Image Rendering Analysis for PBRT and LuxCore

Ujjwal Prazapati(uprazap1@umbc.edu), Ranjan Jaiswal(Ranjanj1@umbc.edu)

***Abstract*—Rendering is the process of generating images which are either based on the flow of light in the real world using reflectance distribution function and mathematical rendering equation or generating the images which are based on different styles of digital art which is encouraged from various artistic categories like animated cartoons, drawings, paintings and technical illustrations. Various Analysis shows that an average 1080p/60fps video takes almost 3 times the video length and an 1440p/60 fps takes almost 5 times the video length for rendering on a i5 quad core CPU and definitely this time will improve as we improve the processor specs. We know that rendering is a resource intensive process and its execution time can be reduced by chipping in more cores and processors but this also brings the problem of cost. In layman terms, if we increase the number of labors then definitely time will improve but instead of that we want to limit the number of laborers and make them productive or skillful i.e. Can we find any other way like changes in hardware which are specific for rendering requirements or make rendering algo better. This report first introduces the concept of rendering and introduces you to the two rendering algorithms named PBRT and LuxCore. Moving on further, this report describes the source origin and steps to build the binaries of these algo which will be used for rendering purposes. Further, this report will explore the related work in the rendering and check whether any significant work has been done regarding rendering analysis or not and if yes, how it is related to our work and if not, then whether it has any relation to the work that we are doing in this report. After that, this report introduces a profiling tool which will be used for the profiling purpose of binaries generated through PBRT and LuxCore using various internal tools like CahceGrind, CallGrind and KCachegrind. The tools will be giving results for metrics like cache miss, branch misprediction etc. After this, the report will show the results generated using valgrind profiling. Further, the results generated by valgrind will be used for analysis. Our analysis will show that there is no scope of optimization regarding code optimization to reduce cache misses which are higher in PBRT but insignificant in LuxCore. Also, how high branch misprediction which is high in both LuxCore and PBRT can be reduced using alternative branch prediction techniques. Further, it will show the code analysis and show what code segments are causing problems and what kind of code section can be improved and is fit for a certain kind of instruction set architecture which can be used for rendering in a better way. Moving on, we do some mathematical calculations which shows how the new instruction set architecture decreases the number of instructions in the function which is getting called most using Amdahl's Law by taking a certain example for a set of instructions. We then give a conclusion of what we have achieved in brief and how it can be useful to make the process of rendering very economical so that the scope of rendering can reach a wider audience. At the end, we discuss the future scope of this project and also share some references for this report.**

I. Introduction and Motivation

A. What is Rendering?

Rendering is a process of converting a raw input into a realistic image which can be displayed on screen by means of computer software. Rendering begins with shading and texturing objects and lighting the scene and ends when surfaces, materials, light and motions are processed into a final image or sequence of images. Rendering is one the final steps in the production pipeline. Rendering is a resource intensive process which mostly depends on the period of rendering and use of external software. This process is used in almost all the fields such architectural design, gaming, movies and product design. Figure 1 depicts a variety of rendering techniques applied on a single scene.

B. Why this Problem?

Rendering is an incredibly resource intensive process, performing complex calculations which usually demands a computer with high specifications. High specifications come with the high cost for rendering. Also, time to produce high quality rendered images increases if an average user tries to run it on an average computer, which limits the rendering to limited users. Most of the research performed earlier either focus on improving the computer specifications, rendering algorithms or some configuration tuning in multicore and many core systems. Whereas our approach is to focus on improving the architecture of the system by implementing better approaches such as vector processing, loop unrolling or multiprocessing.

C. What is PBRT?

PBRT is a physically based ray tracing algorithm that renders the image in a way that is similar to the flow of light in the real world. The book "Physically Based Rendering: From Theory to Implementation, Third Edition" explains the theory and mathematics behind the photorealistic rendering system as well as its practical implementation on sample scenes. PBRT works by taking '.pbrt' as an input file and produces an image file as an output by the means of a ray tracing algorithm.

D. What is LuxCore?

LuxCore is an open-source physically based ray tracing rendering software written in C++ and Python. It simulates the flow of lights according to physical equations and produces realistic images. It bundles one or more light sources into groups and changes the color and intensity of the scene during the rendering process. Virtual film - a feature used in Luxcore which allows users to pause and used in Luxcore also supports High Dynamic Range (HDR) rendering and interactive scene editing.

E. Rendering Algorithm

There are various rendering algorithms. Some of them are discussed and used in this study.

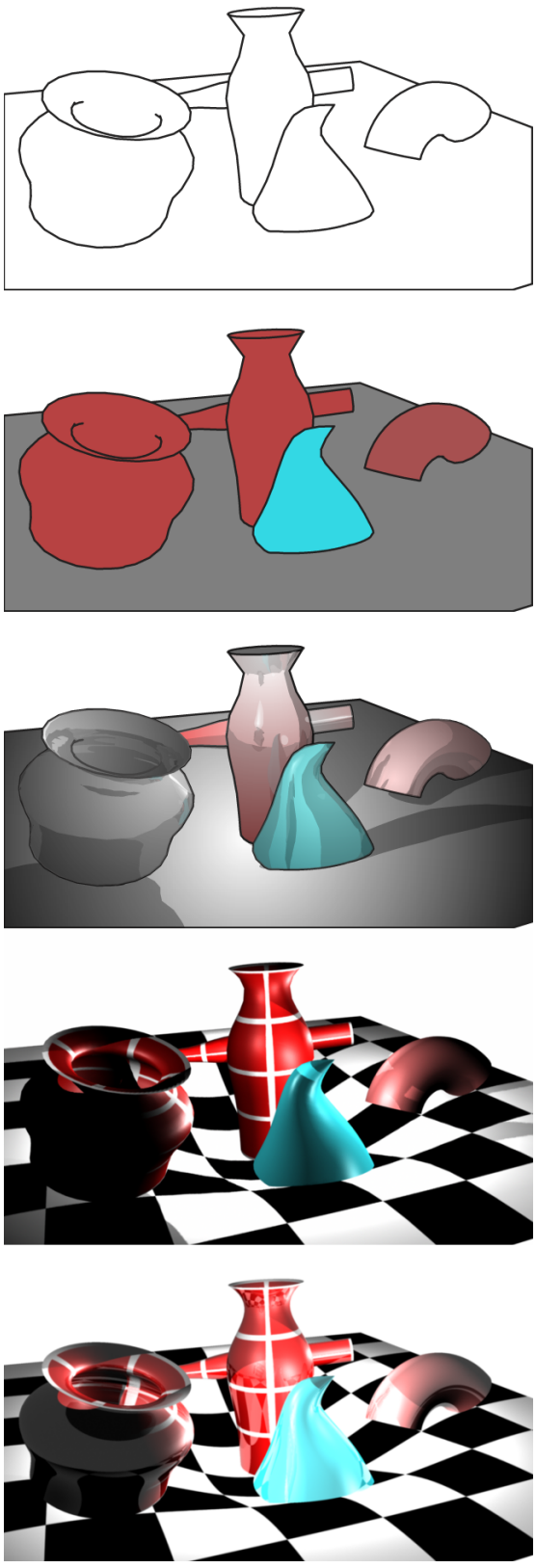


Fig. 1. A variety of rendering techniques applied to a single scene

1) ***Rasterisation*** *– It takes vector data and transforms it into pixels – It is a process by which vector graphics are converted into raster images (a series of pixels) which are then displayed to the computer screen. Rasterisation works by breaking 3D polygon into triangles and then applying triangle Rasterisation algorithms on two adjacent triangles by leaving no gaps between triangles and rasterizing one pixel only once. After all pixels are processed, the computer is able to show 2D representation of 3D objects.:*

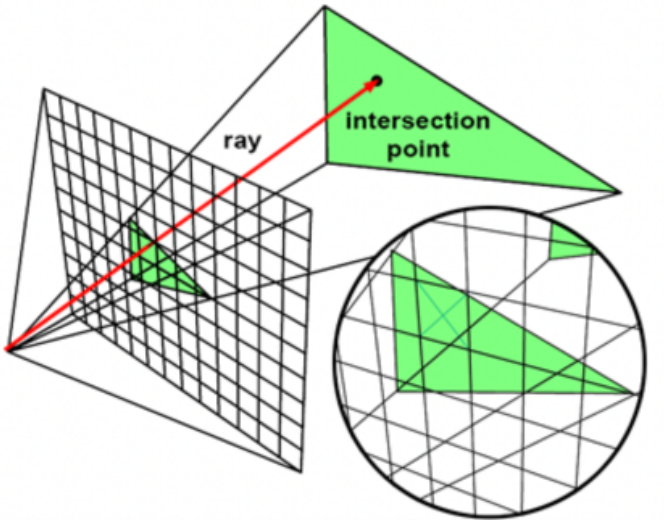


Fig. 2. Rasterisation

2) ***Ray Casting*** *– Ray casting is the most basic rendering algorithm which uses a geometric method of ray tracing. In this algorithm, rays are casted through camera one through every pixel, when the objects in the scene are intersected by rays, the computer records the closest point to the camera and compute-colored values of the texture based on surface radiance. This method is fast as compared to ray tracing but requires a massive number of calculations. This is limited to*

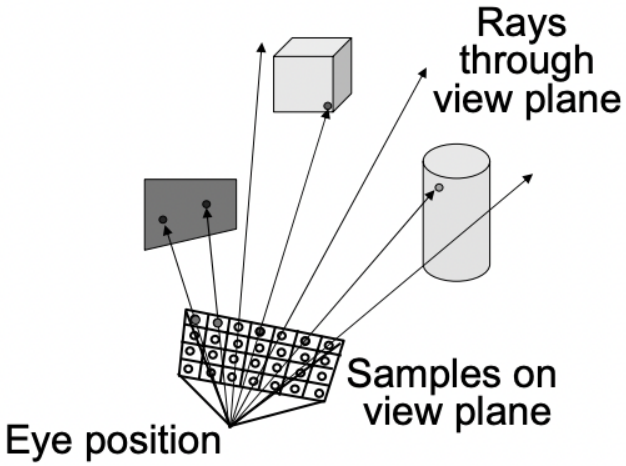


Fig. 3. Ray Casting

its creativity as it only has colors that define the texture of geometry and rays going in one direction which means no reflection or shadow possible.:

3) ***Ray Tracing*** *– This method works on bouncing back of the light rays. Computer sends the rays through every pixel to the object but when rays hit the object, unlike ray casting instead of recording data straight away, the computer sends more rays one through each light source to find out how these lights are affecting the color or brightness of the point. Now the point could have shading, and illumination data calculated at the intersection by the behavior of objects on each light source. A material dictates how light should behave whether it's reflected or absorbed or passes through the materials. There are billions of these bounces which result in scene appearance. This is widely used in real time rendering.:*

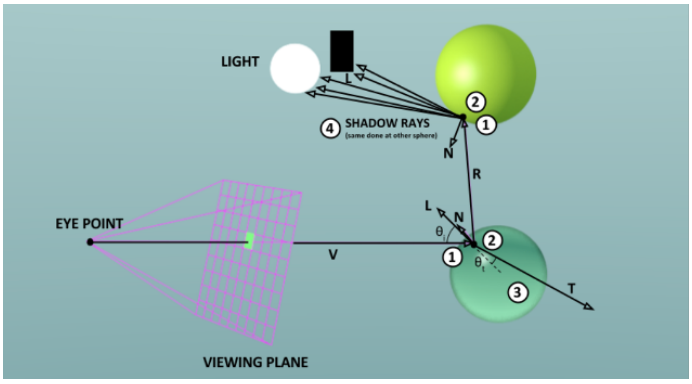


Fig. 4. Ray Tracing

4) ***Path Tracing*** *– Similar to Ray Casting and Ray Tracing, computer send the rays to an object but when the rays hit the object, instead of sending more rays from the point of light source it keeps bouncing the same ray between the scene objects as many time as possible to gather information objects close to each other, find out the color and texture and see if the color affects the previous objects. In general, it's similar to how light behaves in the real world. This method not only focuses on the position of light but also the size of the light. This is mostly used in offline rendering.:*

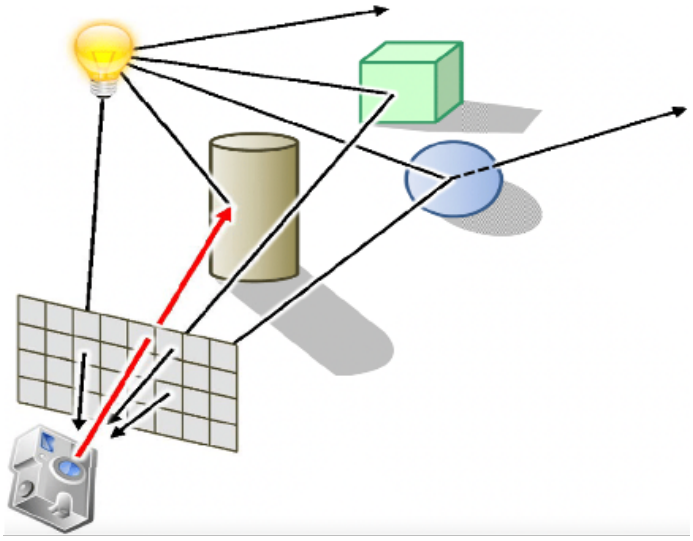


Fig. 5. Path Tracing

II. Related Work

In this section we will be discussing what are the related rendering softwares used by industry and why it is useful to work on analysis of PBRT and LuxCore.

A. ***Rednerman -*** *If we look at some of the rendering softwares used by industry then we see that Pixar Animation Studios uses Renderman for rendering of visual effect.The path tracing architecture of the new generation renderman is designed on the inspiration of PBRT book. And everybody know how much Pixar is famous.*

Fig. 6. Renderman

B. ***NVIDIA IRay -*** *In 2015, NVIDIA bought Physically Based Rendering (PBR) to the mainstream. Because, it wanted to visualize the patterns with right amount of lightening in the 3D objects. For example, some kind of glare on the mirror or lightening flowing the person standing in front of the light or let's say any shining bead.*

Fig. 7. Nvidia

All this required, physically based rendering and a very fast and efficient GPU to process more data in 3D objects. The other renderer LuxCore is also based on PBRT and is an unbiased ray tracer written in C++ and Python but the

main part here is that it is based on PBRT. So, because of all these reasons we can say that the rendering software which are mostly used in industry are based on PBRT and LuxCore.

III. Source Building and Deployment

A. Operating Systems and Specifications

This section describes the operating system and its specifications used in this project. To avoid any CPU or memory usage by background applications – which would add impurity in our profiling data, we decided to go with a new virtual machine provided by Microsoft Azure. We built and deployed the Linux operating system with 2 vcpus and 8 GB memory on cloud and made the environment ready installation process.

B. Build Dependencies and Installation

We downloaded the latest source code from GitHub using the 'git clone' command in a new directory. Before we started the build process, we had to fix multiple dependencies like CMake version, python version compatible with cmake, C++ libraries and GCC dependencies. Then we created binaries files using the 'make' command and rendered sample scenes to test LuxCoreUI. Similarly, we downloaded the source code for PBRT from GitHub in a different directory. Then we ran 'make' command to build pbrt, obj2pbrt and imgtool utilities. It also created an executable file that performs pbrt unit tests on freely available scenes.

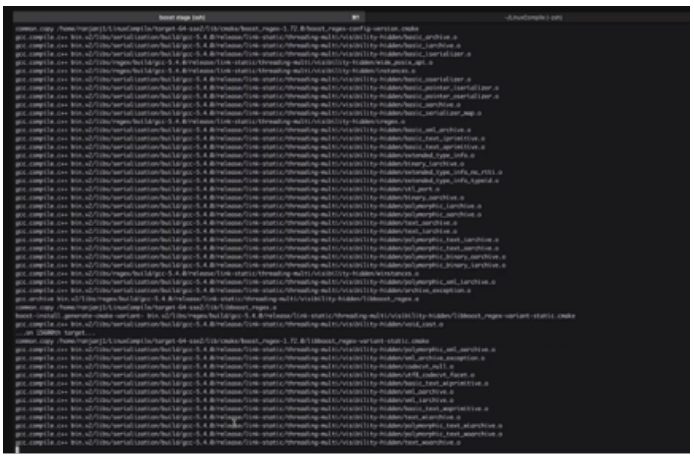


Fig. 8. Building in Luxcore

Moreover, Profiling tools used in this project such as Valgrind, Cachegrind, Callgrind and KCachegrind are installed using the 'sudo apt' command in the Linux terminal. More about profiling tools are discussed in section IV.

C. Running the Application

There are various ways to run the applications but we followed the command line tool to launch the application and gather the data for our analysis.

- Run pbrt without valgrind: ./pbrt ../../scenes/killeroo-simple.pbrt
- Run pbrt with valgrind and cachegrind: valgrind –tool=cachegrind –branch-sim=yes ./pbrt ../../scenes/killeroo-simple.pbrt
- Run Luxcore without valgrind: ./luxcoreui ./scenes/cornell/cornell.cfg
- Run Luxcore with valgrind and cachegrind: valgrind –tool=cachegrind –branch-sim=yes ./luxcoreui ./scenes/cornell/cornell.cfg
- Run Luxcore with valgrind and callgrind: valgrind –tool=callgrind –dump-instr=yes ./luxcoreui ./scenes/cornell/cornell.cfg
- Run Kcachegrind for visualization: kcachegrind callgrind.out.<pid>

IV. Profiling Tool

Here, we see the list of profiling tools that we have used for the purpose of analyzing PBRT and LuxCore and what purpose they server and why they are useful. We will go through each of them specifically and describe about them

A. Valgrind

Valgrind is a debugging tool or software for profiling purpose of a binary on various metrics like memory debugging, call trace, number of instructions, percentage of functions called or memory leak detection. It translates any program into some intermediate representation called IR too and this process is independent of the fact which processor is used by the system. On the basis of these representations, it helps us in gathering multiple stats regarding a running program.

We believe that this tool is helpful because we know that rendering is a resource intensive task and by just looking at it source and then creating an idea is not a good approach so we needed something which could reduce our scope and give us some pointers which can narrow down our path for analysis and even whether there is any path to focus on analysis or not. Plus, this one also one of the famous and recommended tool for any debugging and analysis purposes.

B. Cachegrind

This tool is part of Valgrind and while running Valgrind you just need to mention –tool=cachegrind to use this parameter. This tool is helpful for cache simulation and is helpful in analyzing the cache misses line by line in your source code when your program is executing. It tells you following cache misses:

- L1 cache misses of Instruction
- L1 cache misses for both read and write data
- L2 cache misses for instruction
- L2 cache misses for both read and write data

This is important because one L1 cache miss gives you a penalty of approximately 10 CPU cycles and one L2 cache miss gives you a penalty of 200 CPU cycles and these are not small penalties and one cache miss will be for one instruction. This makes it an important metric so that you can check your source and analyze the number of instruction that are running

for one source code line because this also gives you cache misses per source line.

Moreover, here we can also enable the parameter of branch misprediction. Once we enable this parameter along with cachegrind we would be able to see how many total branch misprediction were as a percentage of total instructions executed. Also, you can see source to source relations and what are the source where

We can see how this tool is narrowing down our focus where we can target particular function or some specific source lines in a function like some kind of loop or any source line which can be optimized. That's why this makes it an important tool for the purpose of adding it in our metric to analyze the source code of PBRT and LuxCore. Moreover, it will also save time to analyze things.

C. Callgrind

This tool is also a part of Valgrind and can be enabled using the following command in argos of Valgrind i.e. –tool=Callgrind. This tool tracks the call history of the running source which is known as a call graph where you can see the relation between the caller and callee which would be helpful in computing the cost. If a function func1 calls func2 then it would also take the cost of func2 while calculating the cost of func1 which is really practical and helpful. It also gives the number of calls made to a particular function which again seems to be very crucial for our analyzation as it can direct us to particular set of source lines and narrow down our focus.

There are two sub command tool provided in callgrind which just have different set of presentation to show the output data. Following are these two ways:

1) Callgrind annotate: This subtool will scan the data collected through profiling and print them in sorted order of the number of times it has been called. E.g. if the total number of function calls are 100 and this function got called 52 times then it will come on top because it has been called most of the time and correspondingly other functions will come. This also has the facility to give you the annotation of sources which are linked to its call. This would be helpful specifically when you want to understand the logic of the function

2) Callgrind control: This subtool is a dynamic tool as it works dynamically on the running program and it can control the state of the program. This is very helpful when you want to observe the behavior of your program in a certain situation such as what happens in a particular function or what is the value of a counter when your function get a [articular input or satisfy a particular condition and is more of a debugging thing. You can get the stack trace of the called function to avoid all long list of stack trace when you program reaches termination. This gives you an advantage of not going through all the stack trace when your program stops.

D. KCachegrind

This sub-tool can be thought of as just a GUI layer over the cachegrind which could help any user visualize what is happening in a more interactive way which would be more easy to understand.

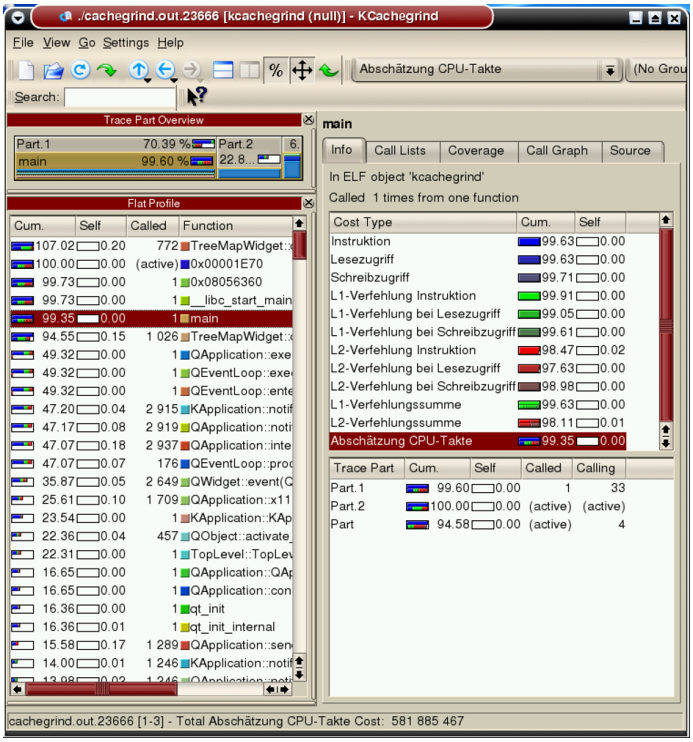


Fig. 9. KCachegrind

The GUI is based on Qt and QT4.x or higher is required for its support. There are some other run time dependencies like dot binary for the creation of call graph and for annotation purpose of the source code it needs objdump binary.

V. Results

Amount of time spent rendering an input file until it produces an output image which meets user requirements is called execution time. It may vary depending on size and quality of input file, hardware configuration, software parameter settings and Renderer in use. In pbrt and LuxCore, execution time could be anywhere from a few seconds to several hours. This section discusses the results obtained during testing and suggests the approaches for improvement in current ISA.

A. *PBRT*

Rendering process in pbrt produces an image file as an output by processing the input file using configuration specified in .pbrt file. In order to keep the execution time less, we decided to use the input file 'killeroo-simple.pbrt' , a sample scene provided along with the pbrt installation. Then we ran pbrt program multiple times for the same input file with and without Valigrid and gathered profiling data for analysis. The results were slow when the program was run with Valgrind, which is expected as Valgrind inserts its own instructions and modifies the application in use to perform profiling and advanced debugging. The results of the testing are shown below:

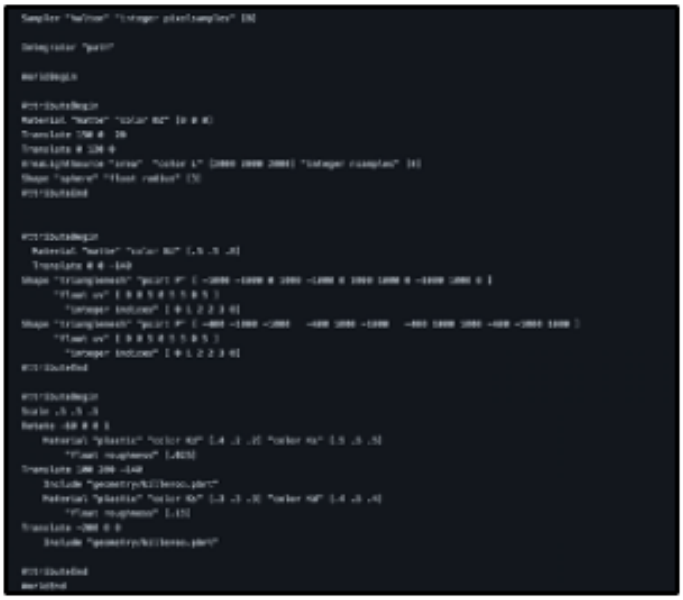


Fig. 10. PBRT Input File

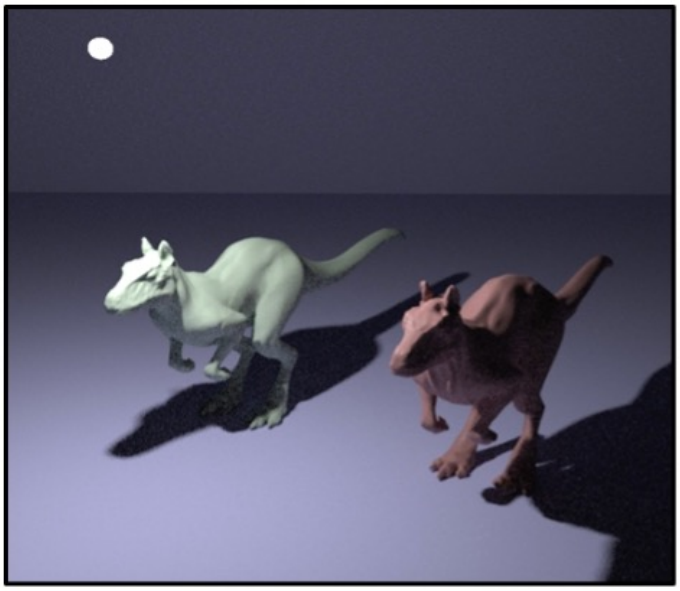


Fig. 11. PBRT Output File

*1) **Statistics**:*

- Execution time without Valgrind: 12.5 seconds
- Execution time with Valgrind: 1581.2 seconds
- Total Instructions: 112,443,107,170
- Function with most instructions executed: Accelerator::Intersect()
- No. of Instructions by function Accelerator::Intersect(): 2,811,077,679
- Percent of total instruction for Accelerator::Intersect(): 25.27

As per profiled data, cache is observed to be performing very well with a level cache miss rate of 4.30% only. These cache misses cannot be eliminated completely because of compulsory misses – a miss occurs when a block from memory is brought first time into the cache. Nevertheless, branch misprediction rate was 8.3%, approximately double the rate of cache miss rate and not performing quite as good as cache rates. Additionally, most of the execution time was spent performing Intersection calculation. Some of the frequently called Intersection functions were Intersect::Accelerator() followed by Intersect::Triangle() with percent share of 25.27% and 11.96% respectively.

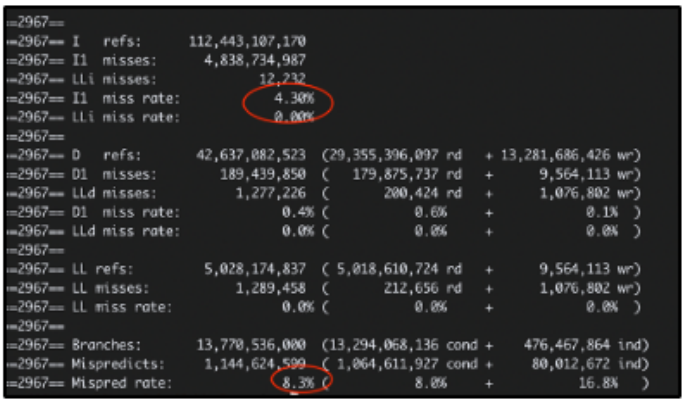


Fig. 12. Cache and Branch miss rate

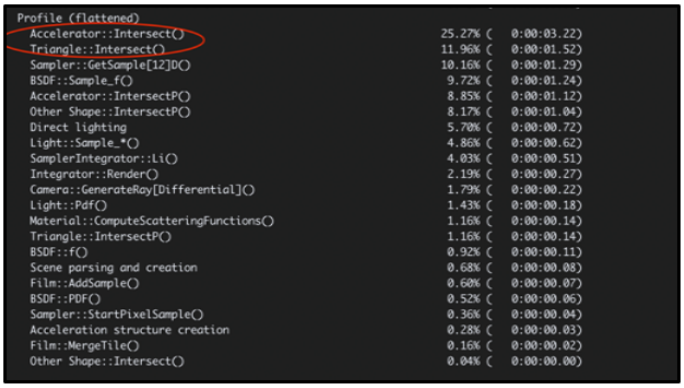


Fig. 13. Weightage of each function

Branch misprediction significantly reduces the performance, as in case, if any branch is mispredicted then the entire pipeline is flushed, which means a lot of wasted clock cycles. We figured out that execution time can be reduced by reducing the number of instructions that are required to compute ray intersection. Our approaches to include vectors in the current ISA will have less penalties, fewer branches and branch mispredictions, faster arithmetic calculations and less dependencies compared to non-vector ISA.

B. LuxCore

Unlike pbrt, Luxcore uses an UI interface and keeps displaying the image as rendering progresses over time. Image quality gets better with time and the user has to decide when

to stop rendering by setting a parameter 'halt time'. So, for the purpose of our analysis we decided to choose a rendering time of 30 minutes and then ran the Luxcore program for sample input files with Valgrind. The results of the testing are shown below:

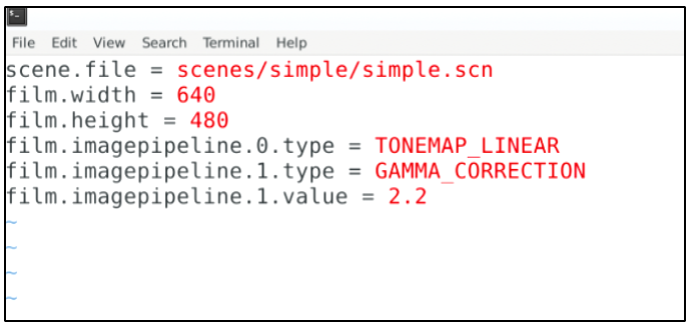


Fig. 14. LuxCore Input File

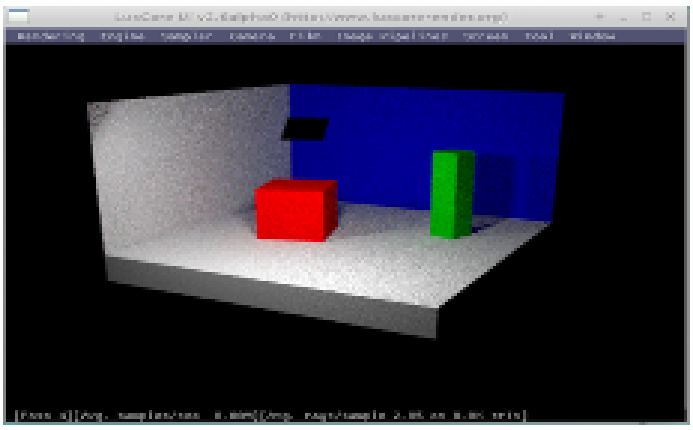


Fig. 15. LuxCore Output File

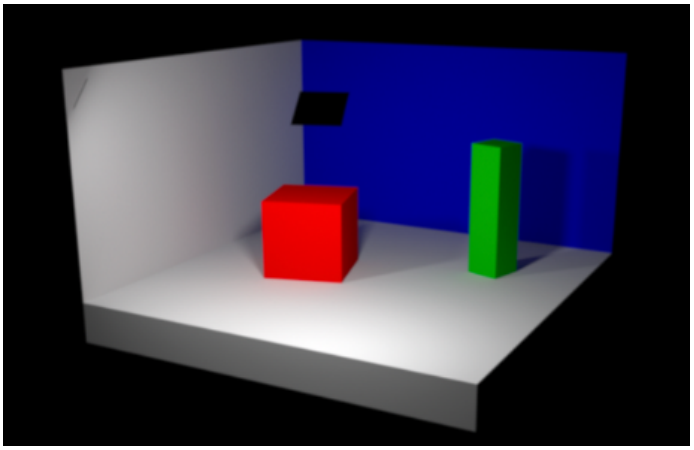


Fig. 16. LuxCore Output Without Valgrind

*1) **Statistics**:*

- Rendering time set by user: 30 minutes
- Total Instructions: 187,801,351,119
- Function with most instruction executed: RendFunc()
- Percent of total instructions for RundFunc(): 54.70

As per the profiled data, cache is seen to be performing very well with a level miss rate of 0.61% only. However, branch misprediction rate was 4.1%, which was significantly higher compared to cache miss rate but in general it was still low. Additionally, most of the execution time was spent in path tracing meaning finding position and size of light.

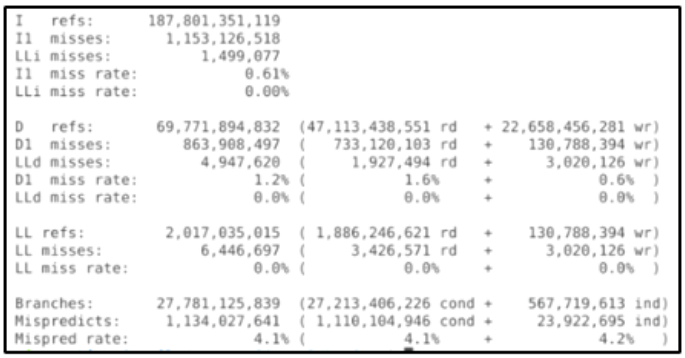


Fig. 17. Cache and Branch miss rate

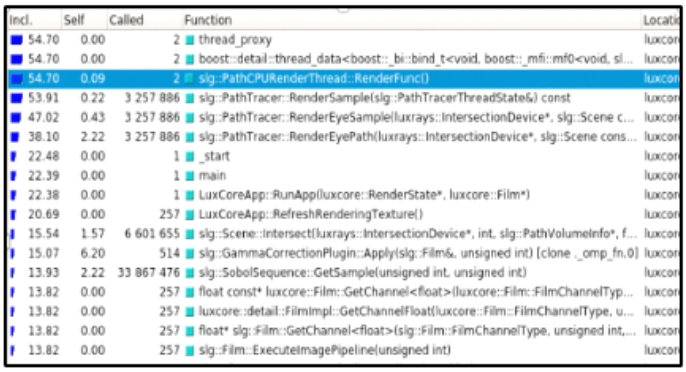


Fig. 18. Weightage of each function

The code in Luxcore is written to be multithreaded and all the instructions are divided into 9 threads. In multithreading, each thread runs in parallel to each other and shares the same code, data and heap segments but different stack segment and register values, which eventually make it a process. Instead of dividing the program into multiple threads, our approach is to create many processes and implement multiprocessing in current ISA, which is more suitable for rendering multiple frames simultaneously, eventually decreasing the overall render time.

VI. Analysis of Results

To analyze the result we have started making some meaningful relations and areas which could be giving some deviated result E.g. high cache, very high branch misprediction, bad use of resources through multithreading etc. For PBRT, the cache miss rate is 4.3 percent and for LuxCore it is 0.51% which is pretty decent and improving in these areas which looks like a very less scope won't bring any significant change as

these metrics were very good. Now, when you see the branch misprediction of both the rendering algo then it is 8.3 percent for PBRT and 4.1 percent for LuxCore. Now, again these still are under 10 percent and for Luxcore it is even under 5 percent and surely these can't go to zero percent so bringing any change in codes to improve branch misprediction would just give us a change of 5-6 percent which again is not a significant change but we still tried to look for the scope of improving branch misprediction in the code and also thought of looking into the architecture of code to find out if some particular kind of computer architecture can be well suited for these instruction sets.

While looking at the code of LuxCore we observed that there are various threads running in parallel and many of the threads are stuck in waiting conditions most of the time and we found this through the below shared image.

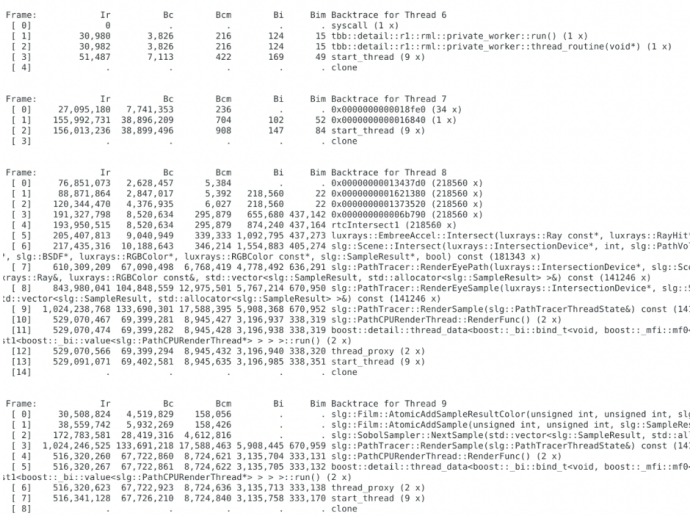


Fig. 19. Function called in each Thread

Then we started looking at the source of LuxCore to see the code segment that is causing this problem and we could see the while loop which is shared in the below shared image which is running in multiple threads and has to wait on the same resource and is waiting many times. In Linux, it is always recommended to have separate processes instead of separate threads to have a better resource utilization and performance.

```
for (u_int steps = 0; !boost::this_thread::interruption_requested(); ++steps) {
        // Check if we are in pause mode
        if (engine->pauseMode) {
                // Check every 100ms if I have to continue the rendering
                while (!boost::this_thread::interruption_requested() && engine->pauseMode)
                        boost::this_thread::sleep(boost::posix_time::millisec(100));

                if (boost::this_thread::interruption_requested())
                        break;
        }
```

Fig. 20. Code snippet of while loop

For the PBRT we didn't find anything that could bring any improvement in it execution time. So, we started digging more deeper into the source and tried to better understand its architecture and look into its primitive elements. And interestingly, we have observed following things in its source code.

- From the analysis we observe that the function Intersect() in PBRT consumes almost 46% of the total instructions executed during rendering process. - We go deep and analyzed the Triangle::Intersect() function and Accelerator::Intersect() function and find that these functions are using most of its operations on vectors and there are many inner function calls inside these functions where many local vectors are getting passed. - For Vectors Point3¡float¿ templated class was used which handles 3 vectors inside it and all math operations on those vectors were performed by sequentially placing the each element. - And almost 60% of the operations are done using call to overloaded operator like (+,-,*) of class Point3<float>

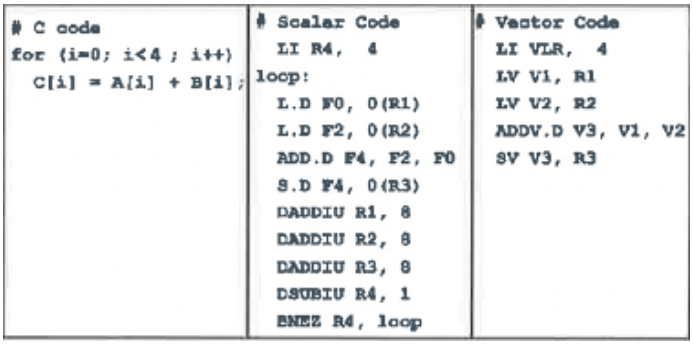
Check the below shared image to see where you can see all the elements of Point3 templates class and how operator are overloaded.

```
Point3<T> &operator+=(const Vector3<T> &v) {
    DCHECK(!v.HasNaNs());
    x += v.x;
    y += v.y;
    z += v.z;
    return *this;
}
Vector3<T> operator-(const Point3<T> &p) const {
    DCHECK(!p.HasNaNs());
    return Vector3<T>(x - p.x, y - p.y, z - p.z);
}
Point3<T> operator-(const Vector3<T> &v) const {
    DCHECK(!v.HasNaNs());
    return Point3<T>(x - v.x, y - v.y, z - v.z);
}
Point3<T> &operator-=(const Vector3<T> &v) {
```

Fig. 21. Code snippet of PBRT function

Now, to take advantage of this, we have thought of taking vector register which can handle all these operations in just one go and no separate loading would be required. Once we modify the instruction set register to handle these vectors then we can load the member variables of class Float3 into just one instruction. Instead of loading a,b,c separately in different instructions, we would be able to load abcd all together. Now, do note that the speed of memory bus and frequency of RAM are same so those factors also needs to be considered but definitely the instruction counter will be reduced which is a major factor in reduction of execution because these overloaded operations are major in the whole instruction set.

Major improvement will be seen in the ALU operation which will be loading all the 4 elements of class Float3 into just one register in one load operation for all function calls of class like add, subtract and multiply. It would require just one instruction instead of three. For example, take the example from the below mentioned image



```
# C code
for (i=0; i<4 ; i++)
  C[i] = A[i] + B[i];
```

```
# Scalar Code
  LI R4,  4
loop:
  L.D F0, 0(R1)
  L.D F2, 0(R2)
  ADD.D F4, F2, F0
  S.D F4, 0(R3)
  DADDIU R1, 8
  DADDIU R2, 8
  DADDIU R3, 8
  DSUBIU R4, 1
  BNEZ R4, loop
```

```
# Vector Code
  LI VLR,  4
  LV V1, R1
  LV V2, R2
  ADDV.D V3, V1, V2
  SV V3, R3
```

Fig. 22. Comparison of Scalar and Vector code

Figure 22 is divided into three section, first section is the source code in C and second section shows the instruction set formed if scalar registers are going to get used and third section shows if we used vector register. We can see that just a simple addition when performed will take 1 load instruction and 9 instruction for addition per iteration which in total for 4 iteration would be 9*4+1=37 instructions for normal architecture. For the vector architecture it will simply be handled in just 5 instructions for all the 4 iterations. This definitely is a drastic improvement for the same CPU with same CPI and just a vector Instruction Set Register. We can calculate the total speed up using the Amdhal's law which is as follows:

SpeedUp=(InstrcutionCount[scalar]*CPI[scalar]* CycleTime[scalar])/(InstructionCount[vector]*CPI[vector]* CycleTime[vector])

CPI[scalar] = CPI[vector] and CycleTime[scalar] = CycleTime[vector]

SpeedUp = InstructionCount [scalar] / InstructionCount [vector] = 37/5 , approx 7.5

Hence, we can say that this speed is definitely huge. This is the ideal scenario since we have not considered the memory bus speed and RAM frequency but definitely change in ISA would bring speedup closer to this.

VII. Conclusion

For pbrt, we used the implementation given in "Physically Based Rendering: From Theory To Implementation", by Matt Pharr,Wenzel Jakob, and Greg Humphreys and LuxCoreRenderWiki for Luxcore render. To evaluate the implementation, performance and hardware required for optimal execution, we analyzed the statistical data collected during testing with the help of Valgrind profiling tool. After analyzing various reports generated by different tools provided by Valgrind, we observed that cache is already optimized to its best possible level and there were almost 0% cache misses and cannot be optimized further. However, after several executions we realized that there was a little scope of optimization in branch mis-predictions which can further be optimized to improve overall speedup of the renderer. Currently, pbrt and Luxcore architecture performs a branch mis-predictions rate of 8.3% and 4.1% respectively, which can further be reduced to almost 0% with the help of better branch prediction techniques. Our implementation involves vector architecture with an increased number of iterations to help process the image pixel by pixel, a tournament predictor with inner level loop predictor can be very useful to reduce branch mis-prediction and improve performance in the organization. Even if we bring branch-misprediction rate close to 0%, we could not observe the desired performance to result in immediate rendering of images. Some of the high configuration images took more than 10 hours to render in the pbrt system, which is too slow in case of image rendering. After closely analyzing the scenario and implementation for multiple iterations, we observed that almost all the operations carried out on scaler architecture resulted in huge performance setbacks. Thus, to reduce the number of instructions and clock cycles, all the operations require carrying out vector operations. This is major improvements toward optimization as it will reduce the number of operations required for rendering and eventually increase the speedup. However, introduction to vector register may impact the modules which are already performing well in the current architecture. Thus, there is a need of modifying the cache and other architecture in order to handle vector operations. More about this is discussed in next section.
After performing multiple tests and analysis, it was observed that Luxcore renders faster and produces better output compared to PBRT-v3.

VIII. Future Scope

The paper mainly discuss the implementation of PBRT and LuxCore and suggests the ways to optimize with respect to introduction of vector registers to the current architecture, reducing the number of operations and branches, and bringing branch misprediction rate close to 0%. These theoretical optimizations may impact some of the better performing modules of the current architecture. These new changes require the incorporation of many other changes within the architecture which are already working efficiently. For example, cache is already preforming well in PBRT and LuxCore but on incorporation of vector registers, cache should be modified to preform vector operations with the same efficiency. Making cache more set associate with the size equal to vector size, results in faster calculations. Implementing such techniques may give better performance in rendering compare to current architecture.

References

[1] https://luxcorerender.org/physically-based-rendering/
[2] https://www.pbrt.org/
[3] https://github.com/mmp/pbrt-v3
[4] http://dirkmittler.homeip.net/blog/archives/6704more-6704
[5] https://pbr-book.org/3ed-2018/contents

[6] https://en.wikipedia.org/wiki/Physically_based_rendering
[7] https://luxcorerender.org/
[8] https://wiki.luxcorerender.org/LuxCoreRender_Wiki
[9] https://github.com/LuxCoreRender/LuxCore
[10] https://cmake.org/
[11] https://en.wikipedia.org/wiki/Rendering_(computer_graphic)
[12] https://azure.microsoft.com/en-us/services/app-service/web/
[13] https://github.com/seed-labs/seed-labs/blob/master/manuals/cloud/seedvm-cloud.md
[14] https://github.com/seed-labs/seed-labs/blob/master/manuals/cloud/create_vm_azure.md
[15] https://valgrind.org/docs/manual/cl-manual.htmlcl-manual.functionality
[16] http://kcachegrind.sourceforge.net/html/Home.html
[17] https://www.sciencedirect.com/topics/computer-science/rendering-algorithm