Towards an Automatic Prediction of Image Processing Algorithms Performances on Embedded Heterogeneous Architectures

Romain Saussard, Boubker Bouzid, Marius Vasiliu, Roger Reynaud

▶ **To cite this version:**

Romain Saussard, Boubker Bouzid, Marius Vasiliu, Roger Reynaud. Towards an Automatic Prediction of Image Processing Algorithms Performances on Embedded Heterogeneous Architectures. 2015 International Conference on Parallel Processing Workshops (ICPPW), Sep 2015, Beijing, China. 10.1109/ICPPW.2015.14 . hal-01244398

HAL Id: hal-01244398
https://hal.science/hal-01244398v1

Submitted on 15 Dec 2015

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Towards an Automatic Prediction of Image Processing Algorithms Performances on Embedded Heterogeneous 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*—Image processing algorithms are widely used in the automotive field for ADAS (Advanced Driver Assistance System) purposes. To embed these algorithms, semiconductor companies offer heterogeneous architectures which are composed of different processing units, often with massively parallel computing unit. However, embedding complex algorithms on these SoCs (System on Chip) remains a difficult task due to heterogeneity, it is not easy to decide how to allocate parts of a given algorithm on processing 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 on different computing units of a given heterogeneous SoC. 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 to embed, and a few characteristics of computing units.**

***Keywords*-Heterogeneous Architectures; Performance Prediction; Image Processing;**

I. Introduction

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

In this work, we introduce a novel methodology to meet automotive industry needs, to predict performances of image processing algorithms on heterogeneous architectures. Firstly, we describe in section II the Nvidia K1 and TDA2x heterogeneous architectures, 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 state-of-the-art of 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 works 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 for embedding 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 on Nvidia Tegra K1 heterogeneous SoC.

A. Nvidia Tegra K1 Architecture

The Nvidia Tegra K1 SoC is composed of quad-core ARM Cortex A15 CPU (1.5 GHz clock rate) providing ARMv7 instruction set and out-of-order speculative issue 3-way superscalar execution pipeline, see Fig. 1, each core has 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). GPU has three memories reachable by all threads:

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

The global memory of the K1 is the same for GPU and ARM, both can potentially access to the same data, see Fig. 2. Memory area of each processing unit is handled by the OS (L4T or Vibrante, which are both based on Linux kernel).

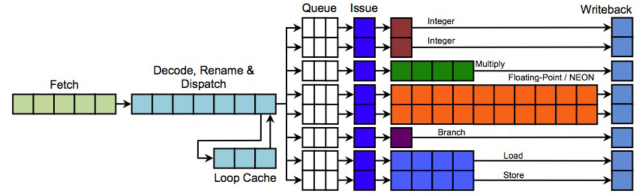

Figure 1. ARM A15 3-way superscalar instruction pipeline.

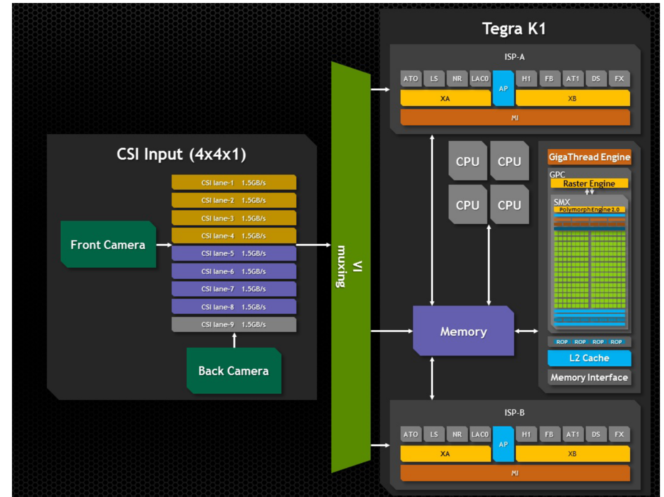

Figure 2. Tegra K1 heterogeneous architecture, global memory is the same for CPU, GPU and ISP.

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 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.

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. The architecture is given in Fig. 3. Firstly, it provides a 750 MHz dual core ARM A15 and a 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 EVE (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.

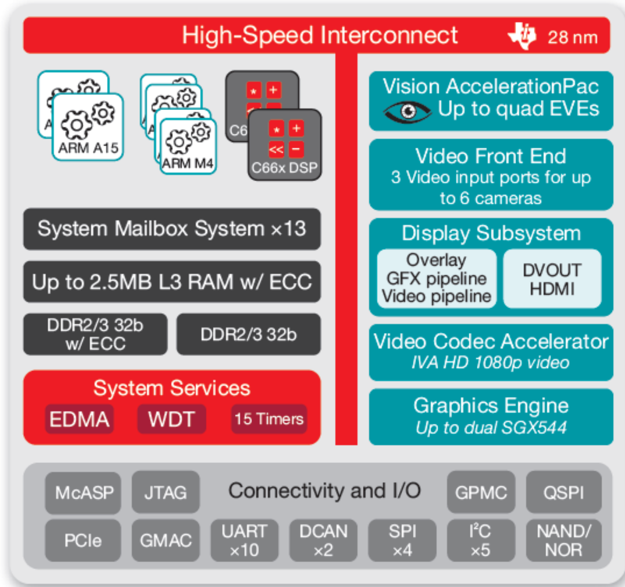

Figure 3. Block diagram of TDA2x heterogeneous architecture.

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, it 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. We propose four simple classes to characterize kernels parallelism degree:

- *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 in the same time. Thus mutexes, 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 sum of all values of a vector.
- *Iterative kernels* P_3: kernels for which each iteration depends on the 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.

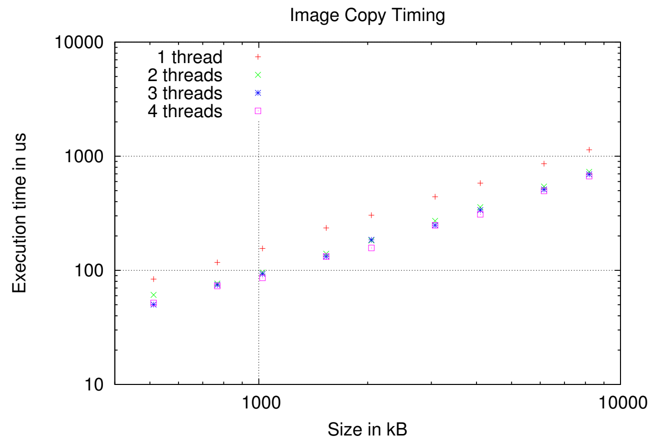

Figure 4. Execution time for an image copy kernel for different image size, using 1, 2, 3, and 4 cores of the ARM.

B. Kernel Concurrency

Multi-core and multi-computing units parallelisms enable to execute multiple kernels concurrently, however execution time of each concurrent kernel may be higher than execution time of one kernel without concurrency. In fact, depending on the architecture, some resources (e.g. global memory or cache) may be shared between cores or computing units, limiting performance of kernels execution.

In the K1 SoC case, we implemented a simple benchmark in order to identify this kind of limits for the ARM processor. This benchmark is a simple kernel which copies an input image to an output image, it processes by using 1, 2, 3, or 4 threads (image is divided by the number of threads, each thread copy one part of the image). As shown in Fig. 4, copy using 1 core is slower than copy using 2 cores, whereas performance is the same when using 2, 3, or 4 cores. Thus cache bandwidth of each core is slower than global memory bandwidth, so executing more than 2 memory transfers concurrently can degrade global execution time.

C. 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].

Let P be the vector of processing units of a given heterogeneous architecture of size $m \times 1$, K be the vector of kernels of a given algorithm of size $n \times 1$:

$$P = \begin{pmatrix} p_1 \\ p_2 \\ \vdots \\ p_m \end{pmatrix}; \quad K = \begin{pmatrix} k_1 \\ k_2 \\ \vdots \\ k_n \end{pmatrix}. \tag{1}$$

The matrix M, of size $m \times n$, constitutes the mapping of K on P , given by

$$P = MK, \tag{2}$$

with $\forall(i,j) \in [\![1,m]\!] \times [\![1,n]\!], M_{i,j} \in \{0,1\}$. Thus $M_{i,j} = 1$ implies that kernel k_j is mapped on processor p_i.

Let φ be the dependency matrix (dependencies between kernels K), with a $n \times n$ size, defined as $\forall(i,j) \in [\![1,n]\!] \times [\![1,n]\!], \varphi_{i,j} \in \{0,1\}$, $\varphi_{i,j} = 1$ implies that kernel k_i depends on k_j results.

Let:

- $\tau(M)$ be the execution function, returning a $n \times 1$ vector corresponding to the execution time of each kernel with mapping M;
- $\delta(M)$ be the transfer function, returning a $n \times 1$ vector corresponding to the transfer delay needed for each kernel with mapping M;
- $\eta(M)$ be the occupancy function, returning a $m \times 1$ vector corresponding to the occupancy rate of each processor with mapping M;

Then, let $f(P, K, \varphi, \tau(M), \delta(M), \eta(M))$ be the cost function, returning the global execution time of the algorithm executed on the heterogeneous architecture with mapping M. The aim of kernel mapping optimization is to find M minimizing f:

$$\arg\min_{M} \left[f\left(P, K, \varphi, \tau(M), \delta(M), \eta(M)\right)\right]. \tag{3}$$

Parameters of function f can be measured or predicted for different M. This evaluation needs all kernels implementations on all processing units (very time consuming), but prediction only needs kernels and architectures analysis (very efficient but needs deep SW / HW knowledge). In this paper, we present a methodology to estimate the execution $\tau(M)$ and transfer $\delta(M)$ times with little knowledge of target architecture.

Our approach is based on two levels of accuracy. The first one uses all basic information provided by manufacturers; but this level cannot deal with cache effects, compiler optimizations, memory latency, concurrent memory access, resources starvation, etc. Thus the second level overcomes these limitations by using a generic benchmark-vectors set which automatically extracts parameters (e.g. L1, L2, L3 caches, concurrent access, or compiler auto-vectorization effects on execution and transfer times, etc.). These parameters can be extracted for any architectures and compilers.

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 a characterization of computing unit (provided by manufacturer or by benchmark results), and a characterization of kernel (high level description, source code or binary file analysis).

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 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 than embedding kernel on 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 a 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 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,
- does not need a deep knowledge of architectures.

Our methodology is based on a kernel descriptor, 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: the computing instructions, and the 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, e.g. *add*, *sub*, *cmp*, etc.
- *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 member of an array.
- *Memory*: load and store instructions.

The arithmetic intensity [23] is defined as the number of operations per memory instruction (load and store). Given N_M, the number of memory instructions and N_C the number of computing instructions, the arithmetic intensity Ia is defined as:

$$Ia = \frac{N_C}{N_M}. \tag{4}$$

This metric can be used to identify the way to optimize the algorithm, for example with the roofline model [20]. Thus, performance of algorithm with small arithmetic intensity is limited by memory bandwidth and latency, whereas performance is limited by computing capability for algorithm with high arithmetic intensity. In fact, memory access latency can be hidden by multiple computing operations.

On state-of-the-art architectures, it is not easy to predict the number of global memory access and the delay of each one because of sophisticated cache systems. Indeed there are different levels of cache with different latencies and bandwidths.

B. Computing Profile and Throughput

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):

$$C = \{S_int, M_int, Float, Specific, Branch, Address\} \tag{5}$$

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:

$$r_c = \frac{N_c}{\sum_{i \in C} N_i}. \tag{6}$$

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 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:

$$T_{c,a} = \frac{N_c}{p_{c,a}}. \tag{7}$$

If architecture a is scalar, different instructions of different classes cannot be executed simultaneously, so total

computation time is the sum of all $T_{c,a}$:

$$t_a = \sum_{i \in C} T_{i,a}. \tag{8}$$

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), computation time is given by the maximum of all $T_{c,a}$:

$$t_a = \max_{\{i \in C\}} T_{i,a}. \tag{9}$$

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 $T_{c,a}$:

$$t_{min,a} = \max_{\{i \in C\}} T_{i,a} \quad t_{max,a} = \sum_{i \in C} T_{i,a}. \tag{10}$$

Given a kernel and a computing unit, we are able to estimate a computation time interval. In our case, that can be helpful to estimate the best computing unit for a given kernel. Indeed, our method is applicable to different kind of computing units, and results can be compared by knowing clock frequency of each computing unit.

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 implies 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 value for the execution time as a uniform distribution over the predicted interval $[t_{min}, t_{max}]$. As predictions of two different kernels are independent (as long as they are not executed concurrently), summing two predicted execution times implies 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:

$$f_{X_i}(p) = \begin{cases} \frac{1}{b_i - a_1} & \text{for } a_i \leq p \leq b_i, \\ 0 & \text{for } p < a_i \text{ or } p > b_i \end{cases}, i \in \{1, 2\}. \tag{11}$$

Thus:

$$Pr[p_1 \leq X_i \leq p_2] = \int_{p_1}^{p_2} f_{X_i}(p) dp. \tag{12}$$

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:

$$f_{X_1+X_2}(p) = \int_{-\infty}^{\infty} f_{X_1}(p-u) f_{X_2}(u) du. \tag{13}$$

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 approaches, 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 (12), thus we obtain a better precision for performance prediction.

VI. Case of Study: a Lane Detection Algorithm Embedded on the K1 SoC

To illustrate our method for performance prediction, we take an automotive use-case: lane detection application. The image processing part of our application takes about 95% of computational needs. It is divided into four levels:

1) Gradient computation: the input color image is converted to grayscale, the gradient is computed with a horizontal Sobel operator, and finally a threshold is applied in order to get a binary image.
2) A Bottom-Hat filter is applied to reduce the number of points for the next step.
3) The lane detection is performed by a Hough transform.
4) High level kernel for decision-making.

The gradient computation and the Bottom-Hat filter outputs are illustrated in Fig. 5.

A. Throughputs of the K1 SoC

As seen in previous section, the model is based on throughputs of architectures, $p_{c,a}$. Manufacturers may give information about throughput for each class, e.g. for Nvidia this can be found in [9].

Moreover, we establish throughputs for different computing units by using a benchmark, which measures the time to execute a known number of instructions. Time is measured with performance counter registers of each architecture.

(a) Input image

(b) Gradient

(c) BottomHat

Figure 5. Successive steps of the lane detection application, and their computing profile.

Throughputs for ARM and GPU of the K1 SoC are given in Tab. I. As *Specific* operations on ARM imply multiple instructions, there is not a unique throughput for this class. Throughputs are given for one core, or one SMX for the GPU. Speed increasing provided by using 4 cores on ARM vary depending on parallelism degree of kernel and API used. For a P_0 kernel with high Ai, we assume that using the best acceleration technique provides a speed increasing equal to the number of threads (e.g. using 4 cores reduces execution time by 4).

The throughputs of NEON operations on ARM are the same than corresponding regular operations, e.g. NEON unit can execute 2 SIMD additions per cycle. However using NEON implies that data have to be loaded into NEON registers from ARM registers, and have to be stored from NEON registers to ARM registers, throughput for loading and storing NEON instructions can vary depending on data alignment.

B. Example of Performance Prediction with Horizontal Gradient

In order to illustrate our methodology for performance prediction, we present an example with the Gradient algorithm of our lane detection application. The algorithm processes in two steps, first a RGB to gray conversion and then a horizontal Sobel filter. In order to achieve high Ai, we

Table I
THROUGHPUTS FOR ARM AND GPU OF THE K1 ARCHITECTURE. VALUES ARE GIVEN IN INSTRUCTIONS PER CYCLE.

Class of instructions	ARM A15	GPU Kepler
S_Int	2	160
M_Int	1	32
Float	1	192
Specific	$*$	32
Branch	1	32
Address	1	32
NEON load & store	0.5	/

choose to store temporary values in a small buffer, benefiting from low latency of L1 cache. Thus complete algorithm performs only 3 loads (one for each color component) and 1 store from / to global memory for each pixel.

RGB to gray conversion is given by listing 1, with pIn pointer on input image, and $pOut$ pointer on output image. For each pixel, it performs 3 *M_Int*, 5 *S_Int* plus 2 if we consider that multiplication-accumulation is not used, and 1 *S_Int* plus 1 *Branch* for the *for* loop.

Listing 1. RGB to Gray conversion
```
*(pOut++) = (*(pIn)*28 + *(pIn+1)*151 + *(pIn+2)
    *77) >> 8;
pIn+=3;
```

We choose to separate the Sobel filter into two filters. The first one is given by listing 2, for each pixel it performs 5 *S_Int*, and 1 *S_Int* plus 1 *Branch* for the *for* loop.

Listing 2. First Sobel filter
```
*(pOut++) = *(pIn-1) - *(pIn+1);
pIn++;
```

The second one is given by listing 3, the *Step* variable is the distance between vertically consecutive pixels. For each pixel it performs 7 *S_Int* (multiplication by 2 is counted as *S_Int*, because corresponds to an arithmetic left shift), 1 *S_Int* plus 1 *Branch* for the *if* condition, 1 *S_Int* for the *abs* operator , and 1 *S_Int* plus 1 *Branch* for the *for* loop.

Listing 3. Second Sobel filter
```
var = abs(*(pIn-Step) + *(pIn)*2 + *(pIn+Step))
if (var > 255)
        *(pOut++) = 255;
else
        *(pOut++) = var;
pIn++;
```

1) Prediction on ARM: First, we apply prediction model for ARM using one core and no vectorization. We assume Ai is high enough to consider performance is limited by computing capability of the architecture, not by memory bandwidth. Predicted execution time, t, is given in number of clock ticks per pixel:

- RGB to Gray: $t \in [4, 8]$
- First Sobel Filter: $t \in [3, 4]$
- Second Sobel Filter: $t \in [5, 7]$

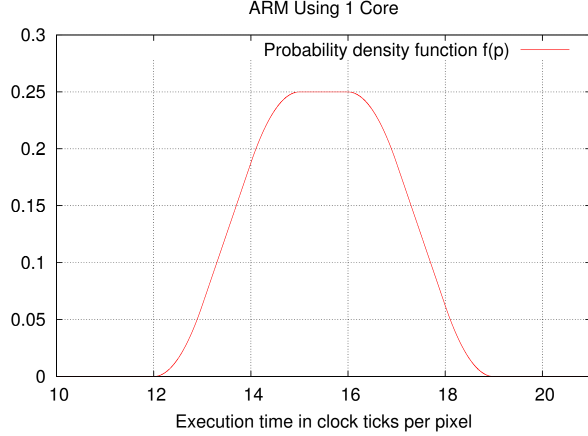

Figure 6. Probability density function of predicted execution time for gradient algorithm using 1 core and no vectorization on ARM. Measured execution time is 15.87 clock ticks per pixel.

The global execution time, obtained by summing the intervals, is $t \in [12, 19]$, or $t = 15.5 \pm 22.6\%$ clock ticks per pixel.

The probability density distribution $f(p)$, obtained by convolving probability density distribution of each kernel, is:

$$f(p) = \begin{cases} (p-12)^2/16 & \text{for } 12 \leq p \leq 13, \\ (p-12.5)/8 & \text{for } 13 \leq p \leq 14, \\ 0.25-(p-15)^2/16 & \text{for } 14 \leq p \leq 15, \\ 0.25 & \text{for } 15 \leq p \leq 16, \\ 0.25-(p-16)^2/16 & \text{for } 16 \leq p \leq 17, \\ (p-18.5)/8 & \text{for } 17 \leq p \leq 18, \\ (p-19)^2/16 & \text{for } 18 \leq p \leq 19, \\ 0 & \text{for } p < 12 \text{ or } p > 19. \end{cases} \quad (14)$$

Distribution $f(p)$, shown in Fig 6, is maximum for $p \in [15, 16]$. According to the density function, the probability that execution time $t \in [14, 17]$ (or $t = 15,5 \pm 9.7\%$) is 70%.

We implement Gradient algorithm on ARM, and apply it on a 1280×720 image. Measured execution time is 9.750 ms, 15.87 clock ticks per pixel which is in the interval with 70% confidence, and represents a difference of 2.5% compared with the average of the interval.

To benefit from multi-core and decrease execution time, image is divided into four parts, each thread processes one part of the image. In order to avoid side effect, each thread processes their parts plus one line, in our case this represents blocks of 1280×181 pixels. Based on prediction with 1 core, we can estimate execution time using 4 cores. As all kernels used for Gradient algorithm are P_0, and pthread is used, we assume there is no waste of time.

One thread processes on 1280×181 pixels, thus predicted time is $t \in ([12, 19] \times 1280 \times 181)$ clock ticks. If we divide by the total number of pixels, then $t \in [3.017, 4.776]$ clock ticks per pixel. Measured value is 2.560 ms, 4.17 clock ticks per pixel, it represents a difference of 7% compared with the average of the interval.

Gradient algorithm can also be accelerated by using NEON. Performance with auto-vectorization depends on compiler capabilities, so our methodology using computing profile can not work. Moreover, in this example memory delay is hidden by computing time. In order to address prediction where performance depends on other parameters than computing profile and throughput, we are using the second level of our estimation approach (a generic benchmark-vectors set injected on this specific architecture). The extracted parameters can be acceleration provided with auto-vectorization, memory access delays, transfers delays between computing units, etc. Then, both information from computing profile and benchmarks are used to obtain predicted performances (cost function f) and the best mapping for a given algorithm and heterogeneous SoC.

2) Prediction on GPU: We apply the same methodology on the same algorithm for the GPU of K1. However, in order to maximize occupancy [24], each thread computes four pixels, and Sobel filter is not separated. Moreover RGB to Gray conversion uses floating point operations. Temporary values are stored in shared memory in order to minimize the number of global memory access.

With CUDA API, image indexes are obtained with threads IDs, there is no need of *for* loop. Thus in RGB to Gray kernel, each thread performs 12 *Float*, 32 *S_Int* and 1 *Branch* (to handle borders of blocks). In Sobel kernel, each thread performs 29 *S_Int*, 4 *Branch*, and 4 *Specific* (absolute value operator belongs to *Specific* class for the GPU). Thus prediction, in clock ticks per pixel, for each kernel are:

- RGB to Gray: $t \in [0.2, 0.29275]$
- Sobel: $t \in [0.18125, 0.43125]$

Predicted global execution time is $t \in [0.38125, 0.724]$ clock ticks per pixel, or $t \in [829, 1574]$ Mp/s (clock rate of K1 GPU is 600 MHz). According to the probability density function, given in Fig. 7, $t \in [0.474, 0.63125]$ clock ticks per pixel with 63% of confidence.

Measured execution time on a 1280×720 image is 0.606 clock ticks per pixel, 990 Mp/s, which is in the interval with 63% of confidence, and represents a difference of 9.66% compared with the average of the interval. Fig. 8 shows results of predictions and measured execution times for both ARM 4 cores and GPU with different image sizes.

C. Kernel Mapping

Our application is divided into four parts and K1 SoC has two programmable computing units, thus more than 128 different configurations can be explored (different mapping, number of core to use, execution pipeline, etc). On TDA2x,

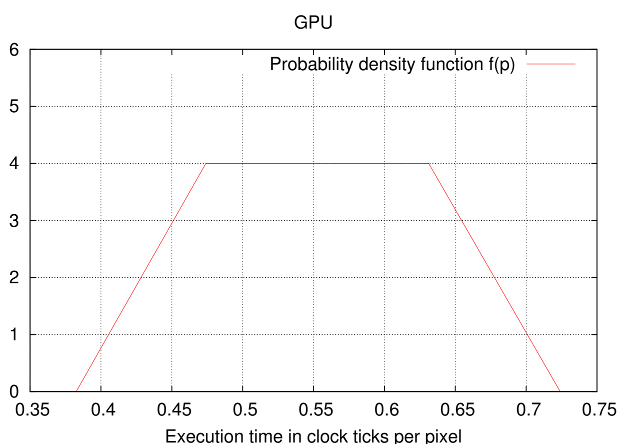

Figure 7. Probability density function of predicted execution time for gradient algorithm using GPU. Measured execution time is 0.606 clock ticks per pixel.

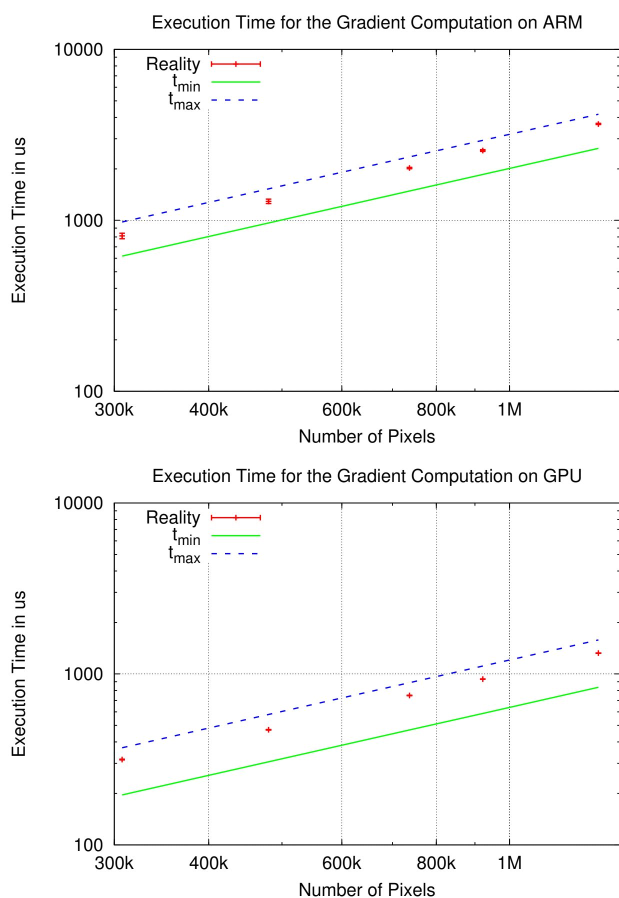

Figure 8. Measured and predicted performances for the Gradient computation on ARM using 4 cores (top) and GPU (bottom). Both axis are in log scale.

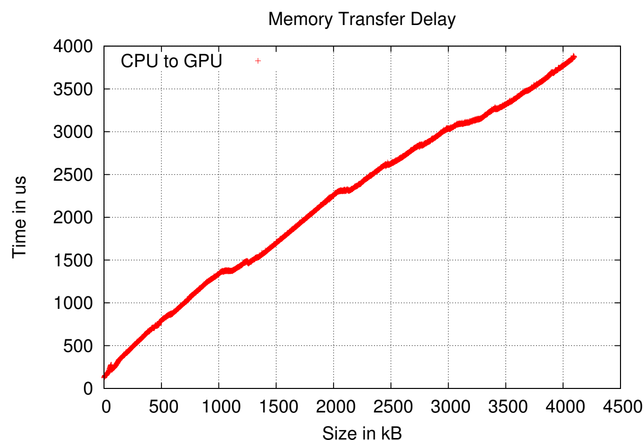

Figure 9. Result of our benchmark measuring delay needed to transfer data from CPU to GPU global memory. Nonlinearities are due to cache effects.

there are four programmable computing units, so more than 1024 different configurations can be studied. Our methodology aims to automatically find one of the best mapping without porting any kernel.

On K1, predictions show that computing time is slower on GPU than on ARM for all kernels except for the high level decision making. Acquisition is handle by the ARM, thus executing a kernel on GPU implies copying data to GPU memory. According to our benchmarks (see Fig. 9), copying a RGB image and running the kernel on GPU is more time consuming than running it on ARM. The Hough transform kernel only need few pixels coordinates as input data, returns lines coordinates, and prediction shows it has good performance on GPU. Thus one of the best mapping for our application is to embed Hough transform on GPU and everything else on ARM with 4 cores. With this mapping, the application runs in 10.4 ms on a 1920×1080 HD image.

On TDA2x, we are working to embed lane detection application. Our methodology gives us that one of the best configuration is to run the two first kernels on DSP, then Hough transform on EVE, and finally high level decision making on ARM.

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 ker-

nels 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 method 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 (3). Thus, we aim to deal with performance prediction of a complex application for different heterogeneous SoC, 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 Computer Society Conference 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 International Conference 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 Computer Society Conference 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 Transactions 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 *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*. 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 *Proceedings of the 15th international conference 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 Computer 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 *Proceedings of the 9th conference on Computing 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.

[23] M. Harris, "Mapping computational concepts to GPUs," in *ACM SIGGRAPH 2005 Courses*. ACM, 2005, p. 50.

[24] M. Harris, "Optimizing cuda," *SC07: High Performance Computing With CUDA*, 2007.