Optimal Performance Prediction of ADAS Algorithms on Embedded Parallel Architectures

Romain Saussard, Boubker Bouzid, Marius Vasiliu, Roger Reynaud

Optimal Performance Prediction of ADAS Algorithms on Embedded Parallel Architectures

Romain Saussard, Boubker Bouzid
Renault S.A.S.
Guyancourt, France
{romain.saussard, boubker.bouzid}@renault.com

Marius Vasiliu, Roger Reynaud
Instutut d'Électronique Fondamentale
Université Paris Sud
Orsay, France
{marius.vasiliu, roger.reynaud}@u-psud.fr

***Abstract*—ADAS (Advanced Driver Assistance Systems) algorithms increasingly use heavy image processing operations. To embed this type of algorithms, semiconductor companies offer many heterogeneous architectures. These SoCs (System on Chip) are composed of different processing units, with different capabilities, and often with massively parallel computing unit. Due to the complexity of these SoCs, predicting if a given algorithm can be executed in real time on a given architecture is not trivial. In fact it is not a simple task for automotive industry actors to choose the most suited heterogeneous SoC for a given application. Moreover, embedding complex algorithms on these systems remains a difficult task due to heterogeneity, it is not easy to decide how to allocate parts of a given algorithm on the different computing units of a given SoC. In order to help automotive industry in embedding algorithms on heterogeneous architectures, we propose a novel approach to predict performances of image processing algorithms applicable on different types of computing units. Our methodology is able to predict a more or less wide interval of execution time with a degree of confidence using only high level description of algorithms, and a few characteristics of computing units.**

Keywords*—*Heterogeneous Architectures; Embedded Systems; Performance Prediction; Image Processing;

I. Introduction

Vehicles increasingly provide ADAS (Advance Driver Assistance System) capabilities, as passive systems (e.g. lane departure warning system alerts the driver when the vehicle crosses a lane) or active systems (e.g. lane centering assist system controls car trajectory). Some of these systems use cameras and image processing algorithms to sense the environment and detect potential obstacles, we can cite lane detection [1], obstacle detection [2], pedestrian detection [3], [4], etc. As they process high amount of data, image processing algorithms need high computational capabilities. Most of state-of-the-art ADAS algorithms run on powerful computers but not in real time. However, image processing algorithms can often be parallelized, so they can benefit from using hardware accelerators like GPUs or multi-core CPUs. Automotive industry needs low-power high computational embedded systems to embed image processing applications on vehicles.

Semiconductor companies are moving into the market of embedded systems for ADAS with heterogeneous architectures. These architectures embed several processing units with different capabilities on the same SoC (System on Chip), often with massively parallel computing unit. We can cite the Tegra K1 SoC, or more recently the Tegra X1, of Nvidia (which embed ARM, GPU and ISP), the TDA2x SoC [5] of Texas Instrument (ARM, DSP and EVE vectorial processor), or the EyeQ of Mobileye [6].

The problem of this type of architecture is the complexity to embed algorithms. First, finding the most suited heterogeneous SoC for a given application is not trivial. Then, as there are several processing units, it is not easy to find the best mapping between algorithms and processing units. Moreover, adjusting an algorithm to embed it on a single processing unit is quite time-consuming, that is why adjusting the algorithm for all processing units to determine the best association is unachievable. In light of that fact, there is a need for car manufacturers and suppliers to predict performances of algorithms on different processing units to help them to choose the best algorithm–processing unit association and the most suited SoC for a given application.

In this work, we introduce a novel methodology to meet automotive industry needs, to predict performances of computationally expensive ADAS algorithms on heterogeneous architectures. First, in section II, we describe the Nvidia K1 and TDA2x heterogeneous SoCs, their characteristics and capabilities. Then, in section III, we introduce our parallelism classification and the problem of kernel mapping optimization. In section IV, we present existing work on performance prediction, followed by our novel approach description in section V. We illustrate our methodology with an example extracted from an ADAS application in section VI. Finally, we discuss future work and conclude in section VII.

II. Embedded Heterogeneous Architecture

Semiconductor companies such as Nvidia, Texas Instrument and Freescale propose heterogeneous architectures to meet automotive industry needs to embed image processing algorithms for ADAS. These architectures handle high performance computing with massively parallel computing unit (e.g. GPU or vectorial processor) and low-power consumption. In this paper, we propose a general approach which can fit with any of these SoCs, and we show some preliminary results obtained with Nvidia Tegra K1 heterogeneous SoC.

A. Nvidia Tegra K1 Architecture

The Nvidia Tegra K1 SoC is composed of quad-core ARM Cortex A15 CPU (1.8 GHz clock rate) providing ARMv7

instruction set and out-of-order speculative issue 3-way superscalar execution pipeline, each core has a NEON and FPU unit [7].

The Kepler GPU of the K1 is composed of one streaming multiprocessor (SMX) of 192 cores [8], accessible with CUDA [9] and the new standard OpenVX [10]. This framework is a hardware abstraction layer for image processing applications supporting modern hardware architectures such as embedded heterogeneous SoCs. OpenVX is based on the implementation of image processing kernels designed by SoCs manufacturers, beneficing from hardware acceleration of architectures.

The parallelization with CUDA is qualified as SIMT (Single Instruction Multiple Threads). The GPU has three memories reachable by all threads:

- Global memory: read and write access.
- Constant memory: read access only.
- Texture memory: read access only, can interpolate adjacent data value, always handles boundary issues.

The global memory of the K1 is the same for GPU and ARM, and both can potentially access to the same data. Memory area of each processing unit is handled by the OS (L4T or Vibrante, which are both based on Linux kernel).

B. K1 Specific Features

The K1 also has several hardware processing units like Image Signal Processor (ISP) and Image Processing Accelerator providing fast and specific image processing algorithms such as debayering, noise reduction, lens correction etc. These units are very fast but the user can only control a limited set of parameters and the chaining order of kernels is restricted.

In addition of basic instructions, ARM and Kepler GPU have specific features which can be used to accelerate image processing algorithms. Thus, ARM processor provides SIMD instructions with NEON units [11]. In order to handle the issues of concurrent reading and writing, CUDA provides atomic instructions based on hardware design.

The Nvidia GPU texture memory provides couple of advantages for reading images data. When accessing to non-integer pixel coordinates (e.g. Im[i-0.5][j-0.5]), texture units handle linear interpolation. This is cost-free because it is computed by hardware design. Moreover texture units handle access to pixels outside the image (e.g. Im[-1][-1]), also cost-free.

C. Texas Instrument TDA2x SoC

The TDA2x is composed of four different types of programmable units. First, it provides 750 MHz dual core ARM A15 and dual-Cortex-M4. These computing units do not bring that much computing capability (A15 core on TDA2x is 4 times less powerful than the one in K1), but they can be used for data management, video acquisition control / rendering, high level decision making, etc.

To handle heavy image processing tasks, TDA2x provides a mix of Texas Instrument fixed and floating point TMS320C66x DSP (Digital Signal Processor) and up to four EVEs (Embedded Vision Engine) cores. TMS320C66x is the most recent DSP from Texas Instrument, it can handle up to 32 multiply accumulate operations per cycle. Each EVE is a 650 MHz core, optimized for image processing, composed of one specific RISC processor and one 512-bit vector coprocessor.

III. Hardware Acceleration and Kernel Mapping

In order to increase performance of a given algorithm, we have to identify parts of code which can benefit from hardware accelerations of a given architecture. This implies to separate the algorithm in more or less fine blocks called kernels.

A. Parallelization Level and Classification

Execution time of a given kernel can be accelerated by using different levels of parallelization of a given architecture. First, one can address parallelism at register level, which we usually call SIMD instructions or vectorization (e.g. NEON for ARM). Some compilers can handle automatic vectorization, e.g. GCC [12], but optimization result can vary depending on compiler.

Secondly, computing units often provide multi-core (ARM provides 4 cores and GPU 192 CUDA cores on K1 SoC). This parallelism can be accessed by using different API, e.g. for multi-core CPU, with pthread (for linux), or at more generic level with OpenMP [13] and C++11/14 parallel features. Pthread is a low level API, which enables fine-grained control over thread management but may be difficult to implement, sometimes too much complicated (depending on code complexity). On the other hand, OpenMP is a high level API, easier to implement, but may have different performances than pthread.

At higher level, heterogeneous architectures offer multi-computing units with different capabilities. Each of these computing units may execute different kernels concurrently.

Speed increasing brought by parallelism is not the same for all kernels, of course it depends on how kernel can be parallelized. To characterize kernels parallelism degree, we propose four simple classes, each one with a specific prediction model:

- *Simple parallelism* P_0: kernels for which there is no dependency between data, each operation can be executed without the result of another one, so concurrently, e.g. convolution.
- *Parallelism using atomic instructions* P_1: kernels for which different threads may write on the same shared data at the same time. Thus mutex, memory barriers or atomic instructions should be used, e.g. histogram construction.
- *Parallel reduction* P_2: kernels for which parallel reduction can be used, e.g. computing the sum of vector elements.
- *Iterative kernels* P_3: kernels for which each iteration depends the on result of the previous one, e.g. if $u_i = f(u_{i-1})$. Due to this dependency, kernels in their original form cannot be parallelized. However, operations executed at each iteration, e.g. $f(u)$, may be parallelized.

Complex algorithms provide a mix of kernels which belong to different parallelism classes.

B. Kernel Mapping Optimization

Embedding a given algorithm on a heterogeneous architecture is a difficult task because one can not easily find how to allocate kernels on the different processing units (kernel mapping) [14]. We propose a metric to evaluate a static kernel mapping efficiency.

Let P be the vector of the processing units of a given heterogeneous architecture, K be the vector of the kernels of a given algorithm, the matrix M constitutes the mapping of K on P , given by $P = M.K$. Let φ be the spatiotemporal dependency matrix: a kernel instance $K_i(t)$ may depends on the execution of another kernel $K_j(t)$, and its previous instance $K_i(t-1)$; φ is used to find possible execution pipelines. Let $\tau(M)$ be the execution function (returning the execution time for each kernel), $\delta(M)$ be the transfer function (returning the transfer delay needed for each kernel), $\eta(M)$ be the occupancy function (returning the occupancy for each computing unit) and $f(P, K, \varphi, \tau(M), \delta(M), \eta(M))$ be the cost function (returning the global execution time of the algorithm). The aim of kernel mapping optimization is to find M minimizing f :

$$\arg\min_{M} \left[f\left(P, K, \varphi, \tau(M), \delta(M), \eta(M)\right)\right]. \qquad (1)$$

Parameters of function f can be measured or predicted for different M. Measure needs all kernels implementation on all processing units (very time consuming), but prediction needs kernels and architectures analysis (very efficient but needs deep SW / HW knowledge).

IV. Related Work

The aim of performance prediction is to estimate the execution time of a given kernel on a given computing unit. It needs characterization of the computing unit (provided by manufacturer or by benchmark results), and characterization of the kernel (high level description, source code analysis, or analysis of binary file).

The difficulty for performance prediction techniques is to find the best compromise between complexity of model and precision of predictions. In fact if technique needs a full optimized source code in order to estimate execution time on one target, it does not bring that much gain compared to measure real performance on real embedded system. Whereas, if technique needs only a high level description of a kernel in order to predict performance for different computing unit, one can rapidly find what target will bring the best performance for that kernel.

A survey of performance modeling techniques is given in [15], according to it three main approaches for performance modeling can be found in literature: analytical modeling, machine learning, and simulation. A performance simulator is able to reproduce a computing unit behavior, it can give a lot of information, identify bottlenecks, predict performance, etc, and generally obtains a fine estimation. Some simulators address hybrid architectures (CPU+GPU), e.g. [16].

Using an architecture simulator implies to port kernel to simulator, which represents about the same amount of work as embedding kernel on the architecture. Moreover, a fine simulation of the system implies a very long execution time.

An analytical model is a set of equations which represents characteristics of the system (kernel and computing unit). Machine learning techniques extract some characteristics by using a set of code and hardware features, then it performs by using feature selection, clustering and regressions techniques to estimate execution times of a kernel, e.g. [17] which addresses CPU and GPU.

Most of analytical model handle performance prediction for only one type of architecture. In [18], authors propose a model based on 47 architecture independent characteristics. Then performances are predicted by using programs in a benchmark suite and measuring similarity between benchmark programs and application, using the 47 architecture independent characteristics. Authors show results on different CPU architectures, but do not address neither GPU nor embedded architectures like DSP.

The famous model of Hong and Kim [19] addresses performance prediction for GPU with two metrics: Memory Warp Parallelism (*MWP*) and Computation Warp Parallelism (*CWP*). The aim of this two metrics is that if $MWP \leq CWP$ then performance is limited by memory bandwidth and latency, but if $CWP > MWP$ then memory latency is hidden by computing operations. Authors report 13.3% mean error on execution time estimation.

The roofline model [20] uses approximatively the same approach by studying the arithmetic intensity of application and memory / computational bandwidths of architecture. However, it is not used for performance prediction, only for bottleneck highlighting and code optimization. The boat hull model [21] adapts the roofline model to provide performance prediction. It is based on a set of primitives, kernel complexity, and memory / computational bandwidths of architecture. It shows good results, 3 and 8% of error for 2 applications. However, the model is limited to kernels which fit in classification given in [22], and cannot handle arbitrary code.

V. Performance Prediction Using the Computing Profile

We propose a novel model for performance prediction which can address multiple architectures, is not limited to a set of primitives, can be used without specific optimized source code and does not need a deep knowledge of architectures. Our methodology is based on a kernel descriptor, called the computing profile.

A. Basic Level of Classification

A kernel can be defined as a set of well-defined instructions, and each instruction can be classified in two classes: computing instructions, and memory instructions. Secondly, we can classify each computing instruction. The key is that one class of computing instructions is executed by one type of unit on the computing unit, e.g. additions will be executed by the ALU, and floating point operations by the FPU.

Classes of instruction are:

- *S_Int*: simple operations on integer: *add*, *sub*, *cmp*...

- *M_Int*: operations with multiplications on integers, this includes multiply accumulate operations.
- *Float*: floating points operations, instructions computed by the FPU.
- *Specific*: specific operations which often encapsulate several instructions, e.g. *div*, *sqrt*, etc.
- *Branch*: branching instructions, e.g. *for* loop, *if*, etc.
- *Address*: addressing operations, e.g. accessing to the ith element of an array.
- *Memory*: load and store instructions.

B. Computing Time Prediction with the Computing Profile

The computing profile of a kernel is an illustration of resources needed by this kernel. The aim is not to get the complexity (number of operations), but to know which classes of instructions are used.

The profile is the ratio of each class, this is the number of instructions of one class divided by the number of instructions of all computing classes. Let C be the set of computing instructions (no memory instructions):

$$C = \{S\_int, M\_int, Float, Specific, Branch, Address\} \tag{2}$$

Let N_c be the number of instructions associated with class c, with $c \in C$. The ratio for the class c, r_c is:

$$r_c = \frac{N_c}{\sum_{i \in C} N_i}. \tag{3}$$

The computing profile shows us which class of instructions is the most used (the maximum of the r_c for all $c \in C$). This information can be used to choose the best architecture for the algorithm. For example an algorithm with a lot of branching operations will have poor performance if embedded on GPU, but could have good performance on ARM, with speculative execution.

Moreover, if we know the throughputs of different classes of instructions for a given architecture, we are able to estimate the computation time for each class of instructions on this architecture. Let $p_{c,a}$ be the throughput (in operations per cycle) of architecture a for class of instructions c, computation time (in cycles) $T_{c,a}$ is:

$$T_{c,a} = \frac{N_c}{p_{c,a}}. \tag{4}$$

If architecture a is scalar, different instructions of different classes cannot be executed simultaneously, so the total computation time is the sum of all $T_{c,a}$:

$$t_a = \sum_{i \in C} T_{i,a}. \tag{5}$$

If architecture a is superscalar, and in the optimal case (when instructions of different classes follow each other with good timing in order to get the maximum benefit from the superscalar capability of the architecture), the computation time is given by the maximum of all $T_{c,a}$:

$$t_a = \max_{\{i \in C\}} T_{i,a}. \tag{6}$$

State-of-the-art architectures are all superscalar, thus we can estimate an execution time interval for the kernel associated with architecture a. In the best case (when the superscalar capability is fully exploited), the computation time is given by the maximum of all $T_{c,a}$. In the worst case (when the superscalar capability is not exploited), the total computation time is the sum of all $T_{c,a}$:

$$t_{min,a} = \max_{\{i \in C\}} T_{i,a} \quad t_{max,a} = \sum_{i \in C} T_{i,a}. \tag{7}$$

Given a kernel and a computing unit, we are able to estimate a computation time interval. The model is based on throughputs of architectures, $p_{c,a}$, this information may be given by manufacturers. Moreover, we establish throughputs for different computing units by using a benchmark, which measures the time to execute a known number of instructions. As our methodology is applicable to different types of computing units, it is possible to compare prediction for different architectures by knowing the clock frequency of each computing unit. If the arithmetic intensity of kernel is high enough then memory access latency are hidden by computation time, thus execution time is equal to computation time.

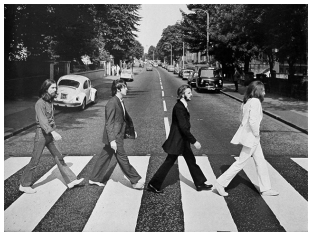
However, if the execution time is limited by memory delay then execution time is greater than predicted computation time. In order to address prediction where performance depends on other parameters than computing profile and throughput, we are using a generic benchmark-vectors set which automatically extracts parameters of the system. The extracted parameters are specific for a given architecture and compiler, it can be acceleration provided with auto-vectorization, memory access delays, transfers delays between computing units, etc. Thus, both information from computing profile and benchmark-vectors set are used to obtain predicted performances and the best mapping for a given algorithm and heterogeneous SoC.

C. Prediction for Consecutive Kernels

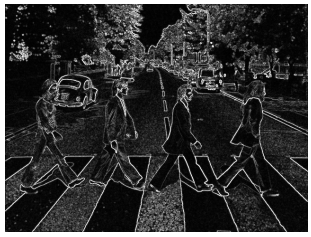
With computing profile prediction, we can obtain an interval of predicted execution time for a given kernel and different computing units. However a complete algorithm is composed of consecutive kernels, so we have to associate predictions of multiple kernels in order to predict performance of a complete algorithm.

First, we can simply use interval arithmetic, to sum intervals of the different kernels in order to obtain another interval. This imply a loss of precision because of a larger interval. Let an algorithm be composed of two consecutive kernels k_1 and k_2, let x_1, x_2 be the real execution time of k_1 and k_2 on a given computing unit. Let $[a_1, b_1]$, $[a_2, b_2]$ be the predicted interval obtained with computing profile for k_1 and k_2, such as $x_1 \in [a_1, b_1]$ and $x_2 \in [a_2, b_2]$. This approach returns another interval for the predicted total execution time: $(x_1 + x_2) \in [a_1 + a_2, b_1 + b_2]$.

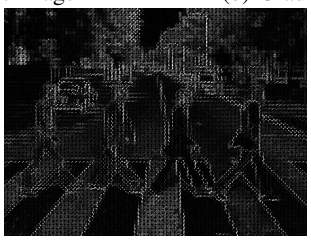
The exact value of execution time is unknown (until we measure it in real condition), so this value belongs to the predicted interval $[t_{min}, t_{max}]$. Without any other specific information, we can modelize the possible values for the execution time as a uniform distribution over the predicted interval $[t_{min}, t_{max}]$. As predictions of two different kernels



(a) Input image



(b) Gradient norm



(c) HOG

Fig. 1: Successive steps of the HOG descriptor.

are independent (as long as they are not executed concurrently), summing two predicted execution times imply summing two independent random variables. Let us now consider two independent random variables X_1 and X_2 representing the estimated execution times, and two uniform distributions $f_{X_1}(p)$ and $f_{X_2}(p)$, describing the relative likelihood that $X_1 = x_1$ and $X_2 = x_2$, defined such as:

$$f_{X_i}(p) = \begin{cases} \frac{1}{b_i - a_1} & \text{for } a_i \leq p \leq b_i, \\ 0 & \text{for } p < a_i \text{ or } p > b_i \end{cases}, i \in \{1, 2\}. \quad (8)$$

Thus:

$$Pr[p_1 \leq X_i \leq p_2] = \int_{p_1}^{p_2} f_{X_i}(p)dp. \quad (9)$$

As X_1 and X_2 are independent, the probability density function of their sum, $f_{X_1+X_2}$, is given by the convolution of the two density functions:

$$f_{X_1+X_2}(p) = \int_{-\infty}^{\infty} f_{X_1}(p-u) f_{X_2}(u)du. \quad (10)$$

The density function $f_{X_1+X_2}(p)$ is non-zero for $p \in (a_1 + a_2, b_1 + b_2)$ and is maximum for $p \in [\min(a_1 + b_2, a_2 + b_1), \max(a_1 + b_2, a_2 + b_1)]$.

For both approach predicted interval become larger each time we add a kernel. However the probabilistic approach enables to reduce the estimated interval with a degree of confidence, as shown in (9), thus we obtain a better precision for performance prediction.

VI. Case of Study: Histogram of Oriented Gradients Embedded on the K1 SoC

To illustrate our method for performance prediction, we take an automotive use-case: pedestrian detection application. We choose to implement the famous algorithm of Dalal and Triggs, which uses histogram of oriented gradients (HOG) descriptor for pedestrian detection [3]. We choose to focus on HOG construction, the algorithm processes in 3 steps:

1) Horizontal and vertical gradient computation obtained by convolution and L1-norm: $\{S\_Int = 0.86, Branch = 0.14\}$.
2) Gradient orientation computation: $\{S\_Int = 0.52, Branch = 0.33, M\_Int = 0.15\}$.
3) Construction of the histogram of orientation for each cell (8×8 pixels in our case): $\{S\_Int = 0.82, Branch = 0.18\}$.

The different steps are illustrated in Fig. 1. Both step 1 and 2 belong to P_0, so are easy to parallelize. However step 3 computes a histogram for each cell (8×8 pixels), this imply to accumulate shared values for pixels of a same cell, thus this kernel is P_1 at pixel granularity.

In order to achieve high arithmetic intensity, temporary values are stored in small buffer, benefiting from low latency of local memory (L1 cache or register).

A. Prediction on ARM

First, we apply prediction model for ARM using one core and no vectorization. Horizontal, vertical and norm of the gradient computation needs 24 *S_Int* and 4 *Branch* operations, so predicted execution time is $t \in [12, 16]$ clock ticks per pixel.

To achieve good performance for the orientation computation, we choose to replace $atan()$ operations by successive branches and to use integer instead of floating point numbers. Thus predicted execution time for orientation computation is $t \in [27, 38]$ clock ticks per pixel.

In the last step, each pixel of a cell vote for an orientation. As one thread handle all pixel of one cell, there is no need of atomic instructions, so predicted time is $t \in [5, 7]$.

Finally, the probability density function is:

$$f(p) = \begin{cases} \frac{1}{11} & 50 \leq p \leq 55 \\ \frac{60-p}{44} & 57 \leq p < 59 \\ \frac{p-45}{44} & 46 < p \leq 48 \\ -\frac{1}{176}(p-54)(p-46) & 48 < p < 50 \\ -\frac{1}{176}(p-59)(p-51) & 55 < p < 57 \\ \frac{1}{176}(p-61)^2 & 59 \leq p < 61 \\ \frac{1}{176}(p-44)^2 & 44 < p \leq 46 \end{cases}. \quad (11)$$

For example, we can affirm execution time $t \in [48, 57]$ with 79% of confidence.

As all kernels are P_0, execution time ARM using 4 cores is 4 times faster than ARM using 1 core. For our test, we work with 640×480 image, on 1.8 GHz ARM using 4 cores. Thus prediction gives us $t = 2240\mu s \pm 8.6\%$ with 79% of confidence. Real execution time is $t_{real} = 2194\mu s$.

B. Prediction on GPU

For GPU, we choose to focus only on the two first steps because histogram construction is P_1 and needs atomic instructions whereas prediction for P_1 kernels are not discussed in this paper. In order to maximize occupancy, each thread computes 4 pixels. For the same operations, prediction on GPU gives us $t \in [0.75, 1.65625]$ clock ticks per pixel, or

$t = 134\mu s \pm 30\%$ to compute gradient norm and orientation for a 640×480 image (clock rate of GPU is 696 Mhz).

The kernel needs to load 4 Words (16 pixels) per thread from global memory, according to our benchmarks these memory operations cost $230\mu s$, which is greater than predicted computation time. Thus execution of our algorithm will be limited by memory bandwidth and latency. Real execution time is $t_{real} = 244\mu s$. We also measure execution time without load operations: $t_{computation} = 132\mu s$. Full execution time, included all steps and memory transfer, is $1030\mu s$.

C. Example of Kernel Mapping

Our application is composed of 3 kernels, and the K1 is composed of 5 computing units (1 GPU and 4 ARM cores), thus 125 different mapping can be tested, and different execution pipelines can be explored. For TDA2x, there are 512 different mapping. Instead of testing all these configurations, it is possible to use our performance prediction model to find the best mapping.

Thus, for K1 our methodology gives us as best mapping to execute the first two steps on GPU and histogram construction on ARM. By using a shared memory area and by optimizing execution pipeline (both ARM and GPU can run at the same time), we achieve execution time $t = 670\mu s$.

VII. Conclusion and Future Work

In this work, we have introduced a novel approach for performance prediction addressing multi-architectures, and operating without specific and optimized source code of application for each architecture. Our methodology is based on a high level description of kernels, the computing profile, and computing throughputs of architectures. We have shown, with an example extracted from an ADAS application, our approach is able to predict a more or less wide interval of execution time with a degree of confidence.

We are working to improve our methodology by taking into account memory delay to predict performances of kernels with small arithmetic intensity. Thus, we are adapting the approach of boat hull model to our methodology, classifying type of memory access found in image processing algorithms and studying there behaviors. Moreover, we are applying our methodology to TDA2x SoC to confirm the general approach for computing units like vectorial processor.

In a future work, we will propose results on kernels which belong to other parallelism classes than P_0. Finally we will show performance prediction using both computing profile and parameters extracted with our set of benchmarks, in order to predict all terms of (1). Thus, we aim to deal with performance prediction of a complex application for different heterogeneous architectures, to help with kernel mapping optimization, and to choose the best suited SoC for a given application.

References

[1] M. Bertozzi and A. Broggi, "Gold: A parallel real-time stereo vision system for generic obstacle and lane detection," *Image Processing, IEEE Transactions on*, vol. 7, no. 1, pp. 62–81, 1998.

[2] E. Dagan, O. Mano, G. P. Stein, and A. Shashua, "Forward collision warning with a single camera," in *Intelligent Vehicles Symposium, 2004 IEEE*. IEEE, 2004, pp. 37–42.

[3] N. Dalal and B. Triggs, "Histograms of oriented gradients for human detection," in *Computer Vision and Pattern Recognition, 2005. CVPR 2005. IEEE Comp. Soc. Conf. on*, vol. 1. IEEE, 2005, pp. 886–893.

[4] D. Geronimo, A. M. Lopez, A. D. Sappa, and T. Graf, "Survey of pedestrian detection for advanced driver assistance systems," *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, vol. 32, no. 7, pp. 1239–1258, 2010.

[5] J. Sankaran and N. Zoran, "TDA2X, a SoC optimized for advanced driver assistance systems," in *Acoustics, Speech and Signal Processing (ICASSP), 2014 IEEE Int. Conf. on*. IEEE, 2014, pp. 2204–2208.

[6] G. P. Stein, E. Rushinek, G. Hayun, and A. Shashua, "A computer vision system on a chip: a case study from the automotive domain," in *Computer Vision and Pattern Recognition-Workshops, 2005. CVPR Workshops. IEEE Comp. Soc. Conf. on*. IEEE, 2005, pp. 130–130.

[7] T. Lanier, "Exploring the design of the cortex-A15 processor," *URL: http://www.arm.com/files/pdf/at-exploring_the_design_of_the_cortex-a15.pdf*, 2011.

[8] NVIDIA, "Nvidia kepler GK110 architecture whitepaper," 2012.

[9] NVIDIA, "Cuda C programming guide," 2014.

[10] E. Rainey, J. Villarreal, G. Dedeoglu, K. Pulli, T. Lepley, and F. Brill, "Addressing system-level optimization with OpenVX graphs," in *Computer Vision and Pattern Recognition Workshops (CVPRW), 2014 IEEE Conference on*. IEEE, 2014.

[11] G. Mitra, B. Johnston, A. P. Rendell, E. McCreath, and J. Zhou, "Use of SIMD vector operations to accelerate application code performance on low-powered ARM and Intel platforms," in *Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2013 IEEE 27th International*. IEEE, 2013, pp. 1107–1116.

[12] D. Naishlos, "Autovectorization in GCC," in *Proceedings of the 2004 GCC Developers Summit*, 2004, pp. 105–118.

[13] L. Dagum and R. Menon, "OpenMP: an industry standard API for shared-memory programming," *Computational Science & Engineering, IEEE*, vol. 5, no. 1, pp. 46–55, 1998.

[14] H. Zhou and C. Liu, "Task mapping in heterogeneous embedded systems for fast completion time," in *Embedded Software (EMSOFT), 2014 International Conference on*. IEEE, 2014, pp. 1–10.

[15] U. Lopez-Novoa, A. Mendiburu, and J. Miguel-Alonso, "A survey of performance modeling and simulation techniques for accelerator-based computing," *Parallel and Distributed Systems, IEEE Trans. on*, 2014.

[16] R. Ubal, B. Jang, P. Mistry, D. Schaa, and D. Kaeli, "Multi2sim: a simulation framework for CPU-GPU computing," in *Proceedings of the 21st international conference on Parallel architectures and compilation techniques*. ACM, 2012, pp. 335–344.

[17] A. Kerr, G. Diamos, and S. Yalamanchili, "Modeling GPU-CPU workloads and systems," in *Proc. of the 3rd Workshop on General-Purpose Computation on GPU*. ACM, 2010, pp. 31–42.

[18] K. Hoste, A. Phansalkar, L. Eeckhout, A. Georges, L. K. John, and K. De Bosschere, "Performance prediction based on inherent program similarity," in *Proc. of the 15th Int. Conf. on Parallel Architectures and Compilation Techniques*. ACM, 2006, pp. 114–122.

[19] S. Hong and H. Kim, "An analytical model for a GPU architecture with memory-level and thread-level parallelism awareness," in *ACM SIGARCH Comp. Architecture News*, vol. 37, no. 3. ACM, 2009, pp. 152–163.

[20] S. Williams, A. Waterman, and D. Patterson, "Roofline: an insightful visual performance model for multicore architectures," *Communications of the ACM*, vol. 52, no. 4, pp. 65–76, 2009.

[21] C. Nugteren and H. Corporaal, "The boat hull model: enabling performance prediction for parallel computing prior to code development," in *Proc. of the 9th Conf. on Comp. Frontiers*. ACM, 2012, pp. 203–212.

[22] C. Nugteren and H. Corporaal, "A modular and parameterisable classification of algorithms," *Eindhoven University of Technology, Tech. Rep. ESR-2011-02*, 2011.