRP instances for other domains put a greater emphasis on ease of development than efficiency. That said, higher-level programming constructs can cause performance issues if it generates access patterns that map unfavourably to the memory system [12].

1.4.2.1 The Medical Instance

The Medical instance was developed in the context of a mobile C-arm which is an X-ray system used during surgery. This enables the surgeon to use real-time X-ray images as a guide while operating and thereby making the incisions as small as possible. Chapter **??** provides more information about the medical TRP instance and its application.

Obviously, achieving real time operation is critical for this application as delay (or delay variation) may result in harm to the patient. This challenge is exacerbated by regulatory requirements which dictate that all information captured though radiation is presented to the surgeon. This significantly limits the degree to which compression can be used. Further, power consumption is a critical requirement as the TRP instance is placed close to the X-ray sensor. If the sensor is heated too much, image quality deteriorates.

These requirements result in the medical platform being very light in terms of added components. The data acquisition subsystem provides the input images using GigE Vision and the same interface is used to pass the enhanced images to the display subsystem. The HDMI and SD-card components were added to simplify testing and development.

1.4.2.2 The Space Instance

The Space instance is developed in the context of an image acquisition satellite, and the utility of the application is to filter out uninteresting images and thereby better utilise the bandwidth-limited link to Earth (see Chapter **??** for more details). For instance, transmitting pictures of clouds is inefficient if the objective of the satellite is to look for objects on the ground. For this platform, the peak power consumption of the complete system cannot exceed 30 W. Further, the volume and weight of the platform is restricted as the platform must fit inside the satellite and launch weight is a significant cost driver.

The Space instance uses the DDR memory to isolate system components. The sensor system retrieves the acquired image from a high-resolution camera over the RS422 interface and writes it to memory. Then, the FPGA-based accelerator reads the image from memory, analyses if the image should be transmitted, and writes the

result to memory. If the analysis concludes that the image should be transmitted, the transmission system transfers the image to Earth over an optical link (which comes with a USB interface). The system does not contain any additional components due to strict power, weight, and volume constraints.

1.4.2.3 The Automotive Instance

The Automotive instance focused on pedestrian detection using the Viola-Jones algorithm. Detection latency is a critical constraint as it determines how quickly the car's driver assistance system can react. Further, power consumption is constrained since the image processing system is commonly placed alongside the cameras behind the car's rear-view mirror. The confined space and high degree of sun exposure makes it very difficult to keep the system sufficiently cool.

The pedestrian detection system is a single system within the driver assistance pipeline. Thus, we assumed that input images are available in memory and that bounding boxes of the detected pedestrians is also written to memory. HDMI, USB, and JTAG are supported to simplify development. These are commonly disabled when the system is not in test-mode to reduce power consumption.

1.4.2.4 The UAV Instance

The key application of the UAV instance computes a depth map using a stereo camera setup (see Chapter **??**). Low latency is a critical requirement since the depth map is used to avoid colliding with objects. Thus, the latency of the depth map computation limits the speed at which the drone can fly. Larger drones are not severely limited by energy or power consumption since the energy consumption of the computing systems is low compared to the engines and the abundant airflow can be used for cooling. For smaller drones, the smaller volume available may change this picture. That said, the limited performance of on-board compute platforms can significantly increase the overall energy consumed to complete the mission due to selecting suboptimal trajectories [10].

The key components of the UAV instance are the CameraLink interface to the camera setup and MAVLink for integrating with the flight control system. Since the instance controls the complete drone, the system complexity is larger than for instance the Medical and Automotive instances that purely process an incoming image stream. Thus, the UAV instance supports the Linux operating system which then also brings with it a number of other components. It also supports the OpenCV image processing library as the productivity benefits of including it outweighs its overheads (e.g., increased memory requirements).

1.4.2.5 The Robotics Instance

The Robotics instance has been used in multiple robots including the VineScout robot for monitoring of vineyards (see Chapter **??** for more details). The main constraint for a robot is that it is able to fulfil its intended purpose. This typically requires advanced software subsystems that for instance perform mapping, motion planning, and control physical movement, and it is impractical to re-implement these systems for every robot. Thus, enabling software reuse is critical. The de facto standard for robotics computing is the Robot Operating System (ROS) [14] which is a set of software libraries and tools that help build robot applications. Performance, power, and energy requirements are robot specific, but in general they are less stringent than for many other domains. Most contemporary robots have the option to move slower to match the performance of the computing system or add more batteries (cooling) to overcome energy (power) constraints.

ROS requires Linux which leads to the robotics instance supporting a wide variety of software components. Further, the ability to work around performance, power, and energy constraints means that there are limited downsides to supporting a rich set of hardware components. Thus, the Robotics instance supports the most components of all the current TRP instances. As the robotics domain matures, we foresee that robots will become smaller and faster. This will likely create a need for a new TRP instance that scale down the number of supported components to improve efficiency.

1.5 The Guidelines Concept

Assembling domain-specific TRP instances can be a daunting task since over-provisioning results in suboptimal efficiency while not supporting the required interfaces makes the instance difficult or impossible to use in the target domain. Fortunately, there are similarities between image processing domains that can be leveraged. In TULIPP, we propose guidelines as a mechanism to codify domain-knowledge and make it accessible to stakeholders with different expertise – thereby enabling creators of new TRP instances to build upon lessons learned other domains.

1.5.1 Guideline Definition

A guideline is an encapsulation of an advice, the insights the advice is based upon, and a recommended implementation method. The advice captures an expert's insights in a precise, context-based formulation and orients the follower (the person reading the advice) towards a goal. The recommended implementation method indicates a practical approach for following the advice in the context of a TRP instance. Both the advice and the recommended implementation methods are supported by theoretical

Table 1.3: Guideline information table for TULIPP Guideline #26 [17].

Item	Value
Guideline Number	26
Guideline Responsible (Name, Affiliation)	Boitumelo Ruf, Fraunhofer
Guideline Reviewer (Name, Affiliation)	Magnus Jahre, NTNU
Guideline Audience (Category)	Application developers
Guideline Expertise (Category)	Hardware designers, System architects
Guideline Keywords (Category)	Code optimization, GPU, FPGA

or experimental evidence that is either produced specifically for the evaluation of the guideline or is pre-existing in the community.

Designing embedded image processing systems requires expertise within a number of different fields. For this reason, we specify (i) the field of expertise from which the guideline was generated, and (ii) the field of expertise of the persons that are expected to find the guideline most useful (i.e., its intended audience). This is specified for all guidelines (see Table 1.3 for an example).

More specifically, we select the expertise and the audience from the following groups:

- **Hardware designers:** This group deals with the design of the hardware platform, component selection, and component interfacing according to system requirements.
- **Operating System (OS) designers:** This groups deals with OS design and development. Examples are defining Application Programming Interfaces (APIs) for applications, ensuring that the OS works with a particular hardware configuration, and providing the application with the means to efficiently control hardware behaviour.
- **Tool-chain designers:** This group deals with supporting application developers by providing tools that automate recurring tasks. They must supply a comprehensive tool-chain that helps application developers efficiently map their applications onto the hardware platform.
- **Application developers:** This group consists of experts in image processing. They know the algorithms used in an application and have the ability to implement this algorithm on a suitable hardware platform. They understand the complete software stack, and can leverage tool support to faster develop the application.
- **System architects:** This group deals with the complete system definition. They are involved in a broad set of issues from the identification of the constraints that come from the final product, making sure that the system adheres to its price constraints, and are able to understand integration issues that arise due to choices made by the four other groups.

Table 1.4 exemplifies the guideline concept with Guideline #26 [17] from the TULIPP guideline repository [19]. Here, the advice orients the follower towards the goal of efficiently implementing kernels that contain a significant number of branches on a Graphics Processing Unit (GPU). The recommended implementation method

Table 1.4: Guideline example: TULIPP Guideline #26 [17].

Guideline advice:	Conditional branching such as if-then-else is vital to most image processing applications, e.g. in finding maximum similarity between pixels or handling image boarders when filters exceed the dimensions of the image. In terms of processing speed and performance overhead the use of conditional branching on CPUs and FPGAs is cheap. However, if the HLS tool cannot group branches, i.e. if branches are likely to diverge, the use of conditional branching can result in a resource overhead when optimizing code for FPGAs. When leveraging the processing power of GPUs by GPGPU (general computation on a GPU), conditional branching is to be used with caution. If branches diverge within a warp, i.e. if some evaluate to true and others to false, the instructions are executed twice, resulting in a processing overhead [13, 16].
Insights that led to the guideline:	CPUs are designed for general purpose processing and are equipped with optimization strategies such as branch prediction, which allow a fast response to conditional input. In order to achieve parallel processing on CPUs, the programmer instantiates different threads and processes which can run concurrently on the different processing cores. The scheduler of the CPU is free to pause the processing of certain threads in order to react to important interrupts and inputs. Hence, it is not guaranteed that all threads will run synchronously. Furthermore, due to its flexibility, the CPU, unlike GPUs, is able to only process the branch for which the conditional directive resolved to true. FPGAs can also cope well with conditional branching in terms of processing speed, as HLS will create different paths for each conditional branch. However if the branches cannot be grouped efficiently the use of many conditional branches leads to a resource overhead on FPGAs In order to achieve great parallelism and high data throughput, GPUs run numerous (>100) kernels on a large number of processing units. The key aspect of this processing is that each instantiation of the kernel is performing the same processing but on different subsets of data. The GPGPU programming model calls this paradigm Single Instruction Multiple Threads (SIMT) which is similar to Single Instruction Multiple Data (SIMD). SIMT processing requires that all threads within in one warp (a group of threads running on one processor, sharing resources) run synchronously. This means that when kernels have conditional branching, all branches are evaluated and processed in order to keep the threads from diverging. At the end, the result of the particular branch is chosen for which the conditional expression resulted in true. Hence, conditional branching with large bodies to save processing time is to be avoided, as all branches will be processed anyway. Furthermore, a divergence of the branches, which occurs when the conditional branch evaluates to true for some threads of the warp and for others to false, will result in processing overhead as the instructions are executed twice. See [16, 13] for more information.
Recommended implementation method:	Avoid conditional branching with possibly divergent branches. Use multiple loops to perform different operations in different areas of the image. When accelerating code with GPGPUs instantiate different kernels instead of using if-then-else statements for image areas which need specific processing.
Instantiation of the recommended implementation method in the reference platform:	This method is actually true for almost all accelerators and particularly with GPGPUs and FPGAs. Accelerators are often based on long pipeline chains and can manage big chunks of data with less hardware involved than standard CPUs. This must particularity be taken into account during the development of the algorithm as branches will cut the execution pipeline and will also have effects on the data to be served to the application and therefore their distribution in the system.
Evaluation of the guideline in reference applications:	There was no evaluation done as part of the TULIPP project as the guideline is common practice when employing GPGPU. However, the authors of [2] did a thorough evaluation on the effect of divergent threads. However, the TULIPP use case followed this method for the development of their application on GPGPU and FPGA.

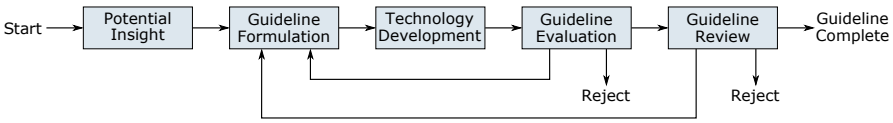


Fig. 1.3: The TULIPP guideline generation methodology.

advocates grouping branches such that all branches within a warp branch in the same direction. The guideline further discusses how the guideline is instantiated and evaluated within the TRP.

There is no one-to-one mapping between a guideline and an insight. Multiple insights can serve as the basis for a single guideline or a single insight can result in multiple guidelines. For example, the same insight can have different implications on different audiences (e.g., hardware designers and application developers). In this case, it can be appropriate to capture each perspective in its own guideline.

1.5.2 Guideline Generation Methodology

Generating guidelines is not straightforward. The main difficulty is to define insightful guidelines that will impact a wide number of developers. While the guideline-creation process typically starts from a particular practice or specific issue, a more general and global view of the problem as well as a higher level of information content is required for a guideline to be broadly applicable.

To address this challenge, we derived the guideline generation methodology shown in Figure 1.3. The process starts when a developer understands something regarding the implementation of embedded image processing systems that he or she believes can be of somewhat general interest. The developer then adds a new page to the guideline repository (see [19]), and fills in an initial draft of the "Advice", "Insights that led to the guideline", and "Recommended implementation method" sections (see Table 1.4). This draft reflects the developer's initial understanding and insight, but may contain significant inaccuracies or flaws. Thus, further analysis and refinement is typically required.

The purpose of the next steps of the guideline generation methodology is to transform the initial formulation into a meaningful guideline. Commonly, some form of technology development must be carried out in order to appropriately evaluate the guideline insight and advice. With this in place, the developer qualitatively evaluates the guideline on a relevant TRP instance. The evaluation has three outcomes. The first possible outcome is that the evaluation matches perfectly with the developer's expectations and the guideline can pass to the review stage without reformulation. The second case is that the evaluation results in deeper insight – enabling the developer to rectify the flaws of the initial guideline formulation. This commonly leads to further technology development, and a new evaluation. The third option is that the developer understands that the insight of the guideline is fundamentally flawed. In this case,

the developer rejects the guideline and removes it from the repository. Both the first and the third cases are rare. For example, none of the guidelines generated during the TULIPP project were rejected.

From a leadership perspective, it is challenging to motivate developers to create guidelines. One obvious reason can be that creating guidelines is extra work that easily gets low priority. Since TULIPP was a research project, we were able to correct for this by explicitly pressuring developers prioritise generating guidelines. For us, the key problem was that the developers felt that their guideline ideas were not sufficiently insightful to serve as meaningful guidelines. The problem is that when a developer has (finally) solved a problem, the solution is obvious to the developer – which quickly gets generalised into obvious for anybody. This is an aspect of the Dunning-Kruger effect [11]: Competent people tend to assume that tasks that are easy for them are also easy for everybody else. In the end, we spent considerable time convincing developers that their insights were worth writing up as guidelines. When they first got started with proposing guidelines, they produced guidelines at a somewhat regular rate.

1.5.3 Guideline Quality Assurance

The final step of the guideline generation methodology is to review the guideline (see Figure 1.3). The review is necessary to ensure that the guideline is soundly formulated – both from the audience and the expert perspectives. To ensure this, we assign a reviewer that (i) has previously not been involved in the formulation of the guideline, and (ii) that has sufficient expertise to assess the quality of the guideline from both the audience and the expert perspectives. If we cannot find a single person that fits these requirements, we assign additional reviewers.

The outcome of the evaluation is an evaluation report which is added to the guideline repository [19]. Again, there are three possibilities. The most common outcome is that the reviewer identifies aspects of the guideline that needs to be reformulated. This may in turn lead to further technology development and more in-depth evaluation. When the guideline has been refined, it is reviewed again. This commonly leads to the second possible outcome: The guideline is accepted. Although a guideline can be accepted after the first review, this is not the most likely case. The reason is that developers tend to struggle with creating sufficient distance to their own work to formulate the guideline such that it is generally applicable. The final option is that the reviewer discover a fundamental and unrectifiable flaw in the guideline which leads to its rejection. This did not happen in the TULIPP project.

1.6 Conclusion

We have now presented the foundational TULIPP Reference Platform (TRP) and its instances. The TRP enables appropriately balancing the specificity and generality of embedded image processing systems while staying within the typical constraints imposed on a particular domain. Currently, TRP instances have been defined for the space, medical, automotive, UAV, and robotics domains. To aid designers when defining new TRP instances, we proposed the guidelines concept. A guideline is a specific, context-sensitive formulation of a TRP-relevant insight. Collectively, the guidelines enable a designer to build on experience from previously defined TRP instances and thereby define a TRP instance for the new domain with minimal trial-and-error.

Acknowledgements We would like to thank Ananya Muddukrishna for his contributions to the initial formulation of the guidelines concept. This work has been funded in part by the European Horizon 2020 project TULIPP (grant agreement #688403).

References

1. Bacon, D.F., Rabbah, R., Shukla, S.: FPGA programming for the masses. Commun. ACM **56**(4), 56–63 (2013)
2. Bialas, P., Strzelecki, A.: Benchmarking the cost of thread divergence in CUDA. arXiv:1504.01650 [cs] (2015). URL `http://arxiv.org/abs/1504.01650`. ArXiv: 1504.01650
3. Canis, A., Choi, J., Aldham, M., Zhang, V., Kammoona, A., Czajkowski, T., Brown, S.D., Anderson, J.H.: LegUp: An Open-source High-level Synthesis Tool for FPGA-based Processor/Accelerator Systems. ACM Trans. Embed. Comput. Syst. **13**(2), 24:1–24:27 (2013)
4. Horowitz, M., Indermaur, T., Gonzalez, R.: Low-power digital design. Symposium on Low Power Electronics pp. 8–11 (1994)
5. Jahre, M., Eeckhout, L.: GDP: Using dataflow properties to accurately estimate interference-free performance at runtime. In: International Symposium on High Performance Computer Architecture (HPCA), pp. 296–309 (2018)
6. Jahre, M., Grannaes, M., Natvig, L.: A quantitative study of memory system interference in chip multiprocessor architectures. In: Proceedings of the International Conference on High Performance Computing and Communications (HPCC) (2009)
7. Jahre, M., Natvig, L.: A high performance adaptive miss handling architecture for chip multiprocessors. Transactions on High Performance Embedded Architecture and Compilation **4**(1) (2009)
8. Kalb, T., Kalms, L., Göhringer, D., Pons, C., Marty, F., Muddukrishna, A., Jahre, M., Kjeldsberg, P.G., Ruf, B., Schuchert, T., Tchouchenkov, I., Ehrenstrahle, C., Christensen, F., Paolillo, A., Lemer, C., Bernard, G., Duhem, F., Millet, P.: TULIPP: Towards ubiquitous low-power image processing platforms. In: Proceedings of the International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation (SAMOS), pp. 306–311 (2016)
9. Koraei, M., Fatemi, O., Jahre, M.: DCMI: A scalable strategy for accelerating iterative stencil loops on FPGAs. ACM Transactions on Architecture and Code Optimization **16**(4), 36:1–36:24 (2019)

10. Krishnan, S., Borojerdian, B., Fu, W., Faust, A., Reddi, V.J.: Air Learning: An AI research platform for algorithm-hardware benchmarking of autonomous aerial robots. arXiv:1906.00421 [cs] (2019). URL http://arxi.org/abs/1906.00421
11. Kruger, J., Dunning, D.: Unskilled and unaware of it: How difficulties in recognizing one's own incompetence lead to inflated self-assessments. Journal of personality and social psychology **77**(6), 1121 (1999)
12. Liu, Y., Zhao, X., Jahre, M., Wang, Z., Wang, X., Luo, Y., Eeckhout, L.: Get out of the valley: Power-efficient address mapping for GPUs. In: Proceedings of the International Symposium on Computer Architecture (ISCA) (2018)
13. NVIDIA: CUDA C++ Programming Guide. Tech. rep. (2019)
14. Open Robotics: Robot Operating System (ROS). https://www.ros.org/ (2020)
15. Qasaimeh, M., Denolf, K., Lo, J., Vissers, K., Zambreno, J., Jones, P.H.: Comparing energy efficiency of CPU, GPU and FPGA implementations for vision kernels. In: Proceedings of the International Conference on Embedded Software and Systems (ICESS) (2019)
16. Reissmann, N., Falch, T.L., Bjørnseth, B.A., Bahmann, H., Meyer, J.C., Jahre, M.: Efficient control flow restructuring for GPUs. In: International Conference on High Performance Computing Simulation (HPCS), pp. 48–57 (2016)
17. Ruf, B.: Guideline 26: Use conditional branching carefully. https://github.com/tulipp-eu/tulipp-guidelines/wiki/Use-conditional-branching-carefully (2019)
18. Sharifian, A., Hojabr, R., Rahimi, N., Liu, S., Guha, A., Nowatzki, T., Shriraman, A.: uIR - An intermediate representation for transforming and optimizing the microarchitecture of application accelerators. In: Proceedings of the International Symposium on Microarchitecture (MICRO) (2019)
19. Tulipp: Tulipp guidelines. https://github.com/tulipp-eu/tulipp-guidelines/wiki (2019)
20. Umuroglu, Y., Fraser, N.J., Gambardella, G., Blott, M., Leong, P., Jahre, M., Vissers, K.: FINN: A framework for fast, scalable binarized neural network inference. In: Proceedings of the International Symposium on Field-Programmable Gate Arrays (FPGA), pp. 65–74 (2017)
21. Umuroglu, Y., Jahre, M.: An energy efficient column-major backend for FPGA SpMV accelerators. In: Proceedings of the International Conference on Computer Design (ICCD), pp. 432–439 (2014)
22. Xilinx: Vivado High-Level Synthesis (2018). URL https://www.xilinx.com/products/design-tools/vivado/integration/esl-design.html