IEIE Transactions on Smart Processing and Computing, vol. 5, no. 4, August 2016
http://dx.doi.org/10.5573/IEIESPC.2016.5.4.289

IEIE Transactions on Smart Processing and Computing

Efficient Process Network Implementation of Ray-Tracing Application on Heterogeneous Multi-Core Systems

Hyeonseok Jung and Hoeseok Yang

Department of Electrical and Computer Engineering, Ajou University / Suwon, Republic of Korea
{hyunsukdn, hyang}@ajou.ac.kr

* Corresponding Author: Hoeseok Yang

Received July 21, 2016; Accepted July 29, 2016; Published August 30, 2016

* *Short Paper*

***Abstract*:** As more mobile devices are equipped with multi-core CPUs and are required to execute many compute-intensive multimedia applications, it is important to optimize the systems, considering the underlying parallel hardware architecture. In this paper, we implement and optimize ray-tracing application tailored to a given mobile computing platform with multiple heterogeneous processing elements. In this paper, a lightweight ray-tracing application is specified and implemented in Kahn process network (KPN) model-of-computation, which is known to be suitable for the description of real-time applications. We take an open-source C/C++ implementation of ray-tracing and adapt it to KPN description in the Distributed Application Layer framework. Then, several possible configurations are evaluated in the target mobile computing platform (Exynos 5422), where eight heterogeneous ARM cores are integrated. We derive the optimal degree of parallelism and a suitable distribution of the replicated tasks tailored to the target architecture.

***Keywords*:** Ray tracing, Multi-core, Process network

1. Introduction

Multi-core CPUs are pervasively used in today's mobile embedded systems, such as smartphones or tablets. In such platforms, it is a key design challenge to distribute the workload over multiple, and possibly heterogeneous, processing elements in an efficient way that satisfies a given performance constraint. A state-of-the-art mobile computing platform (Exynos 5422 0, for instance) includes eight general purpose microprocessors of two types, and a general purpose graphics processing unit (GPGPU) with two compute devices, as shown in Fig. 1.

It is not trivial to program such heterogeneous multi-core systems due to their heterogeneity. That is, there are many possible cases to distribute computational workloads onto such many executable processing elements. Determining how to divide the workload and to assign it to specific processing elements is called *mapping*. Finding the optimal mapping is known to be NP-hard, even when the application's characteristics and the underlying architecture are known a priori.

Several programming frameworks have been proposed for specifying and mapping multimedia applications that can be run in parallel on multiple cores. Traditional multi-processor programming models like OpenMP [2] and MPI [3] were originally devised for cluster-level parallel computers, where multiple homogeneous CPUs collaborate in a single yet distributed system. Recently, Open Computing Language (OpenCL) [4] and the Compute Unified Device Architecture (CUDA) [5] have emerged as suitable programming models for heterogeneous multi-core systems. In particular, they target the graphics processing unit which, thanks to its inherently parallel structure, has been proven to offer better performance for data-parallel applications. However, none of the above-mentioned frameworks becomes mainstream in mobile embedded domains, as they are not suitable for guaranteeing predictable, possibly real-time, performance, which is crucial for multimedia systems.

In order to address this issue, multimedia applications are often described in a formal model-of-computation (MoC). Distributed Application Layer (DAL) [6] is a programming framework based on Kahn Process Network (KPN) [7] MoC, which is capable of offering predictability

when realized on multi-core systems [8]. In this paper, we take the KPN implementation of ray-tracing in DAL from [9] and study how to optimize the workload distribution and mapping decision tailored to a given heterogeneous multi-core platform.

The rest of this paper is organized as follows. In the next two sections, we first review the KPN implementation of the ray-tracing application and then its extension to an advanced process network model. Based on these, various mappings are evaluated for execution time in Section 4, targeting a state-of-the-art mobile computing platform, Exynos 5422. Finally, concluding remarks are drawn in Section 5.

2. Preliminaries

In this section, we first delineate the general Monte-Carlo ray-tracing algorithm [11], which is commonly used in multi-core realization. Then, we describe the target heterogeneous embedded multi-processor system-on-chip, Exynos 5422 0.

Ray-tracing is a computer graphics algorithm that renders extremely realistic images by tracing the path from an image plane to light sources. From each pixel of a two-dimensional image, a ray is projected to the scene (in three-dimensional space) and the color information is determined by simulating the effects of reflections and refractions from the objects that the ray encounters. It is known to be effective for extremely realistic images at the cost of huge computation requirements, as well as considerable energy consumption.

Generally, due to limited computation capability, it is intractable to track all the existing rays that contribute to the color value of each pixel. Thus, Monte-Carlo ray-tracing [11] methods (a common solution to overcome this issue) render a scene by randomly tracing a subset of samples of all possible light paths. As the sampling count grows, the pixel value will eventually converge to the correct solution. Of course, increasing the number of sampling rays may require more computing capability and energy. In mobile embedded systems, which we are targeting, it is crucial to strike a proper balance in this trade-off.

Monte-Carlo ray-tracing operates as follows: a set of ray samples that have slightly different injection angles are randomly chosen for each pixel. For each ray sample, the color information needs to be calculated. If a ray hits no object, it becomes black. Conversely, the color value is determined by a certain illumination algorithm with the material properties of the object and the angle of the ray, if one collides with an object (called *intersection*). If the surface of the object is reflective or translucent, more rays need to be injected into the scene to model the effects of reflection and refraction. Finally, the color information of a pixel is set to the average color of all sample rays. It is common for ray-tracing to be implemented in parallel programming, as it incorporates abundant parallelism by nature. That is, rays from different pixels can be handled simultaneously and independently by separate processing elements.

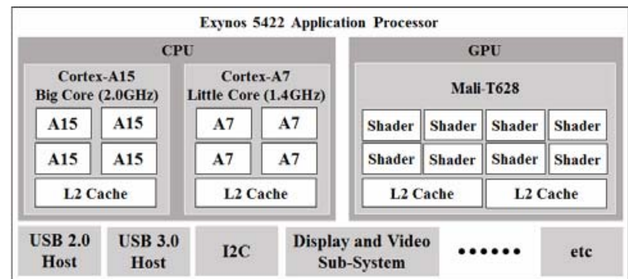

Fig. 1. Exynos 5422 application processor, a heterogeneous multi-core architecture.

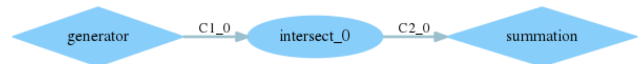

Fig. 2. Kahn Process Network implementation of the ray-tracing application.

Exynos series 0 (the target architecture in this paper) is a typical heterogeneous multi-core platform that incorporates multiple general purpose microprocessors (the ARM Cortex series) and embedded ARM Mali GPGPUs. Those processing elements are asymmetric in compute capabilities and power consumption behavior. In general, it is commonly used in mobile devices like smartphones, embedded devices, and so forth. Fig. 1 denotes the internal structure of Exynos 5422. The CPU is a heterogeneous mixture of four performant processors (Cortex A15) and four energy-efficient processors (Cortex A7), a so-called big.LITTLE architecture. On the GPU side, it incorporates the Mali T628 processor, which has eight shader cores inside. Note that we only utilize Cortex A15 and A7 processors.

3. Process Network Specification

The ray-tracing application is structured as KPN in [9], where three processes are concurrently running as shown in Fig. 2. The first process, *generator*, takes the role of ray vector generations. As stated in the previous section, it generates a fixed number of rays from each pixel to the scene. The second process, *intersect*, is the most computationally hungry one, which is the main rendering function of the ray-tracing application. Lastly, the calculated color values from all the corresponding rays for a pixel are summed up and stored in the corresponding position in an image file in the third process: *summation*.

Note that the first and third processes are associated with input/output operations. The *generator* process calculates all rays injected from a pixel and sends them over a first in, first out (FIFO) channel, *C1_0,* to the *intersect* process. Similarly, the rendering results of the second process are delivered through *C2_0* in a FIFO manner to the summation process. The internal of *intersect* process is implemented in a recursive call. That is, for each ray delivered by *generator*, path tracing takes place in *intersect* by invoking *Vec radiance(const Ray &r, int depth)*. Whenever a ray intersects an object, it recursively

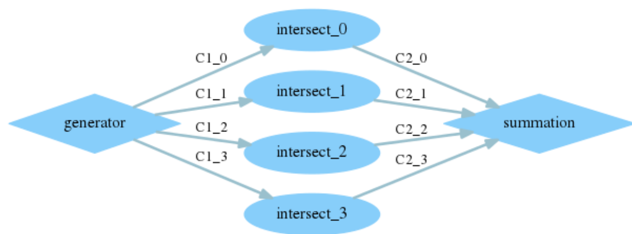

Fig. 3. Illustration of the replication of the *intersect* process in the EPN specification.

calls itself until the newly generated ray hits the background, or the maximum number of recursive calls has been reached. The *summation* process calculates the color information of a pixel, determined as the average of the number of received color values.

The basic KPN specification of the ray-tracing application described above can be modified or extended to have a *proper* degree of parallelism tailored to the underlying hardware architecture. In our specific application, several copies of the *intersect* process can be instantiated and parallelized over multiple cores to take advantage of multi-core platforms. The Expandable Process Networks (EPN) MoC [10] is a model extension to KPN that aims to increase the degree of parallelism by properly expanding the topology of the original graph by means of *replication* and *unfolding*. *Replication* simply replicates a process many times while automatically creating all necessary channels and ports equivalent to the ones that were present in the original KPN specification. On the other hand, *unfolding* is a hierarchical expansion of the process network. That is, a certain process of the original KPN is replaced with another KPN.

In this work, we restrict ourselves to the replication of the *intersect* process in the EPN extension, since the workload of the process has abundant pixel-wise or ray-wise data parallelism. While the hierarchical extension of the *intersect* process is also doable, it was reported as not comparable to the gain obtained from replication [9]. In replication of the *intersect* process, new channels and processes will be created automatically. Fig. 3 shows the modified EPN topology when the second process is replicated by four. It is worthwhile to mention that the modification of the graph topology is automated in the DAL [6] framework, and thus requires no modification of the user-specified code. In DAL, each vertex in KPN/EPN is synthesized as a POSIX thread, and inter-process communication is enabled by UNIX pipes. Thus, in the case illustrated in Fig. 3, six POSIX threads and eight pipes are generated.

4. Evaluations

In this section, we actually map the EPN processes on Exynos 5422 mobile computing platform and evaluate the performance. As a benchmark scene, we used the Cornell Box (http://www.kevinbeason.com/smallpt/) at a resolution of 100x100 pixels with 10 samples per pixel. The maximum recursion depth was set to 25. The number of sub-rows and sub-columns were both set to 2. The testing scene, as well as the source code of the ray-tracing application, is publicly accessible in the DAL[1] package.

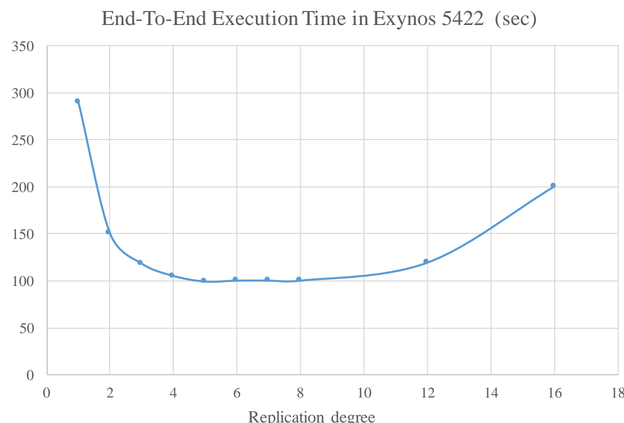

Fig. 4. Execution times using different numbers of *Intersect* processes in Exynos 5422.

4.1 Replication Degree

Thanks to the EPN extension, it is possible to choose an arbitrary number of replication degree in the *intersect* process in our original KPN specification. In this subsection, we will first analyze what degree of replication is optimal for a given hardware platform, and we subsequently analyze how the hardware architecture affects the performance of the parallelized implementation of the ray-tracing application.

We vary the replication degree, and measure the end-to-end execution time for rendering a whole image on Exynos 5422 platform running Linux 3.10.96 operating system. We only utilized the CPU (four A15's and four A7's, in the total eight cores), and the Linux default scheduler was used for the experiment. Fig. 4 depicts the execution times measured for the various replication degrees of the *intersect* process, from 1 to 16. As shown in the figure, when the default Linux scheduler is used, the optimal performance is achieved when the main process is replicated by eight, which is identical to the number of physical cores in the target platform. Beyond this optimal degree, we observed that larger degrees of parallelism result in worse performance, as can be seen in the replication degrees of 12 and 16. If too many process instances are operating, the overhead, including scheduling and inter-process communication, nullifies the benefit of the enhanced parallelism.

The optimal replication degree is highly dependent upon the underlying hardware platform. In order to confirm this, we conducted the same set of experiments on a different machine, a Linux workstation with two Intel Xeon E5-2620 processors, each of which integrates six physical cores. In total, 12 physical cores (24 logical cores powered by the Hyper-threading technique) exist on this platform. As shown in Fig. 5, the optimal performance is again achieved when the replication degree is the same as

[1] Download available here: http://www.dal.ethz.ch/

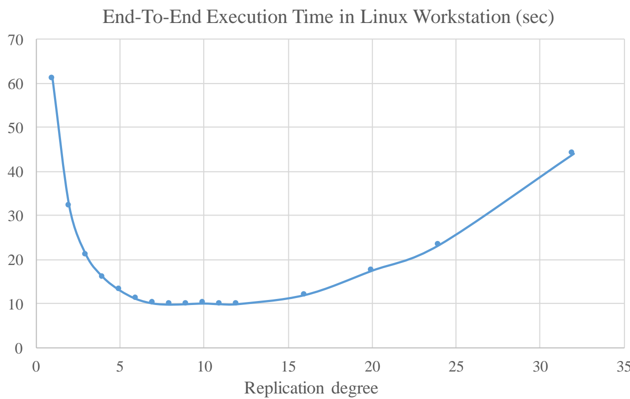

Fig. 5. Execution times using different numbers of *Intersect* processes in a Linux workstation.

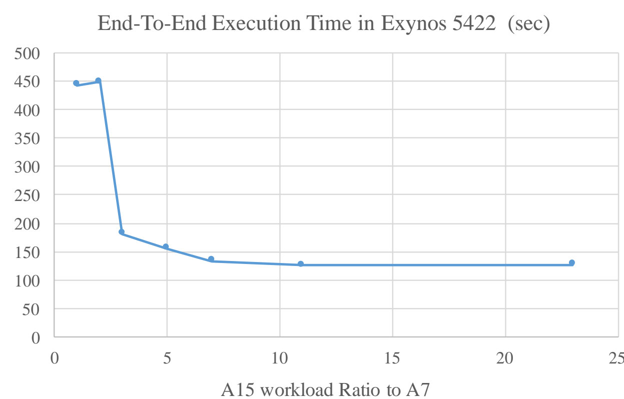

Fig. 6. Execution times of different workload ratios in Exynos 5422.

the number of physical cores: 12.

4.2 Workload Balancing

In the previous experiments, we only varied the degree of parallelism. In other words, the number of generated processes was determined by EPN, but where to execute them remained undecided by the model and was completely driven by the Linux scheduler. This gives us considerable room for optimization, since the workload and amount of computing capability in the constituent cores are known at design time. We fixed the replication degree for *intersect* to 24, then varied the workload ratio of A15 cores to A7 cores by setting the affinity of the generated *intersect* processes to certain cores. When the ratio is three, for instance, six instances of *intersect* are executed on the A7 cores, while the A15 cores are responsible for the remaining 18.

It was clearly observed that this judicious workload assignment complements the Linux scheduler in performance. With a replication degree of 24, the naïve Linux scheduler implementation took 363.98 sec, which is almost three times slower than the fastest one. The optimal performance was achieved when the workload ratio was 11:1.

5. Conclusion

In this paper, we specify the ray-tracing application in the KPN/EPN programming model and optimize it in an embedded heterogeneous multi-core system. Finding the optimal degree of parallelism, as well as the process assignment to cores, was shown to be highly dependent upon the underlying hardware architecture. The optimal degree of replication of the main ray-tracing tends to be identical to the number of physical cores in the computing platform. By balancing the workload, considering the compute capability of the cores, better performance could be achieved.

References

[1] Exynos, Samsung. "Octa 5422." (2015).

[2] Dagum, Leonardo, and Ramesh Menon, "OpenMP: an industry standard API for shared-memory programming." IEEE computational science and engineering 5.1 (1998): 46-55. Article (CrossRef Link)

[3] Gropp, William, et al. "A high-performance, portable implementation of the MPI message passing interface standard." Parallel computing 22.6 (1996): 789-828. Article (CrossRef Link)

[4] Munshi, Aaftab, "The opencl specification." 2009 IEEE Hot Chips 21 Symposium (HCS). IEEE, 2009. Article (CrossRef Link)

[5] Nvidia, C. U. D. A. "Compute unified device architecture programming guide." (2007).

[6] Schor, Lars, et al. "Euretile design flow: Dynamic and fault tolerant mapping of multiple applications onto many-tile systems." 2014 IEEE International Symposium on Parallel and Distributed Processing with Applications. IEEE, 2014. Article (CrossRef Link)

[7] Gilles, Kahn. "The semantics of a simple language for parallel programming." In Information Processing 74 (1974): 471-475.

[8] Schor, Lars, et al. "Scenario-based design flow for mapping streaming applications onto on-chip many-core systems." Proceedings of the 2012 international conference on Compilers, architectures and synthesis for embedded systems. ACM, 2012. Article (CrossRef Link)

[9] Sheikh, Suhel, "Efficient Process Network Specification of Ray-Tracing Application," Master Thesis, ETH Zurich, 2012.

[10] Schor, Lars, et al. "Expandable process networks to efficiently specify and explore task, data, and pipeline parallelism." Compilers, Architecture and Synthesis for Embedded Systems (CASES), 2013 International Conference on. IEEE, 2013. Article (CrossRef Link)

[11] Jensen, Henrik Wann, et al. "Monte Carlo ray tracing." ACM SIGGRAPH. 2003.

Hyeonsuk Jung is an undergraduate student in the department of Electrical and Computer Engineering at Ajou University. His research interests include the design, analysis, and optimization of embedded systems in heterogeneous MPSoCs.

Hoeseok Yang received the B.S. degree in computer science and engineering and the Ph.D. degree in electrical engineering and computer science from Seoul National University, Seoul, Korea, in 2003 and 2010, respectively. He is currently assistant professor at Ajou University, Korea. Before joining Ajou University, he was with D-ITET, ETH Zurich, Switzerland as a postdoctoral researcher. He received the best paper award at international conference on Compilers, Architectures and Synthesis of Embedded Systems (CASES) in 2012. His research interests include HW/SW codesign, design methodologies of MPSoC, and temperature-aware MPSoC design.

Copyrights © 2016 The Institute of Electronics and Information Engineers