Automatic Ray-Tracer Cloud Offloading in OpenMP
Matheus Mortatti, Hervé Yviquel, Guido Araújo

To cite this version:
Matheus Mortatti, Hervé Yviquel, Guido Araújo. Automatic Ray-Tracer Cloud Offloading in OpenMP. the 9th Workshop on Applications for Multi-Core Architectures (WAMCA 2018), Sep 2018, Lyon, France. hal-01984331

HAL Id: hal-01984331
https://hal.archives-ouvertes.fr/hal-01984331
Submitted on 16 Jan 2019

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

L’archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d’enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.
Automatic Ray-Tracer Cloud Offloading in OpenMP

Matheus Mortatti, Hervé Yviquel, and Guido Araujo
Institute of Computing
University of Campinas (UNICAMP)
Campinas, Brazil
ra156740@students.ic.unicamp.br – {herve.yviquel,guido}@ic.unicamp.br

Abstract—Rendering an image from a 3D scene requires a large amount of computation which grows exponentially with the complexity of the scene (e.g. number of objects and light sources). With the increasing demand of high definition content, 3D designers need to use high-performance computer systems to keep the rendering time acceptable. Since owning computer clusters is expensive, designers usually rent computing power directly from cloud service providers (e.g. AWS and Azure). However, even though many cloud providers already propose dedicated rendering services, integrating them within the standard workflow of modeling softwares can become a complex and cumbersome task. It typically requires exporting the project from the design software, dealing with various access control mechanisms from different clouds to upload the project, and executing the rendering remotely through command-line. Offloading computation to the cloud is a technique which can considerably simplify such tasks. To achieve that, this paper uses an extension of OpenMP 4.X to eliminate any major interactions with the end-user, while minimizing the complexity of cloud integration and optimizing the design workflow. It applies such approach to a ray-tracing application, a simplified version of the engines used by professional 3D modeling software (e.g. Blender). It automatically offloads the rendering process from the user computer to computer cluster within the Microsoft Azure cloud, brings the resulting images back after the computation ends and displays them directly on the screen of the user computer, thus providing a transparent programming model and good speed-ups over local execution.

I. INTRODUCTION

Modeling and rendering softwares behind production-grade animated films is available to the public, but the rendering process usually requires high-performance computer systems, called render farms, to generate high quality scenes in reasonable time. For example, a single frame of Monsters University, the 3D computer-animated comedy film produced by Pixar in 2013, took 29 hours to render using a 2,000 nodes computer cluster [1]. One of the most used tools in this domain is ray-tracing, a 3D rendering algorithm that produces ultra-realistic graphics in images and films [2], especially in the area of lighting simulations, which other rendering techniques suffer from. However, the computing complexity of ray-tracer increases exponentially along with the complexity of the scene (e.g. number of objects and light sources).

Since few people have access to the large computing resources required for rendering, cloud providers, such as Amazon AWS or Microsoft Azure, offer dedicated services to setup custom render farms within their datacenters. However, the workflow required to integrate the cloud is still far from being user-friendly, making it hardly accessible to the common graphic designer which is not necessarily a computer expert. To achieve that, the user has to instantiate a cloud cluster and setup the render farm with the software that matches his workflow. Then, he will have to export the scene project files from the modeling software running on his own computer, upload the project files to the cloud by hand, and then run the rendering through remote connection (e.g. SSH). Moreover, integrating cloud services to their software is a cumbersome task for developers since they have to deal with various access control mechanisms from different clouds (e.g. AWS and Azure). Finally, the user has to wait for the rendering of the whole image sequence before being able to download and see it. The possibility of one image taking hours to render makes this workflow far from being optimal.

To address such problem, we have previously developed a novel and yet compatible extension of OpenMP 4.X, called OmpCloud1. This tool allows programmers to easily integrate computation offloading and MapReduce-based parallelization to remote computer clusters within their software project using simple OpenMP directives [3], [4]. Thus, this would considerably simplifies the integration of cloud render farms to the render engines of 3D modeling softwares for both software developers and designers alike.

In this paper, the OmpCloud offloading technique is applied to a simple ray-tracing based engine that we have designed2, a simplified version of the engines used by professional 3D modeling software (e.g. Blender [5] or Renderman [6]). The main contributions of this paper are the following:

- It demonstrates how OmpCloud easies the programmer’s tasks of computation offloading and parallelization: by simply adding basic OpenMP directives to the code of our sequential ray-tracer implementation, the computation is automatically offloaded to the cloud and parallelized within a distributed cloud cluster;
- It also shows how the execution model enabled by cloud offloading can optimize the workflow of the user: as each frame in a sequence can be rendered independently, the application can achieve automatic data retrieval in real time and display the images as soon as they are ready on the screen of the user computer;

1OmpCloud is freely available at http://ompcloud.github.io
2A demonstration of the Ray-Tracer offloaded to the cloud is available at https://youtu.be/qqpNQWW9eT-M
The remainder of this paper is organized as follows. First, we start by introducing the basic concepts involved in directive based programming and cloud computing (Section II). We then describe the proposed approach in Section III. Section V presents and analyzes the experimental results, and Section VI discusses related works. Section VII concludes the work.

```
int MatMul(float *A, float *B, float *C) {
    // Offload code fragment for acceleration on the cloud cluster
    #pragma omp target device(CLOUD)
    #pragma omp map(to: A[:N*N], B[:N*N])
    // Execute loop iterations in parallel on the cluster
    #pragma omp parallel for
    for(int i=0; i < N; ++i)
        for (int j = 0; j < N; ++j)
            for (int k = 0; k < N; ++k)
                C[i * N + j] += A[i * N + k] * B[k * N + j];
    // Resulted matrix 'C' is available locally
    return 0;
}
```

Listing 1: Using the OpenMP accelerator model to run a matrix multiplication into a computer cluster

II. BACKGROUND

OpenMP is a programming model in which program fragments (hot loops) are annotated to make it easier to parallelize the code. The latest version of this standard, OpenMP 4.X [7], includes new directives that allow the transfer of computing to accelerator devices (e.g. GPUs). From a programmer’s point of view, a program starts running on a typical host (e.g. CPU), and when the code snippet annotated with the OpenMP clause is reached, code and data are transferred to the given device for execution, returning the control flow to the host upon completion; this technique is called computation offloading, or simply offloading.

Listing 1 presents a program loop for matrix multiplication that has been annotated with OpenMP directives to offload the computation into an accelerator device. In the OpenMP 4.X standard, the target pragma defines the part of the program that will be executed by the target device. The map clause details the mapping of the data between the host and the device: in Listing 1, inputs (A and B) are sent to the device while output (C) is received from the device after the computation has ended. While the typical devices used in OpenMP 4.X are DSP cores, GPUs, Xeon Phi accelerators, etc. [8], [9], we introduced the cloud as a novel target device available on the computer. This was done within a programming framework we call OmpCloud [3] that extends the OpenMP accelerator model to allow transparent cloud offloading and cluster programming. OmpCloud gives the programmer the ability to quickly expand the computing power of their own computer to a large-scale computer cluster available in the cloud. By using OmpCloud, the programmer can re-use his basic knowledge of OpenMP: it first uses annotations to validate a small computation on a local machine, and then with few modifications (using the device clause), it can migrate it to the cloud for a more expensive computation.

Instead of parallelizing the program into the cores of a single computer, the OmpCloud runtime automatically transfers the annotated program fragments from the local computer to the cloud, thus allowing the parallelization of the iterations of a loop on multiple machines of a cluster. The parallel loop is transformed to match the execution model of MapReduce modeling [10] using the Apache Spark framework [11], which provides seamless execution combined with fault tolerance, data distribution, and workload balancing. In fact, most cloud providers made the deployment of Spark clusters fairly straightforward thanks to the dedicated web interface and custom Linux distribution (for example, Azure HDInsight or Amazon EMR).

The OmpCloud user can easily create their own cluster in just a few clicks without knowing much about parallel programming or cloud computing. The program is started by the user on its own local machine and runs locally until the code fragment annotated with OpenMP is reached. A method is then called to initialize the cloud device. Offloading is done dynamically by the runtime. First, the input data of the kernel are sent to a cloud storage (e.g. AWS S3 or any HDFS server). After all input data has been transmitted, our runtime sends the task to the Spark cluster and locks until the end of the task execution. The driver node reads the kernel input data from the cloud file storage, transmits the input data and distributes the computation to the worker nodes. The output is then collected, rebuilt by the driver and stored into the cloud storage to be transmitted back to the local program, which then continues to run on the local machine. Moreover, OmpCloud directly allows portability over HPC clusters, commercial cloud services and private clouds, making cloud integration easy for the software developer and the user.

III. RAY TRACER

A rendering engine like Ray-Tracer is the perfect matching application to experiment with the OmpCloud cloud offloading execution model. Such engine is usually written in C/C++ and requires a lot of computation that could benefit from been executed in a large remote cloud cluster. This section describes a simple Ray-Tracer we developed based on an algorithm similar to the one used in rendering engines from professional 3D modeling tools such as Blender [5]. In fact, our program emulates the same behavior: the user starts by configuring a scene to render, he then runs the application, the engine performs rendering in a cloud cluster, and as soon as it ends, the resulting images are displayed on the user screen.

A. Implementing the Ray Tracer

A Ray-Tracer is a 3D rendering algorithm that is based on the idea of casting rays from the camera view to the 3D scene. Each ray is mapped to a pixel on the screen and its color is calculated by checking if the ray hits an object. The most simple way to do this is to check if the ray hits an object,
and then color the pixel with the color of the object. Listing 2 shows the implementation of a simple Ray-Tracer algorithm. The sections below describe every aspect of this algorithm in details.

```
// Iterate through every pixel
for (int y = 0; y < CANVAS_HEIGHT; y++) {
    for (int x = 0; x < CANVAS_WIDTH; x++) {
        // Cast a ray from the camera to the pixel
        float px = (2 * ((j + 0.5) / CANVAS_WIDTH) - 1) * tan(fov / 2 * M_PI / 180);
        float py = (1 - 2* ((i + 0.5) / CANVAS_HEIGHT)) * tan(fov / 2 * M_PI / 180);
        float rayP[3] = (px, py, -1.0);
        Object obj = has_intersected(rayP, orig);
        // If the ray intersects an object on the scene, paint the pixel with its color
        if (obj != NULL) {
            // scene, paint the pixel with its color
            pixel[y][x] = obj.color;
        }
    }
}
```

Listing 2: Simple Ray-Tracer algorithm

1) Generating Camera Rays: In the code presented in Listing 2, the loop indexes represent a coordinate from 0 to the width or height of the screen, usually called Raster Space. To generate the rays of the camera, we need to convert these coordinates to the screen space. To do that, the coordinate are first converted to the Normalized Device Coordinate (NDC). which is a coordinate ranging from -1 to 1 starting from the upper left corner of the screen. This is done by means of the following equations:

\[
\begin{align*}
\text{PixelNDC}x &= \frac{(\text{Pixelx} + 0.5)}{\text{CanvasWidth}} \\
\text{PixelNDC}y &= \frac{(\text{Pily} + 0.5)}{\text{CanvasHeight}}
\end{align*}
\]

(1)

Then, we have to fix the origin of the coordinate, since NDC considers the upper left corner of the screen as the origin. The following equation is used to change it to the center of the screen:

\[
\begin{align*}
\text{PixelScreen}x &= 2 * \text{PixelNDC}x - 1 \\
\text{PixelScreen}y &= 1 - 2 * \text{PixelNDC}y
\end{align*}
\]

(2)

The ray vector of the camera can now be generated as shown in lines 8 and 9 of Listing 2. The scene camera is defined as a point in space (usually at the origin) and a canvas representing the screen, which is 1 unit away from the camera. We define the camera ray as a vector starting at the position of the camera, pointing to the direction of a single pixel on the canvas. This ray is used to calculate the color value of that pixel based on the scene.

To compute whether or not the ray has intersected an object, we need to do calculations depending on the type of the object we are using. In our implementation of the Raytracer, we used spheres and triangles in the scene, and thus we needed to find a way to compute the intersection of a line to these two kinds of objects. Below we describe how the computations of the intersection of light rays and an objects are performed.

2) Ray-Triangle Intersection: In order to discover if a ray has intersected a triangle, we need to know if the ray intersected the plane defined by the triangle, the coordinates where it happened and if the intersection point lies inside the triangle. We can determine the point \( P \) where the ray hits the plane defined by the triangle by solving Equation 3 below.

\[
P = O + tR \quad (3)
\]

\[
t = -(\text{dotProduct}(N, O) + D)/(\text{dotProduct}(N, R)) \quad (4)
\]

Where \( O \) is the origin of the ray, \( R \) is the ray’s direction and \( t \) is the distance between the origin of the ray and point \( P \). Given the normal of plane \( N \), we can find \( t \) by solving Equation 4. Where \( D \) is the ray from the origin to the plane. Now that we have \( P \), we can find out if it lies inside the triangle by checking if the point is placed on the left of every side of the triangle. Basically, a cross product is computed with each of the sides of the triangle and the vector associated to point \( P \), as shown in the Figure 1. Since the Right-Hand Coordinate System is used, we only need to check if the result of the cross product returns a positive value. If it does for the cross-products of all triangle sides, then point \( P \) is inside the triangle.

3) Ray-Sphere Intersection: The Ray-Sphere intersection is calculated using an analytic solution. Let us define Equation 5
which describes the sphere such that:

\[ x^2 + y^2 + z^2 = R^2 \]  (5)

If such a point exists, then the sphere equation will have a solution for this point \( P \). If we insert the point in Equation 5 we obtain:

\[ P^2 - R^2 = 0 \]  (6)

\[ P = O + t \cdot D \]  (7)

where \( P \) is the point and \( R \) is the radius of the sphere. As shown in Equation 7, \( P \) can be defined in the parametric form, where \( O \) is the rays’ origin, \( D \) its direction and \( t \) the distance between the origin and point \( P \). By combining together Equations 6 and 7 one can derive Equation 6 as follows:

\[ |O + t \cdot D|^2 - R^2 = 0 \]  (8)

This equation can be seen as a quadratic function of \( t \). One can then compute the discriminant \( \Delta \) of the equation and determine if the equation has a solution. If \( \Delta > 0 \), the equation has two solutions, and the one closest to the ray will be the intersection point \( P \). On the other hand, \( \Delta = 0 \) means that there is a single point of intersection \( P \), and if \( \Delta < 0 \) the ray does not intersect the sphere.

4) Lighting Calculations: Up to this point, the Ray-Tracer uses an unlit mode, with all of the objects in the scene having a flat color. To provide depth to the image, the engine needs to perform lighting calculations on the objects, which is shown in Listing 3. The idea is quite straightforward: having the point where the ray of the camera hits the object surface, the renderer can cast another ray in the direction of each of the light sources (variable \( \text{rayDir} \) on line 6) and if the ray does not hit any other object on its way (line 9), calculate the influence of the light source onto the object. This calculation is done by computing the angle between the light ray and the normal of the object (lines 12-13), and scaling the color with the cosine of the value (line 14). Listing 3 shows the implementation of this algorithm.

IV. INTEGRATING RAY-TRACER TO OMPCLoud

The Ray-Tracer algorithm is an ideal application for OpenMP parallelization. Indeed, each pixel can be rendered independently from the others since the rays created for each pixel are checked for intersection with every object present in the scene. Annotating the code with OpenMP was as simple as placing the parallel pragma on top of the main loop, and sharing the frame buffer to all threads. Now, since OmpCloud aims at making the offloading of a parallel loop to cloud easy, implementing the support for it to our Ray-Tracer was pretty straightforward.

The code in Listing 4 offloads all scene object information automatically to a cloud infrastructure (Microsoft Azure in our case). The target pragma defines the portion of the program that will be executed by the cloud cluster. The map clause details the data mapping between local and cloud memories: scene information, such as object color, triangle and sphere positions, lights, field of view and aspect ratio are sent to the cloud, while the final image pixel information (written in frameBuffer) is downloaded at the end of the computation. Finally, the parallel pragma defines the loop to parallelize on the cloud cluster: the rows of the image will be basically distributed to the cluster nodes for computation since we are only parallelizing the outer loop.

```c
#pragma omp target device(CLOUD)
#pragma omp target
map(to: fov, \colorTri[: t_size * 3], \colorSphere[: s_size * 3], \radius[: s_size], \material[: m_size * 3], \lights[: l_size * 3])
from: frameBuffer[0 * CANVAS_HEIGHT*CANVAS_WIDTH]
#pragma omp target parallel for shared(frameBuffer)
for (int i = 0; i < CANVAS_HEIGHT; i++) {
  for (int j = 0; j < CANVAS_WIDTH; j++) {
    ...
  }
}
```

Listing 4: OmpCloud Implementation of Ray-Tracer

As shown in Figure 2, the user must first create the cluster and store the credentials into an specific configuration file. The program is then started by the user in its own local machine, the program reads the file describing the scene and runs locally until the OpenMP annotated code fragment is reached. A method is then called to initialize the cloud device. The runtime sends the input data required by the kernel as binary files to a cloud storage device (e.g. AWS S3 or any HDFS server) (3). After all the input data has been transmitted, the runtime submits the job to the Spark cluster and blocks until the end of the job execution. The driver node, which is in charge of managing the cluster, reads the input data from the cloud file system (4), transmits the input data and distributes the loop iterations across the Spark worker nodes (5), which are in charge of the computations. Next, the worker nodes run the mapping function that computes the loop body in parallel (6). The output of the loop is then collected, reconstructed by the driver (7) and stored into the cloud storage (8) to be transmitted back to the local program (9), which then displays the resulting image on the local machine (10). The color values of the pixels stored into the framebuffer are converted into an image displayed on the user screen in real time thanks to the Simple DirectMedia Layer (SDL) [12], as shown in Figure 3.

The user can configure the rendering process in different modes: rendering the the whole image and display it at once on the screen (Figure 3a) or split the image in separate tiles with a predefined size that will be displayed as soon as they are rendered, breaking down the process in multiple parts (Figure 3b). Additionally, the user can define multiple frames
by moving the camera within the scene: the frames will be rendered and displayed on the screen one at a time. In fact, such behavior is very similar to the one proposed by most 3D modeling softwares: the rendering can be performed **online** or **offline**. OmpCloud enables cloud integration within the designer workflow even for online rendering with minimal work for the developer: he just need to add some few pragmas to the code.

V. **Experimental Results**

The experiments were performed using is a simple laptop which interacts with a remote cluster from the Microsoft Azure cloud through an Internet connection. Our experiments intend to be a realistic test-case where the client computer is far away from the cloud data-center. The local computer was composed of an Intel Core i7-6700HQ processor (4 cores) running at 2.60GHz and 16GB of RAM, with Ubuntu 16.04. The remote Spark cluster was composed of 5 nodes, all of them running Apache Spark 2.1.0 on top of Ubuntu 16.04, and created using the dedicated Azure HDInsight service. Each node of the cluster was a virtual machine instance of type **D13v2** with 8 cores (executing on Intel Xeon E5-2673 v3 running at 2.4 GHz) and 56GB of RAM. For all experiments, the applications have been compiled with OmpCloud v0.3.1, a custom fork of Clang/LLVM v3.8, using only the standard **O3** optimization flag. The Ray-Tracer was tested with two different configurations: the standard multithreaded OpenMP implementation executed on the local computer, and the OmpCloud implementation which offloads the computation from the laptop to the remote cluster. To properly compare the performance between local and cloud rendering, we created three sets of benchmarks that instantiated different numbers of objects in the scene, as shown in Figure 3. Each benchmark is increasingly heavy on the rendering device, named Benchmin, BenchMid, and BenchMax, instantiating 900, 4,900 and 10,000 spheres respectively. Each benchmark was performed three times for each configuration described.

Figure 4 presents the results of the execution for the Ray-Tracer (RT) multithreaded implementation, named **RT-OMP**, when executing locally and the cloud offloading implementation, named **RT-Cloud**, both compared with the single core execution executed locally, named **RT-Serial**. The execution times of the different dataset for all implementation are presented in Figure 4a. For the serial execution, BenchMin, BenchMid and BenchMax took about 8 min, 44 min and 91 min respectively, with standard deviation varying between 1.7 and 47.3. Result shows that the serial execution on the local computer is always slower than the cloud offloading one, even including the extra overhead due to the communication and the initialization of Spark. However, even if the execution time of **RT-Cloud** reduced with the number of core, the local OpenMP implementation (**RT-OMP**) which use only 4 cores is faster than **RT-Cloud** with 4 and 8 cores because of the extra overhead induced by cloud offloading. In fact, the communication time between the local computer and the cloud infrastructure was about 30s for all benchmarks, showing the overhead is not only due to the remote access.

Figure 4b shows the speedup of the Ray-Tracer over the serial implementation (**RT-Serial**) calculated for the OpenMP and OmpCloud implementations. The results show that the speedup of **RT-OMP** implementation is linear and almost equal for all datasets. This demonstrates the efficiency of OpenMP
parallelization in shared memory architecture. On the other hand, RT-Cloud implementation hardly reaches a speedup of 3x when rendering BenchMax with 16 cores. On 32 cores, the speedup is much better (up to 9.3x) for BenchMid and BenchMax where the overhead of using multiple cores and nodes is small with respect to the benefits of the parallel execution. In order to have a better understanding on how much each core impacts the overall performance, Figure 4c presents the parallel efficiency (speedup / core count) of each implementation. As shown, even though the efficiency of the cloud execution is smaller than the local execution, the former increases as we increase the benchmark size, revealing an improved scalability. Since the overhead can be explained by the load of the Spark runtime which could be reduced using advanced tools for dedicated Spark job submission, like Apache Livy [13] or Spark Job Server [14]. Those tools introduce submission through RESTful APIs and maintain a unique Spark context alive between all jobs, thus enabling lower initialization latency. On the other hand, the overhead is not reducing significantly on larger benchmark, so the load time, more or less constant, cannot explain it entirely. In fact, OmpCloud uses Spark to handle all parallelism: between node and inside each node. Spark creates a dedicated Java Virtual Machine (JVM) on each worker node and use Spark multithreading to parallelize tasks inside a node. Since there is no inter-nodes communication when using 4 and 8 cores (single node), this shows that Spark is not very efficient for multithreading. As a result, future OmpCloud implementations could really benefit from using OpenMP multithreading instead of Spark one to parallelize within each cluster node.

VI. RELATED WORKS

Cloud offloading has been largely studied for mobile computing in order to increase the computational capabilities of cellphones [15]. Some frameworks have been proposed to facilitate the development of mobile applications using cloud resources [16], [17], [18]. Contrary to our approach which relies on C/C++, they mostly rely on .NET or Java which are the most popular environments for mobile device but not really adapted for render engines requiring heavy computation. One of the key challenges of those offloading frameworks is how to dynamically determine if it is worth offloading the computation in terms of communication overhead and energy consumption [19]. Additionally, older works used a similar offloading execution model to accelerate spreadsheet processing in grid computing [20], [21]. In fact, the automatic decision of offloading to a computer cluster versus running on the programmer’s processor is an interesting but complex problem that is not treated in this paper. OpenMP allows programmers to add an if clause to target pragmas to
condition the offloading according to a boolean expression. In our case, the image size and the number of objects in a scene can be considered as parameters to be tested by an if clause to decide between doing offloading to the cloud or not.

Some works have studied the offloading of Blender’s rendering engine to Xeon Phi platforms using OpenMP [22] and MPI [23]. While those works demonstrate good performance, Xeon Phi are quite an exotic platforms not really available to the majority of users. Most of the rest of the literature which strictly follows the OpenMP accelerator model studied GPU offloading and demonstrate good results for heavily parallel applications [8], [9]. Nevertheless, some of them look for more untraditional targets like the Intel Xeon Phi platform or FPGAs [24].

Several interesting works also explore the usability of the programming model as well as the performance portability of the application over different platforms [25], [26], [27]. In particular, Hahnfeld and al. explore programming patterns to allow pipeline parallelism between host-device communication and computation [28]. Such programming patterns could be used to improve the performance of our application, especially when rendering several images and/or tiles.

VII. CONCLUSION

This paper presents the first implementation of a ray-tracing render engine that supports automatic cloud offloading based on the OpenMP accelerator model. Upon analyzing the results and having the experience of implementing OmpCloud support in a real world situation, it is safe to say that, despite its limitations, OmpCloud does provide a simple workflow for the arguably hard existing solutions for cloud offloading. With only a few lines of code and no more required experience than knowing the basics of OpenMP, it was indeed possible to achieve great speedups on a graphical heavy application,
coming closer to a future when cloud resources could be easily programmable. The size of the benchmarks used were not as big as real world graphics applications and still, the speedups were significant. Production grade image rendering require days to complete and using OmpCloud, it would be possible to greatly reduce this time without adding cumbersome tasks to the developer nor the user. In the future, we plan to experiment our methodology with Blender since it is an open-source framework. We also plan to support pipeline parallelism and GPU cloud processing to improve the performance of the rendering engine.

ACKNOWLEDGMENT

This work was respectively supported by the PIBIC-CNPq and CCES CEPID/FAPESP grants 2014/25694-8 and 2017/21339-7. The experiments were also supported by the Microsoft Azure for Research program and the AWS Cloud Credits for Research program.

REFERENCES


