Available online at www.sciencedirect.com

ScienceDirect

Electronic Notes in Theoretical Computer Science

Electronic Notes in Theoretical Computer Science 349 (2020) 103–118

www.elsevier.com/locate/entcs

APU Performance Evaluation for Accelerating Computationally Expensive Workloads

Ernesto Rivera-Alvarado[1,2] Francisco J. Torres-Rojas[3]

Computer Science
Costa Rica Institute of Technology
San Jose, Costa Rica

Abstract

APUs (Accelerated Processing Units) are widely available in personal computers as low-cost processors that have a CPU and an integrated GPU for displaying graphics, in the same die. Using a ray tracing algorithm as a computationally intensive workload and taking advantage of the APU specific characteristics, we compare the performance of this SoC against CPU and GPU solutions in the same price range.

Keywords: accelerated processing unit, computationally intensive workloads, ray tracing, CPU, GPU, acceleration.

1 Introduction

Through the years, computer systems have advanced in integration, performance, functionality, and applications. A clear example of this evolution are CPUs (Central Processing Unit) which are the most widely used resource for computers. Initially, CPUs were conceived to process general purpose tasks but now incorporate application-specific circuitry for functions like video processing, acceleration of vector operations, multimedia decoding, and hardware acceleration for scientific applications [16][26][29]. Nonetheless, CPUs show notorious performance limitations when processing big data sets or when the task at hand has a large number of computing operations [16] [26].

For instance, CPU performance is sub-optimal when rendering computer graphics – a highly parallel and computationally intensive task. This limitation led to the development of application-specific hardware to handle image generation in

[1] Thanks to our colleagues in the "*Happy Few*" research group of the Costa Rica Institute of Technology gave us keen feedback on previous versions of this work.

[2] Email: ernestoriv7@yahoo.com

[3] Email: torresrojas@gmail.com

https://doi.org/10.1016/j.entcs.2020.02.015

computers. Pixel color computation for an image can be performed in parallel (using rasterization) by several processors. This observation guided the design of the Graphics Processing Unit or GPU: a circuit that incorporates a lot of minimalist processors highly optimized for the mathematical operations needed for this specific task. The performance gain obtained in computer graphics with GPUs is massive, and, now, it is widely used in video games and multimedia applications [29].

Due to the limitations of CPU performance in several tasks, a lot of research has been targeted toward the GPU architecture to utilize it as a general purpose processor [20][26][27]. This research has improved the execution time of several computationally intensive workloads (*e.g.*, artificial intelligence, blockchain, bioinformatics, etc. [16]). Thus, the GPU has positioned itself as the default architecture in a significant number of demanding workloads due to its parallel processing capabilities and high optimization for mathematical operations [16][20].

However, GPUs also have several limiting factors that make its use not optimal for a wide variety of tasks. In particular, CPU-GPU communication uses a system bus known as PCI-Express that has a transfer rate lower than the one in CPU-RAM communication, which in itself is also sadly low [18][20].

PCI-Express memory transfer directly impacts the performance of applications that require a lot of communication between the CPU and GPU [36][20]. Another setback is that CPU RAM and GPU VRAM do not have a memory coherence protocol nor an efficient way to share data structures [19]. Also, GPUs are highly inefficient when executing code that is sequential, code that has a high number of branches, code that has recursion or code with unpredictable memory access patterns [18][16][29].

In 2011, with the introduction of novel specialized circuits in the CPU die, a new type of computer architecture known as APU (Accelerated Processing Unit) was born [20]. This setup has a CPU and a GPU that share system memory (RAM) in the same integrated circuit, with a coherence mechanism between their memories and the possibility of efficiently sharing data structures. The APU design brings up the opportunity to create specialized algorithms that make use of these specific characteristics [16][20]. Nonetheless, [20] point out that there is a lot of pending work in evaluating the performance of APUs in tasks where GPU performance is not practical due to constraints like the PCI-Express bus transfer rate, inability of sharing data structures between the CPU and GPU or when the GPU is executing code with a high number of branches [20]. Also, the same authors mention that the design and evaluation of tightly coupled algorithms, that can take advantage of the specific characteristics of the APU architecture to improve performance, are opportunities to be explored [20].

We explored the viability of using APU as an alternative to CPU or GPU architectures in computationally intensive workloads. A ray tracing algorithm was selected as the study case for this evaluation because:

- It is computationally and data-intensive [17].
- There is active research efforts aiming to improve performance [36][27].

- There are state of the art implementations available for CPU and GPU [27].
- It has application in several fields of study like computer graphics, physics simulation, robotic motion planning, among others [12].

These characteristics make the ray tracing algorithm an interesting case for studying its performance in cost and execution time across the CPU and GPU architectures and compare it with an implementation crafted for the APU. The possibility of improving the performance of the ray tracing algorithm in low-cost hardware, like the one that is present in personal computers and video game consoles could open up the opportunity of bringing the benefits of this technique for devices that were out of reach due to its high rendering time [9][10][11].

Section 2 shows the different efforts made to accelerate the ray tracing algorithm through approaches and the obtained results of this initiatives, while Section 3 describes the design and details of our implementation of an acceleration mechanism for the APU. The experiments executed for this research, including the experiment design, the hardware used, and the analysis method for the data obtained are detailed in Section 4. In turn, the results of these experiments are presented in Section 5 and discussed in Section 6. Finally, Section 7 summarizes all the conclusions of his paper and identifies avenues for future work.

2 Background

There have been several attempts to accelerate the ray tracing algorithm through GPUs; nonetheless, these implementations usually incorporate hybrid algorithms that combine rasterization for effects that otherwise would require a lot of computational power, ray tracing, and hardware solutions [32]. This kind of developments makes it challenging to evaluate the performance of a pure ray tracing implementation in GPUs. Also, [32] mention that the actual mechanism for rendering images through ray tracing in GPUs uses complex data structures that require maintenance and acceleration processes to achieve acceptable performance [32]. As far as we know, none of the research efforts have been made in the context of APUs. Instead, they usually utilize specialized high-cost hardware to achieve acceptable rendering times.

Other researchers have focused their work on implementations of ray tracing in limited resources GPUs, like the ones in mobile devices. The results show potential, but they have the disadvantage of not using widely available hardware, since they propose the development of specialized hardware in order to achieve a reasonable performance [22].

Additional investigations incorporate rasterization, ray tracing, and specialized hardware in a single design to accelerate the rendering time [21]. The goal of these developments is to achieve a good enough quality image in a fast manner through mobile GPUs, but with the disadvantage of not using standard hardware and that the use of rasterization provides images with less quality [10][17].

There have been efforts regarding the development of specialized hardware (ASIC) to render images through ray tracing [8]. Since the circuit is dedicated

just for ray tracing, it does not spend time in instruction fetch and instruction decode cycles like a typical CPU. On the other hand, this solution was just a prototype with limited functionality. Besides, it was developed on an FPGA due to the high cost of implementing the design on silicon by the time of the publication.

One of the primary reasons that make it difficult to use the GPU as the default platform for ray tracing is that RAM-VRAM transactions are frequent, and when compared with the CPU, GPU has a small cache, so more bandwidth is needed for memory transactions [27][15]. With that in mind, recent initiatives focus specifically on the acceleration of the ray/object intersection in ray tracing through GPU [36]. They propose a mechanism to compress the data structures of the pre-processed objects that are sent to the GPU memory, which reduces the required communication bandwidth between RAM and VRAM. A disadvantage of this approach is the additional time for data compressing added to each GPU VRAM transaction. Furthermore, this approach only works well with scenes that require a lot of memory transactions.

Another approach of using pre-processed data structures as an acceleration mechanism is the Bounding Volume Hierarchy (BVH). This method analyses each one of the elements in the scene and adds them to bounding boxes that progressively will be added inside bigger containers, creating a binary tree data structure. In this way, when a ray does not crosses a bounding box, all the objects in the sub-tree and in that container are discarded, saving a lot of processing time [27]. A high-performance gain can be obtained through this acceleration mechanism as it avoids the unnecessary calculation of several ray/object intersections [33]. That algorithm was initially conceived to run in CPUs, and it was ported later to GPUs by [6], however, a specific implementation for APUs is yet to be found.

There are developments that aim to position the GPU as a general purpose computing platform [31][10], or to find an optimized way to schedule processing to the GPU (not even the integrated one in the APU) [34]. Both are areas of study on their own, and different from our research.

3 Design

We focus on showing APUs as viable architectures for improving the performance of computationally intensive workloads, in particular ray tracing. This is pertinent because APUs are widely available and present in a lots of commodity hardware [14][20]. We are not just looking for a porting of an algorithm from the CPU to the GPU nor the optimization of an existent algorithm. A novel ray tracing algorithm for the APU that presents an insight into how the specific characteristics of this architecture can be used to improve the performance of computationally intensive workloads was designed. For instance, we depend heavily on the ability to share in an efficient manner data structures between the integrated GPU and the CPU because the memory is shared by both processors, which avoids the communication bottleneck of the PCI-Express bus. The amount of memory can be expanded to the same memory available to the operating system [16]. Furthermore, the tasks

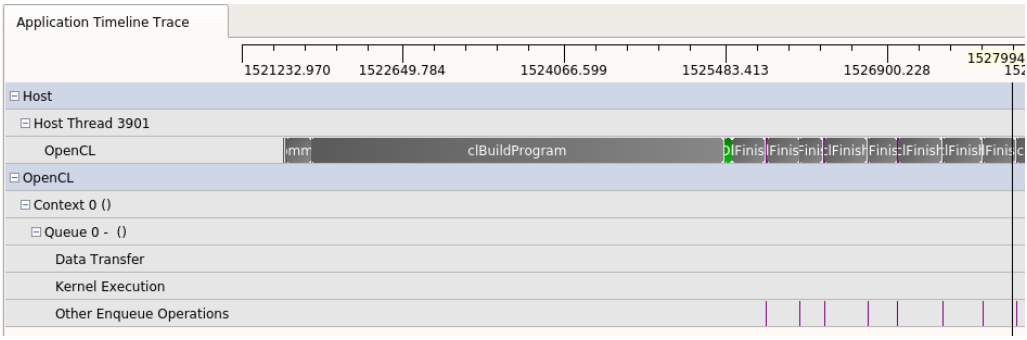

Fig. 1. CodeXL profiling result for the developed APU algorithm with zero data transfer.

for which the GPU is not efficient can be easily delegated to the CPU.

This research presents an initial step towards showing the capabilities of the APU for accelerating computationally expensive workloads, so it was crucial that the three architectures were running precisely the same algorithm (even if the algorithm was not particularly optimized for that platform) in order to show the behavior under the same circumstances and workload. Commercial or state of the art solutions like Nvidia OptiX [24][25] or Intel Embree [5] are out of the scope of this research as it will be difficult to make a fair comparison between the architectures as the solutions differ in programming languages and resources used. Also, those tools are vendor dependent and don't have a direct way to be deployed in AMD APU architecture [24][5].

The very same ray tracing algorithm was developed for CPU (using C) and GPU (using OpenCL). Even the data structures are the same. The implementations do not provide any optimization that favors any architecture. It is based on [27][28][30]. The GPU development followed the recommendations pointed out by [20] to improve the performance when processing a general workload. The code for the APU is exactly the same code as the CPU architecture for its internal CPU rendering, as well as the same code of the discrete GPU for its internal GPU. Thus, any optimization to these components, would be present in all the affected actors.

As pointed out by [20], one of the critical opportunities for the APU is the ability to efficiently share data structures between the CPU and the GPU. Memory traffic is one of the reasons why several algorithms do not perform well in GPUs due to the communication bottleneck that represents the PCI-Express bus, and memory latency [36][20]. The AMD APU architecture used in our experiments offers the ability to have a zero copy mechanism, which allows sharing data structures between the CPU and the integrated GPU in a efficient manner [4][2][3]. The tool `CodeXL` [1][13] was used to profile and check the behavior of the shared memory data structures between the CPU and the integrated GPU. As expected, it was verified that none of the data structures was copied (see **Fig. 1**).

The algorithm to accelerate ray tracing through the APU architecture is presented in **Fig. 2**. The image is rendered using all CPU and GPU cores simultaneously, while sharing the same data structure of the scene which is stored in RAM. Following the recommendations in [27], the image is divided into 16×16 pixel work-

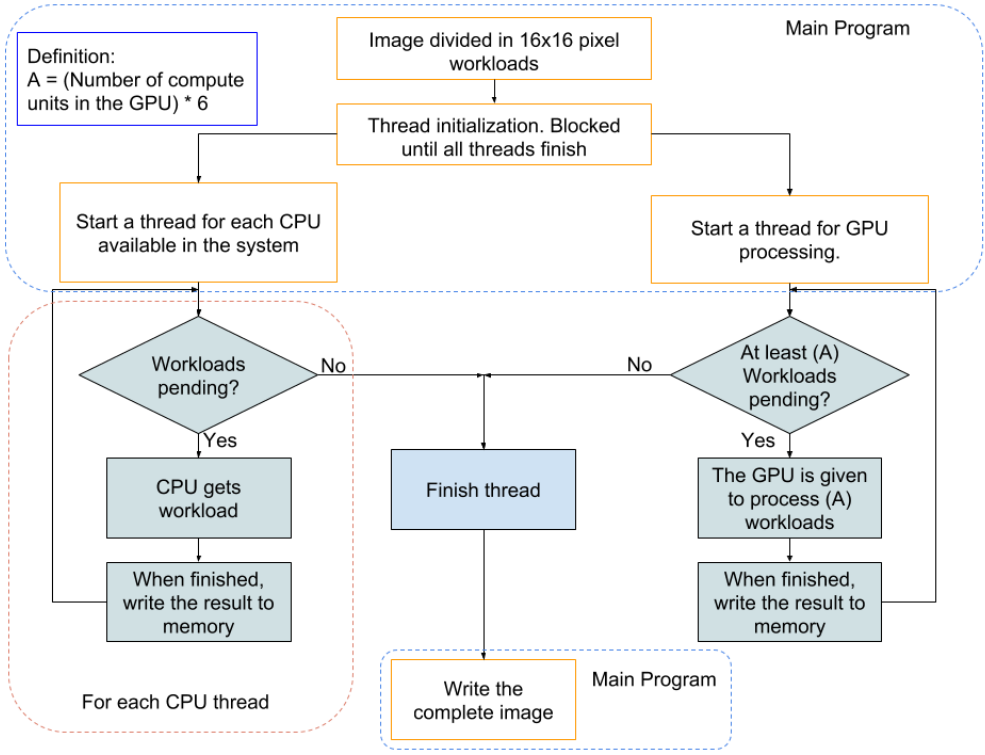

Fig. 2. APU ray tracing algorithm.

loads as the unit to be given to a processor to render. For each available CPU core in the system, a thread that checks if there is any available workload pending is spawned. If this is the case, the thread executes the workload, saves its result to memory and looks for the next workload. Each thread finishes if there is no more pending work. Since each of the work assignments to the GPU component of the APU implies an overhead, there should be at least enough work to maximize the GPU resources utilization, *i.e.*, all its processing units should have enough workload. AMD documentation recommends experimenting with different workload sizes to achieve optimal performance [4][20]. As a rule of thumb, we found that if there are N computing units in the GPU, a total of $6 \times N$ workloads assigned to the GPU did the trick.

The integrated GPU processes the image in parallel with the CPU cores. When the GPU finishes its work, it checks whether there are at least $6 \times n$ workloads available for it to process. If this no the case, the GPU thread finishes, and any remaining work is performed by the CPU cores. When all the threads are finished, the rendered image is copied from RAM to storage memory, and the process ends.

The proposed algorithm nicely fits the requirements for a ray tracing renderer. For other computationally intensive workloads, an in-depth analysis of the task needs to be performed to propose an acceleration mechanism that meets the specific characteristics of that workload.

4 Methodology

Factorial **Analysis of Variance** (ANOVA) experiments are designed to evaluate efficiently the effects that different factors under research might have on a response variable [23]. This methodology was chosen due to its capacity to evaluate the performance of the ray tracing algorithm across the APU, CPU and GPU architecture and several other factors that directly impact rendering time [27]. For practicality, the evaluation of the effects (anti-aliasing, reflections, transparencies) uses a 2^k form, which significantly reduced the number of executed experiments while preserving the influence of those effects in rendering time.

A factor is a component (that has different levels) of the experiment that might impact in some way the response variable [23]. The goal of ANOVA is to determine if that influence is statistically significant. The selected factors and levels for this experiment are:

- **Objects:** The number of objects present in a scene directly impacts rendering time [27]. The following amounts were used:
  - 1000.
  - 4000.
  - 16000.
  - 65000.
  - 260000.
- **Image resolution:** Image quality is dependent on the level of detail that it can contain [17]. A pixel is the smallest possible detail that is present in a digital picture, so it is directly related to the quality. This means that more pixels in an image (more resolution), the better quality it has. Three common computer resolutions were selected for the experiment:
  - 1280×720.
  - 1440×900.
  - 1920×1080.
- **Effects:** Visual effects directly impacts rendering time [27]. All the combinations of following three effects were considered:
  - (**AA**) Anti-aliasing.
  - (**RE**) Reflection (5 levels).
  - (**TR**) Transparency (5 levels).
- **Architecture:** This is the most crucial factor to test the the ability of the APU for accelerating computationally intensive workloads. As has been described in the document, the factors are:
  - APU.
  - CPU.
  - GPU.

There are $5 \times 3 \times 8 \times 3 = 360$ combinations, since it was decided to have 5 replications, we ended up with $360 \times 5 = 1800$ runs of the experiment. Several scripts were developed to automate and recollect the data from the experiments.

Table 1
Hardware description.

Characteristic	**CPU**	**GPU**	**APU**
Vendor	AMD	Nvidia	AMD
Model	Ryzen 2600	GeForce 1050Ti	Ryzen 2400G
Price($)	199	199	169
CPU Cores/Threads	6/12	-	4/8
Power Consumption(W)	65	75	65
CPU Cache L2/L3(MB)	3/16	-	2/4
CPU Frequency(GHz)	3.4 - 3.9	-	3.6 - 3.9
GPU Memory(GB)	-	4	Shared RAM
GPU Frequency(GHz)	-	1.29	1.25
GPU Cores	-	768	704
GPU GFLOPS	-	2138	1736

Two computers were used to obtain the data. One computer held the CPU and GPU architecture, and the other had the APU. Both machines ran with 8 GB RAM clocked at 2400 MHz single channel, a 256 GB SSD, and stock cooling (see **Table 1** for details)

The basic scene used for the experiments is constituted by the quantity and position of the objects. From there, the factors of resolution, effects, and architecture were adjusted to fit a specific combination. A complex ray-traced image has several objects distributed across all the scene [27]. This criterion was used to randomly distribute the objects in the x and y axis of the projection frame and the z axis of the scene. Different object sizes and shapes were used to simulate the unpredictability of the elements contained in a typical ray-traced scene.

Performance can be equated to response time [26]. Thus, the natural response variable for these experiments is the rendering time that the algorithm spends creating the image.

5 Results

Due to the ANOVA adequacy requirements, a square root transformation was applied to the data [23]. However, the charts of this section are presented with the de-transformed (*i.e.*, elevated to its square) data results. **Fig. 3** presents the ANOVA table for this experiment produced by `R` [35]. **Fig. 4** plots the average rendering time in function of the architecture (the most important factor for this research) in all the defined scenarios. **Figs. 5**, **6** and **7** disaggregates this information presenting the behavior of the architecture in relation to each one of the other

Anova Table (Type II tests)

Response: T_sqrt

	Sum Sq	Df	F value	Pr(>F)
Architecture	2980.9	2	2.8440e+05	< 2.2e-16 ***
Objects	21603.4	4	1.0306e+06	< 2.2e-16 ***
Resolution	463.2	2	4.4194e+04	< 2.2e-16 ***
Effects	12945.3	7	3.5288e+05	< 2.2e-16 ***
Architecture:Objects	2944.3	8	7.0228e+04	< 2.2e-16 ***
Architecture:Resolution	41.5	4	1.9806e+03	< 2.2e-16 ***
Objects:Resolution	151.0	8	3.6010e+03	< 2.2e-16 ***
Architecture:Effects	1284.6	14	1.7508e+04	< 2.2e-16 ***
Objects:Effects	5538.6	28	3.7745e+04	< 2.2e-16 ***
Resolution:Effects	111.5	14	1.5191e+03	< 2.2e-16 ***
Architecture:Objects:Resolution	13.2	16	1.5706e+02	< 2.2e-16 ***
Architecture:Objects:Effects	750.7	56	2.5580e+03	< 2.2e-16 ***
Architecture:Resolution:Effects	11.8	28	8.0599e+01	< 2.2e-16 ***
Objects:Resolution:Effects	57.4	56	1.9547e+02	< 2.2e-16 ***
Architecture:Objects:Resolution:Effects	7.4	112	1.2528e+01	< 2.2e-16 ***
Residuals	7.5	1440		

Signif. codes: 0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

Fig. 3. ANOVA Table.

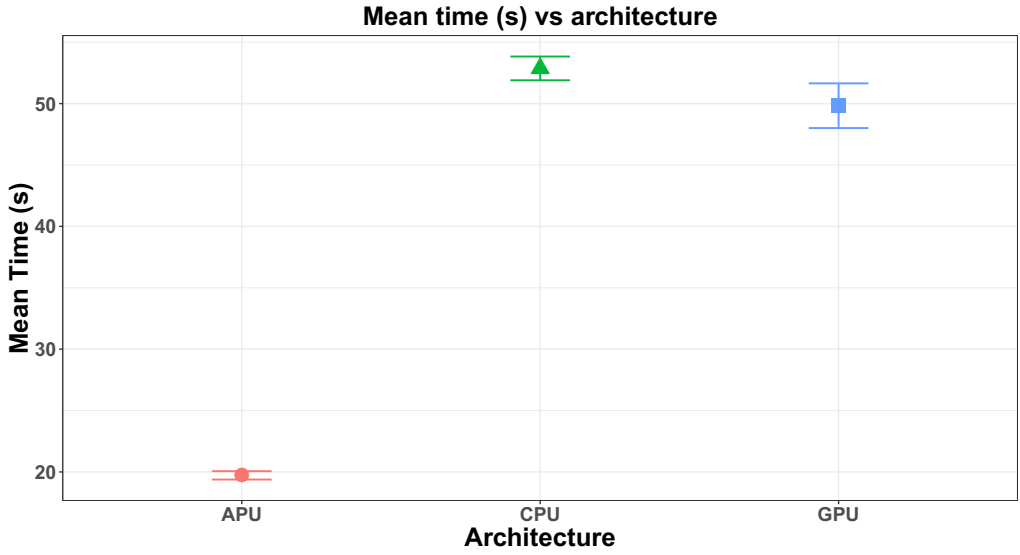

Fig. 4. Average rendering time in function of the architecture.

main factors.

Table 2 shows the average rendering time per architecture (APU, CPU, and GPU) for all the combinations of object quantities, effects and resolution levels. Performance/Pixel was calculated by averaging the number of pixels in the three resolutions used for the experiment. The average time to render an image was divided by the average number of pixels in an image.

Real-world rendering scenarios involve high resolutions, several effects in the scene as well as thousands to millions of objects [27]. We analyzed this case with a 1920×1080 resolution, 260000 objects, and anti-aliasing, reflection and transparency activated. The metrics obtained for this run are displayed in **Table 3**.

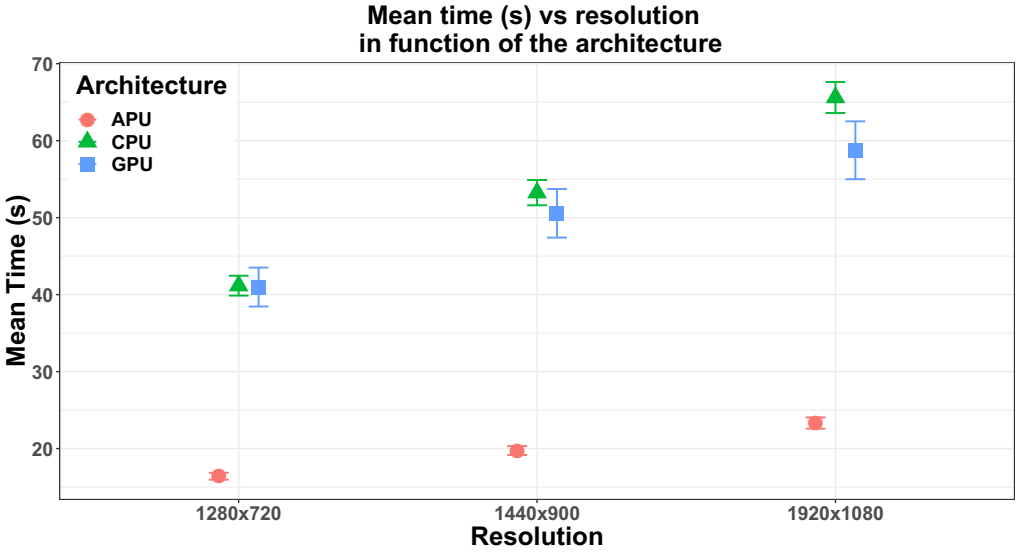

Fig. 5. Average rendering time for resolution vs architecture.

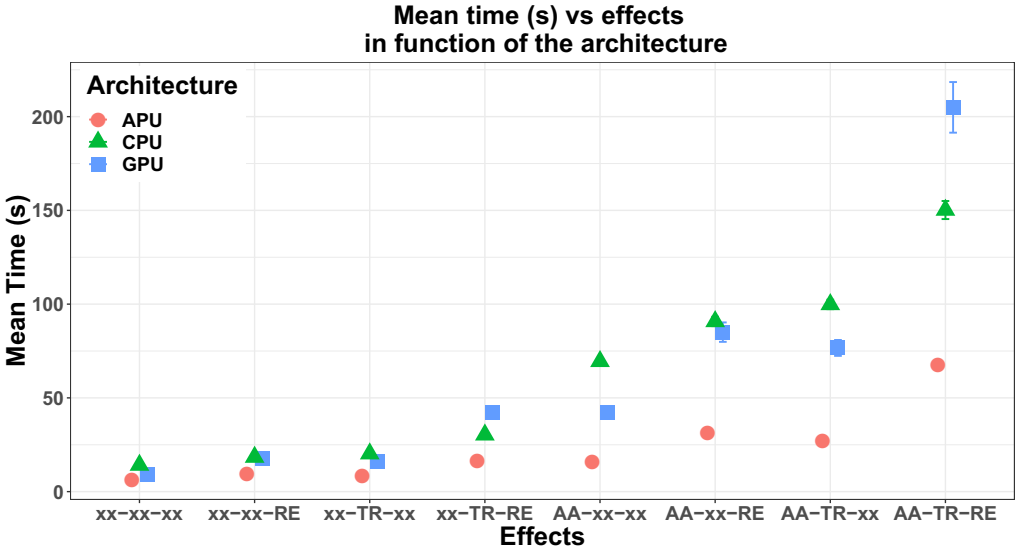

Fig. 6. Average rendering time for effects vs architecture.

Table 2
Obtained metrics for the APU, CPU and GPU architectures (lower is better).

Metric	APU	CPU	GPU
Average Time (s)	19.72	52.86	49.82
Performance/Pixel(μs)	4.59	12.31	11.60

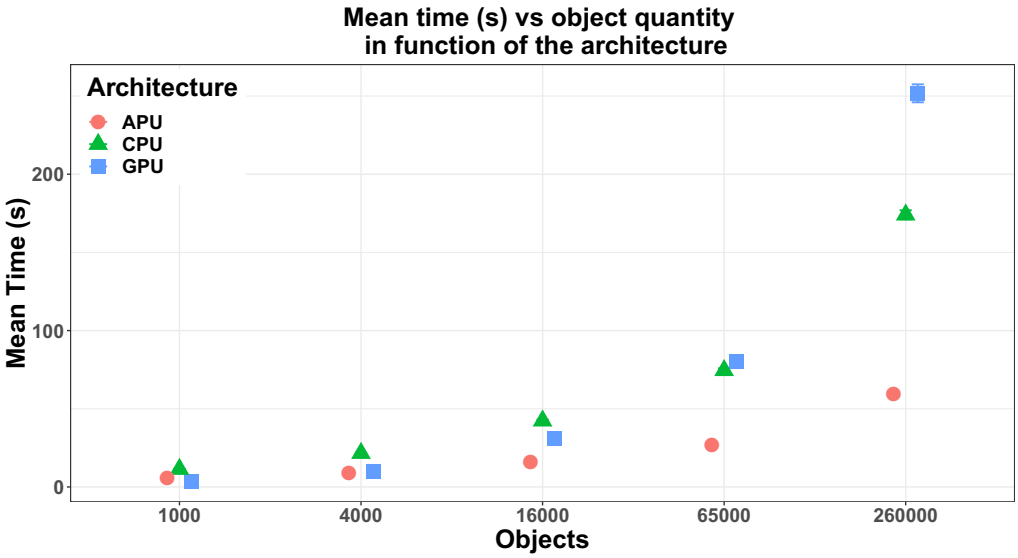

Fig. 7. Average rendering time for object quantity vs architecture.

Table 3
Realistic case metrics.

Metric	APU	CPU	GPU
Average Time (s)	307.02	633.98	1267.14
Performance/Pixel(μs)	148.06	305.70	611.08

The correctness of the proposed algorithm was tested by verifying that all three architectures render exactly the same image for a scene with the same resolution, object quantity, and effects. The `diff` command of UNIX did this nicely. No differences were found between the images generated between the APU, CPU and GPU architectures. **Figs.** 8, 9 and 10 shows the typical images generated by the experiments.

6 Discussion

The ANOVA table of **Fig.** 3 shows that all the main factors and their combinations have very low p values, which means that they are statistically significant. Thus, all they influence the response variable. **Fig.** 4 presents the average of all the interactions of object quantities, effects, and resolutions. A Welch's t pairwise test of this data indicated that the APU is statistically different from the other two architectures, being the one with the lowest rendering time and therefore the best overall performance. **Table** 2 shows the average rendering time per architecture (which are statistically different with a p below 0.05). This information is consistent with this result and indicates that APU is the best architecture for rendering animations as it delivers the best rendering time per pixel, when compared against the time of the CPU and GPU platforms.

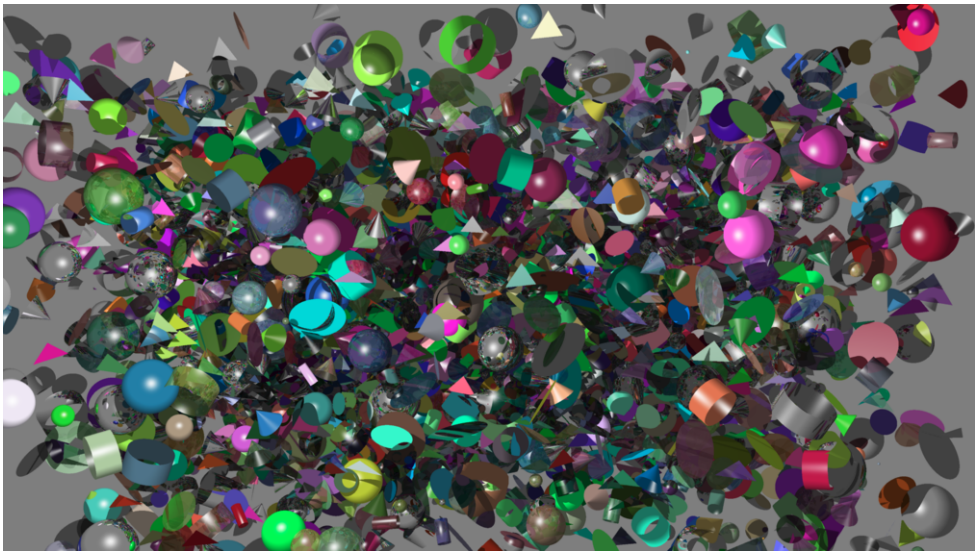

Fig. 8. 16000 objects

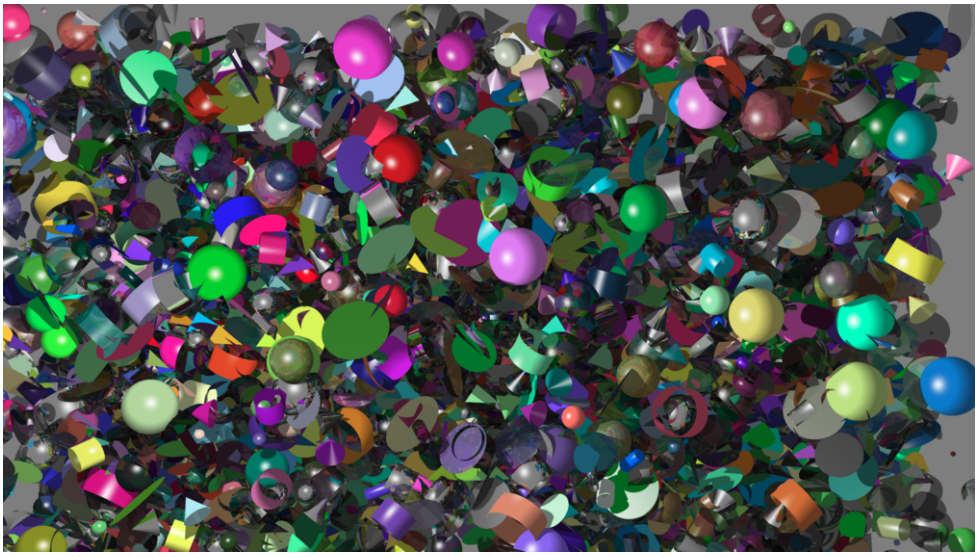

Fig. 9. 65000 objects

The impact in rendering time of resolution, object quantity, effects and their interactions is extensively explained in [30][28][27][7][17]. The results presented in **Fig. 3** and **Figs. 5**, **6**, **7** corroborate these observations. In this regard, ANOVA and Welch's t pairwise tests applied to our data established that:

- APU is statistically different and superior in performance to the other two architectures at any level of resolution. The resolution level impacts on how many pixels are processed in the image, and each pixel implies at least one ray (more if anti-aliasing is present) which in itself triggers intersection detection and effects calculation, through additional secondary rays. Hence, **Fig. 5** evidences that

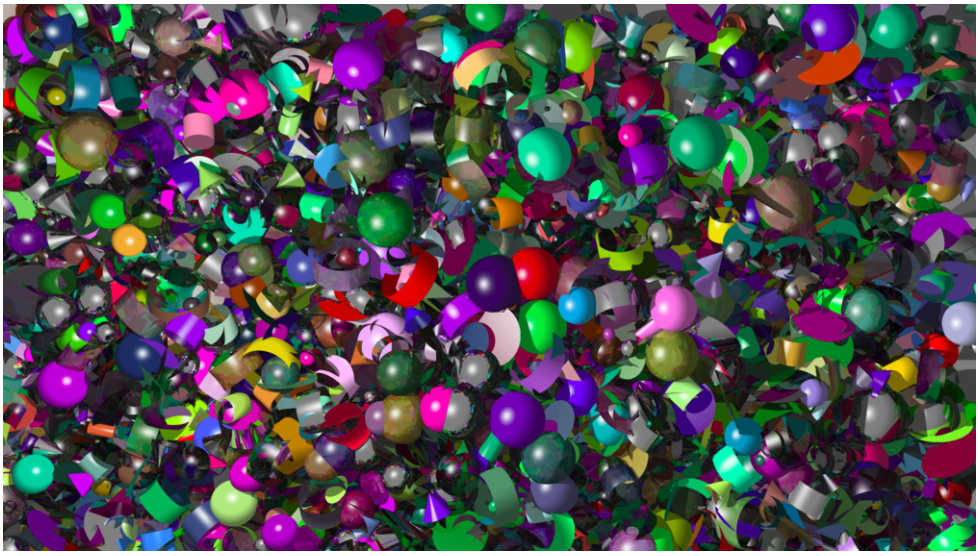

Fig. 10. 260000 objects

APU appears to be less sensitive to the increase of raw work trough resolution.

- The effects of anti-aliasing, reflections, and transparencies are achieved through the generation of more rays for the color calculation of one pixel, and as a consequence, more mathematical operations are performed and rendering time increases. APU is statistically different and superior in performance when the anti-aliasing effect is activated. Adding more effects when anti-aliasing is present enlarges the rendering time difference between architectures, and **Fig. 6** shows that the APU is the one with the best performance as it is less sensitive to this increment.
- In a ray tracing algorithm, intersections between each ray and objects in the scene must be calculated. Therefore, the more objects in the scene, the more intersections that need to be detected, which generates more memory accesses and mathematical operations. Object quantity makes APU rendering time statistically different and better than CPU or GPU, for 16000 or more objects (**Fig. 7**). This phenomenon can be explained by the ease of access that the APU architecture has to the RAM where objects are stored, while in a GPU implementation, as object quantity increases, transfers to the GPU via the PCI-Express bus also increase, negatively impacting the rendering time. Besides, ray tracing processing with GPUs is challenging because of compute units sub-utilization and inefficient memory access patterns due to the unpredictability of which the objects data structure needs to be accessed during collision detection [12].

Table 3 indicates that APU performance excels in average rendering time and performance per pixel in complex scenarios that are more closely to a real-world rendering problem (scenes with the higher resolution, object quantity, and all the effects present). This points out that the APU holds the performance advantage even in cases that require the most computation power, which provides an insight

that this architecture has the potential for accelerating general computationally intensive workloads. In both the general and most complex scenarios, the APU performance is more than 50% higher than the other computing platforms for the defined experiment (**Table 4**).

In order to render correct images like the ones illustrated in **Figs. 8, 9** and **10,** FP64 (floating point numbers with 64 bits) operations are required. If FP32 is used, the rendered image shows nasty artifacts. When calculating a pixel color through ray tracing, a significant amount of mathematical operations are performed to obtain the color to be displayed in the screen. If FP32 is used, its rounding error accumulates through the mathematical operations which cause the generation of incorrect pixel colors. These correct outputs and the metrics obtained from the analysis provides evidence that the proposed acceleration mechanism crafted specifically for the APU, positions this architecture as a viable option for accelerating computationally intensive workloads like ray tracing.

7 Conclusions and Future Work

We presented an alternative low-cost architecture for accelerating computationally intensive workloads. Our design utilizes all available computing resources in the APU (integrated GPU and CPU) to process a computationally intensive task, in this specific case, ray tracing. Our approach takes advantage of the particular characteristics of the APU architecture, for instance, its ability to share data structures from RAM and the ability to efficiently coordinate work within its internal processors.

Our experiments provided valuable information that shows the potential of the APU architecture as a viable alternative for accelerating computationally expensive workloads while being the most cost-efficient when compared against the CPU and GPU architectures. The obtained data and the realistic case metrics show that the APU performance advantage becomes more significant as the memory access and mathematical operations of the task increases.

Floating point operations resolution directly impacted the correctness of ray tracing. We found that FP64 is required to obtain a correct result. Thus, platforms with optimized FP64 operations will improve the performance of the selected task of this research.

For future work, additional ray tracing features and algorithms must be considered. New APU generations should be evaluated as they could change the results of this research. Another computationally intensive workloads besides ray tracing

Table 4
Performance improvement of the APU against CPU and GPU architectures

Scenario	**CPU**	**GPU**
General case (%)	62.69	65.69
Most complex scenario (%)	51.56	79.00

should be studied to measure whether the APU keeps its performance and cost-efficiency advantage in those tasks. As the APU uses the same code as their CPU and GPU counterparts, it would be interesting to explore the effects of extensive optimization of the rendering code in the three architectures. Also, an in-depth exploration of the PCI-Express and memory transfers in the discrete GPU solution could provide an interesting insight into ways to obtain more performance from the APU. Finally, as there are several techniques to minimize the FP32 rounding error [27], the experiments could be run focusing on FP32 operations.

References

[1] Advanced Micro Devices, "Getting Started with CodeXL," AMD, 2012.

[2] Advanced Micro Devices, "AMD Accelerated Parallel Processing, OpenCL Programming Guide," AMD, 2013.

[3] Advanced Micro Devices, "AMD APP SDK, OpenCL User Guide," AMD, 2015.

[4] Advanced Micro Devices, "OpenCL Optimization Guide," AMD, 2015.

[5] Áfra, A. T., I. Wald, C. Benthin and S. Woop, *Embree ray tracing kernels: Overview and new features*, in: *ACM SIGGRAPH 2016 Talks*, SIGGRAPH '16 (2016), pp. 52:1–52:2.

[6] Aila, T. and S. Laine, *Understanding the Efficiency of Ray Traversal on GPUs*, in: *Proceedings of the Conference on High Performance Graphics 2009*, HPG '09 (2009), pp. 145–149.

[7] Akenine-Möller, T., E. Haines and N. Hoffman, "Real-Time Rendering, Fourth Edition," A K Peters/CRC Press, 2018.

[8] Alvarado, M. J., D. Valderde, G. Randolph, Steinvorth and F. J. Torres-Rojas, *RTUCR: Hardware para Ray Tracing*, in: *Tiempo Compartido, Volumen 8, Número 1*, Tiempo Compartido (2008), pp. 6–13.

[9] Angel, E. and D. Shreiner, "Interactive Computer Graphics: A Top-Down Approach with WebGL (7th Edition)," Pearson, 2014, 7 edition.

[10] Bikker, J., "Ray Tracing in Real-Time Games," Ph.D. thesis, NHTV University of Applied Sciences, Reduitlaan 41, 4814DC, Breda, The Netherlands (2012).

[11] Bikker, J. and J. van Schijndel, *The Brigade Renderer: A Path Tracer for Real-Time Games*, International Journal of Computer Games Technology **2013** (2013).

[12] Chitalu, Floyd M. and Dubach, Christophe and Komura, Taku, *Bulk-synchronous Parallel Simultaneous BVH Traversal for Collision Detection on GPUs*, in: *Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, I3D '18 (2018), pp. 4:1–4:9.

[13] Fare, C., *Enabling Profiling For SYCL Applications*, in: *Proceedings of the International Workshop on OpenCL*, IWOCL '18 (2018), pp. 12:1–12:1.

[14] Gaster, B., L. Howes, D. R. Kaeli, P. Mistry and D. Schaa, "Heterogeneous Computing with OpenCL: Revised OpenCL 1.2 Edition," Morgan Kaufmann, 2012.

[15] Haines, E. and T. Akenine-Möller, "Ray Tracing Gems: High-Quality and Real-Time Rendering with DXR and Other APIs," Apress, 2019.

[16] Hennessy, J., "Computer Architecture : A Quantitative Approach," Morgan Kaufmann Publishers, an imprint of Elsevier, Cambridge, MA, 2018.

[17] Hughes, J. F., A. van Dam, M. McGuire, D. F. Sklar, J. D. Foley, S. K. Feiner and K. Akeley, "Computer Graphics: Principles and Practice (3rd Edition)," Addison-Wesley Professional, 2013, 3 edition.

[18] Intel Corporation, "OpenCL™ Developer Guide for Intel® Processor Graphics," Intel Corporation, 2015.

[19] Junkins, S., "The Compute Architecture of Intel ® Processor Graphics Gen9," Version 1.0, Intel Corporation, 2018.

[20] Kaeli, D. R., P. Mistry, D. Schaa and D. P. Zhang, "Heterogeneous Computing with OpenCL 2.0," Morgan Kaufmann, 2015.

[21] Lee, W.-J., S. J. Hwang, Y. Shin, J.-J. Yoo and S. Ryu, *An Efficient Hybrid Ray Tracing and Rasterizer Architecture for Mobile GPU*, in: *SIGGRAPH Asia 2015 Mobile Graphics and Interactive Applications*, SA '15 (2015), pp. 2:1–2:4.

[22] Lee, W.-J., Y. Shin, J. Lee, J.-W. Kim, J.-H. Nah, S. Jung, S. Lee, H.-S. Park and T.-D. Han, *SGRT: A Mobile GPU Architecture for Real-time Ray Tracing*, in: *Proceedings of the 5th High-Performance Graphics Conference*, HPG '13 (2013), pp. 109–119.

[23] Montgomery, D. C., "Design and Analysis of Experiments," Wiley, 2012.

[24] Parker, S. G., J. Bigler, A. Dietrich, H. Friedrich, J. Hoberock, D. Luebke, D. McAllister, M. McGuire, K. Morley, A. Robison and M. Stich, *Optix: A general purpose ray tracing engine*, ACM Trans. Graph. **29** (2010), pp. 66:1–66:13.

[25] Parker, S. G., J. Bigler, A. Dietrich, H. Friedrich, J. Hoberock, D. Luebke, D. McAllister, M. McGuire, K. Morley, A. Robison and M. Stich, *Optix: A general purpose ray tracing engine*, in: *ACM SIGGRAPH 2010 Papers*, SIGGRAPH '10 (2010), pp. 66:1–66:13.

[26] Patterson, D., "Computer Organization and Design : The Hardware/Software Interface," Morgan Kaufmann, Oxford Waltham, MA, USA, 2014.

[27] Pharr, M., W. Jakob and G. Humphreys, "Physically Based Rendering, Third Edition: From Theory to Implementation," Morgan Kaufmann, 2016, 3 edition.

[28] Shirley, P., "Ray Tracing in One Weekend," Amazon Digital Services LLC, 2016, 1 edition.

[29] Stallings, W., "Computer Organization and Architecture (10th Edition)," Pearson, 2015.

[30] Suffern, K., "Ray Tracing from the Ground Up," A K Peters/CRC Press, 2007.

[31] Tristram, D. and K. Bradshaw, *Evaluating the Acceleration of Typical Scientific Problems on the GPU*, in: *Proceedings of the South African Institute for Computer Scientists and Information Technologists Conference*, SAICSIT '13 (2013), pp. 17–26.

[32] Vardis, K., A. A. Vasilakis and G. Papaioannou, *A Multiview and Multilayer Approach for Interactive Ray Tracing*, in: *Proceedings of the 20th ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, I3D '16 (2016), pp. 171–178.

[33] Wang, Y., C. Liu and Y. Deng, *A Feasibility Study of Ray Tracing on Mobile GPUs*, in: *SIGGRAPH Asia 2014 Mobile Graphics and Interactive Applications*, SA '14 (2014), pp. 3:1–3:5.

[34] Wen, Y., M. F. O'Boyle and C. Fensch, *MaxPair: Enhance OpenCL Concurrent Kernel Execution by Weighted Maximum Matching*, in: *Proceedings of the 11th Workshop on General Purpose GPUs*, GPGPU-11 (2018), pp. 40–49.

[35] Wickham, H. and G. Grolemund, "R for Data Science: Import, Tidy, Transform, Visualize, and Model Data," O'Reilly Media, 2017.

[36] Ylitie, H., T. Karras and S. Laine, *Efficient Incoherent Ray Traversal on GPUs Through Compressed Wide BVHs*, in: *Proceedings of High Performance Graphics*, HPG '17 (2017), pp. 4:1–4:13.