A Robust Methodology for Performance Analysis on Hybrid Embedded Multicore Architectures

Romain Saussard, Boubker Bouzid, Marius Vasiliu, Roger Reynaud

A Robust Methodology for Performance Analysis on Hybrid Embedded Multicore Architectures

Romain Saussard*† Boubker Bouzid* Marius Vasiliu† Roger Reynaud†
*Renault S.A.S. (Guyancourt, France)
Email: {romain.saussard, boubker.bouzid}@renault.com
†SATIE, ENS Cachan, Université Paris Saclay (Orsay, France)
Email: {romain.saussard, marius.vasiliu, roger.reynaud}@u-psud.fr

***Abstract*—Today's vehicles increasingly embed software intelligence in order to be safer for the driver, and to achieve autonomous driving in a close future. To answer the computational needs of these algorithms, system-on-chip (SoC) suppliers propose heterogeneous architectures. With such complex SoCs, embedding applications in vehicle becomes more and more complex for car manufacturers. Indeed, it is not trivial to find the best suited SoC for a given application, and to define load balancing strategies when working with heterogeneous architectures. These difficulties can be overcome by using performance prediction, based on computing architectures models. To build these models, we provide a set of test vectors which automatically extract key characteristics of tested architectures. Our methodology is able to perform a complete computing architecture model, by using 3 different levels of tests, each one characterizing a specific situation representative of real applications. We aim to obtain performance prediction for different applications, for any embedded SoCs based on models performed with this methodology. In this paper, we describe our characterization methodology, and show results obtained with embedded SoCs used for automotive applications.**

I. Introduction

Computational needs for vehicles are increasing with the emergence of advanced driver assistance systems (ADAS) and autonomous driving systems. Indeed, vehicles increasingly use sensors and complex algorithms to sense the environment. Thus, automotive industry uses more and more computational powerful systems-on-chip (SoCs), especially to embed image processing algorithms. Image processing algorithms process high amounts of data, so they need high computational resources. Most of state-of-the-art image processing algorithms for ADAS are designed to be executed on powerful computer, sometimes not in real time. Nevertheless, they often can be easily parallelized, so they can be accelerated by massively parallel architectures. This implies an effort from automotive actors to bridge the gap between prototypes on computer and embedded SoCs on vehicles.

Semiconductor companies are moving into the market of ADAS with heterogeneous SoCs. These systems provide different kind of processors to bring more computational performance and energy efficiency. We can cite Nvidia (Tegra K1 and X1), Texas Instruments (TDA2x), Renesas (R-Car H2 and R-Car H3). In [1], we show that computational intensive image processing algorithms, such as lane detection, can be embedded and executed in real time on heterogeneous SoCs.

When embedding software applications on vehicle, two challenges need to be addressed. First, one needs to guess the best suited SoC for a given application, then one needs to find how to partition the computational load on the different processors of the heterogeneous SoC. In [2], [3] we present a way to address these challenges, by using performance prediction. However, performance prediction models need architecture characteristics to proceed, and this information is not always provided by suppliers.

In this paper, we propose a generic methodology to automatically extract key features of heterogeneous SoCs. Characteristics extracted show few raw performance indicators for the measured architectures, and can be used in a performance prediction model. First, in section II, we briefly present some heterogeneous SoCs used for ADAS applications. Then, in section III, we present existing work on architecture characterization, followed by our novel approach description in section IV. Finally, we show some results obtained with our methodology in section V, and we discuss about future work and conclude in section VI.

II. Heterogeneous SoCs for ADAS

In order to meet automotive industry needs for embedding computational intensive ADAS applications, semiconductor companies such as Nvidia, Renesas, or Texas Instruments propose heterogeneous SoCs. These architectures are often composed of ARM processors, for general processing, and of one or few hardware accelerators to handle computationally intensive operations such as image processing. In this paper, we propose a generic methodology to characterize computing architectures, including modern computers and any of these SoCs. Architectures characteristics are summarized in table I.

A. Nvidia Tegra K1 and X1

Nvidia has released two embedded architectures for ADAS applications: the Tegra K1 and the Tegra X1. Both are composed of ARM processors, GPUs and Image Signal Processors (ISPs). The Tegra K1 consists of a quad-core ARM Cortex A15 CPU providing ARMv7 instruction set and a Kepler GPU, composed of one streaming multiprocessor (SMX) of 192 cores. The development platform is known as Nvidia Jetson TK1. The Tegra X1 is composed of a quad-core ARM Cortex A57, a quad-core ARM Cortex A53 providing ARMv8

TABLE I: Summary of hardware configurations.

SoC	Processors	Parallel Units	Classification	Parallel Degree	Memory Bandwidth	Tools
Tegra K1	4×A15 with NEON	1 SMX Kepler GPU	SIMT	192 CUDA cores	17 GB/s	GCC, CUDA, Visionworks (OpenVX)
Tegra X1	4×A57, 4×A53 with NEON	2 SMX Maxwell GPU	SIMT	2×128 CUDA cores	26.5 GB/s	GCC, CUDA, Visionworks (OpenVX)
R-Car H2	4×A15, 4×A7 with NEON	IMP-X4	MIMD	*Under NDA*	6.4 GB/s	GCC
TDA2x	2×A15 with NEON, 2×C66x DSPs	4×EVE	SIMD	512 bits vector size	n.a.	GCC, cl6x, TI Vision SDK

instruction set, and a Maxwell GPU composed of two SMX of 128 cores. The development platform for autonomous cars, DrivePX, provides 2 Tegra X1.

ARM processors provide out-of-order speculative issue superscalar execution pipeline. Each core handles SIMD instructions with a NEON unit and has a FPU unit. GPU are well known for accelerating image processing application execution time [4]. CUDA framework [5] is used for kernel implementation on the GPU of Tegra K1 and Tegra X1. The ISP provides fast and specific image processing kernels such as delayering, noise reduction, etc. This unit is very fast, but user can only control a limited set of parameters and the chaining order of kernels is restricted.

B. Renesas R-Car

Renesas has released two embedded architectures for ADAS applications: the R-Car H2 and the R-Car H3. Both are composed of ARM processors, programmable hardware accelerators to handle image processing kernels, and ISPs.

The R-Car H2 provides a quad-core ARM A15 and a quad-core ARM A7. It is composed of a programmable hardware accelerator, the high performance real-time image recognition processor (IMP-X4).

C. Texas Instruments TDA2x

The TDA2x [6] is composed of four different types of programmable units. First, 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 Instruments 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 Instruments, 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.

D. Software Environment

Each SoCs manufacturer has its own software environment to develop on its platform. For example Nvidia provides the CUDA API for GPU programming, Texas Instruments proposes cl6x (DSP compiler), specific tools to program EVE processors, and TI Vision SDK [7]. Thus, a user who wants to work on different platforms has to learn different software API, languages, etc. In addition, porting an algorithm from PC to an embedded SoC, with a specific API, is not trivial.

To facilitate portability and genericity, the Khronos Group has released the OpenVX standard. OpenVX [8] is an open, royalty-free standard for cross platform acceleration of computer vision applications. It enables performance and power-optimized computer vision processing, it is based on the implementation of image processing kernels designed by SoC manufacturers, beneficing from hardware acceleration of the architecture.

Nvidia, Renesas and Texas Instruments take parts to the OpenVX specification definition. In addition, Nvidia has released an OpenVX implementation to address Tegra K1 and Tegra X1: VisionWorks. To our knowledge, Texas Instruments and Renesas has not released yet an OpenVX implementation for their platforms.

E. Kernel Mapping Optimization

When working with heterogeneous architectures, a challenge needs to be addressed: how to partition the different tasks (or kernels) of the algorithm to embed on the different processors of the heterogeneous architecture. That is what we call the kernel mapping problem, introduced in [1]–[3].

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 the kernel mapping optimization is to find M minimizing f :

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

As discussed in [2], [3], the parameters of the function f can be predicted for different M. In these papers we have presented a methodology to estimate the execution $\tau(M)$ and transfer $\delta(M)$ times with little knowledge of target architectures.

Our performance prediction approach is based on two levels of accuracy. The first one builds an architecture model based on 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 test vector set which automatically extracts features, for any architectures and compilers in order to build a more precise model for the association architecture + compiler. In this paper, we explain our methodology for extracting some features of the tested architectures and their software environment.

III. Existing Work on Architecture Characterization

A. Legacy Metrics

One of the most well-known metrics to characterize architecture performance is the FLOPS (Floating-point Operations Per Second). It only measures floating point operations, unlike the MIPS (Millions of Instruction per second) which measures all type of instruction (moving, computing, comparing, etc.) executed in a second. However, MIPS cannot be used to compare different instruction sets for the same workload (e.g. CISC vs RISC).

The Dhrystone benchmark [9] was created to be representative of CPU performance. It does not use floating point operations, the output corresponds to the number of Dhrystones per second (the number of iterations of the main code loop per second), measured in D-MIPS.

B. Benchmarks for Computing

The benchmark consists in measuring execution time of workloads for a given computing architecture. Then, benchmark results can be compared for many architectures. However, results are strongly dependent on workloads (parallelism degree, operations used, etc.).

1) Serial CPU: The SPEC [10] benchmark is made of a set of 49 compute-intensive workloads. It can only measure performance for CPU (no parallelism), it is based on integer and floating-points operations[11]. The Embedded Microprocessor Benchamark Consortium (EMBC) [12] uses the CoreMark benchmark [13]. This benchmark outputs only one scalar, representing computing architecture capability. It is based on non-contiguous memory operations, matrix operations (to illustrate hardware acceleration like MAC, or SIMD instructions), and state machine processing (to illustrate branch prediction performance).

2) Parallel Computing: Some benchmarks are defined to characterize parallel architecture. For example, SPLASH-2 benchmark [14] aims to characterize multi-processor architecture. For different workloads, it measures computing load repartition between all processors, communication/computing ratio, impact of dataset size, etc.

3) Heterogeneous Computing: The famous Rodinia benchmark [15] is composed of multi-core CPU and GPU kernels (using OpenMP and CUDA). Workloads are inspired by Berkeley's dwarf taxonomy [16]. It characterizes communication between processing units, synchronization techniques, power consumption, etc. Parboil [17] is quite similar, but only based on few kernels. Scalable Heterogeneous computing (SHOC) [18] uses OpenCL and CUDA implementation to target CPU+GPU architectures. It is based on two different types of test: Level zero (low level characteristics of the architecture, e.g. PCIe Bus speed, Memory Bandwith, Kernel Compilation, Peak FLOPS, etc.), and Level One (based on parallel kernels, e.g. FFT, matrix multiplication, etc.).

C. Benchmarks for Memory

Some benchmarks aim to characterize memory performance. For example the STREAM benchmark [19] measures performance for 4 different kernels: Copy $a[i] = b[i]$, Scale $a[i] = q \times b[i]$, Sum $a[i] = b[i] + c[i]$, and Triad $a[i] = b[i] + q \times c[i]$. The Low level architectural characterization benchmark suite (LLCBench) [20] characterizes architecture with a set of 3 benchmarks: MPBench for MPI messaging, BLASBench for computing performance based on BLAS routines, and CacheBench for memory performance. It only outputs raw data and graphs.

IV. Our Methodology

Most of the state-of-the-art benchmarks focus on specific applications, or focus on one type of operation (floating-point or integer). In order to obtain low level characteristics of tested architectures, we developed a novel methodology which is based on a set of performance test vectors. Our approach is generic in the sense that it can target any programmable architectures (software meaning), including PC and any of the heterogeneous SoCs listed in section II. Our methodology is composed of 3 levels: low level, mid level, and high level test vectors. Test vectors are all

A. Low Level Test Vectors

These test vectors aim to extract low level characteristics of the tested architecture. It is divided into two parts computing test vectors (extracting computing capability of the architecture) and memory test vectors (extracting bandwidth and latency).

1) Memory: To start with, let us define memory operation that we want to characterize:

- *Read - Read*: 2 consecutive reading on the same data.
- *Read - Write*: 1 reading, then 1 writing on the same data.
- *Write - Read*: 1 writing, then 1 reading on the same data.
- *Write - Write*: 2 consecutive writing on the same data.
- *Mcpy*: memcopy with $@src \neq @dst$.
- *Trsf*: memory transfer from a computing unit to another.

The tests consist in measuring time needed to execute one of the six memory operation for different sizes. Two strategies

are used: in one test, allocated data size is fixed and we vary the numbers of memory operations; in the other test, we vary the allocated memory size. In all the tests, we use contiguous memory access in order to obtain maximum bandwidth and to be close to image processing use-case. However, it is possible to apply the same methodology with non-contiguous access.

2) Computing: We want to characterize a computing operation (e.g. sum, multiplication, multiply accumulate, bitwise operators, etc.) for different data type (*int32*, *int16*, *int8*, *float64*, *float32*). Thus, for a given operator, for each data type, we measure the execution time of a known number of operations while varying the arithmetic intensity (I_A). The I_A is the ratio of computing operations to memory operations.

So, we obtain an execution time depending on the I_A as described in the Boat Hull Model[21]:

- Execution time is constant when the I_A is low, performance is limited by memory bandwidth and latency.
- Execution time is proportional to the I_A when the I_A is high enough, memory bandwidth and latency are hidden by computing operations. The slope represents the throughput of the processor for the tested operation and the variable type used.

Note that the limit between a low and a high I_A is specific for each processor, it depends on its memory bandwidth and computing capabilities. These tests are based on few loops with a known size at compilation time. So they also highlight compiler capabilities, illustrating the auto-vectorization and loop-unrolling performances.

These tests are designed to achieve high parallelism degree, to fully exploit massively parallel processors such as GPU, or multi-core CPU with SIMD instructions. In order to characterize sequential execution, we also run these tests without parallelism acceleration (e.g. for CPU, by using only 1 core without SIMD instructions).

3) Conflict Management: Conflict management ensure isolation of shared data from concurrent threads, while causing performance decrease when two or more thread simultaneously try to operate on the same data. These test vectors study the behavior of these specific mechanisms for different architectures (e.g Mutex for CPU, or Atomic instructions for GPU).

B. Mid Level Test Vectors

Mid level test vectors consist in characterizing concurrency execution on a processor, i.e. the impact of executing few independent tasks at the same time on the same processor. We use the previously defined low level test vectors, running two or more low level tests concurrently (e.g. multiplication and addition, or addition and memory, etc.) in two different configurations on the same processor:

- Concurrent execution on different cores: the tests may only share few cache levels, so it only impacts memory access delay.
- Concurrent execution on the same core: the tests share all resources, execution time is fully impacted.

TABLE II: Configurations of tested architectures.

SoC	GCC	Compiler flags	CUDA
K1	4.8	*-O3 -mtune=cortex-a15 -mfpu=neon-vfpv4 -funroll-loops -fopenmp*	7.0
X1	4.9	*-O3 -mtune=cortex-a57 -funroll-loops -fopenmp*	7.5
R-Car	4.8	*-O3 -mtune=cortex-a15 -mfpu=neon-vfpv4 -funroll-loops -fopenmp*	/

These tests highlight some hazards on performance due to concurrency, that are not described in datasheets. Results are integrated in the architecture model, to achieve more accurate performance prediction.

C. High Level Test Vectors

High Level testing uses as vectors real-time programs (e.g. ADAS applications) built from full known source code or using some binary libraries. The last case is the most difficult to evaluate or to predict, because binary blocks must be treated as "black-boxes".

We aim to use characteristics extracted by the low level tests to predict the "black-box" behavior through different architectures. A typical example are applications using the new standard OpenVX. As discussed in section II-D, this new standard enables to execute the same code on different heterogeneous platforms. For example, with VisionWorks (Nvidia implementation of OpenVX), it is possible to compile and run the same code on PC with Nvidia GPU, Jetson and DrivePX platforms without any software adaptation.

V. Results

Our methodology is applicable for any heterogeneous architectures. Nonetheless, in this paper, we focus on three SoCs: K1, X1, and R-Car H2. Software configurations for our tests are summarized in table II.

A. Computing

For each architecture, we present our test results for the ARM processor in 4 different configurations: 1 core, 1 core with handwritten SIMD instructions (NEON), 4 cores using OpenMP, and 4 cores with NEON; plus the results for the massively parallel processors (except for the R-Car H2 SoC due to a NDA).

In this paper, we only show results for *int32* addition and multiplication, and *float64* multiplication. Results are illustrated in Fig. 1. As predicted by the Boat Hull model [21], we observe a constant execution time for low I_A, and a linear execution time for high I_A. In this linear part, the slope can be directly exploited to obtain throughput for a given configuration (operation and variable type).

In addition, results also highlight some compiler effects: for some ARM configurations there is break in the line (I_A=5 for integer), this is due to auto-vectorization and loop-unrolling. Actually, after this threshold, compiler stops auto-vectorizing and loop-unrolling.

Some extracted throughputs are given in table III, compared to throughputs given by manufacturers, obtained values are

close to those provided by datasheets. The gaps between measurements and datasheets for some throughputs can be explained by our test design. In fact in our tests, the operation that we want to characterize is followed by some data transfer instructions. So the instruction pipeline is not fully exploited, but it is more representative of what happened in reality.

Then, in Fig. 2, we show results of concurrent executions on ARM relative to single threaded execution (Fig. 1): in one case we run two processes concurrently on different cores, in the other case we run two processes concurrently on the same core. Running two instances of the same test on different cores on the ARM seems to have no impact on the execution time. However, the execution time of two tasks executed on the same core is 2 times greater than single execution. About K1, the execution time on the same core is bigger than 2 times for $I_A \leq 1$, where performance is limited by memory bandwidth and latency. So we deduce it is due to memory/cache management. To strengthen the K1 model, we take into account this behavior when working with concurrent operations on different cores.

B. Memory

In this paper, we choose to focus on transfer (CPU to GPU, GPU to CPU), and memcopy operations. In Fig. 3, we give the delay needed to transfer different data size from one processor to another. Note that for all measures, data size allocated in memory is fixed to 4096 kB. Transfer delay is piecewise linear, non-linearities are due to cache effects. X1 transfer bandwidth seems to be about 1.7 times faster than K1 transfer bandwidths.

In Fig. 4, we give characterization of memcopy operations for K1, X1, and R-Car SoCs. Two consecutive operations on the same data do not have the same behavior, the second one being faster than the first one. Regarding GPU results, there is a static delay due to the latency of kernel execution on GPU; this delay depends on different parameters (e.g. size of kernel).

In Fig. 5, we show behaviors of concurrent memcopy operations on ARM using 2, 3 or 4 concurrent processes on different cores relative to single threaded execution (Fig. 4). In the 4 cores configuration, both X1 and K1 have a relative execution time lower than $\times 4$, whereas we operate 4 times more memory operations than single core test. It means that, in single core configuration, performance is limited by cache bandwidth, while in 2 or more cores configuration, performance is limited by memory bandwidth. According to these results, K1 memory bandwidth is about 1.33 times greater than cache bandwidth, and X1 memory bandwidth is about 2 times greater than cache bandwidth.

C. OpenVX / VisionWorks

One of the high level test vectors we choose to analyze in this paper is a feature tracker algorithm. Its VisionWorks implementation is composed of 4 kernels: color to gray conversion, Gaussian pyramid construction, Lukas Kanade optical flow, and Harris feature tracker. We focus on color to gray conversion and Harris feature tracker kernels for three different architectures: K1, X1 and a PC equipped with a GTX 970M Nvidia GPU.

In Fig.6, we give execution time of the two kernels versus the number of Mpixels to process, for K1, X1, and PC architectures. Fig.7 shows histograms of execution times for K1. As far as we know, VisionWorks mainly uses GPU for image processing kernels. We observe that execution time increases linearly with image size, with an offset due to kernel launch latency, like what we obtained with memory low level test vectors on GPU (Fig. 4).

As seen in the results for computing test vectors (Fig. 1), GPU needs a very high I_A to be computing bound, so we assume performance is limited by memory bandwidth. Thus, for architecture a, and kernel k, execution time is given by:

$$t_{k,a} = \alpha_a \cdot \gamma_{k,a} \cdot x + \beta_{k,a}. \quad (2)$$

Where $\beta_{k,a}$ denotes the kernel launch latency, α_a the memory bandwidth (in μs/Byte) obtained with our low level tests, x the number of pixels, and $\gamma_{k,a}$ the number of memory access for each pixels. By deriving the previous equation, we obtain:

$$\frac{dt_{k,a}}{dx} / \alpha_a = \gamma_{k,a}. \quad (3)$$

As the source code is identical for each architectures (only the VisionWorks library binary is specific for each architecture), each implementation should have the same amount of memory access, so γ_k should be constant. With measured $\frac{dt_{k,a}}{dx}$, and α_a obtained from our low level tests, we have for:

- Color kernel: $\gamma_{1,PC} = 3.2$; $\gamma_{1,X1} = 8.6$; $\gamma_{1,K1} = 8.7$.
- Track kernel: $\gamma_{2,PC} = 5.3$; $\gamma_{2,X1} = 15.3$; $\gamma_{2,K1} = 15.9$.

For K1 and X1, γ_k are close enough to consider them as similar. Thus, a known execution time on K1 can be directly extrapolated on X1 and vice versa. However, γ_k for PC is not equal but roughly proportional for different kernels. This can be explained by different memory access strategies, specific compiler optimizations, etc.

VI. Conclusion and Future Work

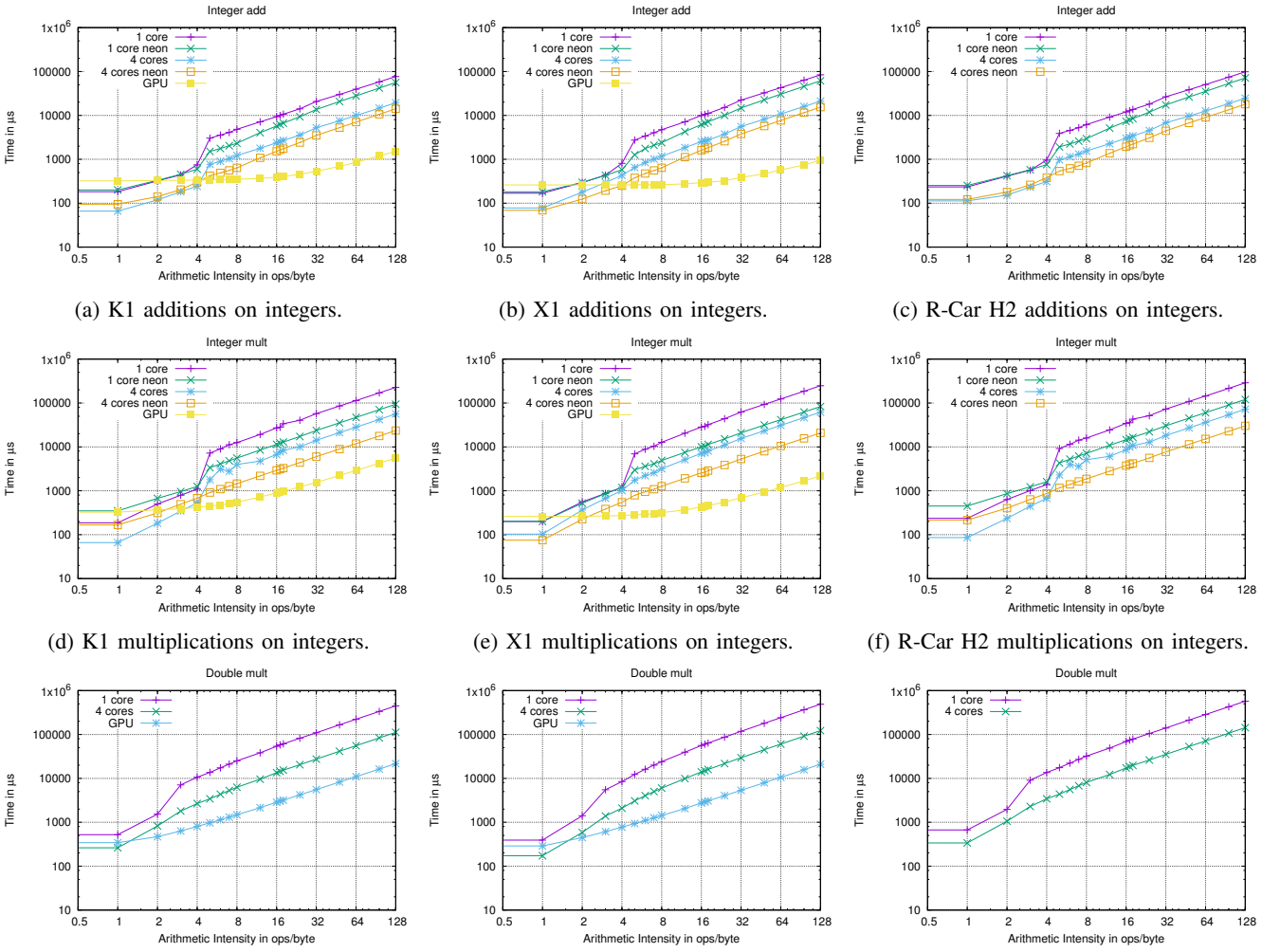
In this paper, we have introduced a novel methodology for characterizing computing architectures. Our test vectors are based on 3 levels. Low level vectors extract some key features of the tested architecture, mid level vectors study behaviors in concurrency conditions, and high level vectors analyze performances of real applications.

We are currently working on applying our methodology for other architectures, like TDA2x, R-Car H3, or the hardware accelerator of the R-Car H2 (IMP-X4). We are also studying behaviors of other "black box" kernels, in order to validate our approach with the high level test vectors (e.g. OpenVX / VisionWorks) and performance prediction.

The characteristics extracted with our approach can be directly used to strengthen a model of a computing architecture, in order to be applied for performance prediction. Actually, our main goal is to define a global methodology to help in finding the best suited SoC for a given application, and defining load balancing strategies for heterogeneous architectures, by using performance prediction.

TABLE III: Computing throughputs extracted from our test vectors compared to theoretical values. Throughputs are given in operation per cycle for ARM-A15 1 core, the GPU of the K1, and the GPU of the X1. The "$*$" denotes multiple instructions.

Throughput→ Operation↓	**A15 (datasheet)**	**A15 (measured)**	**GPU K1 (datasheet)**	**GPU K1 (measured)**	**GPU X1 (datasheet)**	**GPU X1 (measured)**
int8 add	n.a.	1	n.a.	124	n.a.	200
int16 add	n.a.	1	n.a.	124	n.a.	200
int32 add	2	2	160	128	256	225
float32 add	0.25	0.22	192	128	256	225
float64 add	0.25	0.22	8	8	8	8
int8 mult	n.a.	0.22	n.a.	30	n.a.	60
int16 mult	n.a.	0.22	n.a.	30	n.a.	60
int32 mult	0.5	0.4	32	32	$*$	84
float32 mult	0.2	0.18	192	128	256	225
float64 mult	0.2	0.18	8	8	8	8

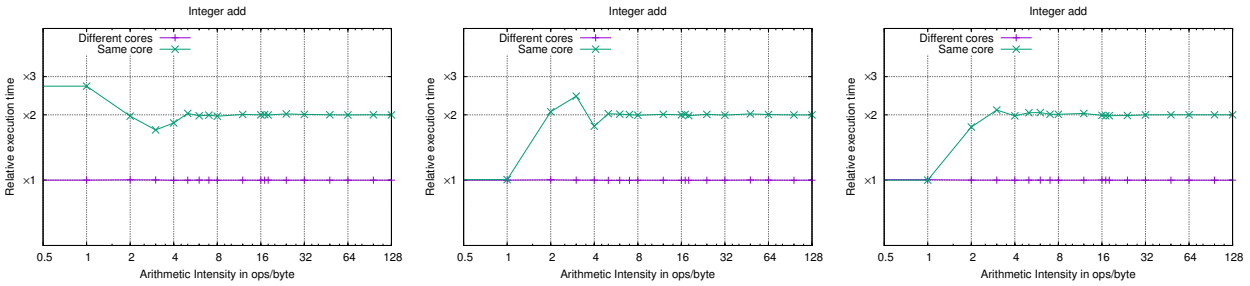


(a) K1 additions on integers. (b) X1 additions on integers. (c) R-Car H2 additions on integers.

(d) K1 multiplications on integers. (e) X1 multiplications on integers. (f) R-Car H2 multiplications on integers.

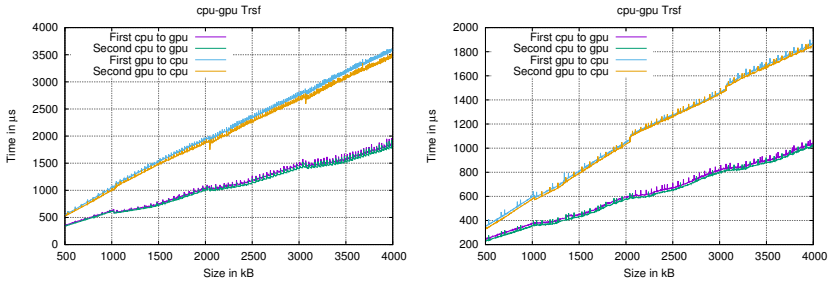
(g) K1 multiplications on doubles. (h) X1 multiplications on doubles. (i) R-Car H2 multiplications on doubles.

Fig. 1: Computing characterization examples, given for K1, X1 and R-Car H2 SoCs. We only show here a small sample of our results (integer and double variables). Note that we managed to have the same amount of memory access for all variable types.



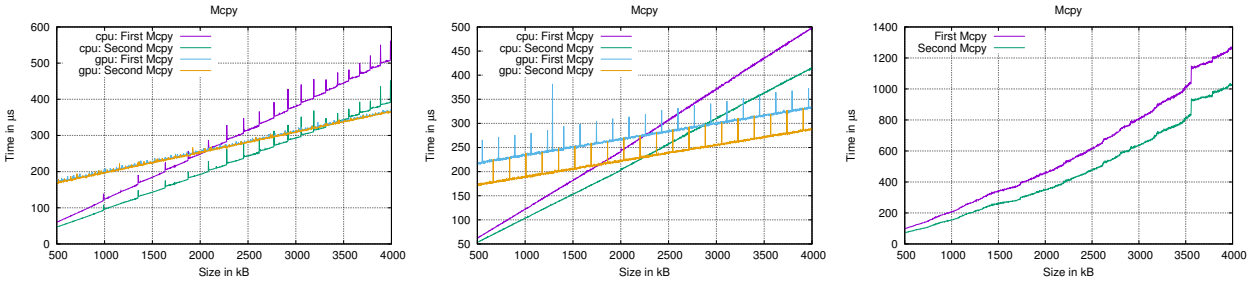
(a) K1 concurrent additions on integers. (b) X1 concurrent additions on integers. (c) R-Car concurrent additions on integers.

Fig. 2: Concurrent operations characterization on ARM, given for K1, X1 and R-Car H2 SoCs. Even if two ARM cores share L2 cache, running two tests concurrently on different cores seems to have no impact on the execution time. However, if the tests are run on the same core, execution time is about 2 times greater.



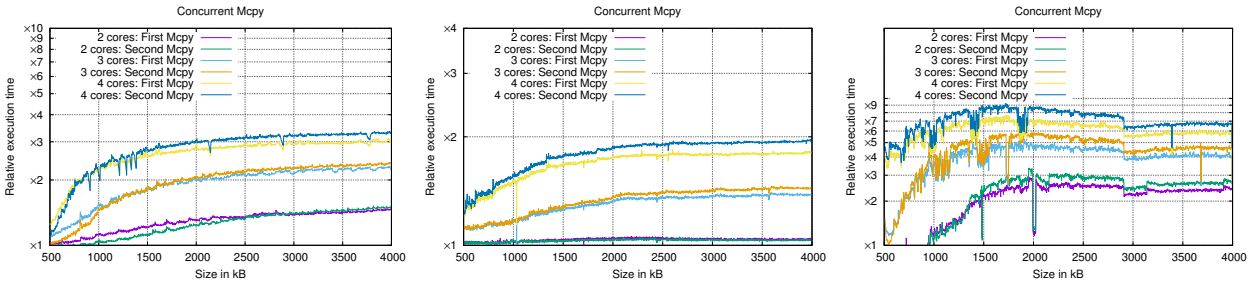
(a) K1 Transfer operation. (b) X1 Transfer operation.

Fig. 3: Transfer CPU-GPU characterization, given for K1 and X1 SoCs. Allocated size is fixed to 4096 kB for all measures.



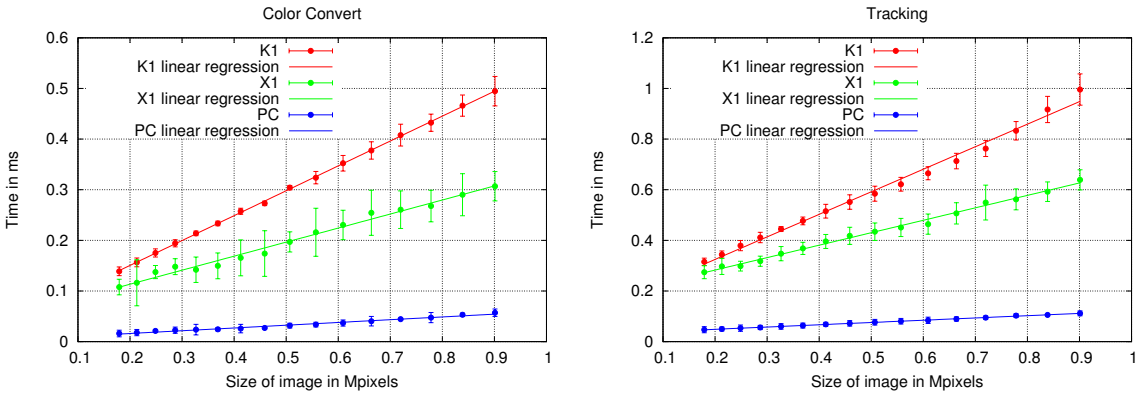
(a) K1 memcpy operation. (b) X1 memcpy operation. (c) R-Car memcpy operation.

Fig. 4: Memcpy characterization, given for K1, X1 and R-Car H2 SoCs. Allocated size is fixed to 4096 kB for all measures.



(a) K1 concurrent memcpy. (b) X1 concurrent memcpy. (c) R-Car concurrent memcpy.

Fig. 5: Concurrent memcpy operations on ARM, using 2, 3 or 4 cores. Allocated size is fixed to 4096 kB for all measures.



(a) Color conversion kernel.

(b) Harris feature tracker kernel.

Fig. 6: Results of OpenVX/VisionWorks tests for two kernels, given for three different architectures. Execution time is proportional to the number of pixels, with a static offset due to kernel launch latency on GPU.

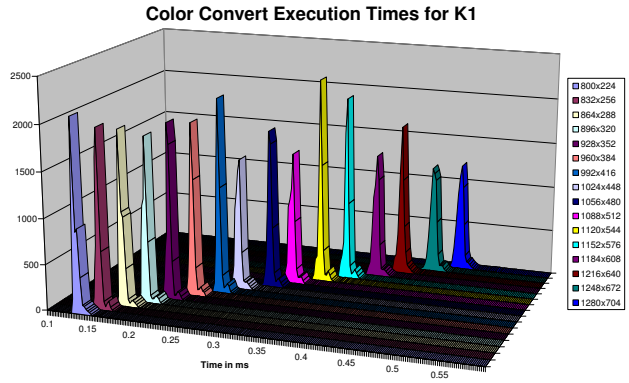


Fig. 7: Histogram of execution times of Color Conversion kernel (on K1) for different video stream resolutions.

References

[1] R. Saussard, B. Bouzid, M. Vasiliu, and R. Reynaud, "The embeddability of lane detection algorithms on heterogeneous architectures," in *2015 IEEE International Conference on Image Processing (ICIP)*. IEEE, 2015, pp. 4694–4697.

[2] R. Saussard, B. Bouzid, M. Vasiliu, and R. Reynaud, "Towards an automatic prediction of image processing algorithms performances on embedded heterogeneous architectures," in *2015 44th International Conference on Parallel Processing Workshops (ICPPW)*. IEEE, 2015, pp. 27–36.

[3] R. Saussard, B. Bouzid, M. Vasiliu, and R. Reynaud, "Optimal performance prediction of ADAS algorithms on embedded parallel architectures," in *2015 IEEE 17th International Conference on High Performance Computing and Communications (HPCC)*. IEEE, 2015, pp. 213–218.

[4] D. Castaño-Díez, D. Moser, A. Schoenegger, S. Pruggnaller, and A. S. Frangakis, "Performance evaluation of image processing algorithms on the gpu," *Journal of structural biology*, vol. 164, no. 1, pp. 153–160, 2008.

[5] NVIDIA, "Cuda C programming guide," 2015.

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

[7] K. Chitnis, R. Staszewski, and G. Agarwal, "TI vision SDK, optimized vision libraries for ADAS systems," Web ressource. http://www.ti.com/lit/wp/spry260/spry260.pdf, Texas Instrument, Tech. Rep., 2014.

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

[9] R. P. Weicker, "Dhrystone: a synthetic systems programming benchmark," *Communications of the ACM*, vol. 27, no. 10, pp. 1013–1030, 1984.

[10] "The Standard Performance Evaluation Corporation (SPEC)," Web ressource. http://www.spec.org.

[11] J. L. Henning, "Spec cpu2006 benchmark descriptions," *ACM SIGARCH Computer Architecture News*, vol. 34, no. 4, pp. 1–17, 2006.

[12] "Embedded Microprocessor Benchmark Consortium (EMBC)," Web ressource. http://www.eembc.org.

[13] S. Gal-on and M. Levy, "Exploring coremark a benchmark maximizing simplicity and efficacy," Web ressource. http://www.eembc.org/techlit/coremark-whitepaper.pdf.

[14] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta, "The splash-2 programs: Characterization and methodological considerations," in *ACM SIGARCH Computer Architecture News*, vol. 23, no. 2. ACM, 1995, pp. 24–36.

[15] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *2009 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 2009, pp. 44–54.

[16] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams *et al.*, "The landscape of parallel computing research: A view from Berkeley," UCB/EECS-2006-183, EECS Department, University of California, Berkeley, Tech. Rep., 2006.

[17] J. A. Stratton, C. Rodrigues, I.-J. Sung, N. Obeid, L.-W. Chang, N. Anssari, G. D. Liu, and W.-M. W. Hwu, "Parboil: A revised benchmark suite for scientific and commercial throughput computing," *Center for Reliable and High-Performance Computing*, 2012.

[18] A. Danalis, G. Marin, C. McCurdy, J. S. Meredith, P. C. Roth, K. Spafford, V. Tipparaju, and J. S. Vetter, "The scalable heterogeneous computing (shoc) benchmark suite," in *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*. ACM, 2010, pp. 63–74.

[19] J. D. McCalpin, "Memory bandwidth and machine balance in current high performance computers," *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter*, pp. 19–25, 1995.

[20] P. Mucci, "Llcbench-low level architectural characterization benchmark suite," Web ressource. http://icl.cs.utk.edu/projects/llcbench, 2009.

[21] C. Nugteren and H. Corporaal, "The boat hull model: adapting the roofline model to enable performance prediction for parallel computing," in *ACM Sigplan Notices*, vol. 47, no. 8. ACM, 2012, pp. 291–292.