A Framework for Optimizing CPU-iGPU Communication on Embedded Platforms

Francesco Lumpp
Dept. Computer Science
Univ. of Verona, Italy

Hiren D. Patel
Dept. Electrical and Computer Engineering
Univ. of Waterloo, Canada

Nicola Bombieri
Dept. Computer Science
Univ. of Verona, Italy

***Abstract*—Many modern programmable embedded devices contain CPUs and a GPU that share the same system memory on a single die. Such a unified memory architecture allows the explicit data copying between CPU and integrated GPU (iGPU) to be eliminated with the benefit of significantly improving performance and energy savings. However, to enable such a "zero-copy" communication model, many devices either implement intricate cache coherence protocols or they may disable the last level caches. This often leads to strong performance degradation of cache-dependent applications, for which CPU-iGPU data transfer based on standard copy remains the best solution. This paper presents a framework based on a performance model, a set of micro-benchmarks, and a novel zero-copy communication pattern to accurately estimate the potential speedup a CPU-iGPU application may have by considering different communication models (i.e., standard copy, unified memory, or pinned "zero-copy"). It shows how the framework can be combined with standard profiler information to efficiently drive the application tuning for a given programmable embedded device.**

***Index Terms*—CPU-GPU communication, edge computing, performance tuning, I/O cache coherence**

I. Introduction

Unlike traditional CPU-GPU communication models, which require copying data from the CPU memory to the GPU memory and back, *zero-copy* (ZC) allows CPU and GPU to access concurrently to a shared memory space. Communication based on ZC consists of passing data through pointers to the *pinned* shared address space. When CPU and the iGPU physically share the memory space, their communication does not rely on the PCIe bus and communication through ZC can be even more efficient [1], [2].

Such a communication model is instrumental for performance and energy efficiency in many real-time applications. Examples are camera- or sensor-based applications, in which the CPU offloads streams of data to the GPU for high-performance and high-efficiency processing [1], [2].

On the other hand, zero-copy through shared address space requires the system to guarantee cache coherency across the memory hierarchy of the two processing elements. Since the overhead involved by SW coherency protocols applied to CPU-iGPU devices may elude the benefit of the zero-copy communication, many systems address the problem by disabling the last level caches (LLC) (see Fig. 1.a) [3].

Although zero-copy best applies to many SW applications (e.g., applications in which CPU is the data producer and the GPU is the consumer), it often leads to strong performance degradation when the applications make intensive use of the GPU cache (i.e., *cache-dependent applications*). To reduce such a limitation, more recent embedded devices include hardware-implemented I/O coherence, by which the iGPU

This work has been supported by the project Dipartimenti di Eccellenza 2018-2022 funded by the Italian Ministry of Education, Univ. and Research (MIUR).

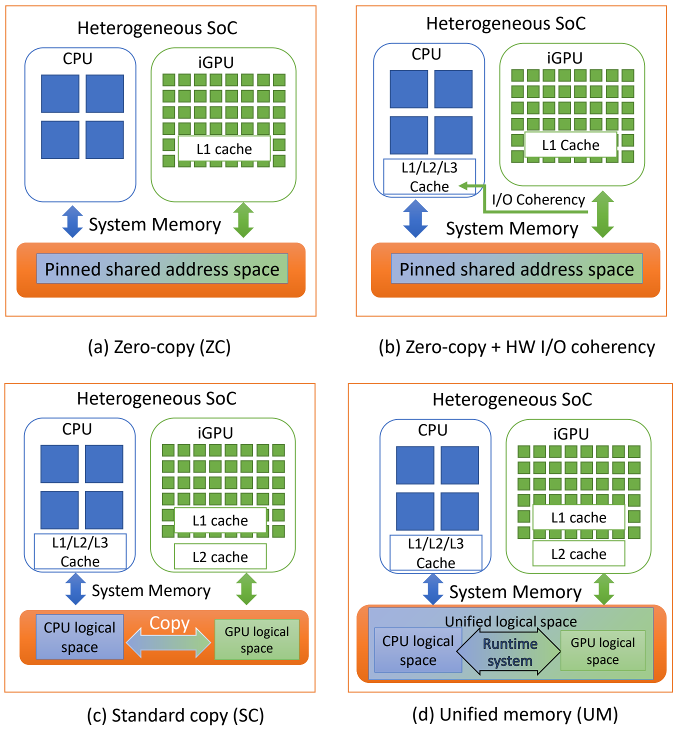

Fig. 1: CPU-iGPU communication models.

directly accesses the CPU cache, while the GPU cache is still disabled (see Fig. 1.b). Even though these solutions limit the performance loss, traditional communication models based on data copy, which we call *standard copy* (SC), are often the best solution with cache-dependent SW applications. With SC between CPU and iGPU, the physically shared memory space is partitioned into different logical spaces and the CPU copies the data from its own partitions to the iGPU partitions (see Fig. 1.c). The caches, which are all enabled, hide the data copy overhead, and cache coherence is guaranteed implicitly by flushing the caches before and after each GPU kernel invocation.

To ease the CPU-GPU programming and avoid explicit data transfer invocations, user-friendly solutions allow the programmer to implement CPU-iGPU communication through data pointers. In this communication model, which we call *unified memory* (UM), the physically shared memory is still partitioned into CPU and GPU logic spaces although they are abstracted and used by the programmer as a virtually unified logic space. The runtime system maintains cache coherence through on-demand page migration (see Fig. 1.d) [4].

Choosing the most efficient CPU-iGPU communication model amongst SC, UM, and ZC depends on both the SW application characteristics (i.e., compute vs. memory bound, usage of caches, etc.) as well as the characteristics of the target embedded device. Indeed, the overhead introduced by the cache coherence driver or the advantages provided by any I/O coherence implemented in hardware may affect the overall performance.

In this paper, we present a framework that, through the use of a set of micro-benchmarks and of a performance model, analyses the characteristics of the SW application and the target device to provide the potential performance the system can reach by changing the communication model. Switching from one communication model to another is often a time consuming and error prone task. Even more challenging is the switching from the models in which CPU routines and iGPU kernels are implicitly synchronized (i.e., SC and UM) to the model in which synchronization and task overlapping is the responsibility of the programmer. This framework (which is available for download at `github.com/FrancescoL96/Cache-Benchmark`) endeavours to support the programmer during the development and tuning of CPU-iGPU applications by proposing the most suited communication model and, in case of zero-copy, a communication pattern to best exploit this communication model's potential for improved performance.

This work's main contributions are the key elements of the framework as summarized next.

- A *set of micro-benchmarks* to characterize the CPU-iGPU communication performance of the device. The micro-benchmarks mix different amounts of computation and memory accesses on both the CPU and the iGPU to measure the performance impact of each communication model on the given embedded device.
- *A performance model* that combines the information provided by the micro-benchmarks and by any standard profiling tool to extrapolate the potential speedup the system performance can reach by switching from one model to another.
- A *zero-copy communication pattern* to enhance the system performance by taking advantage of a synchronized and overlapped execution of CPU and iGPU tasks.

The paper presents the results obtained by applying the framework to optimize the performance of different real world edge computing applications for different NVIDIA Jetson devices (i.e., Nano, TX2, AGX Xavier).

The paper is organized as follows. Section II presents the related work. Section III presents the the overall framework, the micro-benchmarks, the performance model, and the zero-copy communication pattern. Section IV presents the results, while Section V is devoted to the concluding remarks.

II. Background and Related Work

Cache coherency for GPU accelerators has been investigated in many research works. In [5], the authors propose a push-based, coherence mechanism that explicitly exploits the CPU and GPU's producer-consumer relationship by automatically moving data from CPU to GPU's last-level cache. In [6], the authors propose a cache coherence protocol designed for forward-looking multi-GPU systems. HALCONE [7] is a timestamp-based coherence protocol for multi-GPU systems. It replaces the compute unit level logical time counters with cache level logical time counters to reduce coherence traffic. In [8], the authors propose selective caching, by which they disallow GPU caching of any memory that would require coherence updates to propagate between CPU and GPU. A survey of additional techniques for managing and leveraging caches in GPUs proposed more in the past is presented in [9].

In contrast to the abovementioned works that propose cache coherency protocols for CPU-iGPU or multi-node GPUs, we propose a framework to accurately estimate the potential speedup a CPU-iGPU application may have on a given device by considering different communication models.

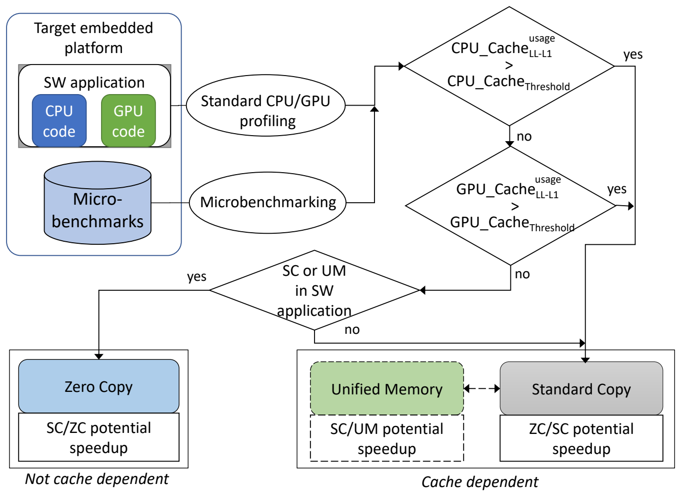

Fig. 2: Overview of the proposed framework.

A performance model for tuning GPU applications has been proposed in [10]. The model relies on a suite of micro-benchmarks to extrapolate the characteristics of specific GPU device components (e.g., arithmetic instruction units, memories, etc.) in terms of throughput, power, and energy consumption. GPUPerfML [11] combines decision trees and theoretical analytical models to locate performance bottlenecks of GPU applications and guide the application optimization. A comprehensive review of previous works addressing performance models for GPUs is presented in [12]. All the analysed contributions focus on GPU computation and memory access patterns over different platforms.

Unlike these prior works that target the tuning of GPU applications (i.e., kernels), the framework proposed in this work targets the tuning of CPU-iGPU communication on physically shared memory that, to the best of our knowledge, has not been addresses in prior works.

III. The Framework

Fig. 2 shows the overview of the proposed framework. Given a SW application and a target embedded platform, a standard profiling tool is applied to extrapolate information on the usage levels of both CPU and GPU caches[1].

The idea is that if the application strongly benefits from both caches, then concurrent accesses of CPU and iGPU on the *pinned* shared memory space and thus the ZC communication model cannot provide the best overall performance. This is due to the fact that the cache coherence protocol, or the caches disabled for guaranteeing consistency with the zero-copy model, may elude the benefit of eliminating the data copy. In this case, SC or UM are the best solutions. In contrast, if the cache usage is low on the iGPU, the best model depends on the CPU cache usage. If the CPU cache usage is high, ZC could give the best performance if the device implements a sufficiently efficient cache coherence protocol (e.g., the hardware I/O coherence of the NVIDIA Jetson AGX Xavier). Otherwise, SC or UM are likely the best solutions.

In case the application makes low usage of both the CPU and iGPU caches (i.e., the caches do not affect the application performance), ZC could provide at least equivalent

[1]We consider the last level cache (LLC) for both CPU and iGPU as the caches involved in data coherency protocols for CPU-iGPU communication. We consider they are the only caches disabled with zero-copy.

performance w.r.t. SC and UM. In this case, ZC is generally preferred, as shown in the experimental results section, because it can guarantee lower energy consumption due to the saved data transfers for the copies. The performance model proposed in this work aims at characterizing such a *cache usage* of the application, and correlates this value with the potential performance of each communication model (see Section III-A). As a result, considering the application, the implemented communication model, and the target device, the framework allows the user to accurately estimate the potential speedup the application may have by changing the communication model.

It is important to note that the characteristics of the target device strongly affect the choice of the best communication model and the potential speedup achieved from one model to another. The micro-benchmarks (see Section III-B) aim at extrapolating the device characteristics and accurately estimating such a potential speedup.

Finally, ZC may provide even better performance if combined with an overlapped execution of CPU and iGPU tasks. The performance model allows estimating the maximum performance improvement of ZC with task overlapping. We propose an access pattern to implement such an overlapping while guaranteeing data consistency (see Section III-C).

A. Profiling and performance model

Given a SW application and an embedded device, we define the usage of the last level caches (LLC) of the CPU and iGPU as follows:

$$CPU\_Cache^{usage}_{LL\_L1}(\%) = miss\_rate\_\mathrm{L1}_{CPU} \times (1 - miss\_rate\_LL_{CPU}) \quad (1)$$

$$GPU\_Cache^{usage}_{LL\_L1}(\%) = \frac{t_n * t_{\mathrm{size}} * (1 - hit\_rate\_L1_{GPU})}{kernel\_runtime} \times \frac{1}{GPU\_Cache^{max\_throughput}_{LL\_L1}} \quad (2)$$

where t_n and t_{size} represent the number of memory transactions and their size, respectively. The definitions assume an architecture with L2 as LLC. It can be generalized for different memory architectures. The two equations represent, in percentage, the amount of data that is obtained from the LLC of the CPU and iGPU, respectively out of all the requested data from the CPU/iGPU multiprocessors. All the miss/hit rate information are provided by any standard profiling tool. The maximum throughput in eqn 2 is extrapolated by the micro-benchmarks (see Section III-B).

We define the potential speedup a SW application may have by changing the communication model as follows:

$$\mathrm{SC/ZC}_{speedup} = \frac{SC_{runtime}}{\frac{SC_{runtime} - copy\_time}{1 + (CPU_{time}/GPU_{time})}} \leq SC/ZC_{Max\_speedup} \quad (3)$$

$$\mathrm{ZC/SC}_{speedup} = \frac{ZC_{runtime}}{\frac{ZC_{runtime}}{1/[1 + (CPU_{time}/GPU_{time})]} + copy\_time} \leq ZC/SC_{Max\_speedup} \quad (4)$$

where $SC_{runtime}(ZC_{runtime})$ is the execution time of the SW application with the SC(ZC) communication model. $copy\_time$ is the time spent for CPU-iGPU data transfer, $CPU_{time}(GPU_{time})$ is the runtime of the only CPU task (GPU kernel).

$\mathtt{SC/ZC}_{Max\_speedup}$ and $\mathtt{ZC/SC}_{Max\_speedup}$ represent the maximum speedup that can be obtained by moving from one model to another on the given device. These values are independent from the application and are extrapolated by the micro-benchmarks. Eqn. 3 defines the potential speedup an application that has been classified as not cache-dependent (first checks of the framework flow) may have by replacing the originally implemented SC model with ZC. It takes into account the SC runtime from which the data copy time is removed and the potential overlap between CPU and iGPU computation (CPU_{time}/GPU_{time}).

Eqn. 4 defines the potential speedup an application classified as cache-dependent may have by switching from ZC to SC. It takes into account the overall time needed by SC to explicitly copy data. It also considers the serialization of the CPU and iGPU tasks since their overlapping is not allowed in SC. It is important to note that if an application is cache dependent and originally implemented with SC, the framework does not suggest any change to the communication model and any further potential speedup.

For the sake of space and without loss of generality, we consider the performance of UM similar to SC. In all our micro-benchmarks, the maximum difference between the two model performance ranges between $\pm 8\%$ in all the considered devices. The difference is strictly related to the driver implemented for the on-demand page migration. Compared to the difference between SC(UM) and ZC, we consider negligible, in this paper, the difference of performance between SC and UM. It is also important to note that the programming effort to switch between these two models (SC and UM) is minimum.

B. Micro-benchmarks

The micro-benchmark code is implemented with the aim of satisfying four main properties.

- *Stressing capability*. The micro-benchmarks apply extensive and heavy workloads to the memories and caches. This allows the framework to reach the steady state in which each functional component is fully work-loaded to measure the real (w.r.t. theoretical) peak performance while minimizing the side effects that can incur throughout the measurements.
- *Workload variability*. The communication model and the corresponding components are stressed with different workloads. This allows the framework to quantify the effect of moving from one model to another.
- *Selectivity*. The micro-benchmarks stress, as much as possible, only one target functional component at the time. This allows the framework to extrapolate the potential speedup obtained by moving from one communication model to another, considering that they can involve different functional components.
- *Portability*. The micro-benchmarks are implemented independently from any CPU/iGPU platform.

Since the compiler may optimize the code (e.g., code-block reordering, dead code elimination, etc.) and, by means of the consequent side effects, it may affect the target properties, the code has been checked and refined throughout the different steps of the compilation process.

The first micro-benchmark aims at finding the peak throughput of the GPU LL-L1 cache on the target device ($GPU\_Cache^{max\_throughput}_{LL\_L1}$). This value allows accurately classifying an application as GPU *cache dependent/not cache-dependent* (see eqn. 2). Then, in case the application is cache dependent and originally implemented with the ZC model, this value is used to estimate the potential speedup obtained by switching from ZC to SC ($\mathtt{ZC/SC}_{Max\_speedup}$ in equation

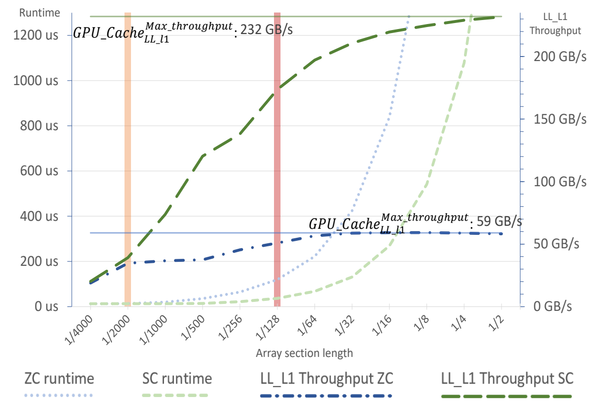

Fig. 3: Second micro-benchmark results on an NVIDIA Jetson Xavier. Relationship between LL_L1_throughput and kernel times of the iGPU.

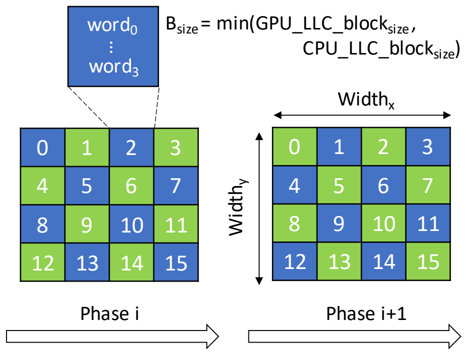

Fig. 4: Overview of the communication pattern for ZC.

4). It implements the elaboration of a matrix data structure computed by both CPU and GPU. In particular, the CPU performs a series of floating point operations with data read and written from and to a single memory address. These operations include square roots as well as divisions and multiplications. The GPU performs a 2D reduction multiple times through linear memory accesses. This is achieved through iterative loading of the operands (i.e., `ld.global` instructions), a sum (i.e., `add.s32`), and the result store (`st.global`). The two classes of operations (CPU and iGPU) allow the micro-benchmark to evaluate the peak usage of the CPU and iGPU caches. The two routines (CPU and GPU) are evaluated by considering the three communication models. ZC makes use of the concurrent execution of the routines and the concurrent access to the shared data structure. SC and UM explicitly exchange the data structure before the routine computation.

The second micro-benchmark implements extensive GPU computations, with varying levels of linear memory accesses. It aims at finding the GPU cache thresholds ($GPU\_Cache_{Threshold}$) used by the framework to suggest the best communication model between ZC and SC/UM (see Fig. 2). The micro-benchmark routine accesses sections of different length of a fixed-size array (e.g., from $1/4000$ to $1/2$), through a single `ld.global` and `st.global`, combined with a `fma.rn` (i.e., fused multiply-add) that uses two locally calculated values. It is implemented with both ZC and SC communication models and extrapolates the $GPU\_Cache_{Threshold}$ value by comparing the LL-L1 caches. Fig. 3 shows, for example, this micro-benchmark results for an NVIDIA Jetson Xavier, in which the threshold has been found at $1/2000$ accesses. A comparable throughput of the two models (from $1/4000$ to $1/2000$) translates into comparable system runtime with the two models (see light dotted lines in the figure). From $1/2000$ onward, the difference between the throughput values and the runtime linearly increases. $GPU\_Cache_{Threshold}$, in percentage, is calculated by considering the last comparable value of the throughput over the peak cache throughput ($GPU\_Cache_{LL\_L1}^{max\_throughput}$) provided by the first micro-benchmark (i.e., 20 GB/s and 59 GB/s, respectively, for the SC model in the example). The micro-benchmark extrapolates the $CPU\_Cache_{Threshold}$ in a similar way.

The third micro-benchmark performs a balanced CPU+iGPU computation through a routine whose performance are fully independent from the GPU cache. The GPU kernel implements repetitive memory accesses with sufficiently sparse single read access (`ld.global`) and single write access (`st.global`) in order to guarantee the maximum miss rate. It implements a concurrent access pattern (see Section III-C) and a perfect overlapping of the CPU and iGPU computations to extrapolate the maximum communication performance (and thus $SC/ZC_{Max\_speedup}$ and $ZC/SC_{Max\_speedup}$) the given embedded platform can provide by considering both SC and ZC.

C. Zero-copy communication pattern

With ZC, data copy is removed while CPU and iGPU access concurrently to the same pinned logical and physical space. Concurrent accesses by heterogeneous processors requires both data consistency and race conditions to be solved by the programmer. Even though explicit synchronization points would allow these issues to be easily addressed, the overhead involved by synchronization strongly affects the overall system performance. To avoid explicit synchronizations at every memory access while guaranteeing deterministic results, we propose a concurrent pattern based on tiling [13], which is accessed by CPU and iGPU through alternate producer-consumer phases.

Fig. 4 shows an overview of the communication pattern. An n-dimensional data structure, where n depends on the problem (2D matrix for images in the example of Fig. 4), is created and its size ($Width_x \times Width_y$) is calculated depending on the available GPU LL cache. The data structure is partitioned into data blocks (tiles) which size (B_{size}) corresponds to the smaller size between GPU and CPU LLC cache block size. This allows each access to a tile to be performed by a coalesced memory transaction.

CPU-iGPU communication relies on a pipelined sequence of access phases in which at phase i the CPU first reads and then writes onto the even blocks while the iGPU reads and writes on the odd blocks. At phase $i+1$, even and odd blocks are inverted for CPU and iGPU.

IV. Experimental results

We applied the proposed framework for the tuning of two different real cases of study: An application for the extraction of centroids in Shack–Hartmann wave front sensors [14] and an ORB-SLAM application for the simultaneous localization and mapping [15]. Both the applications have been tuned for three edge computing devices, i.e., NVIDIA Jetson Nano, TX2, and AGX Xavier.

A. Device micro-benchmarking

Fig. 5 shows the results of the first micro-benchmark on the Jetson TX2 and Xavier. ZC has side-by-side bars to show the overlapping execution. For the sake of space, the results on the Nano, which are equivalent to those of the TX2, have been omitted. The figure shows that the execution time of both the CPU routine and GPU kernel of the micro-benchmark with

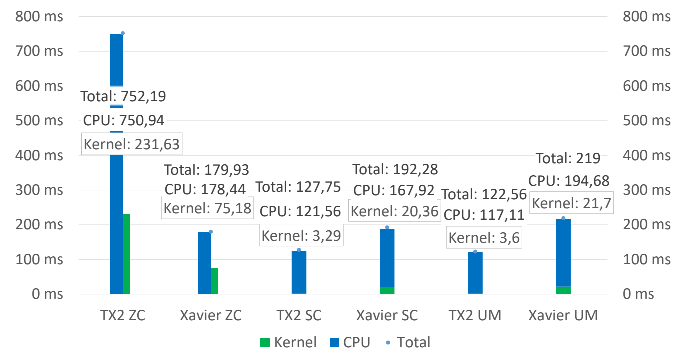

Fig. 5: First benchmark results: Execution times on the Jetson TX2 and Xavier with ZC, SC, and UM.

Board	$GPU\_Cache_{LL\_L1}^{max\_throughput}$		
	Zero Copy	Unified Memory	Standard Copy
TX2	1.28 GB/s	97.34 GB/s	104.15 GB/s
Xavier	32.29 GB/s	214.64 GB/s	231.14 GB/s

TABLE I: First benchmark results: Maximum throughput of the GPU cache on the Jetson TX2 and Xavier.

ZC are higher than those of SC or UM. This is due to the fact that the system disables the GPU cache when adopting the concurrent accesses of ZC.

With TX2, the performance difference is sensibly higher (up to 70%) since, differently from Xavier, TX2 disables also the CPU cache with ZC. The results shown in Table I ($GPU\_Cache_{LL\_L1}^{max\_throughput}$) confirm that the GPU memory accesses with ZC form an important bottleneck with a GPU throughput that is up to 77 times lower than the GPU throughput provided by SC and UM.

With Xavier, which implements I/O coherency and the CPU cache is always enabled, the difference between the GPU kernel performance with ZC and SC is "limited" to 3.7 times. Table I shows that the $GPU\_Cache_{LL\_L1}^{max\_throughput}$ of ZC in Xavier is still significantly worse than that in SC (or UM), even though the difference is sensibly reduced when compared to TX2 (i.e., 7 times lower in Xavier vs. 77 times lower in TX2).

In conclusion, when considering cache-dependent applications originally implemented with ZC, the ZC to SC switching can lead to a $\texttt{Max}_{ZC/SC\_speedup}$ equal to 70 in the TX2, while equal to 3.7 in Xavier. Since these values represent upperbounds, the micro-benchmark results underline that Xavier likely gives positive performance by adopting ZC *also* in many cache-dependent applications.

Figures 6 and 3 show the results of the second benchmark on the Jetson TX2 and Xavier, respectively. With TX2 (Fig. 6), from 1/16,000 to 1/8000 accesses, the GPU cache throughput between ZC and SC is comparable. This allows us to identify the GPU cache threshold (2.7%). Over 1/8000, the difference on throughput as well as on performance sensibly increases.

With Xavier (Fig. 3), we identified three zones (delimited by the vertical lines in the figure). In the first zone (left-most), ZC and SC provide the same performance. This allows us to identify the GPU cache threshold (16.2%) to switch to ZC. In the second zone (between the two vertical lines) the performance difference is below 200% (cache usage between 16.2% and 57.1%). In this case, the device may still provide equal or better performance by switching to ZC. In the third zone the performance difference sensibly increases over 200%, which suggests to adopt SC. This underlines that, when compared to SC, ZC can offer identical performance when there is limited cache usage, with linear performance degradation up

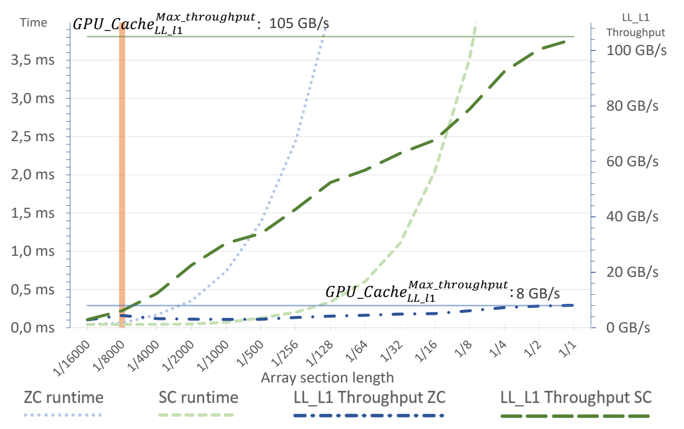

Fig. 6: Second benchmark results on the NVIDIA Jetson TX2.

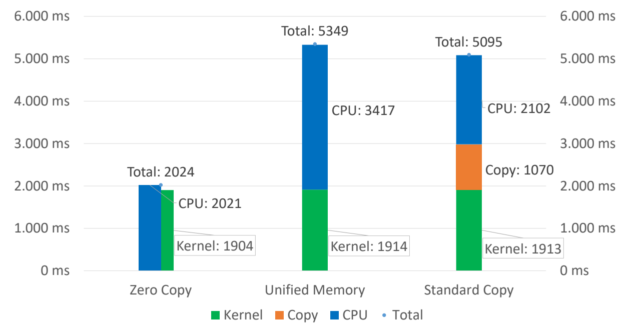

Fig. 7: Third benchmark results.

to a hard limit for bandwidth (i.e., 59 GB/s on Xavier). The closer we move towards the third zone, the higher must be the time the application gains with concurrent execution and task overlapping. After 57.1% of GPU cache usage, the GPU is severely bottlenecked and the recommendation is to not use ZC.

Fig. 7 shows the results of the third benchmark, which are used to extrapolate the $\texttt{SC/ZC}_{Max\_speedup}$. The runtime of the CPU and GPU tasks are comparable and the tasks can be fully overlapped. Due to the large data set used, 2^{27} floats (i.e., 512 MB), transfer times contribute significantly to the system performance. ZC is up to 164% faster than UM and up to 152% faster than SC.

Board	$CPU\ Cache_{LL\_L1}^{usage}$ (%)	CPU cache thresh. (%)	$GPU\ Cache_{LL\_L1}^{usage}$ (%)	GPU cache thresh. (%)	Kernel times (μs)	Copy time per kernel (μs)	SC/ZC *speedup* (up to, %)
Nano	19.8	15.6	1.7	2.5	453.5	44.8	–
TX2	19.8	15.6	3.7	2.7	175.2	22.4	–
Xavier	6.1	100	7.0	16.2-57.1	41.2	16.88	69.3

TABLE II: Profiling results of the SH-WFS application.

B. Tuning the Shack–Hartmann adaptive optics application

Adaptive optic algorithms measure and compensate optical aberrations when capturing images. We applied the proposed framework to tune an implementation of the adaptive optics with Shack-Hartmann sensors algorithm for edge computing [14]. Table II shows the profiling results, which quantify the dependency of the application performance on the CPU and GPU caches. In particular, the CPU cache usage of the application on Nano and TX2 exceed the threshold. This suggests that the application on these devices likely takes more advantage from the SC or UM communication models. With Xavier, the framework suggests to switch to ZC, with an estimated potential speedup up to 69%.

Board	SC time (CPU only)	SC kernel time	UM time (CPU only)	UM kernel time	UM speedup (vs SC)	UM kernel speedup (vs SC)	ZC time (CPU only)	ZC kernel time	SC/ZC *speedup* (actual vs SC)	ZC kernel speedup (vs SC)
Nano	1070.1μs (238.6μs)	453.54μs	1021.5μs (259.7μs)	454.92μs	5%	0%	1796.1μs (1120.7μs)	467.21μs	-67%	-3%
TX2	765.04μs (79.6μs)	175.18μs	783.67μs (217.2μs)	177.16μs	-2%	-1%	801.24μs (307.4μs)	244.17μs	-5%	-39%
Xavier	304.57μs (41.9μs)	41.24μs	305.80μs (88.8μs)	47.08μs	0%	-14%	220.15μs (45.4μs)	47.14μs	38%	-14%

TABLE III: SH-WFS centroid extraction algorithm performance results.

Board	$CPU\ Cache^{usage}_{LL\_L1}$ (%)	CPU cache thresh. (%)	$GPU\ Cache^{usage}_{LL\_L1}$ (%)	GPU cache thresh. (%)	Kernel times (μs)	Copy time per kernel (μs)	SC/ZC *speedup* (up to, %)
TX2	0	15.6	25.3	2.7	93.56	1.57	–
Xavier	0	100	20.1	16.2-57.1	24.22	1.35	5.9

TABLE IV: Profiling results of the ORB-SLAM application.

Board	SC time	SC kernel time	ZC time	ZC kernel time	SC/ZC *speedup* (actual)	ZC kernel speedup
TX2	70ms	93.56μs	521ms	824.20μs	-744%	-880%
Xavier	30ms	24.22μs	30ms	26.99μs	0%	-10%

TABLE V: ORB-SLAM performance results.

To evaluate the performance model, we implemented the application with the three communication models. Table III shows the results. As expected, the difference between SC and UM is negligible (below $\pm5\%$). Switching from SC to ZC on Nano and TX2 lead to a system performance degradation. A sensible loss of performance has been measured with Nano (-67%), which was expected as, in that board, the micro-benchmarks classified the application CPU cache dependent while not GPU cache dependent. A loss of performance has been measured and was expected on TX2, since both the CPU and GPU cache usage were behind the corresponding thresholds. With Xavier, we observed a performance improvement of the system equal to 38%. Thanks to ZC and the corresponding data transfer elimination, we measured, in average, 0.12J and 0.09J per second energy saving on Xavier and TX2, respectively, w.r.t. SC. We did not consider the energy saving on Nano as the performance loss is not negligible.

C. ORB-SLAM application

For the sake of space, we report the comparison between the ORB-SLAM application considering only SC and ZC. We do not report the result with the Nano device as it does not allow satisfying the real time constraints of the (heavy) application. Table IV shows the profiling results, which classify the application as GPU cache-dependant with both TX2 and Xavier. However, with Xavier, the profiling maps the application in the second zone of the GPU cache usage (see Section IV-A and Fig. 3).

Table V shows the application performance we obtained with the application implemented with both SC and ZC on TX2 and Xavier. As expected, ZC on TX2 strongly limits the application performance. On the other hand, Xavier provides the same performance by considering the application implemented with SC and ZC. In this case, the performance of the GPU kernel that slightly decreases (-10%) is fully compensated by the absence of data transfers and task overlapping. With a 30Hz camera as input sensor, we measured, on average, 0.17J per second energy saving on Xavier.

V. Conclusions and future work

This paper presented a framework to accurately estimate the potential speedup a CPU-iGPU application under tuning may have by switching from one communication model to another. The experimental analysis conducted on different real cases of study underlined that the characteristics of both application and target device strongly affect the choice of the best communication model, thus motivating the key role of the micro-benchmarks and of the performance model in the proposed decision framework.

References

[1] J. Fickenscher, S. Reinhart, F. Hannig, J. Teich, and M. Bouzouraa, "Convoy tracking for adas on embedded gpus," in *Proc. of IEEE Intelligent Vehicles Symposium*, 2017, pp. 959–965.

[2] X. Wang, K. Huang, and A. Knoll, "Performance optimisation of parallelized adas applications in fpga-gpu heterogeneous systems: A case study with lane detection," *IEEE Transactions on Intelligent Vehicles*, vol. 4, no. 4, pp. 519–531, 2019.

[3] N. Otterness, M. Yang, S. Rust, E. Park, J. Anderson, F. Smith, A. Berg, and S. Wang, "An evaluation of the nvidia tx1 for supporting real-time computer-vision workloads," in *Proc. of the IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS*, 2017, pp. 353–363.

[4] Nvidia Inc., "Nvidia tootlkit documentation, Unified Memory," https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#um-unified-memory-programming-hd.

[5] A. Yudha, R. Pulungan, H. Hoffmann, and Y. Solihin, "A simple cache coherence scheme for integrated CPU-GPU systems," in *Proc. of ACM/IEEE Design Automation Conference (DAC)*, 2020.

[6] X. Ren, D. Lustig, E. Bolotin, A. Jaleel, O. Villa, and D. Nellans, "Hmg: Extending cache coherence protocols across modern hierarchical multi-gpu systems," in *Proc. of IEEE International Symposium on High Performance Computer Architecture, HPCA 2020*, 2020, pp. 582–595.

[7] S. A. Mojumder, Y. Sun, L. Delshadtehrani, Y. Ma, T. Baruah, J. L. Abellán, J. Kim, D. R. Kaeli, and A. Joshi, "HALCONE : A hardware-level timestamp-based cache coherence scheme for multi-gpu systems," *CoRR*, vol. abs/2007.04292, 2020.

[8] N. Agarwal, D. Nellans, E. Ebrahimi, T. Wenisch, J. Danskin, and S. Keckler, "Selective gpu caches to eliminate cpu-gpu hw cache coherence," in *Proc. of International Symposium on High-Performance Computer Architecture*, 2016, pp. 494–506.

[9] S. Mittal, "A survey of techniques for managing and leveraging caches in gpus," *Journal of Circuits, Systems and Computers*, vol. 23, no. 8, 2014.

[10] N. Bombieri, F. Busato, and F. Fummi, "Power-aware performance tuning of gpu applications through microbenchmarking," in *Proc. of ACM/IEEE Design Automation Conference*, 2017.

[11] R. Zheng, Q. Hu, and H. Jin, "Gpuperfml: A performance analytical model based on decision tree for GPU architectures," in *Proc. of International Conference on High Performance Computing and Communications*, 2019, pp. 602–609.

[12] S. Madougou, A. Varbanescu, C. De Laat, and R. Van Nieuwpoort, "The landscape of GPGPU performance modeling tools," *Parallel Computing*, vol. 56, pp. 18–33, 2016.

[13] J. Cheng, M. Grossman, and T. McKercher, *Professional Cuda C Programming*. John Wiley & Sons, 2014.

[14] F. Kong, M. Polo, and A. Lambert, "Centroid estimation for a shack-hartmann wavefront sensor based on stream processing," *Applied optics*, vol. 56, no. 23, 2017.

[15] R. Mur-Artal and J. D. Tardós, "ORB-SLAM2: an open-source SLAM system for monocular, stereo and RGB-D cameras," *IEEE Transactions on Robotics*, vol. 33, no. 5, pp. 1255–1262, 2017.