University of California

Los Angeles

A Multi-Accelerator Architecture for Photon Mapping

A dissertation submitted in partial satisfaction

of the requirements for the degree

Doctor of Philosophy in Computer Science

by

Seung hyun Pan

2014

Abstract of the Dissertation

A Multi-Accelerator Architecture for Photon Mapping

by

Seung hyun Pan

Doctor of Philosophy in Computer Science

University of California, Los Angeles, 2014

Professor Miloš D. Ercegovac, Chair

Real-time rendering of photorealistic images has always been an important goal in Computer Graphics. The most computationally expensive part of this process is obtaining the effects of global illumination. Photon mapping is a well-known technique for calculation of realistic global illumination, and also shows characteristics which we believe make it favorable for dedicated hardware acceleration.

Online arithmetic is a digit-serial form of arithmetic, where input vectors are processed from the most significant digit down to the least, and the result is also produced one digit at each step. Pipelined online arithmetic circuits are extremely regular while only requiring simple calculations between registers, which allows for high clock speeds and low power dissipation with a huge potential for parallel execution.

Combining these two concepts, we design and evaluate MAPM (Multi-Accelerator for Photon Mapping), a multi-accelerator architecture that employs pipelined online arithmetic to accelerate the two most time consuming operations in photon mapping: the tree search and shader operation. On a VHDL implementation, we perform behavioral verification using ModelSim, examine hardware cost with Synopsys tools and evaluate throughput gain and scalability of the architecture

using a custom built cycle-accurate simulator based on the Intel Pin tool.

By employing two MAPMs set to a configuration of 16 Tree Search Modules, 16 Shader Operation Modules and 2 Shader Operation Accelerators per Shader Operation Module, we observed a throughput increase of 1384× over an optimized software setup, and an increase of 4.78× over a recent MPSoC implementation. This is achieved using an acceptable hardware cost of 28.8% of the bandwidth, 22.2% of the area, and 5.6% of the power consumption of the low-end Intel Celeron G1820T.

The MAPM also shows a significant reduction in power dissipation. Compared to a conventional parallel circuit with equivalent functionality, the MAPM showed a synthesizable clock speed at about 3.5×, dynamic power consumption of 0.104×, and area cost of 1.799×.

The dissertation of Seung hyun Pan is approved.

Yuval Tamir

Glenn Reinman

Behzad Razavi

Miloš D. Ercegovac, Committee Chair

University of California, Los Angeles

2014

To my father and mother,

who have nurtured me with unending love

into the person I am today.

For that, I am eternally grateful.

Table of Contents

List of Figures

List of Tables

Acknowledgments

First and foremost, I am sincerely grateful and forever indebted to my advisor, Professor Miloš D. Ercegovac, for without him none of this work would have been possible. Ever patient and focused, with a sparkling sense of humor, he was an inspiration and role model throughout my grad school years. The years of learning from him has indeed been an awe-inspiring experience which I shall never forget all my life.

I also wish to thank my committee Professors. I have learned so much from all my classes with Professors Yuval Tamir and Glenn Reinman, both attending and working as a TA. Thanks to Professors Bezhad Razavi and Dejan Marković, who were kind enough to attend my oral exams and offer their insights and moral support. Their feedback was crucial in raising the scientific integrity of my dissertation.

A huge thank you goes to Shawn Singh, whose joint work with me has been the firm basis of my dissertation work. On top of that, his friendship and hours of work and discussion that we shared was an essential part of this project. I also extend thanks to my labmates and fellow CS students past and present, wonderful people who would always help me with anything and everything without question; in no particular order, Suk-bok Lee, Seung Hoon Lee, Sung Chul Choi, Pouya Dormiani, Hojat Parta, Seok Won Heo and Young-kyu Choi.

Special thanks to my family also, whose unending love is always a warm haven even in the gloomiest of times. My dear mother and father who are always worrying over and praying for me from afar, and my only brother who I know always has my back. And my profound gratitude to my fiancée, who has stood beside me during all the dark hours, even through the burden of her own inspiring research work. Without her support, there is no doubt that this would have been a much

harder and less successful journey for me.

There are countless people that have helped and supported me throughout my grad school years, so many that I do not have enough space here to thank individually. Also for anyone that I may have carelessly overlooked, please know that you all have my most sincere word of thanks, and no one was left out on purpose. This dissertation is not a mere token of my personal accomplishment; it is truly a monument that represents how fortunate I am to have all the great people that are around me.

Vita

2006 B. S., Electrical Engineering, Seoul National University
Seoul, Republic of Korea.

Publications

S. Singh, S. h. Pan, and M. Ercegovac, "Accelerating the Photon Mapping Algorithm and its Hardware Implementation," in *Application-Specific Systems, Architectures and Processors (ASAP), 2011 IEEE International Conference on*, September 2011, pp. 149–157.

CHAPTER 1

Introduction

Rendering photorealistic high-quality images in real-time has long been an important and classic goal in Computer Graphics. Due to a rising demand on applications such as interactive entertainment, virtual reality, and 3D displays, the need for fast high-quality image rendering is rapidly on the rise.

In general, the mechanism of how light travels through the physical environment and reaches the optic nerves in the retina of human eyes is well known. To achieve photorealistic rendering, it is necessary to calculate the effects of light transmission to emulate the physics of light, by a precise yet feasible method. The main obstacle is the massive amount of calculation that is required in order to obtain high-quality rendering. The main bulk of the calculation load is the process required for obtaining the effects of global illumination. To calculate this, one needs to consider light reflections from all parts of the scene that end up at points visible in the final image.

Current solutions usually fall in one of the two groups; one, they offer high-quality imagery but is too slow for real-time, or two, their throughput is good enough for real-time but the light emulation is approximated and thus produce images that are not photorealistic. The two most common rendering algorithm classes are rasterization-based approaches and ray tracing-based approaches [59]. The rasterization-based approach is a very fast but physically approximated method, where it goes through each polygon in the scene one by one to determine the color of each pixel in the final image. By making use of the inherent parallelism

in this operation, rasterization has seen a lot of success by running on graphics processing units (GPUs) for massively parallel execution. On the other hand, ray tracing-based methods emulate physical light rays more accurately, and can produce extremely realistic light effects. This makes ray tracing a better candidate for the goal of high-quality global illumination. The main drawback of ray tracing algorithms is that it is comparatively slower than rasterization methods, where even with all the algorithmic advances and improved hardware support, current performance of ray tracing is still a few magnitudes away from our ultimate goal of real-time photorealistic rendering.

Photon mapping [28] is a ray tracing-based approach which we believe to be an efficient method for global illumination calculation, and also shows promise of showing real time performance [52]. The focus of this dissertation will be on the design and verification of an application-specific accelerator for obtaining massively parallel execution of the photon mapping rendering algorithm. By using digit-serial online arithmetic, we can process a large amount of pipelined concurrent operations, in addition to parallel execution using multiple instances of execution modules. Additionally, the simple and structurally regular circuits of online arithmetic allows for a fast clock speed and low power consumption.

1.1 Main research problem

Regarding our final goal, let us examine some numbers regarding actual performance for current systems. Full HD, which is also referred to as the 1080p, is a widely used mode for screen resolutions. This uses a widescreen aspect ratio of 16:9, with a resolution of 1,920 pixels wide by 1,080 high, which is 2,073,600 pixels total. Full HD is a commonly used standard for various multimedia applications, including digital films, HDTV, and the most recent generation of video game consoles.

For photon mapping in general, the number of final gather rays (FGRs) that need to be calculated for each pixel is around 600 to 4,255 as noted in [24]. From these numbers, we can calculate the maximum number of total rays for Full HD to be $2,073,600 \times 4,255 = 8,823,168,000$, which is almost 9 billion rays. If we can get a throughput capable of calculating around 10 billion rays per frame at around 20-30 frames per second, this would be enough performance to achieve real-time rendering for this resolution. We can consider this as the minimum performance required for real-time photorealistic image rendering in Full HD.

As a side note, future applications will need to consider 2160p (4K UHD) and 4320p (8K UHD). 2160p has a resolution of 3,840 pixels wide by 2,160 high, which is 8,294,400 pixels total, which is 4 times the count of 1080p. And 4320p has a resolution of 7,680 pixels wide by 4,320 high, which is 33,177,600 pixels total, which is 16 times the count of 1080p. These will obviously have a higher ray process requirement compared to those of the 1080p.

Regarding the goal of 10 billion rays per frame, let us consider the following points:

- As can be seen in the International Technology Roadmap for Semiconductors annual report [27], speedup obtained from the advancement of CMOS technology is subtly but surely slowing down. This trend is evident as a lot of recent processors do not obtain speedup from increased clock time but through an increased number of cores or processing units, and increasing the amount of parallel work that the processor can efficiently handle at the same time. Simply waiting for the hardware to improve to the necessary performance point is not a viable solution, even before we consider the amount of time it will take to get there.

- Looking at [59], we can see that a wide range of algorithmic improvements have been thoroughly explored for ray tracing. However, the amount of re-

quired processing is still too much to overcome by purely algorithmic methods. As mentioned in [63], performances of up to several hundred million rays per second is expected to be reachable in a few years, but this still falls a few magnitudes short of the performance goal for real-time photorealistic rendering in Full HD.

- There have been attempts to implement photorealistic photon mapping in real time as we shall examine in more detail in 2.2, but the current methods still come short of our performance goal.

- While parallelism will definitely be one of the techniques that we will need to utilize, the work necessary for the handling of each individual thread is far too complex and slow that this performance goal cannot be achieved by simply employing a large number of duplicate execution modules that are avaliable.

Considering these points, to move closer to realizing the goal of real-time photorealistic image rendering, one of the the main issues that we need to tackle is improving the efficiency of resource usage overall, including power, area, and memory bandwidth.

In order to achieve this goal, we propose an accelerator architecture with application-specific modules which can speed up each individual thread, and can be run in parallel to get high throughput. The main goal of this dissertation is to design and evaluate an application specific Multi-Accelerator for Photon Mapping (MAPM) architecture.

1.2 Contributions of this dissertation

In this dissertation, we develop and analyze a hardware design for accelerating the 3D rendering algorithm of photon mapping.

For the realization of this end target, we worked on the following tasks.

1. **Novel architecture and design** The MAPM focuses on the back-end part of the photon mapping process, where all data structures are complete and processing them is all that remains. The MAPM is a co-processor like structure; it shares data paths to the main memory with the CPU, receives new instructions from the CPU, and outputs the results to the frame buffer or the main memory. For the detailed design, we need to consider where the actual calculation is processed, which data is utilized at that point, and all required data paths needed to supply this information.

   The Tree Search logic module design is adapted from [47], and online multiplier and divider designs are adapted from [16].

2. **HDL implementation** After the details of the design are finalized, it is vital to make sure the design functions as intended. To do this, the MAPM design is implemented using VHDL. Key components have gone through rigorous behavioral verification with ModelSim using test benches with Python generated inputs.

   After making sure that the functionality of the design is valid, we obtained ASIC hardware cost numbers by synthesizing the VHDL designs using Synopsys Design Compiler.

3. **Cycle-accurate simulation** To examine how well the MAPM scales, and to obtain more detailed performance numbers from cycle count, we built a cycle-accurate simulator. The simulator was created using Intel Pin libraries [44], an instrumentation tool that allows insertion of arbitrary C++ code in an executable. The Pin tool inserts calls to the simulator into a software photon mapping executable code created in [49], and the simulator returns exact cycle count and cache simulation results.

4. **Putting it all together** With all the information from the tasks listed above, we have enough information to evaluate the MAPM by making detailed comparisons with other recent systems and photon mapping implementations. We compared hardware cost to recent CPUs to verify whether the MAPM's hardware cost is affordable. Also, we evaluated the performance of the MAPM by looking at throughput numbers and comparing it to other recent photon mapping implementations.

5. **The results** In brief, using two MAPMs with a configuration having 16 Tree Search Modules, 16 Shader Operation Modules and 2 Shader Operation Accelerators per Shader Operation Module demonstrated a throughput increase of 1384× over the optimized software implementation of [48], and a throughput increase of 4.78× compared to the MPSoC setup in [17]. This is achieved while only using 28.8% of the bandwidth, 22.2% of the area, and 5.6% of the power consumption of the low-end Intel Celeron G1820T. These results represent the fastest throughput of all photon mapping architectures known to us.

   The MAPM also shows a significant reduction in power dissipation. Compared to a conventional parallel circuit with equivalent functionality, the MAPM showed a synthesizable clock speed at about 3.5×, dynamic power consumption of 0.104×, and area cost of 1.799×.

1.2.1 Collaborator contribution

This work is based on research in photon mapping architectures investigated jointly with Shawn Singh in [49]. Ideas and results from his doctoral dissertation [47] have been used in the design of the MAPM. In particular, we used his photon mapping C++ code for the software executable used in the cycle accurate simulation, and his Pin code written for [49] served as a great learning example

for writing the cycle-accurate simulator.

1.3 Organization of dissertation

The organization of the dissertation is as follows. Chapter 2 examines some background and related work. Chapter 3 briefly introduces the photon mapping algorithm and presents the details of the MAPM architecture. Chapter 4 looks at evaluation methods and results, and Chapter 5 provides a summary and future work.

CHAPTER 2

Background and related work

The problem of rendering a 3D scene can be described as follows. Given a description of a scene, where we are given the objects in the scene, light sources, and a camera, we wish to output an image. This image is an array of color information which reflects what the scene will look like as viewed from the camera location.

For this rendering problem, there are two standards we need to consider, which are the quality of the final image, and the time required to produce the image. In general, higher photorealism would require more complex calculations which leads to more time required for processing. Leaving out part of the required processing, or approximating it to reduce time would lead to a drop in the quality of the final image. There already exist algorithms which are oriented to enhancing one of the criteria, but at the cost of a high quality drop in the other criterion.

With this in mind, we examine some of the algorithmic approaches to rendering and offer some justification on our selection of photon mapping.

2.1 Rendering algorithms

There are various classes of algorithms which try to solve this rendering problem. Each has its own strengths and weaknesses, and here we shall examine the positive and negative aspects of photon mapping and why we think photon mapping is a great candidate for real-time photorealistic rendering.

2.1.1 Rasterization

Rasterization is a widely used method of rendering due to its high speed. Most of the GPU architectures on the market are based on this.

For each pixel, the rasterization pipeline [18, 39] goes through each polygon in turn and determines whether the polygon intersects a straight line from the camera, and if it does, which one is the closest to the camera point. Once this is determined, the color of the closest point is calculated using shader functions, and the result becomes the pixel color for the final image. A large part of this can be performed independently of each other. For example, comparing the distance to each polygon with the smallest value found so far does not require data from other polygons.

The GPU architecture takes advantage of this instruction independence by having a huge number of simple cores where each instruction is passed through in parallel. Each instruction is assigned to a core on the GPU, leading to massively parallel execution. This leads to a very high throughput performance, to the point where it is possible to get real-time frame rates.

However, rasterization is not a good solution for photorealistic rendering due to a few drawbacks:

1. When using only rasterization, it is difficult to account for secondary rays and physically correct global illumination. The main reason is that rasterization is optimized to calculate visibility of the entire scene from a single camera point, and to obtain illumination from secondary rays, a large number of multiple passes through the pipeline is necessary. This would weaken the timing performance of rasterization, and still likely not yield physically accurate global illumination.

   Another way to obtain global illumination using rasterization would be to have some approximation at the shader phase, but again, this would result

in a physically inaccurate rendering and would yield lower quality images.

2. The workload does not scale well due to the fact that rasterization needs to go through all polygons to determine visibility from each pixel, therefore having a linear dependency on the number of polygons in the scene. There has been some previous work that tries to deal with this issue such as the irregular z-buffer [29], but a lot of the calculation still ends up not being used in the final image. This affects the scalability of rasterization in terms of the the number of polygons in the scene.

In short, rasterization offers many speed benefits, but for the goal of rendering physically correct photorealistic images, is not a well-suited class of algorithm.

2.1.2 Ray tracing

This brings us to ray tracing, which is a method much more appropriate for physically based rendering than rasterization.

Ray tracing is a rendering method where the actual light paths are traced through the scene using a general ray path calculation. Compared to other rendering methods, ray tracing has the benefit of being able to account for all possible light paths, and this is done with a generalized and unified algorithm, i. e. tracing rays. Ray tracing closely emulates how light physically operates in the real world, which makes it easy to account for various illumination effects such as reflection, refraction, scattering and dispersion. This allows ray tracing to achieve levels of realism not possible in rasterization, making it the most suitable method for photorealistic rendering.

Ray tracing can also reduce the amount of wasted calculation, due to being able to pinpoint parts of the scenery that is directly visible from the camera. From this, ray tracing can eliminate unnecessary processing for areas not used in the final image. This is a weak point for rasterization, where it is difficult to identify

which calculations will be used for the final image, and the workload increases linearly with the number of polygons in the scene.

However, the higher quality of rendering comes at the cost of a much larger computation load compared to rasterization. Looking at some recent work, RPU (Ray Processing Unit) by Woop et al. [64] shows a hardware design for an architecture with programmable shaders. On a 66 MHz FPGA, the RPU shows a performance of up to 20 FPS (frames per second) for an image resolution of 512×384 pixels using primary rays only. This is around 100 million rays/s for simple scenes with no lighting and shading, and 15 million rays/s for moderately complex scenes with simple lighting. OptiX, a general purpose ray tracing engine by Parker et al. [40] that utilizes GPUs for ray tracing, shows performance of 1.5 FPS on the GTX480 for path tracing, and 4.5 FPS for the less-accurate Whitted-style tracing. While this is quite an advancement, for the goal of photorealistic real-time rendering for Full HD, the state of the art still comes a bit short.

2.1.3 Photon mapping

Photon mapping is a ray tracing based method that is effective for calculating global illumination. It was originally proposed by Jensen [28] as an algorithm for offline rendering. Since then, various research has been carried out, including exploration of algorithmic improvements such as reverse photon mapping [24] and SIMD (Single Instruction, Multiple Data) packets with data reordering [48], and analysis on required cache bandwidth as in [51]. Recently, Singh has shown in his dissertation [47] that compared to other ray tracing methods, photon mapping can produce better quality images from a much smaller number of rays, and it has great potential for a speedup similar to rasterization, as the main workload can be divided up into fine-grain operations. This speedup was explored in the collaborative work [49] where the simulation of an online hardware accelerator shows a potential speedup of 100× throughput compared to the software performance.

Instead of treating particles as virtual light sources, photon mapping uses density estimation [46] to compute contribution to a camera point more effectively. By actually creating and bouncing photons around the scene to emulate light transport, photon mapping is also capable of producing a wide range of complex illumination effects. Some examples of light effects that can be captured efficiently using photon mapping are caustics (concentrated patches of light of nearby surfaces), diffuse interreflection (light reflection from diffuse surfaces, example: color bleeding), and subsurface scattering (light enters material and is scattered before being absorbed or reflected).

Photon mapping has the following strong points:

1. Compared to other ray tracing algorithms, photon mapping requires a smaller number of rays. As explored in detail in [47], photon mapping can produce a similar or better quality image from processing an order of magnitude smaller number of rays. By using an efficient search data structure such as KD-trees (k-Dimensional trees) [9] to sort through the photons, photon mapping can calculate global illumination by processing a fewer number of rays compared to that of a ray traced image of similar quality.

2. The main bulk of photon mapping can be divided into fine-grain operations. The biggest reason for the fast processing speed of rasterization is that each polygon and shader process can be divided up into small independent sections that can be run in a massively parallel manner. Photon mapping is similar in that each search into the data structure and each shader operation can be processed independent of each other. By utilizing this parallelism, photon mapping can be efficiently mapped to separate many-core accelerators to achieve high throughput.

3. The number of photons is unlikely to be intractable. Each photon carries information that is highly likely to be used in the final image, and due to

density estimation, processing photons is much more efficient than processing light rays of the same effect [47], allowing us to render higher quality images from a smaller number of samples.

These characteristics all combine to allow a smaller workload, and can be mostly achieved without having to sacrifice the quality of the final image. Therefore, we assert that development of an accelerator to take full advantage of the fine-grain characteristic of photon mapping would be an important step towards real-time photorealistic rendering.

However, there are still a number of obstacles that prevent us from having real-time Full HD photon mapping. For one, processing each ray in photon mapping is more complex compared to other ray tracing methods. Each separate shader operation needs to process a handful of floating-point multiplications and divisions, and simply using conventional floating-point units will not be enough to yield the massive throughput necessary for real-time rendering. Also, photon mapping relies on the KD-tree data structure to accelerate the photon search process and reduce the time spent in searching for shader operations. Unfortunately, building the tree is relatively slow and updates are costly, compared to other data structures used in rendering such as grid accelerators or bounding volume hierarchies [59]. In general, with more time spent in building the data structure, a better search performance can be obtained. The tree build is not a focus of this dissertation, however a lot of research is being done to speed up the KD-tree build, such as speeding up the surface area heuristic used in the tree build [26,42], utilizing multi-threading and multi-core parallelism [45], adaptive fast rebuilding from the scene graph [11], or motion decomposition [21].

2.2 Hardware acceleration for photon mapping

Here we examine some previous work on using hardware to accelerate photon mapping.

In [43] by Purcell et al., photon mapping was implemented on the GPU architecture. Unfortunately, as it was difficult to fit the main search data structure on the old, less programmable GPU architecture, a different search method had to be used, and this affected the overall quality of the image.

Work by Larsen and Christensen in [32] implements photon mapping on a combination of a CPU and GPU. This work uses textures to store global illumination information, and they reduce the number of necessary samples by tracking coherence for frame to frame photon tracing. This works in 35+ FPS for the images in the paper at 512×512, but the need to have multiple passes for each global illumination texture may limit the scalability to larger resolution images.

McGuire and Luebke has an interesting paper in [36], where an algorithm named Image Space Photon Mapping was implemented on an 8-core CPU and a GPU. Rasterization methods are used to produce photons and query points very quickly. The rendering rate is very impressive at up to 26 FPS for Full HD images. However, this is only an estimation of physically based rendering, and no secondary rays are traced, resulting in a low quality image and limiting the method's scalability to photorealism.

T-ReX by Kim et al. [30] also uses a heterogeneous setup of a CPU and a GPU to implement photon mapping. Using decoupled data for each, and progressive photon mapping methods [22], they were able to reduce required data transfer between the CPU and GPU and obtain performance numbers of 3-20 million rays/s with response time at 15-67 ms.

Work by Fallahpour et al. [17] presents a homogeneous MPSoC implementation for photon mapping, which includes the full process including all data structure

builds. The maximum performance obtained here is about 639 million cycles to render a 320×240 resolution image using an 8x8 ARMv4 core network.

As far as we know, there has not been another approach for accelerating photon mapping using application-specific hardware that utilizes fine-grain parallelism, without heavy approximations which affect the quality of the rendering. In this work, the MAPM approaches this problem using online arithmetic.

2.3 Previous work on photon mapping using online arithmetic

Here we look at previous work on accelerating photon mapping using online application-specific hardware.

In the collaborative work [49], we built on the main idea of focusing on the fine-granularity of photon mapping using online arithmetic. We drew the large picture of the co-processor accelerator architecture and implemented the Tree Search Accelerator (TSA) and Shader Operation Accelerator (SOA) modules in detail using VHDL. Using Synopsys tools, hardware cost comparison was done with conventional circuits generated using FloPoCo [10]. Software simulation was done using Pin [5] to obtain instruction count.

The hardware cost comparison at 200 MHz showed a power cost of about 20% and area cost of about 130% compared to conventional circuits. Also, the online implementation synthesis could be synthesized up to 1.5 GHz, while the parallel conventional implementation showed a much slower clock speed. Perfomance-wise, simulation results showed a 100× factor throughput improvement over a pure software implementation in [48] while needing an estimated power cost of less than 200 mW, and area cost of around 1% of 150 mm^2.

In Singh's dissertation [47], in addition to the above collaborative work, he

evaluates how photon mapping compares to other ray tracing algorithms in terms of rays traced and output quality, and also shows mathematical proof of the fine-granularity of photon mapping. Also covered in this work is his methods of optimizing the photon mapping algorithm using SIMD data packets with data reordering to enhance cache behavior.

This dissertation expands and elaborates on these previous works by:

1. **Detailed architecture design** In previous work, no details of the architecture surrounding the TSA and SOA were implemented; power and area cost did not take these circuits into account. In this work, all data paths to memory and between modules, and the instruction flow of input parameters through the accelerators are figured out in detail. The arithmetic inside the SOA is implemented in detail, including calculation trees and all bit shifts needed to match the input data range of online operators. All necessary modules are shown with working details so that the TSA and SOA modules created in [49] are supplied with the necessary information and can process the tree search and shader operations.

2. **HDL implementation** With the detailed design for the MAPM figured out, we can now better evaluate the hardware cost. All key modules with any updated design details are implemented in VHDL and their functionality is verified using ModelSim. The SOA especially needs meticulous evaluation, using a large amount of test benches generated with the help of Python [6]. Again, we enlist the help of FloPoCo to implement a conventional parallel circuit with the same functionality, and Synopsys tools to obtain hardware cost numbers.

3. **Cycle-accurate simulation** Cycle-accurate simulation is necessary to evaluate scalability of the MAPM, along with obtaining cycle count and cache bandwidth. The simulator was created in C++ using Intel Pin libraries [44],

carefully taking into account how the internal data would flow through each module in the detailed design.

4. **Evaluation** With the information we obtain from HDL implementation and cycle-accurate simulation, we have enough information to evaluate the scalability of the MAPM architecture, and compare the performance and hardware cost of the MAPM with other recent solutions.

2.4 Online arithmetic

Online arithmetic [16] is a digit-serial form of arithmetic, where input vectors are processed digit by digit from the most significant digit down to the least, with the results also being produced one digit at a time. Due to the the digit-serial characteristic of online arithmetic, by using a pipelined form, it is possible to issue one application specific operation every clock cycle. This gives us a large concurrency potential at the operator level even before we consider parallelism by using duplicate accelerators.

2.4.1 Timing of online arithmetic circuits

There are two components to the delay of a single online arithmetic operation [16].

1. **Initial delay**: The initial delay δ is also called the online delay. This is the number of additional digits required to determine the value of the first result digit. The first output digit is delivered after $\delta + 1$ cycles.

2. **Time to deliver n output digits**: After the initial delay, one output digit is delivered every cycle. For the circuit to output n output digits, n cycles is required.

Therefore the cycle delay of a single online operand is $T_n = \delta + 1 + n$. These numbers may seem to suggest that the timing characteristics of online arithmetic is worse than traditional arithmetic circuits. However, the main benefit of online arithmetic is that we can easily pipeline the architecture in a digit-serial manner. Combined with the fact that pipelined online arithmetic consists of a very regular structure, with simpler circuits between registers allowing a faster clock speed compared to traditional circuits, this allows for a very high throughput overall.

Also, at first glance, online arithmetic may seem to be adding cycle latency for a single operation. However, the faster clock speed that is possible with online arithemetic more than makes up for the added cycles, and as a result it is possible to actually reduce the latency of a single operation in terms of time, depending on the synthesized clock speed.

Another way that throughput of online circuits can be enhanced is by replication of online modules for calculations with no operation-level feedback loops. For iterations such as $y(i) = f(y(i-1))$, it is possible to push the throughput rate up to

$$\frac{1}{\text{cycle time}}$$

at the cost of more replicated online modules. This is highly useful as the designer can choose any point between the two extremes for a tradeoff between throughput and hardware cost.

For a compound network of operators, more cycles can be saved for a single operation by starting the next operation before the previous one is finished. Online arithmetic works in a digit-serial method where the inputs are fed into the circuit one digit at a time, and the outputs are also produced digit by digit.

Given the following compound network,

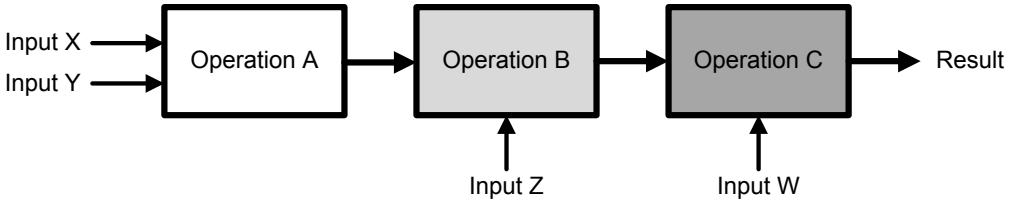


Figure 2.1: An example compound operation

with a conventional parallel arithmetic architecture, we need to wait until the result of Operation A is available. Only then can we start the next operation, which is Operation B. Therefore, the timing diagram for a parallel circuit would look like something shown in figure 2.2. Here, t_{PA} is the time it takes for Operation A to finish in the parallel operator.

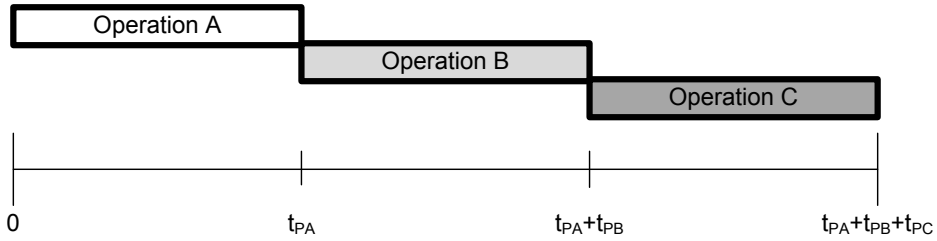


Figure 2.2: Timing diagram of conventional parallel arithmetic

On the other hand, if we use online arithmetic, the second Operation B will be able to start as soon as the most significant result digit is obtained from the previous operation. The delay for this is the preprocessing delay δ. So the timing diagram will look similar to this.

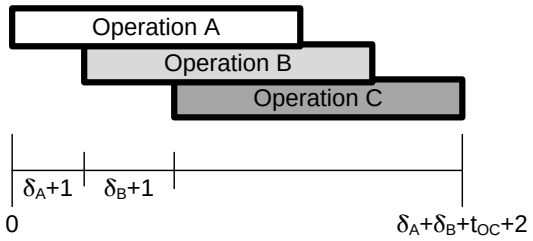


Figure 2.3: Timing diagram of online arithmetic

It is easy to see that with online arithmetic, the more operators we have in the compound operator, the more cycles we can save from this digit-serial form of execution.

2.4.2 Online arithmetic applications

Online arithmetic has been implemented for a large number of applications.

Online algorithms for fixed-point division and multiplication were initially proposed in [55], with proof of correctness and expansion to a radix-4 scheme in [56]. The square root operation was implemented in [38]. Floating-point algorithms for addition, subtraction and multiplication, along with the notion of quasi-normalization is presented in works such as [60–62]. Multiplicative normalization of fractions for evaluation of elementary functions was implemented in online form in [20]. An algorithm and VLSI implementation of a radix-2 evaluation of the function $Y = AX + B$ can be found in [58]. Computation of rotation factors for matrix transformations in online form is shown in [14]. Online arithmetic was used in implementing a general hardware oriented algorithm that can be used for a wide application domain, including evaluation of polynomials, certain rational functions and arithmetic expressions, solving linear equation systems, and basic arithmetic operations [13].

Details on established online arithmetic algorithms for basic fundamental operations including addition, multiplication, division and square root can be found in [16]. More complex operations such as the sum of three squares used for vector normalization [25] or primitives used in Fast Fourier Transform and Discrete Cosine Transform operations [33] are also implemented in online arithmetic. The advantages of online arithmetic for high throughput operations has been explored in gene sequence profiling [37], and motion estimation for video encoding standards such as H.26x and MPEG-1, -2 or -4 [53].

CHAPTER 3

The Multi-Accelerator architecture

3.1 Photon mapping algorithm overview

Before examining the details of the MAPM architecture, we first need a brief understanding of how the photon mapping algorithm works. The discussion in this section is based on implementation details from [41] and the actual implementation code from [49]. The following explains the details of the photon mapping algorithm that is the basis of the MAPM architecture implementation, with some examples where applicable. Note that the algorithm here is for calculating indirect illumination only. Direct illumination is usually calculated separately from indirect illumination and are added together for the final image. As direct illumination does not account for secondary rays, the workload is smaller than indirect illumination by many orders of magnitude, and it will not be considered for acceleration here.

3.1.1 Goal of the algorithm

The end goal of the PM (Photon Mapping) algorithm is to produce a 2D image of the given scene. The image has a preset aspect ratio and resolution, given as the number of pixels on the x and y axis. Each pixel is a square block that displays a single unified color, specified by three floating-point values. Each of these stands for the color intensity of red, green and blue light components, and combined together, determines the single color of each pixel that make up the image. In

short, the process of rendering a 3D image using PM is determining the color of each pixel of the image, so that it faithfully represents what the given scene would look like. The color components are calculated in floating-point values, and are converted into the data format that fits the display's color palette data format (usually integers) if necessary.

3.1.2 Input data

For the image, we are given information on the 3D scene, along with light sources and camera information.

1. **3D scene** Information of the 3D scene is given as a set of triangles, along with color and texture information of its surface. For flat surfaces with straight edges, it is trivial to divide it up into triangular parts, and curved surfaces or edges are divided up into a sufficiently large number of small triangles, so that they look curved from a distance.

2. **Light sources** Light sources are the origin points of illumination in the scene. Light travels in straight lines from the sources and bounce from the surfaces to project direct and indirect illumination into the scene. Without any illumination, nothing would be visible.

   Information on the light sources include the (x, y, z) coordinates of the location, and the intensity of the light's RGB (Red, Green, Blue) components.

3. **Camera** The camera is the viewpoint into the scene. Basically, objects in front of it will be visible in the image, and objects not in the direction that the camera is facing or obscured by another object will not be shown in the final image. The location of the camera is given in (x, y, z) coordinates, and a direction vector indicates where the camera is pointing.

For the example, the scene used is a variation of the Cornell box, a well known 3D test model that is widely used for various image rendering techniques and first introduced in [19].

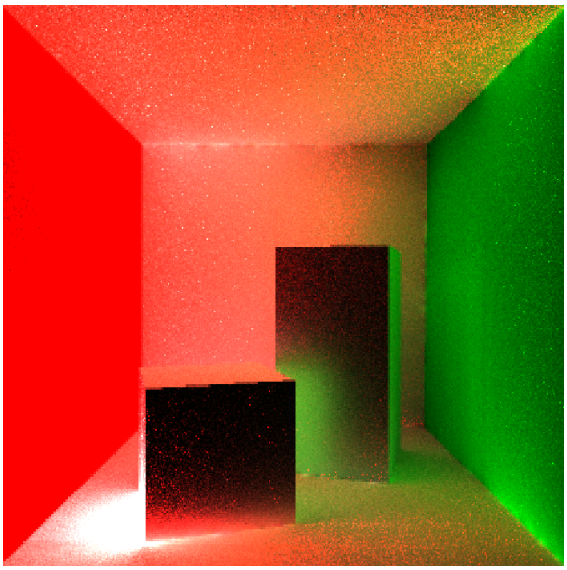


Figure 3.1: Cornell box (adapted from [19])

Our example model consists of two cuboid blocks placed inside five outer walls (left, right, ceiling, floor and rear wall). The left wall is colored red, the right wall is colored green, and all other walls of the room and the two blocks are white. The colored walls will create a similarly colored hue on the white walls with proper global illumination.

The scene file cornell-box.obj holds information on the x, y, z coordinates of all vertices in the scene, and each triangle is defined by its material and the three indices of the vertices that make up the triangle. This information is parsed and built into a polygon map, which is the KD-tree where all triangle information is stored in order to speed up the ray intersection process.

The camera of our example is located at (278.0, 273.0, -530). This is parallel to the center of the overall bounds on the xy plane, with a negative z coordinate to distance the camera from the objects in the scene. The camera's direction vector

is pointing straight in the direction of the scene, parallel to the z axis.

For the sake of our example, a single light source is placed at (1.0, 25.0, 1.0). Seen from the camera point, this is at the corner closer to the camera of the two lower-left ones of the scene. Looking at the image, we can see a circle of intense light on the floor due to it being very close to the light. The light itself is not visible as this image is indirect illumination only.

3.1.3 The KD-tree

A KD-tree is a binary space partitioning tree that subdivides a k-dimensional space into irregularly sized regions [9]. In computer graphics, it is used to store and efficiently search for objects inside the 3D space, and is generally considered to be the best data structure for accelerating the tracing of rays [23] (from this point on, we shall assume that all KD-trees are set to a 3D space unless stated otherwise). This is an essential part of the PM algorithm, since the process requires a large number of searches to find query point and photon pairs.

Every node in the KD-tree represents a cuboid region inside the 3D space. Each non-leaf node divides the current cell in two with a splitting hyperplane that is perpendicular to the coordinate axes. The splitting hyperplane can be plotted using one of the following equations: $x = a$, $y = b$, or $z = c$.

Figure 3.2 shows a simplified example of a KD-tree in 2D space.

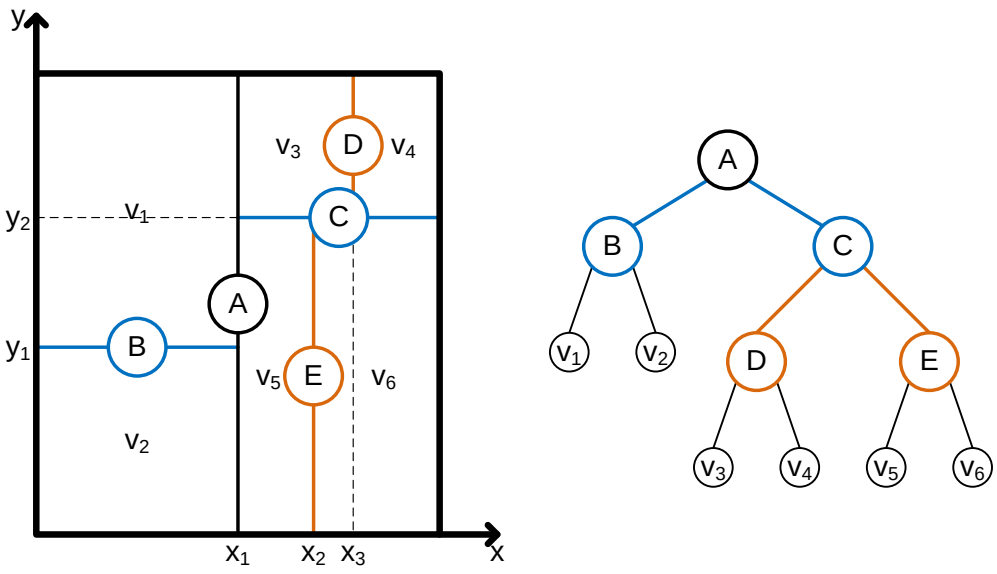


Figure 3.2: KD-tree example on a 2-dimensional space

From the 2D example, if we are looking for points that have an x coordinate larger than x_1, we can easily determine that it will not exist under node B, since all points under B have $x < x_1$. As each node has at most 2 children, the average search time is O(log n), with n being the number of total search points in the scene.

3.1.4 Building the photon KD-tree

1. **Parsing the scene file** The .obj file holding the polygon information is parsed, and all triangle information is loaded into memory. All triangles are sorted and built into the polygon map, which is the KD-tree where all triangle information is stored in order to speed up the ray intersection process. To build the polygon map, a well known heuristic called the surface area heuristic (SAH) [35] is used. At each division of the triangles into children nodes, the SAH selects the division that minimizes the total surface area for the two children. This is based on the idea that the relative probability of

a ray passing through a subspace is linear to its surface area.

Since this is not a part of the PM process that is accelerated in the MAPM, we shall not delve into the details any further here.

2. **Photon tracing** With the polygon tree finished, rays are recursively traced from the light source into the scene to create photons. In essence, the photons represent a sampling of light at various locations in the scene.

   The initial ray starts from the light source into a random direction. If an intersection is found, a photon is created at the intersection point. Related information is stored in an array, which includes the x, y, z coordinates, incoming direction of the light, and RGB intensity values. Once this is done, a reflected bounce ray is created with RGB intensity values adjusted in accordance to the surface material, and the ray intersection process is repeated using the new bounce ray.

   This process ends if it reaches a preset hard depth limit. Also, it is beneficial to make use of the Monte Carlo method [12] Russian Roulette, by randomly selecting photons to terminate early after a smaller depth limit. This is due to the photon mapping process inherently being a random sampling problem.

   The photon tracing is repeated by creating a new sample ray from the light source, until the number of total photons created reaches a preset number count.

3. **Photon KD-tree build** Once all photons are found, the photons are sorted by location to form the photon map, which is a KD-tree that holds all the photon information. At each tree node, all contained photons are sorted along the longest axis and divided into two children at the median, with the one photon at the median being saved to the current node. This goes on until a node has only one photon, whereas that node now is a leaf node

without any further children. We can see that a photon map containing n photons will result in a balanced and dense binary tree, with n total nodes and $\lceil log(n) \rceil$ levels.

Below we see a small example of a 7 photon tree build process. The initial bounding box is the bounding box for the whole scene, and it starts at $x(0, 555.036)$, $y(0, 548.8)$, $z(0.769442, 559.2)$.

(a) Node 0 (root node)

Initial photons are:

slot	x	y	z
0	312.808	548.8	8.8027
1	264.63	428.347	559.2
2	0	44.8259	495.07
3	28.2439	0	7.08738
4	72.7003	548.8	191.099
5	0	509.39	171.588
6	114.27	84.3132	117.435

Length of bounding box axes are $x(555.036 - 0 = 555.036)$, $y(548.8 - 0 = 548.8)$, $z(559.2 - 0.769442 = 558.431)$, and longest is z. Sorting through z gives us:

slot	x	y	z
0	28.2439	0	7.08738
1	312.808	548.8	8.8027
2	114.27	84.3132	117.435
3	0	509.39	171.588
4	72.7003	548.8	191.099
5	0	44.8259	495.07
6	264.63	428.347	559.2

Median is 171.588 at slot 3 with split axis z, photon at slot 3 is placed at current node 0, left child contains photons at slots 0-2 and right child contains photons at slots 4-6.

(b) Node 1 (left child of node 0)

Initial photons are:

slot	x	y	z
0	28.2439	0	7.08738
1	312.808	548.8	8.8027
2	114.27	84.3132	117.435

Length of bounding box axes are $x(555.036 - 0 = 555.036)$, $y(548.8 - 0 = 548.8)$, $z(171.588 - 0.769442 = 170.819)$, and longest is x. Sorting through x gives us:

slot	x	y	z
0	28.2439	0	7.08738
1	114.27	84.3132	117.435
2	312.808	548.8	8.8027

Median is 114.27 at slot 1 with split axis x, photon at slot 1 is placed at current node 1, left child contains photon at slot 0 and right child contains photon at slot 2.

(c) Node 2 (left child of node 1)

As there is only 1 photon given, node 2 becomes a leaf node that contains the photon at $(28.2439, 0, 7.08738)$.

(d) Node 3 (right child of node 1)

As there is only 1 photon given, node 3 becomes a leaf node that contains the photon at $(312.808, 548.8, 8.8027)$.

(e) Node 4 (right child of node 0)

Initial photons are:

slot	x	y	z
0	72.7003	548.8	191.099
1	0	44.8259	495.07
2	264.63	428.347	559.2

Length of bounding box axes are $x(555.036 - 0 = 555.036)$, $y(548.8 - 0 = 548.8)$, $z(559.2 - 171.588 = 387.612)$, and longest is x. Sorting through x gives us:

slot	x	y	z
0	0	44.8259	495.07
1	72.7003	548.8	191.099
2	264.63	428.347	559.2

Median is 72.7003 at slot 1 with split axis x, photon at slot 1 is placed at current node 4, left child contains photon at slot 0 and right child contains photon at slot 2.

(f) Node 5 (left child of node 4)

As there is only 1 photon given, node 5 becomes a leaf node that contains the photon at $(0, 44.8259, 495.07)$.

(g) Node 6 (right child of node 4)

As there is only 1 photon given, node 6 becomes a leaf node that contains the photon at $(264.63, 428.347, 559.2)$.

Now all photons are accounted for, and the final tree looks like Table 3.1.

Index	LC	RC	Photon coordinates
0	1	4	$0, 509.39, 171.588$
1	2	3	$114.27, 84.3132, 117.435$
2	–	–	$28.2439, 0, 7.08738$
3	–	–	$312.808, 548.8, 8.8027$
4	5	6	$72.7003, 548.8, 191.099$
5	–	–	$0, 44.8259, 495.07$
6	–	–	$264.63, 428.347, 559.2$

Table 3.1: Tree contents for the example photon tree

3.1.5 Final gathering and query points

1. **Tracing primary rays** As mentioned in the previous section, we are given the location of the camera and the direction it is pointing at. From this information, we create a plane in front of the camera that is perpendicular to the camera direction vector. A rectangle with the same aspect ratio of the final image is drawn on this plane, which is then divided up into the same number of pixels as the image. This rectangle acts as a virtual frame of the final image, and by calculating the color of each pixel on the image plane, we can determine the color of the pixel on the final image.

   To do this, rays that originate from the camera are created. Each ray is directed to pass a single pixel on the rectangular frame. These rays are called primary rays.

   Each primary ray is traced into the scene to find an intersection point with the 3D scene. Again, the polygon map is utilized for faster search through the scene. The color of this intersection point then becomes the final color of the image pixel that corresponds to this primary ray. The process of

determining this value is final gathering.

2. **Final gathering and query points** Final gathering starts from the intersection points of primary rays. From each intersection point, a number of sampling rays are shot into the scene in random directions. Intersections with these rays become query points, where we need to find nearby photons in order to calculate the indirect lighting that is sent towards the intersection point of the primary ray, and provides global illumination. In preparation of this search and shade process, the query points are stored in an array.

3.1.6 Tree Search and Shader Operation

With the above processes finished and the photon KD-tree and query point array complete, we are now ready to run tree search and find shader operations. This is the main part of the PM algorithm that is accelerated by the MAPM architecture.

1. **Tree search** For each query point in the array, we would like to find all photons that are close enough to affect the illumination at the query point. We determine this by checking the distance from the query point to the photon. In relation to the location of the query point, we wish to find all photons that have a distance equal to or smaller than a fixed distance h, which is the radius of the kernel. This is done in an efficient method by starting from the root node of the photon KD-tree and moving down the tree node by node, excluding any branches that contain spaces that are further than h from the query point. Any photons that are found during and at the end the search are paired up with the query point to form shader operations.

   Each step through the KD-tree processes a single node. Every node represents a subset of the 3D space containing photons. As we examined in Section 3.1.3, each node divides the space in two via its splitting plane, and

all photons inside the 3D space fall into one of 3 distinct sets, the left child, right child, or on the splitting plane.

At each node, we wish to determine which of the 3 sets hold the photons that we seek for this given query point. With this information, we can determine whether or not we need to search further into the children, or if we need to assign any shader operations from this node. In order to determine this in an efficient manner, we calculate the distance between the query point and the split position of each cell along a single axis, and derive a pair of simple comparisons.

For each query point and photon KD-tree node pair, we work with the following floating-point values:

$$\begin{aligned} p &: \text{query point position} \\ h &: \text{radius of the kernel} \\ s &: \text{split position of the KD-tree node} \end{aligned}$$

The query point position p is the x, y or z coordinate of the query point. This is selected depending on the orientation of the splitting plane of the current tree node, for instance, if the splitting plane is $x = a$, p is equal to the x coordinate of the query point.

The radius of the kernel h is a pre-computed value which affects the overall image quality. Discussion on calculation of this value can be found in [47]. In terms of the tree search operation, h is the maximum distance between a photon and a query point for the photon to affect the color at the query point. Any photon that lies outside this distance can be ignored, without any effect on the final image.

And lastly, the split position s is the coordinate of the splitting plane that divides the space of the current node between the two children. For example, if the splitting plane crosses the x axis, its plot equation would be $x = s$.

Put simply, we wish to exclude all photons that lie outside the area marked by the dark rectangle in Figure 3.3. The distance to any photon that lies outside this range will be greater than h and do not need to be considered for a shader operation.

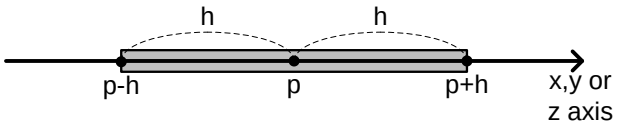


Figure 3.3: Photon search range

With this in mind, there are only 3 possible cases regarding the the split position s of the KD-tree node:

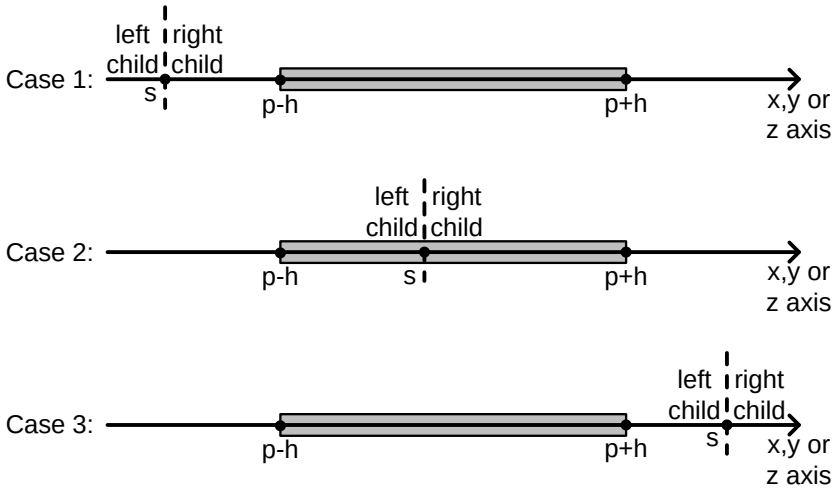


Figure 3.4: Scenarios for the split position location

(a) **Case 1** The rectangle lies completely to the right of the split position. We can safely exclude any photons that are in the left child, meaning that we continue searching the right child.

(b) **Case 2** The rectangle lies on both sides of the split position. No photons can be safely excluded, meaning that we continue searching both the left and right child.

(c) **Case 3** The rectangle lies completely to the left of the split position. We can safely exclude any photons that are in the right child, meaning that we continue searching the left child.

To determine which one of these cases we have, we evaluate two inequalities, $p-h < s$ and $p+h > s$. From Figure 3.3, $p-h$ is the left end of the rectangle, and $p + h$ is the right end of the rectangle. From the two values, we can determine which case this search belongs to:

(a) $p - h < s$ is true and $p + h > s$ is false. Which means that the left end of the rectangle is on the left side of s, and the right end of the rectangle is NOT on the right side of s. This indicates that the whole rectangle is on the left side of s, which would be Case 3 of Figure 3.4. We would need to traverse the left child for this case.

(b) $p - h < s$ is false and $p + h > s$ is true. Which means that the left end of the rectangle is NOT on the left side of s, and the right end of the rectangle is on the right side of s. This indicates that the whole rectangle is on the right side of s, which would be Case 1 of Figure 3.4. We would need to traverse the right child for this case.

(c) $p-h < s$ is true and $p+h > s$ is true. Which means that the left end of the rectangle is on the left side of s, and the right end of the rectangle is on the right side of s. This indicates that the whole rectangle is on both sides of s, which would be Case 2 of Figure 3.4. We would need to traverse both the left and right child for this case.

(d) $p-h < s$ is false and $p+h > s$ is false. Which means that the left end of the rectangle is NOT on the left side of s, and the right end of the rectangle is NOT on the right side of s. This cannot possibly be true unless h is a negative number, but since $h > 0$, we will never have this condition.

For reasons that are obvious, we shall call the inequality $p - h < s$ as TC_L (Traverse Child Left) and the inequality $p + h > s$ as TC_R (Traverse Child Right).

Some example tree search queries into our minimal photon KD-tree at Table 3.1 is shown below.

(a) Tree Search with query point at $(0, 358.739, 280.736)$

- Examining node 0: not a leaf, calculate TC_L and TC_R using split axis z.

  We have $p = 280.736$, $h = 100$, $s = 171.588$, $p - h = 180.736$, $p + h = 380.736$.

  TC_L =false and TC_R =true, traverse right child (4) only.
- Examining node 4: not a leaf, calculate TC_L and TC_R using split axis x.

  We have $p = 0$, $h = 100$, $s = 72.7003$, $p - h = -100$, $p + h = 100$.

  TC_L =true and TC_R =true, traverse both children (5,6).

  Also, create Shader Operation for photon in node 4.
- Examining node 5: leaf, create Shader Operation for photon in node 5.
- Examining node 6: leaf, create Shader Operation for photon in node 6.
- End of query

(b) Tree Search with query point at $(0, 473.547, 255.165)$

- Examining node 0: not a leaf, calculate TC_L and TC_R using split axis z.

  We have $p = 255.165$, $h = 100$, $s = 171.588$, $p - h = 155.165$, $p + h = 355.165$.

  TC_L =true and TC_R =true, traverse both children (1,4).

Also, create Shader Operation for photon in node 0.

- Examining node 1: not a leaf, calculate TC_L and TC_R using split axis x.

  We have $p = 0$, $h = 100$, $s = 114.27$, $p - h = -100$, $p + h = 100$.

  TC_L =true and TC_R =false, traverse left child (2) only.

- Examining node 4: not a leaf, calculate TC_L and TC_R using split axis x.

  We have $p = 0$, $h = 100$, $s = 72.7003$, $p - h = -100$, $p + h = 100$.

  TC_L =true and TC_R =true, traverse both children (5,6).

  Also, create Shader Operation for photon in node 4.

- Examining node 2: leaf, create Shader Operation for photon in node 2.

- Examining node 5: leaf, create Shader Operation for photon in node 5.

- Examining node 6: leaf, create Shader Operation for photon in node 6.

- End of query

(c) Tree Search with query point at $(213.296, 0, 26.6906)$

- Examining node 0: not a leaf, calculate TC_L and TC_R using split axis z.

  We have $p = 26.6906$, $h = 100$, $s = 171.588$, $p - h = -73.3094$, $p + h = 126.691$.

  TC_L =true and TC_R =false, traverse left child (1) only.

- Examining node 1: not a leaf, calculate TC_L and TC_R using split axis x.

  We have $p = 213.296$, $h = 100$, $s = 114.27$, $p - h = 113.296$, $p + h = 313.296$.

  TC_L =true and TC_R =true, traverse both children (2,3).

- Examining node 2: leaf, create Shader Operation for photon in node 2.
- Examining node 3: leaf, create Shader Operation for photon in node 3.
- End of query

The pseudo code for Tree Search can be written as shown:

```
obtain p from query point, s from current node
calculate TC_L = (p-h)<s, TC_R = (p+h)>s
if( TC_L is true ) then:
  if( left child is not leaf ) then:
    issue TS with query point and left child
  else:
    issue SO for query point and left child photon
end if
if( TC_R is true ) then:
  if( right child is not leaf ) then:
    issue TS with query point and right child
  else:
    issue SO for query point and right child photon
end if
if( TC_L is true and TC_R is true ) then:
  issue SO for query point and current child photon
end if
```

2. **Shader operation**

Once we have found a photon that is close enough to affect the light at the query point, we need to calculate the color values that the photon adds to

the query point. The equations below show how this is calculated for each of the three colors, red, green and blue:

$$\begin{aligned}
\texttt{color.red} &= \frac{\texttt{kernel} \times \texttt{BRDF}}{h} \times \texttt{power.red} \times \texttt{contribution.red} \\
\texttt{color.green} &= \frac{\texttt{kernel} \times \texttt{BRDF}}{h} \times \texttt{power.green} \times \texttt{contribution.green} \\
\texttt{color.blue} &= \frac{\texttt{kernel} \times \texttt{BRDF}}{h} \times \texttt{power.blue} \times \texttt{contribution.blue}
\end{aligned}$$

The theory behind the calculations is essentially a density estimation, and the `kernel` value is from a kernel function such as a Gaussian distribution. The `BRDF` is a function that calculates how much of the light that entered a certain point in the scene is reflected out towards a given direction. h is the value of the kernel radius, the `power.(color)` values are each color's power components of the photons, and the `contribution.(color)` values are the scaling factors. The `BRDF` and `contribution.(color)` values are in the range [0,1]. The `power.(color)`, the `kernel` and kernel radius h are non-zero positive values. These values are calculated and directly added to the corresponding pixel of the query point in the frame buffer.

For more detailed discussion regarding these values, refer to [47].

3.2 Architecture overview

Figure 3.5 shows an overhead schematic of the MAPM. The MAPM is a co-processor like structure; it shares data paths to the main memory with the CPU, along with instruction issues from the CPU and outputs the results directly to the frame buffer or the main memory. The MAPM receives tree search instuctions from the CPU in terms of query points, and returns the calculated color value of each query point.

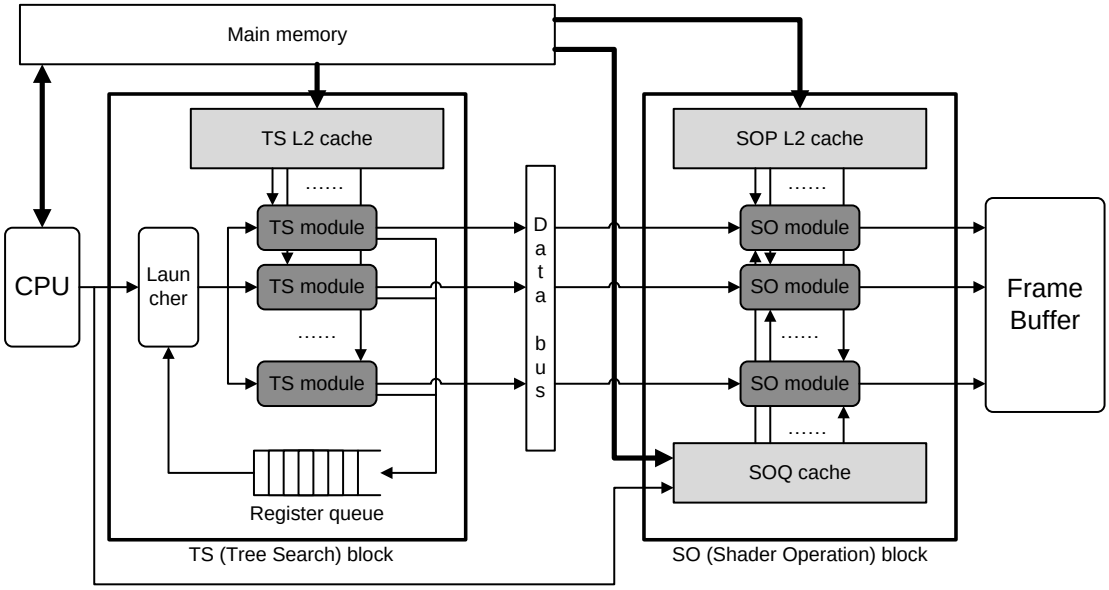


Figure 3.5: MAPM architecture overview

The high level specification of the MAPM is shown below.

inputs:

$$
\begin{aligned}
\underline{px} &= (px_{31}, px_{30}, \cdots, px_1, px_0),\ px_i \in \{0,1\} \\
&\quad px_{31} \text{ is the sign bit } px.s, \\
&\quad px_{30} \text{ to } px_{23} \text{ is the 8-bit biased exponent } px.e, \\
&\quad \text{and } px_{22} \text{ to } px_0 \text{ is the 23-bit mantissa part } px.m \\
&\quad px_{val} = (-1)^{px.s} 2^{px.e-B}(1.px.m), \text{ where } B = 2^7 - 1 \\
\underline{py} &= (py_{31}, py_{30}, \cdots, py_1, py_0),\ py_i \in \{0,1\} \\
&\quad \text{format is the same as } \underline{px} \\
\underline{pz} &= (pz_{31}, pz_{30}, \cdots, pz_1, pz_0),\ pz_i \in \{0,1\} \\
&\quad \text{format is the same as } \underline{px} \\
\underline{qIn} &= (qIn_{31}, qIn_{30}, \cdots, qIn_1, qIn_0),\ qIn_i \in \{0,1\}
\end{aligned}
$$

outputs:

$\underline{colorR}$ = $(colorR_{31}, colorR_{30}, \cdots, colorR_1, colorR_0)$, $colorR_i \in \{0,1\}$
$colorR_{31}$ is the sign bit $colorR.s$,
$colorR_{30}$ to $colorR_{23}$ is the 8-bit biased exponent $colorR.e$,
and $colorR_{22}$ to $colorR_0$ is the 23-bit mantissa part $colorR.m$
$colorR_{val} = (-1)^{colorR.s} 2^{colorR.e-B}(1.colorR.m)$,
where $B = 2^7 - 1$

$\underline{colorG}$ = $(colorG_{31}, colorG_{30}, \cdots, colorG_1, colorG_0)$, $colorG_i \in \{0,1\}$
format is the same as $\underline{colorR}$

$\underline{colorB}$ = $(colorB_{31}, colorB_{30}, \cdots, colorB_1, colorB_0)$, $colorB_i \in \{0,1\}$
format is the same as $\underline{colorR}$

function:

$\underline{colorR}$ = red color component of query point at index $\underline{qIn}$
(with $t_{MAPM-delay}$ cycle delay)

$\underline{colorG}$ = green color component of query point at index $\underline{qIn}$
(with $t_{MAPM-delay}$ cycle delay)

$\underline{colorB}$ = blue color component of query point at index $\underline{qIn}$
(with $t_{MAPM-delay}$ cycle delay)

We assume that all basic synchronization signals such as clock and reset signals are there for all modules from here on, and will not explain them explicitly, as their operations are quite straightforward. The exact value of $t_{MAPM-delay}$ depends on various factors, including requests to memory and any stall cycles that may happen. However, the value is exactly the same for all three $\underline{color(R/G/B)}$ values, so the three colors are calculated and given as output at the exact same clock cycle.

The CPU issues the tree search instructions by handing over information on each query point. The variables that are given here are $\underline{px}$, $\underline{py}$ and $\underline{pz}$, which are the x, y, z coordinates of the query point, and the query index $\underline{qIn}$, which is the index into the array in the main memory where related information is saved. One

such value is the `contribution` values, that are used in the shader operation and are fetched to the SOQ cache. The position coordinates are given to the Launcher module in the TS (Tree Search) block, and the query index is given to the SOQ cache in the SO (Shader Operation) block.

As we can see in the diagram, the MAPM can be divided into two blocks, where each handles a different phase of the search and shade part of the PM algorithm (Section 3.1.6). The first is the TS block, where query point and photon pairs are acquired, and these are handed over to the SO block, where the actual color values are calculated.

The data bus is the bridge between the two blocks, where it receives the outputs of the TS modules and relays them to SO modules with available space. By shifting the starting index every cycle, the data bus tries to spread out the workload between different SOMs (Shader Operation Modules).

One more thing to note is that the three caches of the MAPM, which are the TS cache, SOP (Shader Operation – Photon) cache and SOQ (Shader Operation – Query point) cache, are read-only and never write anything back to main memory. This allows for a simpler and faster cache design compared to conventional read-and-write cache blocks. This slack can be utilized for higher bandwidth designs, such as multi-port non-blocking caches [50].

More details about each block are explained in the following sections.

3.3 The Tree Search block

3.3.1 Overview

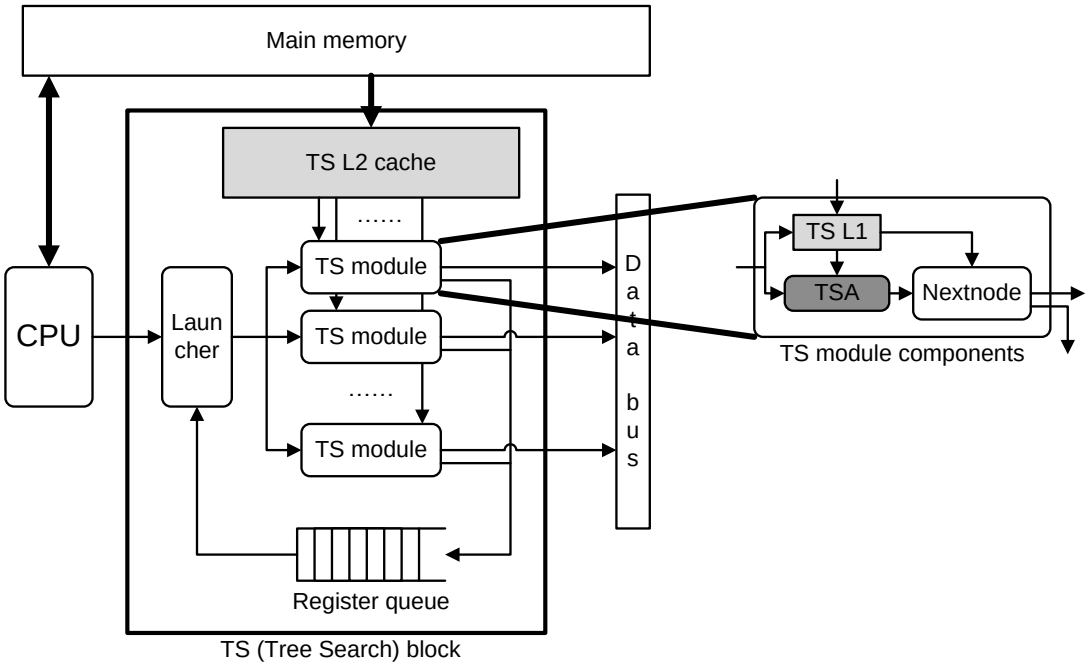


Figure 3.6: The TS block

The first part of the MAPM architecture is the Tree Search block, shown in Figure 3.6. This block processes the Tree Search phase of photon mapping. Once the KD-tree structures are finalized as explained in the algorithm overview (Section 3.1), the photon KD-tree is searched with each query point to find photons that are in close proximity. These photons are paired with the query point to form shader operations which are then handed over to the Shader Operation block.

Tree search instructions are issued from the CPU. For each query point, all related information is saved in an array in the main memory, and the CPU hands over information from the array.

The high level specification of the TS block is shown below.

inputs:

$\underline{px}$ = $(px_{31}, px_{30}, \cdots, px_1, px_0)$, $px_i \in \{0, 1\}$

px_{31} is the sign bit $px.s$,

px_{30} to px_{23} is the 8-bit biased exponent $px.e$,

and px_{22} to px_0 is the 23-bit mantissa part $px.m$

$px_{val} = (-1)^{px.s} 2^{px.e-B}(1.px.m)$, where $B = 2^7 - 1$

$\underline{py}$ = $(py_{31}, py_{30}, \cdots, py_1, py_0)$, $py_i \in \{0, 1\}$

format is the same as $\underline{px}$

$\underline{pz}$ = $(pz_{31}, pz_{30}, \cdots, pz_1, pz_0)$, $pz_i \in \{0, 1\}$

format is the same as $\underline{px}$

$\underline{qIn}$ = $(qIn_{31}, qIn_{30}, \cdots, qIn_1, qIn_0)$, $qIn_i \in \{0, 1\}$

outputs:

$\underline{px}$ = $(px_{31}, px_{30}, \cdots, px_1, px_0)$, $px_i \in \{0, 1\}$

px_{31} is the sign bit $px.s$,

px_{30} to px_{23} is the 8-bit biased exponent $px.e$,

and px_{22} to px_0 is the 23-bit mantissa part $px.m$

$px_{val} = (-1)^{px.s} 2^{px.e-B}(1.px.m)$, where $B = 2^7 - 1$

$\underline{py}$ = $(py_{31}, py_{30}, \cdots, py_1, py_0)$, $py_i \in \{0, 1\}$

format is the same as $\underline{px}$

$\underline{pz}$ = $(pz_{31}, pz_{30}, \cdots, pz_1, pz_0)$, $pz_i \in \{0, 1\}$

format is the same as $\underline{px}$

$\underline{qIn}$ = $(qIn_{31}, qIn_{30}, \cdots, qIn_1, qIn_0)$, $qIn_i \in \{0, 1\}$

$\underline{pIn}$ = $(pIn_{31}, pIn_{30}, \cdots, pIn_1, pIn_0)$, $pIn_i \in \{0, 1\}$

function:

$\underline{pIn}$ = the node index of the photon paired with the query point

each pair is delivered as soon as it is found

each delay $t_{TSB-delay}$ is variable

$\underline{qIn}$ is the integer index of the query point, and $\underline{pIn}$ is the integer index of the

photon paired with the query point.

The exact value of $t_{TSB-delay}$ depends on various factors, including requests to memory, the location of the query point, and any stall cycles that may happen. Also, each query point may have multiple photons that it can be paired with, hence creating multiple outputs for one input. Therefore each output also contains information on the input query point.

Below, we look at the internal modules inside the TS block.

1. **Launcher module**

Figure 3.7: Overview of Launcher

The Launcher module is a wholly combinational data redirector that feeds the inputs of the TSMs (Tree Search modules) with the appropriate values taken from the CPU input and register queue. Each signal bundle in Figure 3.7 is a group of signals as detailed below.

inputs:

(CPU) $\underline{px}$ = $(px_{31}, px_{30}, \cdots, px_1, px_0)$, $px_i \in \{0,1\}$

px_{31} is the sign bit $px.s$,

px_{30} to px_{23} is the 8-bit biased exponent $px.e$,

and px_{22} to px_0 is the 23-bit mantissa part $px.m$

$px_{val} = (-1)^{px.s} 2^{px.e-B}(1.px.m)$, where $B = 2^7 - 1$

(CPU) $\underline{py}$ = $(py_{31}, py_{30}, \cdots, py_1, py_0)$, $py_i \in \{0,1\}$

format is the same as $\underline{px}$

(CPU) $\underline{pz}$ = $(pz_{31}, pz_{30}, \cdots, pz_1, pz_0)$, $pz_i \in \{0,1\}$

format is the same as $\underline{px}$

(CPU) $\underline{qIn}$ = $(qIn_{31}, qIn_{30}, \cdots, qIn_1, qIn_0)$, $qIn_i \in \{0,1\}$

(R_Q) $\underline{px0}$ = $(px0_{31}, px0_{30}, \cdots, px0_1, px0_0)$, $px0_i \in \{0,1\}$

format is the same as $\underline{px}$

(R_Q) $\underline{py0}$ = $(py0_{31}, py0_{30}, \cdots, py0_1, py0_0)$, $py0_i \in \{0,1\}$

format is the same as $\underline{px}$

(R_Q) $\underline{pz0}$ = $(pz0_{31}, pz0_{30}, \cdots, pz0_1, pz0_0)$, $pz0_i \in \{0,1\}$

format is the same as $\underline{px}$

(R_Q) $\underline{qIn0}$ = $(qIn0_{31}, qIn0_{30}, \cdots, qIn0_1, qIn0_0)$, $qIn0_i \in \{0,1\}$

(R_Q) $\underline{nIn0}$ = $(nIn0_{31}, nIn0_{30}, \cdots, nIn0_1, nIn0_0)$, $nIn0_i \in \{0,1\}$

$\cdots$

(R_Q) $\underline{pxk}$ = $(pxk_{31}, pxk_{30}, \cdots, pxk_1, pxk_0)$, $pxk_i \in \{0,1\}$

format is the same as $\underline{px}$

(R_Q) $\underline{pyk}$ = $(pyk_{31}, pyk_{30}, \cdots, pyk_1, pyk_0)$, $pyk_i \in \{0,1\}$

format is the same as $\underline{px}$

(R_Q) $\underline{pzk}$ = $(pzk_{31}, pzk_{30}, \cdots, pzk_1, pzk_0)$, $pzk_i \in \{0,1\}$

format is the same as $\underline{px}$

(R_Q) $\underline{qInk}$ = $(qInk_{31}, qInk_{30}, \cdots, qInk_1, qInk_0)$, $qInk_i \in \{0,1\}$

(R_Q) $\underline{nInk}$ = $(nInk_{31}, nInk_{30}, \cdots, nInk_1, nInk_0)$, $nInk_i \in \{0,1\}$

outputs:

(TSM)	$\underline{px0}$	=	$(px0_{31}, px0_{30}, \cdots, px0_1, px0_0)$, $px0_i \in \{0,1\}$
			format is the same as $\underline{px}$
(TSM)	$\underline{py0}$	=	$(py0_{31}, py0_{30}, \cdots, py0_1, py0_0)$, $py0_i \in \{0,1\}$
			format is the same as $\underline{px}$
(TSM)	$\underline{pz0}$	=	$(pz0_{31}, pz0_{30}, \cdots, pz0_1, pz0_0)$, $pz0_i \in \{0,1\}$
			format is the same as $\underline{px}$
(TSM)	$\underline{qIn0}$	=	$(qIn0_{31}, qIn0_{30}, \cdots, qIn0_1, qIn0_0)$, $qIn0_i \in \{0,1\}$
(TSM)	$\underline{nIn0}$	=	$(nIn0_{31}, nIn0_{30}, \cdots, nIn0_1, nIn0_0)$,
			$nIn0_i \in \{0,1\}$
		...	
(TSM)	$\underline{pxk}$	=	$(pxk_{31}, pxk_{30}, \cdots, pxk_1, pxk_0)$, $pxk_i \in \{0,1\}$
			format is the same as $\underline{px}$
(TSM)	$\underline{pyk}$	=	$(pyk_{31}, pyk_{30}, \cdots, pyk_1, pyk_0)$, $pyk_i \in \{0,1\}$
			format is the same as $\underline{px}$
(TSM)	$\underline{pzk}$	=	$(pzk_{31}, pzk_{30}, \cdots, pzk_1, pzk_0)$, $pzk_i \in \{0,1\}$
			format is the same as $\underline{px}$
(TSM)	$\underline{qInk}$	=	$(qInk_{31}, qInk_{30}, \cdots, qInk_1, qInk_0)$, $qInk_i \in \{0,1\}$
(TSM)	$\underline{nInk}$	=	$(nInk_{31}, nInk_{30}, \cdots, nInk_1, nInk_0)$,
			$nInk_i \in \{0,1\}$

function:

assign searches to TSMs

where $k = n_{tsm} - 1$. $\underline{px}$, $\underline{py}$, $\underline{pz}$ are the query point's coordinates, $\underline{qIn}$ is the index of the query point, and $\underline{nIn}$ is the tree node index.

Figure 3.8 shows the details of the Launcher.

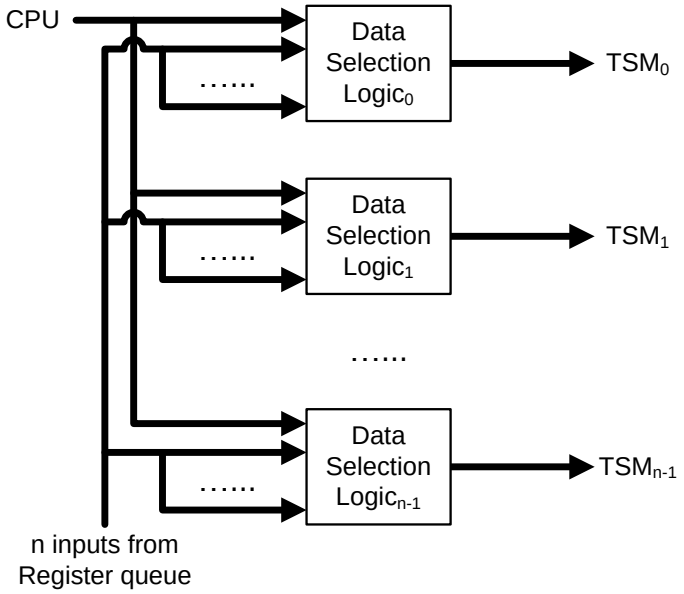


Figure 3.8: Details of Launcher

Each Data Selection Logic block is essentially a multiplexer. The Launcher receives the first n_{tsm} entries of the Register queue and feeds the TSMs with appropriate tree search information, which include the position x, y, z coordinates of the query point, the query point index value, and the tree node index value. If there were less than n_{tsm} entries in the Register queue, at least one TSM will be empty, and the Launcher checks for a tree search instruction from the CPU and assigns it to a free TSM with tree node index 0 (root node). There are no separate flags for marking the validity of the data. Since all index values are positive, by setting the sign bit to 0 or 1 we can include the valid flag inside the data.

The Launcher operation pseudo code is shown here:

```
check first n_tsm entries in register queue
assign all valid searches to TSMs
if not exists (free TSM) end
else if exists (input from CPU) then
  assign new search to TSM
```

2. **Nextnode module**

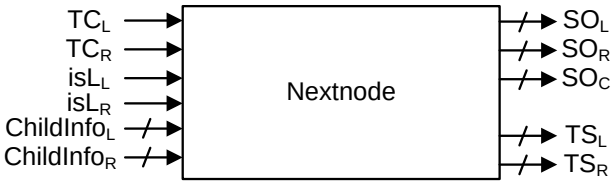


Figure 3.9: Overview of Nextnode

The Nextnode module examines the TC_L and TC_R values calculated in the TSAs and directs the data to its appropriate next stage, which is either the Register queue, or the Shader Operation block. It is similar to the Launcher in that it is a fully combinational block that directs data to the appropriate modules. ChildInfo is a bundle of signals which is explained in detail in the high-level specification below:

inputs:

(TSA) TC_L $TC_L \in \{0,1\}$

(TSA) TC_R $TC_R \in \{0,1\}$

(TSA) isL_L $isL_L \in \{0,1\}$

(TSA) isL_R $isL_R \in \{0,1\}$

(TSA) $\underline{pxL} = (pxL_{31}, pxL_{30}, \cdots, pxL_1, pxL_0)$, $pxL \in \{0,1\}$

pxL_{31} is the sign bit $pxL.s$,

pxL_{30} to pxL_{23} is the 8-bit biased exponent $pxL.e$,

and pxL_{22} to pxL_0 is the 23-bit mantissa $pxL.m$

$pxL_{val} = (-1)^{pxL.s} 2^{pxL.e-B}(1.pxL.m)$,

where $B = 2^7 - 1$

(TSA) $\underline{pyL} = (pyL_{31}, pyL_{30}, \cdots, pyL_1, pyL_0)$, $pyL \in \{0,1\}$

format is the same as $\underline{pxL}$

(TSA) $\underline{pzL} = (pzL_{31}, pzL_{30}, \cdots, pzL_1, pzL_0)$, $pzL \in \{0,1\}$

format is the same as $\underline{pxL}$

(TSA) $\underline{qInL} = (qInL_{31}, qInL_{30}, \cdots, qInL_1, qInL_0)$, $qInL \in \{0,1\}$

(TSA) $\underline{nInL} = (nInL_{31}, nInL_{30}, \cdots, nInL_1, nInL_0)$, $nInL \in \{0,1\}$

(TSA) $\underline{pxR} = (pxR_{31}, pxR_{30}, \cdots, pxR_1, pxR_0)$, $pxR \in \{0,1\}$

format is the same as $\underline{pxL}$

(TSA) $\underline{pyR} = (pyR_{31}, pyR_{30}, \cdots, pyR_1, pyR_0)$, $pyR \in \{0,1\}$

format is the same as $\underline{pxL}$

(TSA) $\underline{pzR} = (pzR_{31}, pzR_{30}, \cdots, pzR_1, pzR_0)$, $pzR \in \{0,1\}$

format is the same as $\underline{pxL}$

(TSA) $\underline{qInR} = (qInR_{31}, qInR_{30}, \cdots, qInR_1, qInR_0)$, $qInR \in \{0,1\}$

(TSA) $\underline{nInR} = (nInR_{31}, nInR_{30}, \cdots, nInR_1, nInR_0)$, $nInR \in \{0,1\}$

outputs:

(DB) $\underline{pxL}$ = $(pxL_{31}, pxL_{30}, \cdots, pxL_1, pxL_0)$, $pxL_i \in \{0,1\}$
pxL_{31} is the sign bit $pxL.s$,
pxL_{30} to pxL_{23} is the 8-bit biased exponent $pxL.e$,
and pxL_{22} to pxL_0 is the 23-bit mantissa $pxL.m$
$pxL_{val} = (-1)^{pxL.s} 2^{pxL.e-B}(1.pxL.m)$,
where $B = 2^7 - 1$

(DB) $\underline{pyL}$ = $(pyL_{31}, pyL_{30}, \cdots, pyL_1, pyL_0)$, $pyL \in \{0,1\}$
format is the same as $\underline{pxL}$

(DB) $\underline{pzL}$ = $(pzL_{31}, pzL_{30}, \cdots, pzL_1, pzL_0)$, $pzL \in \{0,1\}$
format is the same as $\underline{pxL}$

(DB) $\underline{qInL}$ = $(qInL_{31}, qInL_{30}, \cdots, qInL_0)$, $qInL \in \{0,1\}$

(DB) $\underline{nInL}$ = $(nInL_{31}, nInL_{30}, \cdots, nInL_0)$, $nInL \in \{0,1\}$

(DB) $\underline{pxR}$ = $(pxR_{31}, pxR_{30}, \cdots, pxR_1, pxR_0)$, $pxR_i \in \{0,1\}$
format is the same as $\underline{pxL}$

(DB) $\underline{pyR}$ = $(pyR_{31}, pyR_{30}, \cdots, pyR_1, pyR_0)$, $pyR \in \{0,1\}$
format is the same as $\underline{pxL}$

(DB) $\underline{pzR}$ = $(pzR_{31}, pzR_{30}, \cdots, pzR_1, pzR_0)$, $pzR \in \{0,1\}$
format is the same as $\underline{pxL}$

(DB) $\underline{qInR}$ = $(qInR_{31}, qInR_{30}, \cdots, qInR_0)$, $qInR \in \{0,1\}$

(DB) $\underline{nInR}$ = $(nInR_{31}, nInR_{30}, \cdots, nInR_0)$, $nInR \in \{0,1\}$

(DB) $\underline{pxC}$ = $(pxC_{31}, pxC_{30}, \cdots, pxC_1, pxC_0)$, $pxC_i \in \{0,1\}$
format is the same as $\underline{pxL}$

(DB) $\underline{pyC}$ = $(pyC_{31}, pyC_{30}, \cdots, pyC_1, pyC_0)$, $pyC \in \{0,1\}$
format is the same as $\underline{pxL}$

(DB) $\underline{pzC}$ = $(pzC_{31}, pzC_{30}, \cdots, pzC_1, pzC_0)$, $pzC \in \{0,1\}$
format is the same as $\underline{pxL}$

(DB) $\underline{qInC}$ = $(qInC_{31}, qInC_{30}, \cdots, qInC_0)$, $qInC \in \{0,1\}$

(DB) $\underline{nInC}$ = $(nInC_{31}, nInC_{30}, \cdots, nInC_0)$, $nInC \in \{0,1\}$

(R_Q) $\underline{pxL} = (pxL_{31}, pxL_{30}, \cdots, pxL_1, pxL_0)$, $pxL_i \in \{0,1\}$
pxL_{31} is the sign bit $pxL.s$,
pxL_{30} to pxL_{23} is the 8-bit biased exponent $pxL.e$,
and pxL_{22} to pxL_0 is the 23-bit mantissa $pxL.m$
$pxL_{val} = (-1)^{pxL.s} 2^{pxL.e-B}(1.pxL.m)$,
where $B = 2^7 - 1$

(R_Q) $\underline{pyL} = (pyL_{31}, pyL_{30}, \cdots, pyL_1, pyL_0)$, $pyL \in \{0,1\}$
format is the same as $\underline{pxL}$

(R_Q) $\underline{pzL} = (pzL_{31}, pzL_{30}, \cdots, pzL_1, pzL_0)$, $pzL \in \{0,1\}$
format is the same as $\underline{pxL}$

(R_Q) $\underline{qInL} = (qInL_{31}, qInL_{30}, \cdots, qInL_0)$, $qInL \in \{0,1\}$

(R_Q) $\underline{nInL} = (nInL_{31}, nInL_{30}, \cdots, nInL_0)$, $nInL \in \{0,1\}$

(R_Q) $\underline{pxR} = (pxR_{31}, pxR_{30}, \cdots, pxR_1, pxR_0)$, $pxR_i \in \{0,1\}$
format is the same as $\underline{pxL}$

(R_Q) $\underline{pyR} = (pyR_{31}, pyR_{30}, \cdots, pyR_1, pyR_0)$, $pyR \in \{0,1\}$
format is the same as $\underline{pxL}$

(R_Q) $\underline{pzR} = (pzR_{31}, pzR_{30}, \cdots, pzR_1, pzR_0)$, $pzR \in \{0,1\}$
format is the same as $\underline{pxL}$

(R_Q) $\underline{qInR} = (qInR_{31}, qInR_{30}, \cdots, qInR_0)$, $qInR \in \{0,1\}$

(R_Q) $\underline{nInR} = (nInR_{31}, nInR_{30}, \cdots, nInR_0)$, $nInR \in \{0,1\}$

function:
assign Tree Searches to Register queue,
Shader Operations to Data bus

$\underline{px}$, $\underline{py}$, $\underline{pz}$ are the query point's coordinates, $\underline{qIn}$ is the index of the query point, and $\underline{nIn}$ is the tree node index. The subscripts L, R, and C stand for left child, right child and current node, respectively.

Figure 3.10 shows the details of the Nextnode module.

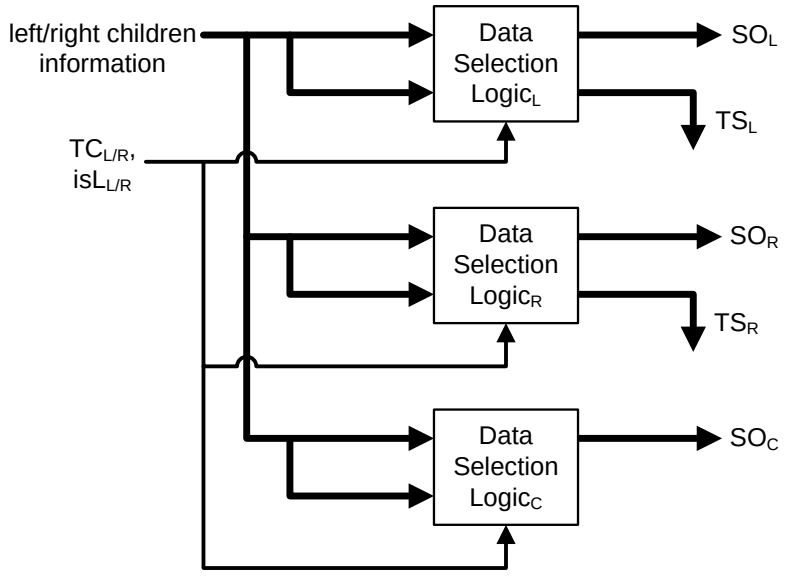


Figure 3.10: Details of Nextnode

Each Data Selection Logic is essentially a multiplexer, with select signals being $TC_{L/R}$ and $isL_{L/R}$ values. From these Nextnode can determine where the information from the left and right children need to move to next. For the outputs, the valid flag is encoded into the data itself by setting the sign bit to 0 or 1, utilizing the fact that the index values will never be a negative value.

The Nextnode operation pseudo code is shown here:

```
if (TCL is true) then
  if (isLL is true) then
    assign new SO for left child to Data bus
  else
    assign TS into left child to Register queue
if (TCR is true) then
  if (isLR is true) then
    assign new SO for right child to Data bus
```

```
    else
      assign TS into right child to Register queue
  if (TCL and TCR is true) then
    assign new SO for current node to Data bus
```

3. **Register queue**

   In order to search for photons that are close inside the photon KD-tree, we need to do searches leading into each node's children. These searches are placed on the Register queue and assigned to free TSMs by the Launcher module. Function-wise, there is not much that goes on here; any new children searches are written to the queue, and any old searches that get assigned to TSMs are removed from the queue. The Register queue basically acts as a First-In First-Out queue for Tree Search instructions.

   Each register entry contains the following information:

$$
\begin{aligned}
\underline{px} &= (px_{31}, px_{30}, \cdots, px_1, px_0),\ px_i \in \{0,1\} \\
&\quad px_{31} \text{ is the sign bit } px.s, \\
&\quad px_{30} \text{ to } px_{23} \text{ is the 8-bit biased exponent } px.e, \\
&\quad \text{and } px_{22} \text{ to } px_0 \text{ is the 23-bit mantissa part } px.m \\
&\quad px_{val} = (-1)^{px.s} 2^{px.e-B}(1.px.m), \text{ where } B = 2^7 - 1 \\
\underline{py} &= (py_{31}, py_{30}, \cdots, py_1, py_0),\ py \in \{0,1\} \\
&\quad \text{format is the same as } \underline{px} \\
\underline{pz} &= (pz_{31}, pz_{30}, \cdots, pz_1, pz_0),\ pz \in \{0,1\} \\
&\quad \text{format is the same as } \underline{px} \\
\underline{qIn} &= (qIn_{31}, qIn_{30}, \cdots, qIn_1, qIn_0),\ qIn \in \{0,1\} \\
\underline{nIn} &= (nIn_{31}, nIn_{30}, \cdots, nIn_1, nIn_0),\ nIn \in \{0,1\}
\end{aligned}
$$

   $\underline{px}$, $\underline{py}$, $\underline{pz}$ are the query point's coordinates, $\underline{qIn}$ is the index of the query point, and $\underline{nIn}$ is the tree node index.

3.4 The Tree Search Accelerator (TSA)

3.4.1 Problem outline

Tree search instructions are issued from the CPU. For each query point, all related information is saved in an array in the main memory, and the CPU hands over information from the array.

For each query point and photon KD-tree node, the following values are obtained:

p : query point position
h : radius of the kernel
s : split position of the KD-tree node

We assume that all three are 32-bit positive floating-point values in IEEE-754 standard form [8], so each variable comprises a sign bit, an 8-bit exponent, and a 23-bit mantissa with 1 hidden bit. In addition to this, we will assume there are no special case inputs such as denormalized numbers, infinity or NaNs (Not a Number).

The pseudo code for the Tree Search block, and where each operation is handled, is shown here:

```
receive p, h (query point) and s (node) [Launcher]
calculate TC_L = (p-h)<s, TC_R = (p+h)>s [TSM]
if( TC_L is true ) then: [Nextnode]
  if( left child is not leaf ) then:
    issue TS with query point and left child
  else:
    issue SO for query point and left child photon
end if
if( TC_R is true ) then: [Nextnode]
```

```
    if( right child is not leaf ) then:
      issue TS with query point and right child
    else:
      issue SO for query point and right child photon
  end if
  if( both TC_L and TC_R is true ) then: [Nextnode]
    issue SO for query point and current node photon
```

Using online arithmetic for the comparison has several advantages, and high throughput is the main point. Throughput is an important performance criterion for computer graphics in general, as the calculation requirement is usually very high and delay of a single operation is not critical. It becomes even more important considering the need to obtain massive fine-grain parallelism as discussed in Section 2.1.3.

By using online arithmetic, we can assign one comparison per module at every clock cycle. There is also the additional benefit of the clock cycle being very short as each stage of online arithmetic is much simpler in comparison to conventional parallel arithmetic circuits. Due to the shader operations being independent of each other, a high throughput performance of locating query point/photon pairs will lead to high throughput launching of shader operation instructions in the SO block.

3.4.2 Overall structure

Figure 3.11 is an overview of the TSA. The TSA takes the values p, h and s, and returns boolean values on whether we need to traverse any of the children nodes, namely TC_L and TC_R.

Figure 3.11: Overview of TSA

The high level specification of the TSA is shown below.

inputs:

$$\begin{aligned}
\underline{p} &= (p_{31}, p_{30}, \cdots, p_1, p_0),\ p_i \in \{0,1\} \\
&\quad p_{31} \text{ is the sign bit } p.s, \\
&\quad p_{30} \text{ to } p_{23} \text{ is the 8-bit biased exponent } p.e, \\
&\quad \text{and } p_{22} \text{ to } p_0 \text{ is the 23-bit mantissa part } p.m \\
&\quad p_{val} = (-1)^{p.s} 2^{p.e-B}(1.p.m), \text{ where } B = 2^7 - 1 \\
\underline{h} &= (h_{31}, h_{30}, \cdots, h_1, h_0),\ h_i \in \{0,1\} \\
&\quad \text{format is the same as } \underline{p} \\
\underline{s} &= (s_{31}, s_{30}, \cdots, s_1, s_0),\ s_i \in \{0,1\} \\
&\quad \text{format is the same as } \underline{p}
\end{aligned}$$

outputs:

$$\begin{aligned}
TC_L &\quad TC_L \in \{0,1\} \\
TC_R &\quad TC_R \in \{0,1\}
\end{aligned}$$

function:

$$\begin{aligned}
TC_L &= \left\{ \begin{array}{ll} 1 & \text{if } p_{val} - h_{val} < s_{val} \\ 0 & \text{otherwise} \end{array} \right\} \text{with } t_{TSA-delay} \text{ cycle delay} \\
TC_R &= \left\{ \begin{array}{ll} 1 & \text{if } p_{val} + h_{val} > s_{val} \\ 0 & \text{otherwise} \end{array} \right\} \text{with } t_{TSA-delay} \text{ cycle delay}
\end{aligned}$$

$t_{TSA-delay}$ is the number of cycles required until the output is given. It is equal to

$$t_{TSA-delay} = [\text{bit width of } p.m] + 1 \text{ (hidden bit)} + 1$$

The TSA is comprised of the following submodules as shown in this diagram. MSB stands for most significant bit.

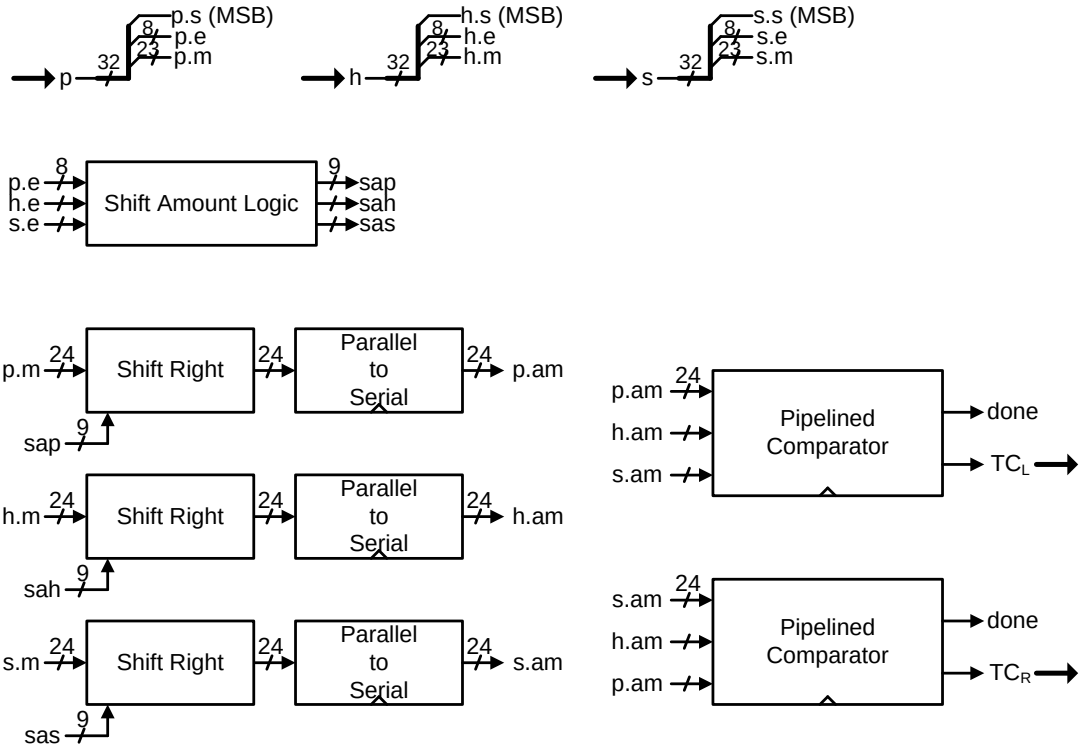


Figure 3.12: Components of TSA

The Shift Amount Logic and Shift Right Logic combine to align the mantissa sections of p, h and s and hand them over to the Parallel-to-Serial (P2S) module. Here, the mantissa values move through a series of registers and are changed from parallel form to bit-serial form. The two pipelined comparators are given these values and determines the TC_L and TC_R values, which are the final outputs of the TSA module. More details of each module are shown in the following sections.

3.4.3 Submodule : Shift Amount Logic

In order to align the mantissa values, we need to calculate the exponent differences and shift the values accordingly. The Shift Amount Logic module takes the

exponent values of p, h and s as inputs, and calculates the number of bits that each mantissa needs to be shifted.

To this goal, the module does two things. First, it determines the largest value of the three exponents. Next, it calculates the shift amount for each variable, which is equal to the input exponent subtracted from the largest value found in the previous step. As all exponents are biased by the same amount ($B = 2^7 - 1$), we do not have to take the bias B into account, only the difference values are relevant. The variable with the largest exponent is not shifted at all, and any variable with an exponent that is smaller than the largest exponent has its mantissa shifted by the difference of the two exponents.

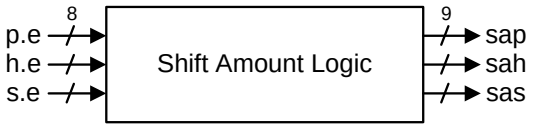


Figure 3.13: Overview of Shift Amount Logic

The high level specification is shown below.

inputs:

$$\begin{aligned}
\underline{p.e} &= (p.e_7, p.e_6, \cdots, p.e_1, p.e_0),\ p.e_i \in \{0,1\} \\
\underline{h.e} &= (h.e_7, h.e_6, \cdots, h.e_1, h.e_0),\ h.e_i \in \{0,1\} \\
\underline{s.e} &= (s.e_7, s.e_6, \cdots, s.e_1, s.e_0),\ s.e_i \in \{0,1\} \\
&\text{the biased exponent parts of } \underline{p}, \underline{h} \text{ and } \underline{s}
\end{aligned}$$

outputs:

$$\begin{aligned}
\underline{sap} &= (sap_8, sap_7, \cdots, sap_1, sap_0),\ sap_i \in \{0,1\} \\
\underline{sah} &= (sah_8, sah_7, \cdots, sah_1, sah_0),\ sah_i \in \{0,1\} \\
\underline{sas} &= (sas_8, sas_7, \cdots, sas_1, sas_0),\ sas_i \in \{0,1\} \\
&\text{shift amount for } p, h \text{ and } s
\end{aligned}$$

function:

$$\underline{sap} = \begin{cases} 0 & \text{if } \underline{p.e} = \underline{max.e} \\ \underline{s.e} - \underline{p.e} & \text{if } \underline{s.e} = \underline{max.e} \\ \underline{p.e} - \underline{h.e} & \text{otherwise} \end{cases}$$

$$\underline{sah} = \begin{cases} 0 & \text{if } \underline{h.e} = \underline{max.e} \\ \underline{p.e} - \underline{h.e} & \text{if } \underline{p.e} = \underline{max.e} \\ \underline{h.e} - \underline{s.e} & \text{otherwise} \end{cases}$$

$$\underline{sas} = \begin{cases} 0 & \text{if } \underline{s.e} = \underline{max.e} \\ \underline{h.e} - \underline{s.e} & \text{if } \underline{h.e} = \underline{max.e} \\ \underline{s.e} - \underline{p.e} & \text{otherwise} \end{cases}$$

where $\underline{max.e} = \text{MAX}(\underline{p.e}, \underline{h.e}, \underline{s.e})$

The output values are a form of encoded information on how many bits each mantissa needs to be shifted in order to align the three for comparison. Details on how this is used is discussed in the next section.

To do this in a efficient manner, the Shift Amount Logic module calculates three values as shown in Figure 3.14. The 8-bit exponent values $p.e$, $h.e$ and $s.e$ are added a 0 at the most significant bit and sign extended into $p.x$, $h.x$ and $s.x$ in order to incorporate for negative result values. Since all exponent values are positive due to the bias, $p.x = p.e$. Therefore, in the discussion below we shall use $p.e$ to represent both terms.

The sgn_i bit is the most significant bit of each adder output. From this we can infer the results shown in Table 3.2, and in turn, find which of the three input exponents is no smaller than the other two exponents.

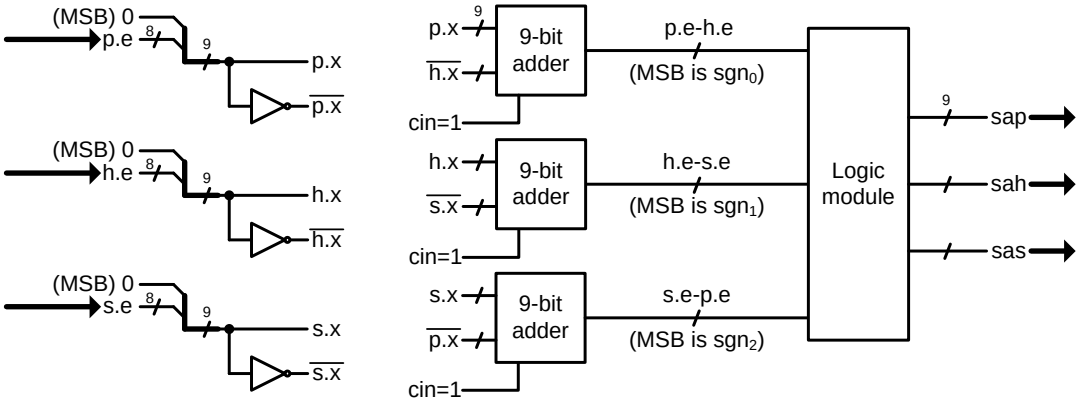


Figure 3.14: Components of Shift Amount Logic

Sign bit	Value	Meaning
sgn_0	0	$\underline{p.e} \geq \underline{h.e}$
	1	$\underline{h.e} > \underline{p.e}$
sgn_1	0	$\underline{h.e} \geq \underline{s.e}$
	1	$\underline{s.e} > \underline{h.e}$
sgn_2	0	$\underline{s.e} \geq \underline{p.e}$
	1	$\underline{p.e} > \underline{s.e}$

Table 3.2: Inequality for each sign bit

Given this information, we can now identify one of the largest exponent variables. Once this is identified, the mantissa shift amount of each variable can be obtained by subtracting the exponent value from the largest exponent. The shift amount values will all be a non-negative integer, with at least one (which belongs to the variable with the largest exponent) being 0.

Here, we examine each case one by one:

1. $sgn_0 = 0$, $sgn_1 = 0$, $sgn_2 = 0$: From the values we have $\underline{p.e} \geq \underline{h.e}$, $\underline{h.e} \geq \underline{s.e}$

and $\underline{s.e} \geq \underline{p.e}$. From the first two, we get $\underline{p.e} \geq \underline{h.e} \geq \underline{s.e}$, and the only way this and $\underline{s.e} \geq \underline{p.e}$ is true is if all three are the same value, so $\underline{p.e} = \underline{h.e} = \underline{s.e}$.

2. $sgn_0 = 0$, $sgn_1 = 0$, $sgn_2 = 1$: From the values we have $\underline{p.e} \geq \underline{h.e}$, $\underline{h.e} \geq \underline{s.e}$ and $\underline{p.e} > \underline{s.e}$. From the first two, we get $\underline{p.e} \geq \underline{h.e} \geq \underline{s.e}$, and since $\underline{p.e} > \underline{s.e}$, either $\underline{p.e} > \underline{h.e}$ or $\underline{h.e} > \underline{s.e}$, or both are true. In any case, we can be sure that $\underline{p.e}$ is not smaller than any other exponent.

3. $sgn_0 = 0$, $sgn_1 = 1$, $sgn_2 = 0$: From the values we have $\underline{p.e} \geq \underline{h.e}$, $\underline{s.e} > \underline{h.e}$ and $\underline{s.e} \geq \underline{p.e}$. From the first and third inequality, we get $\underline{s.e} \geq \underline{p.e} \geq \underline{h.e}$, and since $\underline{s.e} > \underline{h.e}$, either $\underline{s.e} > \underline{p.e}$ or $\underline{p.e} > \underline{h.e}$, or both are true. In any case, we can be sure that $\underline{s.e}$ is not smaller than any other exponent.

4. $sgn_0 = 0$, $sgn_1 = 1$, $sgn_2 = 1$: From the values we have $\underline{p.e} \geq \underline{h.e}$, $\underline{s.e} > \underline{h.e}$ and $\underline{p.e} > \underline{s.e}$. From the second and third inequality, we get $\underline{p.e} > \underline{s.e} > \underline{h.e}$, and this holds true with the first $\underline{p.e} \geq \underline{h.e}$. So we can be sure that $\underline{p.e}$ is the largest exponent.

5. $sgn_0 = 1$, $sgn_1 = 0$, $sgn_2 = 0$: From the values we have $\underline{h.e} > \underline{p.e}$, $\underline{h.e} \geq \underline{s.e}$ and $\underline{s.e} \geq \underline{p.e}$. From the second and third inequality, we get $\underline{h.e} \geq \underline{s.e} \geq \underline{p.e}$, and since $\underline{h.e} > \underline{p.e}$, either $\underline{h.e} > \underline{s.e}$ or $\underline{s.e} > \underline{p.e}$, or both are true. In any case, we can be sure that $\underline{h.e}$ is not smaller than any other exponent.

6. $sgn_0 = 1$, $sgn_1 = 0$, $sgn_2 = 1$: From the values we have $\underline{h.e} > \underline{p.e}$, $\underline{h.e} \geq \underline{s.e}$ and $\underline{p.e} > \underline{s.e}$. From the first and third inequality, we get $\underline{h.e} > \underline{p.e} > \underline{s.e}$, and this holds true with the second $\underline{h.e} \geq \underline{s.e}$. So we can be sure that $\underline{h.e}$ is the largest exponent.

7. $sgn_0 = 1$, $sgn_1 = 1$, $sgn_2 = 0$: From the values we have $\underline{h.e} > \underline{p.e}$, $\underline{s.e} > \underline{h.e}$ and $\underline{s.e} \geq \underline{p.e}$. From the first and second inequality, we get $\underline{s.e} > \underline{h.e} > \underline{p.e}$ and this holds true with the third $\underline{s.e} \geq \underline{p.e}$. So we can be sure that $\underline{s.e}$ is the largest exponent.

8. $sgn_0 = 1$, $sgn_1 = 1$, $sgn_2 = 1$: From the values we have $\underline{h.e} > \underline{p.e}$, $\underline{s.e} > \underline{h.e}$ and $\underline{p.e} > \underline{s.e}$. From the first two, we get $\underline{s.e} > \underline{h.e} > \underline{p.e}$, from which we can derive that $\underline{s.e} > \underline{p.e}$. However, this cannot be true at the same time with the third $\underline{p.e} > \underline{s.e}$. So this is not possible and a don't-care case.

At this point, the three subtraction results that we have on hand are $\underline{p.e} - \underline{h.e}$, $\underline{h.e} - \underline{s.e}$ and $\underline{s.e} - \underline{p.e}$. In order to obtain the value of $\underline{h.e} - \underline{p.e}$, $\underline{s.e} - \underline{h.e}$ or $\underline{p.e} - \underline{s.e}$, we need to perform a change-of-sign operation. This is calculated in two's-complement arithmetic by inverting all bits and adding 1 to the least significant bit. For example, to get $\underline{h.e} - \underline{p.e}$, we invert all bits to get $\overline{\underline{h.e} - \underline{p.e}}$, and add 1 to it. Doing this will require vector multiplexers and one additional stage of a 9-bit carry propagate adder. Instead of placing adders here, the shift information is simply relayed to the Shift Right module in its negative form, and the appropriate number conversion is done previous to the shift right operation. This is a better solution in terms of worst case delay, since instead of a potential 9-bit propagation delay, we have a single 2-bit vector multiplexer.

This information is summed up in Tables 3.3 and 3.4.

$sgn_{0/1/2}$	Inequality	Largest
0 0 0	$\underline{p.e} \geq \underline{h.e} \geq \underline{s.e} \geq \underline{p.e}$	all are equal
0 0 1	$\underline{p.e} \geq \underline{h.e} \geq \underline{s.e}$ and $\underline{p.e} > \underline{s.e}$	$\underline{p.e}$
0 1 0	$\underline{s.e} \geq \underline{p.e} \geq \underline{h.e}$ and $\underline{s.e} > \underline{h.e}$	$\underline{s.e}$
0 1 1	$\underline{p.e} > \underline{s.e} > \underline{h.e}$ and $\underline{p.e} \geq \underline{h.e}$	$\underline{p.e}$
1 0 0	$\underline{h.e} \geq \underline{s.e} \geq \underline{p.e}$ and $\underline{h.e} > \underline{p.e}$	$\underline{h.e}$
1 0 1	$\underline{h.e} > \underline{p.e} > \underline{s.e}$ and $\underline{h.e} \geq \underline{s.e}$	$\underline{h.e}$
1 1 0	$\underline{s.e} > \underline{h.e} > \underline{p.e}$ and $\underline{s.e} \geq \underline{p.e}$	$\underline{s.e}$
1 1 1	$\underline{s.e} > \underline{h.e} > \underline{p.e}$ and $\underline{p.e} > \underline{s.e}$	$\times$

Table 3.3: Determining largest exponent

$sgn_{0/1/2}$	Largest	shift bits p	shift bits h	shift bits s
0 0 0	all	0	0	0
0 0 1	$\underline{p.e}$	0	$\underline{p.e} - \underline{h.e}$	$\overline{\underline{s.e} - \underline{p.e}} + 1$
0 1 0	$\underline{s.e}$	$\underline{s.e} - \underline{p.e}$	$\overline{\underline{h.e} - \underline{s.e}} + 1$	0
0 1 1	$\underline{p.e}$	0	$\underline{p.e} - \underline{h.e}$	$\overline{\underline{s.e} - \underline{p.e}} + 1$
1 0 0	$\underline{h.e}$	$\overline{\underline{p.e} - \underline{h.e}} + 1$	0	$\underline{h.e} - \underline{s.e}$
1 0 1	$\underline{h.e}$	$\overline{\underline{p.e} - \underline{h.e}} + 1$	0	$\underline{h.e} - \underline{s.e}$
1 1 0	$\underline{s.e}$	$\underline{s.e} - \underline{p.e}$	$\overline{\underline{h.e} - \underline{s.e}} + 1$	0
1 1 1	$\times$	$\times$	$\times$	$\times$

Table 3.4: Output logic according to largest exponent

3.4.4 Submodule : Shift Right

Once the shift amounts are determined from the previous module, using this information, the Shift Right module aligns the mantissa bits accordingly.

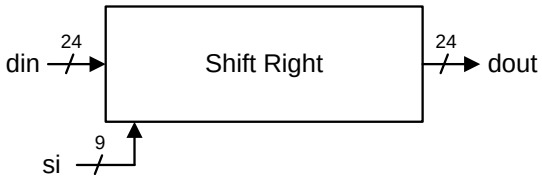


Figure 3.15: Overview of Shift Right

The high level specification for the Shift Right module is shown below.

$$
\begin{aligned}
\text{inputs:} \quad \underline{si} &= (si_8, si_7, \cdots, si_1, si_0),\ si_i \in \{0,1\} \\
\underline{din} &= (din_{23}, din_{22}, \cdots, din_1, din_0),\ din_i \in \{0,1\} \\
\text{outputs:} \quad \underline{dout} &= (dout_{23}, dout_{22}, \cdots, dout_1, dout_0),\ dout_i \in \{0,1\} \\
\text{function:} \quad dout_i &= \begin{cases} din_{i+s} & \text{if } i+s \geq 23 \\ 0 & \text{otherwise} \end{cases} \\
& \text{where } s = \left| \sum_{i=0}^{7} 2^i si_i \right|
\end{aligned}
$$

The main part of the module is a barrel shifter, which shifts the input mantissa $\underline{din}$ to the right, according to the value of $\underline{sd}$ (shift distance) and additional shift. Each 2^i-bit right shifter stage is a 2-input vector multiplexer, which selects whether the output is unchanged from the input or shifted 2^i-bits to the right. All bits that are shifted in are always 0.

The value of $\underline{sd}$ is derived from the input $\underline{si}$. In order to avoid having an extra adder stage in the previous module, the shift amount is handed over as a negative number in some cases. In this case, we need to convert $\underline{sd}$ back into a positive value by inverting all bits and adding 1 to the least significant digit position. Since this would require an 8-bit carry propagate adder to fully calculate, we instead have an additional level at the end of the barrel shifter, which is controlled by the signal named additional shift.

The sign of the input can be easily determined by looking at the most significant bit si_8. From this, we can determine the sd bits as follows:

1. if $si_8 = 0$, $sd_i = si_i$ for $i = 0$ to 7, additional shift = 0
2. if $si_8 = 1$, $sd_i = \overline{si_i}$ for $i = 0$ to 7, additional shift = 1

Note that the additional shift bit is always the same as si_8.

Once the shift distance sd is determined, the actual shifting can begin. Since the mantissa is 24-bits wide, if any bit in the range sd_{7-5} is 1, the input mantissa

is completely shifted out and everything will be 0. If this is not the case, the 5 least significant bits of *sd* are given as control signals to the barrel shifter. There is a separate shifter stage for each of the 5 bits and the final additional shift bit. Each shifter stage in the barrel shifter is a 2-input vector multiplexer. For details on the barrel shifter implementation, one can refer to standard text such as [15].

The module design details can be seen in Figure 3.16.

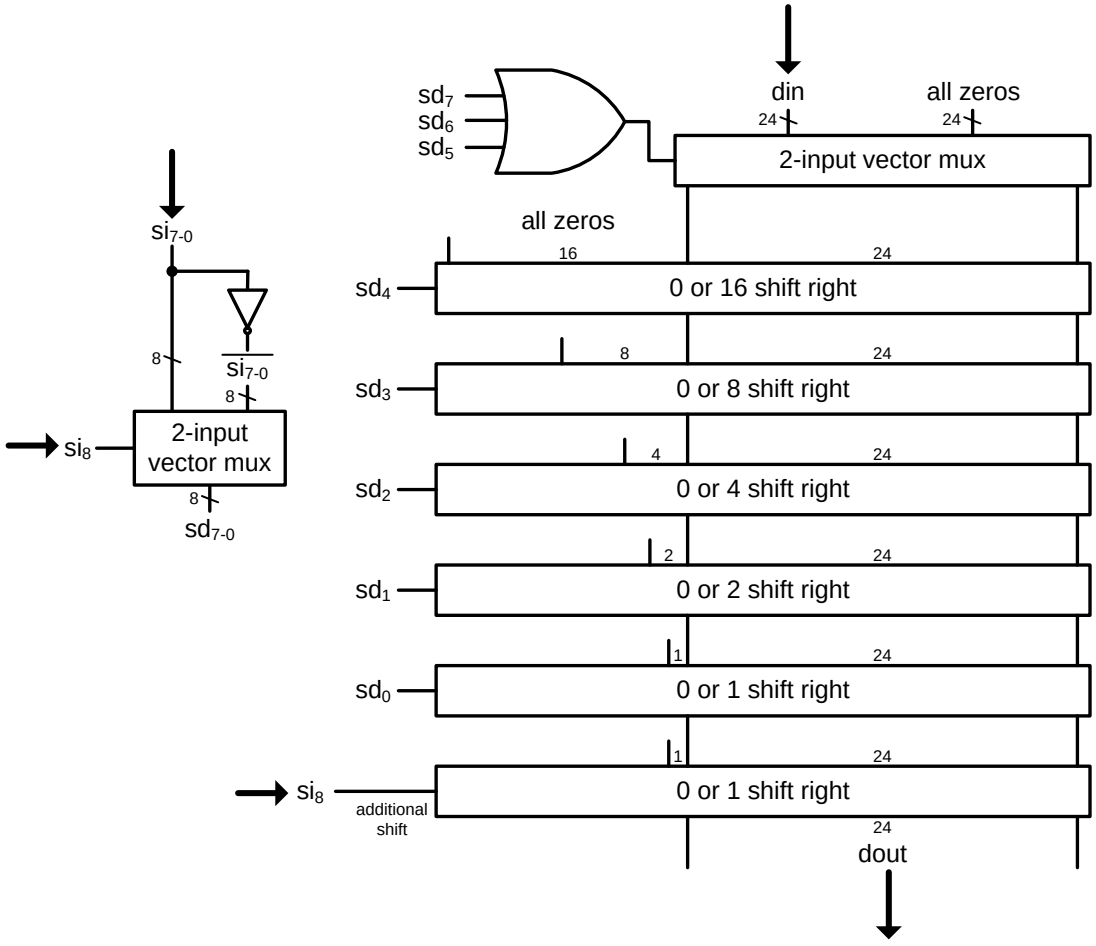


Figure 3.16: Components of Shift Right

Note that the input *din* is the mantissa bits for p, h and s with the hidden bit included. In other words, the shift input *din* is equal to $p.mwh$, $h.mwh$ or $s.mwh$. The output signals are the aligned mantissa including hidden bit for p, h or s.

3.4.5 Submodule : P2S (Parallel to Serial)

The Parallel to Serial module converts parallel inputs into digit-serial form using a series of shift registers.

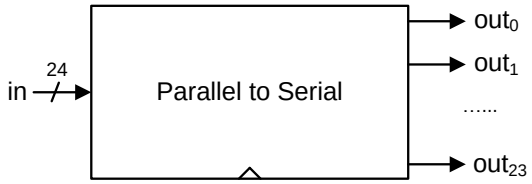


Figure 3.17: Overview of Parallel to Serial

The high level specification for the Parallel to Serial Converter module is shown below.

$$\begin{aligned} \text{inputs:} \quad & \underline{in} &=& \ (in_{23}, in_{22}, \cdots, in_1, in_0),\ in_i \in \{0,1\} \\ \text{outputs:} \quad & \underline{out} &=& \ (out_{23}, out_{22}, \cdots, out_1, out_0),\ out_i \in \{0,1\} \\ \text{function:} \quad & out_i(t) &=& \ in_i(t-(23-i)) \end{aligned}$$

The P2S module is a triangular array of registers delaying each separate bit of the input by the number of clock cycles that corresponds to the distance from the most significant bit, allowing each bit to reach the pipelined module at the correct cycle. For an n-bit input vector in, with the most significant bit at index $n-1$ and the least significant bit at index 0, the bit in_i is delayed by $n-1-i$ cycles.

For example, if the current input is $in(123)$, the output bits at $t = 123$ will be:

$$\begin{aligned} out_{23}(123) &= in_{23}(123) \\ out_{22}(123) &= in_{22}(122) \\ out_{21}(123) &= in_{21}(121) \\ &\cdots \\ out_{1}(123) &= in_{1}(101) \\ out_{0}(123) &= in_{0}(100) \end{aligned}$$

Details of the register connections is shown in Figure 3.18. LSB stands for least significant bit.

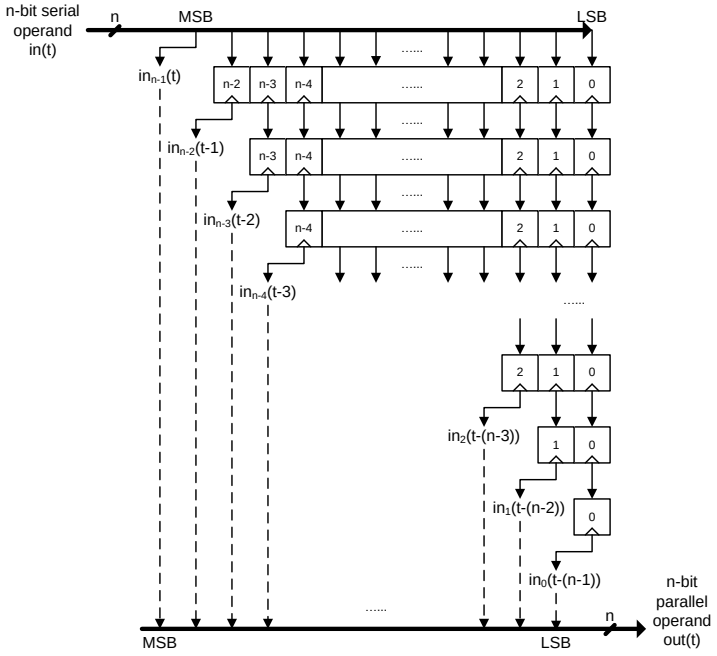


Figure 3.18: Components of Parallel to Serial

The data bits that are being transformed from parallel to serial in the P2S module are the aligned mantissa of either p, h or s, including the hidden bit. We

shall refer to this as $p.am$, $h.am$ and $s.am$ for aligned mantissa. Note that this includes the hidden bit.

3.4.6 Submodule : Pipelined Comparator

The goal of this module is to compare the aligned mantissa with hidden bit values of $p-h$ and s one bit per each clock cycle, and determine which is the larger value. This is handled in a pipelined online manner, where each slice receives information about the previous stages and processes one additional digit per clock cycle, and the results are handed over to the next slice. The main idea for the design of this module is adapted from [47].

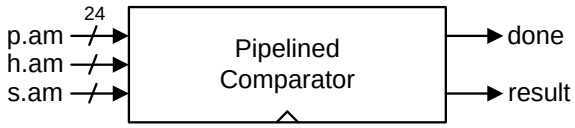


Figure 3.19: Overview of Pipelined Comparator

The high level specification for the Pipelined Comparator is shown below.

$$
\begin{aligned}
\text{inputs:} \quad \underline{p.am} &= (p.am_{23}, p.am_{22}, \cdots, p.am_1, p.am_0),\ p.am_i \in \{0,1\} \\
&\quad \text{at time } t,\ p.am_i(t) = p.am_i(t-(23-i)) \\
\underline{h.am} &= (h.am_{23}, h.am_{22}, \cdots, h.am_1, h.am_0),\ h.am_i \in \{0,1\} \\
&\quad \text{at time } t,\ h.am_i(t) = h.am_i(t-(23-i)) \\
\underline{s.am} &= (s.am_{23}, s.am_{22}, \cdots, s.am_1, s.am_0),\ s.am_i \in \{0,1\} \\
&\quad \text{at time } t,\ s.am_i(t) = s.am_i(t-(23-i)) \\
\text{outputs:} \quad done &= done \in \{0,1\} \\
result &= result \in \{0,1\}
\end{aligned}
$$

$$\text{function:} \quad done \;=\; \begin{cases} 1 & \text{if } p.am(t-24) - h.am(t-24) < s.am(t-24) \\ & \text{or if } p.am(t-24) - h.am(t-24) > s.am(t-24) \\ & \text{and this is determined before the last slice} \\ 0 & \text{otherwise} \end{cases}$$

$$result \;=\; \begin{cases} 1 & \text{if } p.am(t-24) - h.am(t-24) > s.am(t-24) \\ 0 & \text{otherwise} \end{cases}$$

The comparator compares the values of $p.am - h.am$ and $s.am$, and determines whether $p.am - h.am > s.am$, $p.am - h.am = s.am$ or $p.am - h.am < s.am$. This can be determined by looking at the two output bits, *result* and *done*, as shown in Table 3.5.

result	*done*	meaning
0	0	$p.am - h.am = s.am$
0	1	$p.am - h.am < s.am$
1	0	$p.am - h.am > s.am$
1	1	$p.am - h.am > s.am$

Table 3.5: Possible output combinations of the comparator

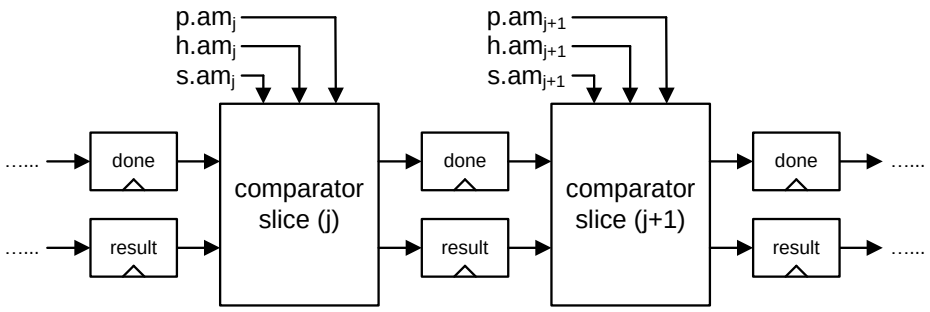


Figure 3.20: Partial view of pipelined slices

Figure 3.20 shows part of the pipelined comparator stages at index j, where there is one bit from the aligned mantissas with hidden bit of p, h and s given

to each slice, along with the d (done) and r (result) bits from the previous stage. Each slice processes this information to produce d and r bits and relay them to the next slice through the pipeline registers.

We shall use $p.am[j]$ to refer to the partial numerical value of $p.am$, represented by the first j bits. The value of $p.am[j] = \sum_{i=0}^{j-1} 2^{j-1-i} p.am_i$ At each index j, the pipeline slice determines the inequality of $p.am[j] - h.am[j]$ and $s.am[j]$ from the information given, and hands over the necessary information to the next slice.

One advantage of examining from the most significant bits is that once we are sure of the result at index j, the same result is true for any index larger than j as the following bits have a smaller weight. Although, we need to consider the fact that the values range at each bit position is $\{0, 1\}$ for $s.am[j]$, but $\{-1, 0, 1\}$ for $p.am[j] - h.am[j]$. Therefore, we cannot assume the inequality finalized when $p.am[j] - h.am[j]$ is larger than $s.am[j]$ by 1, since the value of $p.am[j+1] - h.am[j+1]$ can be -1 and affect the value at slice j. The comparison is complete at slice j only when $s.am[j]$ is larger than $p.am[j] - h.am[j]$ by 1 or more, or if $p.am[j] - h.am[j]$ is larger than $s.am[j]$ by 2 or more. Therefore at least two stages need to be examined before we can obtain the end result. Also, some information from the immediate previous stage needs to be relayed between the slices. This information is transferred through the $carry_j$ bit. The value of $carry_j = (p.am[j-1] - h.am[j-1]) - s.am[j-1]$.

As a result, the logic at each slice compares the inequality of $p.am[j] - h.am[j] + 2carry_j$ and $s.am[j]$. Let us define $p.am[j] - h.am[j] + 2carry_j$ as the value $lval[j]$. This will be the value that $s.am[j]$ is compared to at each slice.

Below, we shall examine the combinations of different inputs and what values need to be produced for each case.

1. $carry_j = 0, p.am[j] = 0, h.am[j] = 0, s.am[j] = 0$: First, $lval[j] = p.am[j] - h.am[j] + 2carry_j = 0 - 0 + 2 \cdot 0 = 0$. As this is equal to $s.am[j] = 0$, the

values are the same so far and we cannot determine the inequality yet. $carry_{j+1} = (p.am[j] - h.am[j] + 2carry_j) - s.am[j] = (0 - 0 + 2 \cdot 0) - 0 = 0$.

2. $carry_j = 0, p.am[j] = 0, h.am[j] = 0, s.am[j] = 1$: First, $lval[j] = p.am[j] - h.am[j] + 2carry_j = 0 - 0 + 2 \cdot 0 = 0$. This is smaller than $s.am[j] = 1$ by 1, therefore we can be sure that $p.am[k] - h.am[k]$ is smaller than $s.am[k]$ for any $k \geq j$. No need to calculate the $carry_{j+1}$ bit since we are done.

3. $carry_j = 0, p.am[j] = 0, h.am[j] = 1, s.am[j] = 0$: First, $lval[j] = p.am[j] - h.am[j] + 2carry_j = 0 - 1 + 2 \cdot 0 = -1$. This is smaller than $s.am[j] = 0$ by 1, therefore we can be sure that $p.am[k] - h.am[k]$ is smaller than $s.am[k]$ for any $k \geq j$. No need to calculate the $carry_{j+1}$ bit since we are done.

4. $carry_j = 0, p.am[j] = 0, h.am[j] = 1, s.am[j] = 1$: First, $lval[j] = p.am[j] - h.am[j] + 2carry_j = 0 - 1 + 2 \cdot 0 = -1$. This is smaller than $s.am[j] = 1$ by 2, therefore we can be sure that $p.am[k] - h.am[k]$ is smaller than $s.am[k]$ for any $k \geq j$. No need to calculate the $carry_{j+1}$ bit since we are done.

5. $carry_j = 0, p.am[j] = 1, h.am[j] = 0, s.am[j] = 0$: First, $lval[j] = p.am[j] - h.am[j] + 2carry_j = 1 - 0 + 2 \cdot 0 = 1$. This is larger than $s.am[j] = 0$ by 1, but the difference is small enough that if $p.am[j+1] - h.am[j+1] = -1$ this may need to be changed to 0. Therefore we cannot determine the inequality yet. $carry_{j+1} = (p.am[j] - h.am[j] + 2carry_j) - s.am[j] = (1 - 0 + 2 \cdot 0) - 0 = 1$.

6. $carry_j = 0, p.am[j] = 1, h.am[j] = 0, s.am[j] = 1$: First, $lval[j] = p.am[j] - h.am[j] + 2carry_j = 1 - 0 + 2 \cdot 0 = 1$. As this is equal to $s.am[j] = 1$, the values are the same so far and we cannot determine the inequality yet. $carry_{j+1} = (p.am[j] - h.am[j] + 2carry_j) - s.am[j] = (1 - 0 + 2 \cdot 0) - 1 = 0$.

7. $carry_j = 0, p.am[j] = 1, h.am[j] = 1, s.am[j] = 0$: First, $lval[j] = p.am[j] - h.am[j] + 2carry_j = 1 - 1 + 2 \cdot 0 = 0$. As this is equal to $s.am[j] = 0$, the

values are the same so far and we cannot determine the inequality yet. $carry_{j+1} = (p.am[j] - h.am[j] + 2carry_j) - s.am[j] = (1 - 1 + 2 \cdot 0) - 0 = 0$.

8. $carry_j = 0, p.am[j] = 1, h.am[j] = 1, s.am[j] = 1$: First, $lval[j] = p.am[j] - h.am[j] + 2carry_j = 1 - 1 + 2 \cdot 0 = 0$. This is smaller than $s.am[j] = 1$ by 1, therefore we can be sure that $p.am[k] - h.am[k]$ is smaller than $s.am[k]$ for any $k \geq j$. No need to calculate the $carry_{j+1}$ bit since we are done.

9. $carry_j = 1, p.am[j] = 0, h.am[j] = 0, s.am[j] = 0$: First, $lval[j] = p.am[j] - h.am[j] + 2carry_j = 0 - 0 + 2 \cdot 1 = 2$. This is larger than $s.am[j] = 0$ by 2, therefore we can be sure that $p.am[k] - h.am[k]$ is larger than $s.am[k]$ for any $k \geq j$. No need to calculate the $carry_{j+1}$ bit since we are done.

10. $carry_j = 1, p.am[j] = 0, h.am[j] = 0, s.am[j] = 1$: First, $lval[j] = p.am[j] - h.am[j] + 2carry_j = 0 - 0 + 2 \cdot 1 = 2$. This is larger than $s.am[j] = 1$ by 1, but the difference is small enough that if $p.am[j+1] - h.am[j+1] = -1$ this may need to be changed to 0. Therefore we cannot determine the inequality yet. $carry_{j+1} = (p.am[j] - h.am[j] + 2carry_j) - s.am[j] = (0 - 0 + 2 \cdot 1) - 1 = 1$.

11. $carry_j = 1, p.am[j] = 0, h.am[j] = 1, s.am[j] = 0$: First, $lval[j] = p.am[j] - h.am[j] + 2carry_j = 0 - 1 + 2 \cdot 1 = 1$. This is larger than $s.am[j] = 0$ by 1, but the difference is small enough that if $p.am[j+1] - h.am[j+1] = -1$ this may need to be changed to 0. Therefore we cannot determine the inequality yet. $carry_{j+1} = (p.am[j] - h.am[j] + 2carry_j) - s.am[j] = (0 - 1 + 2 \cdot 1) - 0 = 1$.

12. $carry_j = 1, p.am[j] = 0, h.am[j] = 1, s.am[j] = 1$: First, $lval[j] = p.am[j] - h.am[j] + 2carry_j = 0 - 1 + 2 \cdot 1 = 1$. As this is equal to $s.am[j] = 1$, the values are the same so far and we cannot determine the inequality yet. $carry_{j+1} = (p.am[j] - h.am[j] + 2carry_j) - s.am[j] = (0 - 1 + 2 \cdot 1) - 1 = 0$.

13. $carry_j = 1, p.am[j] = 1, h.am[j] = 0, s.am[j] = 0$: First, $lval[j] = p.am[j] - h.am[j] + 2carry_j = 1 - 0 + 2 \cdot 1 = 3$. This is larger than $s.am[j] = 0$ by 3,

therefore we can be sure that $p.am[k] - h.am[k]$ is larger than $s.am[k]$ for any $k \geq j$. No need to calculate the $carry_{j+1}$ bit since we are done.

14. $carry_j = 1, p.am[j] = 1, h.am[j] = 0, s.am[j] = 1$: First, $lval[j] = p.am[j] - h.am[j] + 2carry_j = 1 - 0 + 2 \cdot 1 = 3$. This is larger than $s.am[j] = 1$ by 2, therefore we can be sure that $p.am[k] - h.am[k]$ is larger than $s.am[k]$ for any $k \geq j$. No need to calculate the $carry_{j+1}$ bit since we are done.

15. $carry_j = 1, p.am[j] = 1, h.am[j] = 1, s.am[j] = 0$: First, $lval[j] = p.am[j] - h.am[j] + 2carry_j = 1 - 1 + 2 \cdot 1 = 2$. This is larger than $s.am[j] = 0$ by 2, therefore we can be sure that $p.am[k] - h.am[k]$ is larger than $s.am[k]$ for any $k \geq j$. No need to calculate the $carry_{j+1}$ bit since we are done.

16. $carry_j = 1, p.am[j] = 1, h.am[j] = 1, s.am[j] = 1$: First, $lval[j] = p.am[j] - h.am[j] + 2carry_j = 1 - 1 + 2 \cdot 1 = 2$. This is larger than $s.am[j] = 1$ by 1, but the difference is small enough that if $p.am[j+1] - h.am[j+1] = -1$ this may need to be changed to 0. Therefore we cannot determine the inequality yet. $carry_{j+1} = (p.am[j] - h.am[j] + 2carry_j) - s.am[j] = (1 - 1 + 2 \cdot 1) - 1 = 1$.

The different combinations of $carry_j$, $p.am[j]$, $h.am[j]$ and $s.am[j]$, along with the possible outcomes, are shown in Table 3.6. Part of the table is adapted from [47]. Here, the value $eval = lval[j] \ ? \ s.am[j]$.

$carry_j$	$p.am[j]$	$h.am[j]$	$lval[j]$	$s.am[j]$	$eval$	$carry_{j+1}$	done?
0	0	0	0	0	$\times$	0	n
0	0	0	0	1	$<$	$\times$	y
0	0	1	-1	0	$<$	$\times$	y
0	0	1	-1	1	$<$	$\times$	y
0	1	0	1	0	$\times$	1	n
0	1	0	1	1	$\times$	0	n
0	1	1	0	0	$\times$	0	n
0	1	1	0	1	$<$	$\times$	y
1	0	0	2	0	$>$	$\times$	y
1	0	0	2	1	$\times$	1	n
1	0	1	1	0	$\times$	1	n
1	0	1	1	1	$\times$	0	n
1	1	0	3	0	$>$	$\times$	y
1	1	0	3	1	$>$	$\times$	y
1	1	1	2	0	$>$	$\times$	y
1	1	1	2	1	$\times$	1	n

Table 3.6: Logic table for comparator slice (adapted from [47])

In the table, we see that the value $eval = lval[j]\ ?\ s.am[j]$ is meaningful only when the done bit is 'y', and the $carry_{i+1}$ bit is meaningful only when the done bit is 'n'. Therefore, if we combine the $lval[j]\ ?\ s.am[j]$ and $carry_{i+1}$ bits into a single *result* bit, all the information can be expressed using two single-bit outputs.

The truth table for this is shown in Table 3.7. For the *done* flag, 0 is 'n' (inequality is not determined yet) and 1 is 'y' (inequality is determined). The meaning of the *result* flag changes according to the *done* flag, if *done* is 0, *result* is the $carry_j$ bit, and if *done* is 1, 0 stands for $p-h < s$ and 1 stands for $p-h > s$.

$carry_j$	$p.am[j]$	$h.am[j]$	$s.am[j]$	$result$	$done$
0	0	0	0	0	0
0	0	0	1	0	1
0	0	1	0	0	1
0	0	1	1	0	1
0	1	0	0	1	0
0	1	0	1	0	0
0	1	1	0	0	0
0	1	1	1	0	1
1	0	0	0	1	1
1	0	0	1	1	0
1	0	1	0	1	0
1	0	1	1	0	0
1	1	0	0	1	1
1	1	0	1	1	1
1	1	1	0	1	1
1	1	1	1	1	0

Table 3.7: Truth table of comparator slice circuit (adapted from [47])

The implementation diagram is shown in 3.21. The truth table logic is implemented in the module CompLogic. Some additional multiplexers are needed to carry the previous result bit to the next, if the comparison is finished at a previous stage.

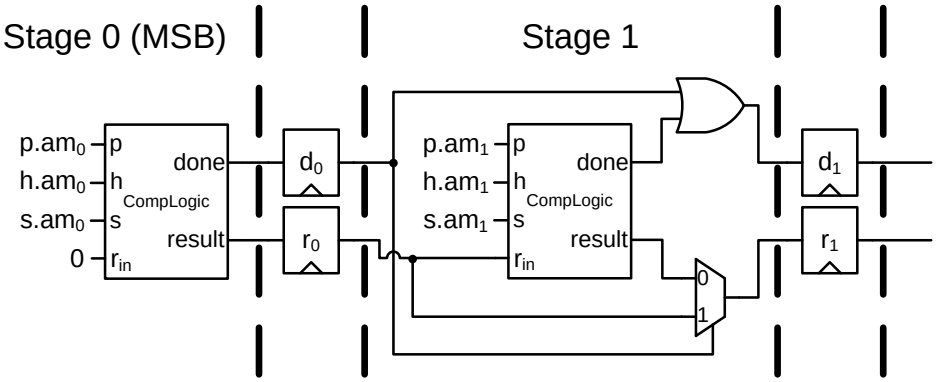


Figure 3.21: Components for Comparator slice (adapted from [47])

In order to do the second comparison between s and $p + h$, we can use the same pipelined comparator, only with the inputs p and s swapped. Now this will calculate whether $s - h < p$ is true, which is equal to $s < p + h$, and this is exactly the inequality we need.

3.5 The Shader Operation block

3.5.1 Overview

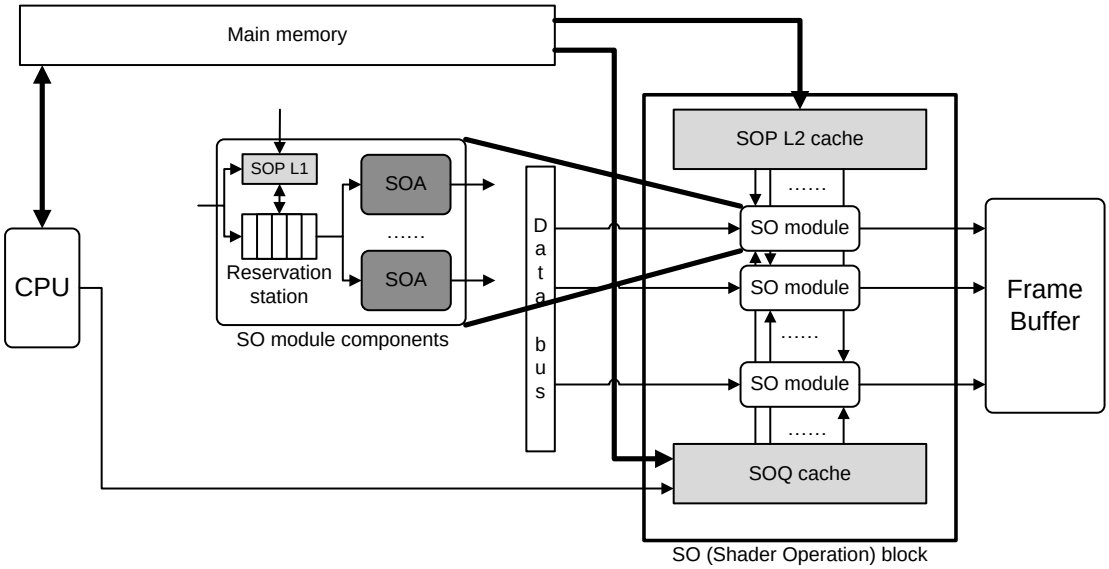


Figure 3.22: The SO block

The second part of the MAPM architecture is the Shader Operation block, shown in Figure 3.22. Once the TS block finds appropriate query point and photon pairs, they are handed over to the SO block via the Data bus. All necessary variables for each SO instruction is fetched, and once they arrive, the color values are calculated using digit-serial online arithmetic.

Below, we can see the high level specifications of the SO block.

inputs:

$\underline{px}$ = $(px_{31}, px_{30}, \cdots, px_1, px_0)$, $px_i \in \{0, 1\}$

px_{31} is the sign bit $px.s$,

px_{30} to px_{23} is the 8-bit biased exponent $px.e$,

and px_{22} to px_0 is the 23-bit mantissa part $px.m$

$px_{val} = (-1)^{px.s} 2^{px.e-B}(1.px.m)$, where $B = 2^7 - 1$

$\underline{py}$ = $(py_{31}, py_{30}, \cdots, py_1, py_0)$, $py_i \in \{0, 1\}$

format is the same as $\underline{px}$

$\underline{pz}$ = $(pz_{31}, pz_{30}, \cdots, pz_1, pz_0)$, $pz_i \in \{0, 1\}$

format is the same as $\underline{px}$

$\underline{qIn}$ = $(qIn_{31}, qIn_{30}, \cdots, qIn_1, qIn_0)$, $qIn_i \in \{0, 1\}$

$\underline{pIn}$ = $(pIn_{31}, pIn_{30}, \cdots, pIn_1, pIn_0)$, $pIn_i \in \{0, 1\}$

outputs:

$\underline{colorR}$ = $(colorR_{31}, colorR_{30}, \cdots, colorR_1, colorR_0)$,

$colorR_i \in \{0, 1\}$

format is the same as $\underline{px}$

$\underline{colorG}$ = $(colorG_{31}, colorG_{30}, \cdots, colorG_1, colorG_0)$,

$colorG_i \in \{0, 1\}$

format is the same as $\underline{px}$

$\underline{colorB}$ = $(colorB_{31}, colorB_{30}, \cdots, colorB_1, colorB_0)$,

$colorB_i \in \{0, 1\}$

format is the same as $\underline{px}$

function:

$\underline{colorR}$ = red color component of query point at index $\underline{qIn}$

(with $t_{SOB-delay}$ cycle delay)

$\underline{colorG}$ = green color component of query point at index $\underline{qIn}$

(with $t_{SOB-delay}$ cycle delay)

$\underline{colorB}$ = blue color component of query point at index $\underline{qIn}$

(with $t_{SOB-delay}$ cycle delay)

$\underline{qIn}$ is the integer index of the query point, and $\underline{pIn}$ is the integer index of the photon paired with the query point.

The exact value of $t_{SOB-delay}$ depends on various factors, including requests to memory and any stall cycles that may happen. However, the value is exactly the same for all three $\underline{colorX}$ values, so the three colors are calculated and given as output at the exact same clock cycle.

In the following sections, we look at the internal modules inside the SO block in more detail.

1. **Reservation station**

   In order to calculate the color values for each query point, we need to read a number of variables that is necessary for the calculation. Each SO instruction sits in the Reservation station until these values are fetched from memory. Their function is similar to the reservation stations utilized in Tomasulo's dynamic scheduling algorithm in hardware [54].

   The variables that are read here, as well as discussion on the data path it takes, can be seen in Section 3.5.2.

3.5.2 The two caches of the SO block

Table 3.8 shows all the variables that are required for a SO.

Name	Data type	Defined by
`kernel`	32-bit floating point	photon and query point
`BRDF`	32-bit floating point	photon and query point
h	32-bit floating point	none
`power.color`	3×32-bit floating point	photon
`contribution.color`	3×32-bit floating point	query point

Table 3.8: Required parameters for a single SO

Here, we can see that the variables can be divided into three distinct groups.

The first is `contribution.color`, which is strictly tied to the query point. The second is `power.color`, which is the power value of the photon and its value is defined by the photon. The final group is `kernel` and `BRDF`, which change according to both the photon and the query point. For each group, the data access pattern is different, and it is beneficial to divide the cache structure accordingly.

The access pattern for queries is sequential and predictable, since the CPU issues new tree searches by accessing the array of query points from the beginning to the end, one by one. This is illustrated in Figure 3.23. Once a tree search instruction enters the TS block, any further searches into its children has a higher priority than a whole new TS instruction, so if there are some children searches inside the Register queue, they are scheduled before a new TS instruction is read from the CPU.

Once a query point passes through the whole tree and ends its search at leaf nodes, that point is done and is never revisited. By having a dedicated cache that is direct-mapped, and an extremely sequential access pattern, the cache operates similar to a First-In First-Out queue. When overwrites happen, the oldest block is the first one to be evicted.

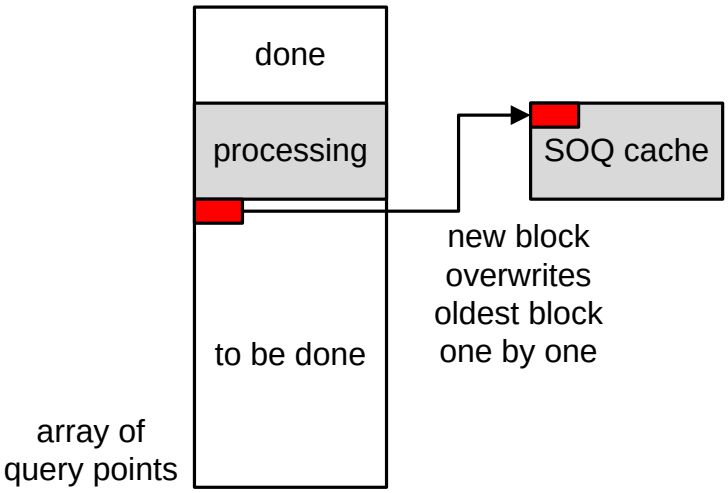


Figure 3.23: Cache operation for the SOQ cache

The array of query points is a large data structure, with up to a few hundred to thousand points per pixel. Each query point data structure is 32 bytes, so even with a hundred points per pixel with a small image at 320×320, this array is over 300 MB in size, and will only grow larger with a higher workload. With the SOQ cache dedicated only to this data array, we can make sure we only read this once into the cache, reducing the required overall memory bandwidth.

Photons are repeatedly accessed in a rather random order. It is difficult to predict which photon will be required at which point in the algorithm. And because every tree search with a new query point starts at the root node, we cannot declare any photon data to be truly 'done'; it is entirely possible that the next query point may need access to it.

Therefore it would be beneficial to have variables that are affected by the photon to have a different data path from the query point variable. The two separate data paths are represented by the SOP L1/L2 caches and SOQ cache.

3.6 The Shader Operation Accelerator (SOA)

3.6.1 Problem definition

Once we have found a photon that is close enough to affect the light at the query point from the Tree Search Accelerator, it is necessary to calculate the actual color value that the photon adds to the query point. This is the shader operation, and it is processed in the Shader Operation accelerator.

For every query point and photon pair that has been found, we need to check that the actual distance is indeed closer than the kernel radius h. This is done by calculating the following and observing the sign bit of the result:

$$\texttt{cond} \;=\; (\texttt{distance}(\texttt{queryPoint.location}, \texttt{photon.location}))^2 - h^2$$

If the distance condition is met, then the photon affects the color of the query point, and we can calculate the effect of the photon on the query point using the following equations.

$$\begin{aligned}
\texttt{color.red} &= \frac{\texttt{kernel} \times \texttt{BRDF}}{h} \times \texttt{power.red} \times \texttt{contribution.red} \\
\texttt{color.green} &= \frac{\texttt{kernel} \times \texttt{BRDF}}{h} \times \texttt{power.green} \times \texttt{contribution.green} \\
\texttt{color.blue} &= \frac{\texttt{kernel} \times \texttt{BRDF}}{h} \times \texttt{power.blue} \times \texttt{contribution.blue}
\end{aligned}$$

The process used here is a density estimation where the `kernel` value is calculated from a kernel function (e.g. a Gaussian distribution). The `BRDF` is a function that calculates how much of the light that entered a certain point in the scenery is reflected out towards a given direction. h is the value of the kernel radius, the `power.(color)` values are the separate color components for photons, and the `contribution.(color)` values are the scaling factors. Again, for more detailed discussion regarding these values, refer to [47].

- The `BRDF` and `contribution.(r/g/b)` values are in the range [0,1].

- The `power`.(`r/g/b`), the `kernel` and kernel radius h are non-zero positive values.

In addition to this, we will assume there are no special case inputs such as denormalized numbers, infinity or NaNs (Not a Number).

Once each `color` value is calculated, the main processor can directly add the three values to the corresponding query point in the frame buffer.

3.6.2 Calculation trees

Excluding the condition calculation, which is handled separately, the shader operation is basically a series of floating-point multiplications and one division as shown in the diagrams below. This is done by handling the mantissa and exponent bits separately, and combining them at the end to obtain the full result.

The multiplication and division calculation tree for the mantissa part is shown in Figure 3.24. The range of each variable is also displayed.

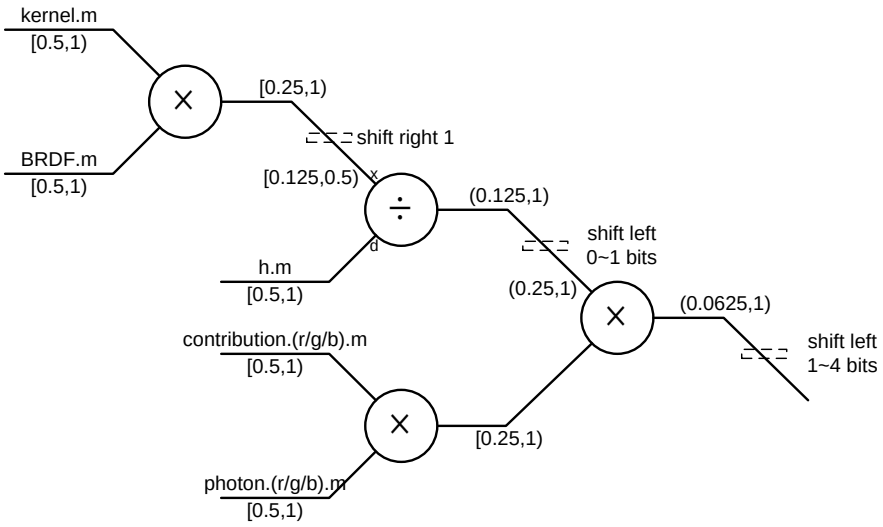


Figure 3.24: Calculation tree for mantissa

Each mantissa input is a 24-bit wide value, and this is handled in a digit-serial online fashion using the radix-2 online multiplier and divider design adapted

from [16] and adjusted to incorporate for the specific inputs we have for this calculation. Since the IEEE standard has a hidden leading 1, all mantissa bits are pushed one location to the right and a 1 is shifted in. Each exponent value is increased by 1 to accomodate for this.

$$1.x_0x_1x_2\ldots \times 2^{exp} = 0.1x_0x_1x_2\ldots \times 2^{exp+1}$$

In order to keep the output of the divider under 1, the x input to the divider is shifted one bit to the right in advance.

Each online multiplication operand needs to be quasi-normalized, as discussed in [61] and [60]. By the definitions in [60], if we have a non-zero redundant floating-point number $x = x_f r^{x_e}$, with k digits of mantissa using a maximally redundant digit set, this number is said to be

1)	normalized	if $r^{-1} \geq \lvert x_f \rvert < 1$
2)	quasi-normalized	if $r^{-2} \geq \lvert x_f \rvert < 1$
3)	pseudo-normalized	if $r^{-k} \geq \lvert x_f \rvert < 1$

As it is possible for the output of the divider to be smaller than 2^{-2} and thus not be quasi-normalized, an extra step is required here to adjust the output if necessary. The process of quasi-normalization can be handled as discussed in the appendix of [57]. The values of the first two quotient digits of the result (q_0', q_1') are examined, and if they are $(1,0)$, $(1,1)$, $(-1,0)$ or $(-1,-1)$, the quotient is already quasi-normalized. If the most significant digits of the quotient are not one of the given combinations, the digits are combined as $nq_0' = 2q_0' + q_1'$, and all remaining quotient bits are shifted so that $nq_1' \leftarrow q_{next}$. This new pair of values (nq_0', nq_1') is now the values of the first two quotient digits, and the process is repeated as necessary.

For our calculations, since no negative values are possible, and the possible quotient value range is larger than 2^{-3}, we need at most 1 quasi-normalization step at this point. Processing quasi-normalization increases the online delay of

each single shader operation by one cycle for each digit shift, but overall it has almost no effect on the net throughput of the SOA. This is because any additional cycles on a single instruction can be hidden behind all other overlapping parallel instructions, and they have no effect at all on the architecture's ability to issue new shader operations into the SOA.

At the end of the calculation, to pack the result back into the IEEE standard format, the output mantissa needs to be in the range of $[1, 2)$. The output stage examines the calculated mantissa and shifts the output to the left. All shift left bits are kept track of and sent to the exponent calculation so that the exponent value can be adjusted as required.

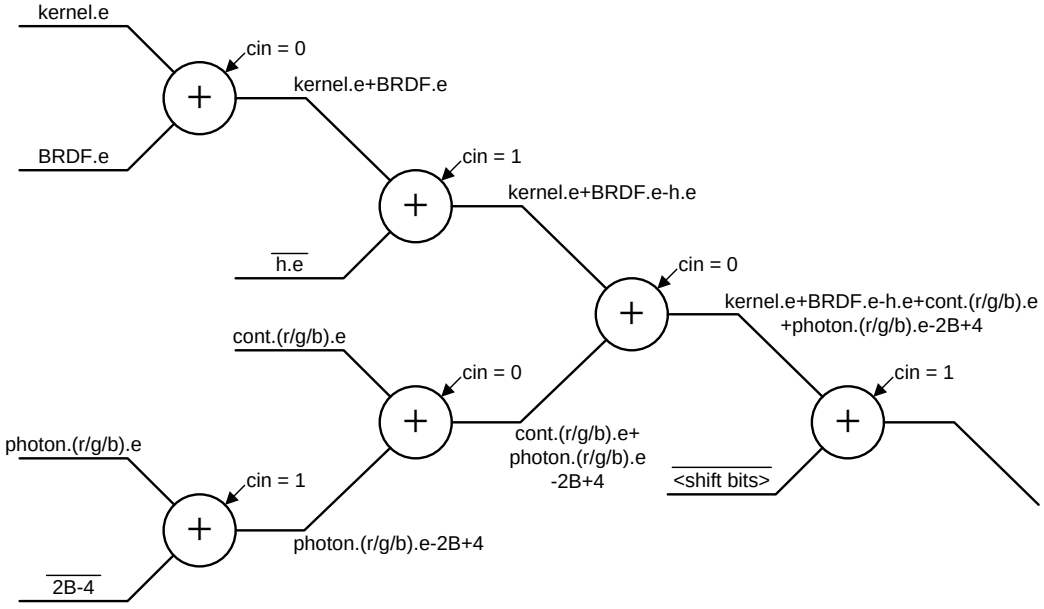


Figure 3.25: Calculation tree for exponent

The addition calculation tree for the exponent part is shown in Figure 3.25. Each exponent input is an 8-bit wide value, and this is handled in a parallel adder and delayed until the mantissa calculations are complete. Due to the IEEE floating-point standard having a biased exponent, we need to take the bias B into account when doing addition and subtraction of exponents. The exponent value

B is:

$$B = 2^{n-1} - 1$$

where n is the number of bits for the exponent. For the current setup, we have $n = 8$.

The actual exponent value $exp_{(\texttt{var})}$ can be calculated using:

$$\begin{aligned} exp_{(\texttt{var})} &= (\texttt{var}).e - B \\ (\texttt{var}).e &= exp_{(\texttt{var})} + B \end{aligned}$$

The result of the adder tree in Figure 3.25 is equal to the exponent value $\texttt{color}.(\mathrm{r/g/b}).e$. Some carry-in values are set to 1 in order to accomodate for the push-to-the-rights done in the mantissa calculation. Also, an additional 1 is added to accomodate for the shift right in the mantissa calculation right before the division.

Taking into account that all mantissa inputs to the multiplier and divider have been shifted one bit to the right, the unbiased exponent value for $\texttt{color}.(\mathrm{r/g/b}).e$ (excluding the final shift adjustment) is equal to:

$$\begin{aligned} & \texttt{color}.(\mathrm{r/g/b}).e \\ = \; & ((exp_{\texttt{kernel}} + 1) + (exp_{\texttt{BRDF}} + 1) + 1) - (exp_h + 1) + (exp_{\texttt{power}.(\mathrm{r/g/b})} + 1) \\ & + (exp_{\texttt{cont}.(\mathrm{r/g/b})} + 1) + B \\ = \; & exp_{\texttt{kernel}} + exp_{\texttt{BRDF}} - exp_h + exp_{\texttt{power}.(\mathrm{r/g/b})} + exp_{\texttt{cont}.(\mathrm{r/g/b})} + B + 4 \\ = \; & (exp_{\texttt{kernel}} + B) + (exp_{\texttt{BRDF}} + B) - (exp_h + B) + (exp_{\texttt{power}.(\mathrm{r/g/b})} + B) \\ & + (exp_{\texttt{cont}.(\mathrm{r/g/b})} + B) - 2B + 4 \\ = \; & \texttt{kernel}.e + \texttt{BRDF}.e - h.e + \texttt{power}.(\mathrm{r/g/b}).e + \texttt{cont}.(\mathrm{r/g/b}).e - (2B - 4) \end{aligned}$$

Since the value of B is determined beforehand, $2B - 4$ is a constant and does not need a separate adder to calculate.

The shift left bits at the end of the divider and last multiplier in the mantissa tree are collected and subtracted from the exponent at the end. The shift amount will be between 1 and 5.

3.6.3 Overall structure

3.6.4 Submodule : P2S (Parallel to Serial)

The P2S module here is the same as the P2S module in the Tree Search accelerator in Section 3.4.5.

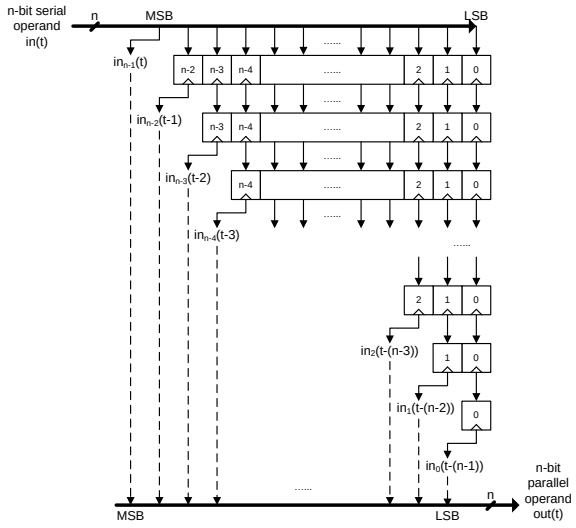


Figure 3.26: Components of Parallel to Serial

3.6.5 Submodule : ADD8 (Parallel Adder)

The exponent is only 8-bits wide and is calculated using a series of parallel adders as seen in the adder tree in Figure 3.25.

3.6.6 Submodule : MUL24 (Pipelined Online Multiplier)

The following outlines the online multiplication algorithm. It is based on the online multiplication in [16], with some changes to fit the characteristics of the application.

---

1. [*Initialize*]

$x[-3] = y[-3] = w[-3] = q[0] = 0$

for $j = -3, -2, -1$

$x[j+1] \leftarrow CA(x[j], x_{j+4});\ y[j+1] \leftarrow CA(y[j], y_{j+4})$

$v[j] = 2w[j] + (x[j]y_{j+4} + y[j+1]x_{j+4})2^{-3}$

$w[j+1] \leftarrow v[j]$

end for

2. [*Recurrence*]

for $j = 0, \ldots, n-1$

$x[j+1] \leftarrow CA(x[j], x_{j+4});\ y[j+1] \leftarrow CA(y[j], y_{j+4})$

$v[j] = 2w[j] + (x[j]y_{j+4} + y[j+1]x_{j+4})2^{-3}$

$p_{j+1} = SELM(\widehat{v}[j])$

$w[j+1] \leftarrow v[j] - p_{j+1}$

$P_{out} \leftarrow p_{j+1}$

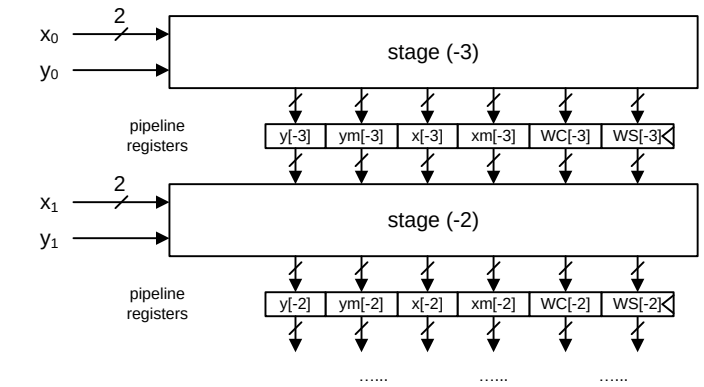
end for

A few notes on the algorithm:

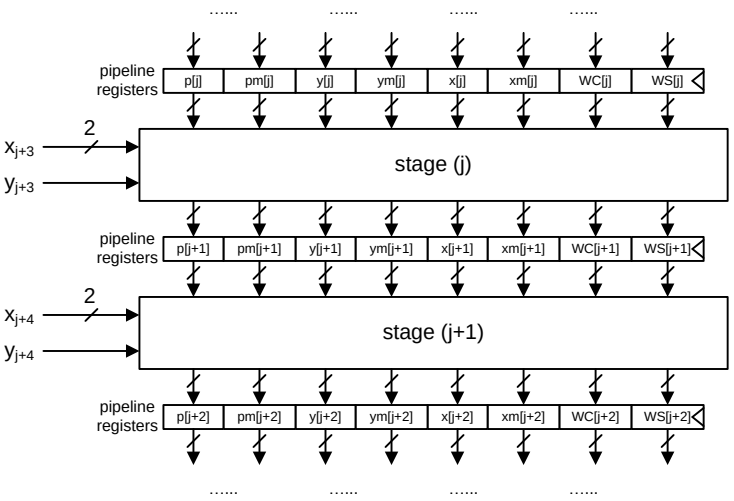
- The residual w is kept in redundant form, where WS is the pseudosum and WC is the stored carry. It is simply shown as $w[j]$ in the above description.
- n is the bit width.

- The online delay $\delta = 3$ for this setup; the estimate $\widehat{v}[j]$ is 2 bits wide.
- CA is either the concatenation of bits (as for direct inputs both x and y will not be in redundant form for our calculation) or on-the-fly conversion, as explained in [16].

The diagrams in Figure 3.27 show overviews of the pipeline logic and register construct of the online multiplier. Each logic stage is entirely combinational, with any required registers all placed between the stages. At each clock, each stage takes inputs from the previous registers and digits from the parallel-to-serial registers, and calculates the register values for the next stage. Each stage is given the x and y digits required at each stage, which enters the Shader Operation module in parallel form and passes through the parallel-to-serial converter shown in Figure 3.26.



(a) Initial stages



(b) Recurrence stages

Figure 3.27: Online multiplier pipeline stages overview

The following two figures show the combinational logic of the stages in more detail. Figure 3.28 is the circuit for the initialize stages and 3.29 is the circuit for the recurrence stages.

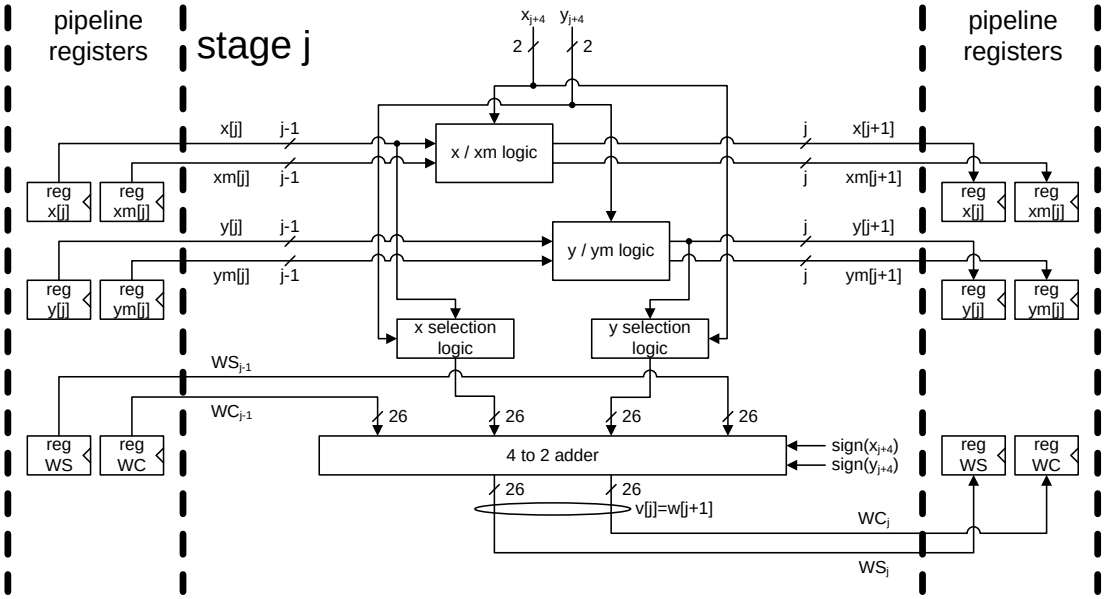


Figure 3.28: Initialize stage details of a pipelined online multiplier (adapted from [16])

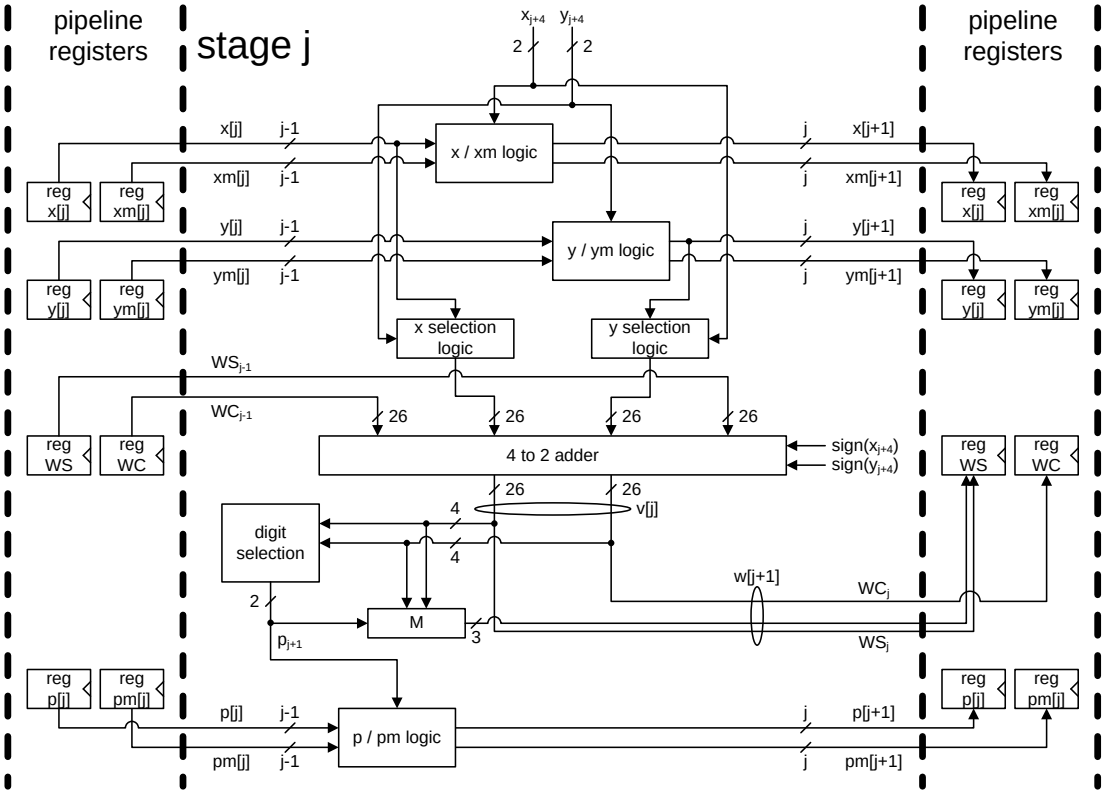


Figure 3.29: Recurrence stage details of a pipelined online multiplier (adapted from [16])

The [4:2] adder computes $v[j]$, the next p_{j+1} is chosen from the top bits of $v[j]$, and the next residual value $w[j+1]$ is updated by concatenating the top bits from the output of the M module. The bit alignments of the adder can be seen in Figure 3.30.

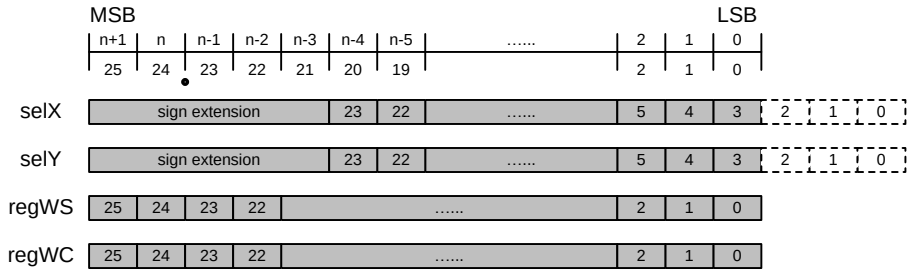


Figure 3.30: [4:2] adder setup for a 24-bit online multiplier (adapted from [16])

Table 3.9 shows the multiplier digit selection table.

$\widehat{v}[j]$	$v_{-1}v_0.v_1$	p_{j+1}
$\frac{3}{2}$	01.1	1
1	01.0	1
$\frac{1}{2}$	00.1	1
0	00.0	0
-$\frac{1}{2}$	11.1	0
-1	11.0	-1
-$\frac{3}{2}$	10.1	-1
-2	10.0	-1

Table 3.9: Digit selection table for online multiplier (adapted from [16])

3.6.7 Submodule : DIV24 (Pipelined Online Divider)

The division operation is shown here.

$$x = q \cdot d + rem$$

where

$$|rem| < |d| \cdot ulp, sign(rem) = sign(x)$$

In the above expression, the dividend x and the divisor d are the operands. The quotient q is the results of the operation. rem needs to be smaller than the divisor times ulp (unit in the last position).

The specification of the pipelined online divider with bit width n is as follows:

$$
\begin{array}{rrcl}
\text{inputs:} & \text{dividend } \underline{x} & = & (x_{n-1}, x_{n-2}, \cdots, x_1, x_0),\ x_i \in \{0,1\} \\
& \text{divisor } \underline{d} & = & (d_{n-1}, d_{n-2}, \cdots, d_1, d_0),\ d_i \in \{0,1\} \\
& & & a_7 \text{ downto } a_0 \text{ are the original input,} \\
& & & a_{14} \text{ downto } a_8 \text{ are bits that get shifted in} \\
& \underline{s} & = & (s_2, s_1, s_0),\ s_i \in \{0,1\} \\
\text{outputs:} & \text{quotient } \underline{q} & = & (q_7, q_6, \cdots, q_1, q_0),\ q_i \in \{0,1\} \\
\text{function:} & y_i & = & \begin{cases} a_{i+s} & \text{if } E = 1 \\ 0 & \text{otherwise} \end{cases} \\
& & & \text{where } s = \sum_{j=0}^{2} s_j 2^j \text{ and } i = 0, 1, \cdots, 7
\end{array}
$$

The following outlines the online division algorithm. It is based on the online division in [16], with some changes to fit the characteristics of the application.

---

1. [*Initialize*]

$x[-4] = d[-4] = w[-4] = q[0] = 0$

for $j = -4, \ldots, -1$

$d[j+1] \leftarrow concat(d[j], d_{j+5})$

$v[j] = 2w[j] + x_{j+5}2^{-4}$

$w[j+1] \leftarrow v[j]$

end for

2. [*Recurrence*]

for $j = 0, \ldots, n-1$

$d[j+1] \leftarrow concat(d[j], d_{j+5})$

$v[j] = 2w[j] + x_{j+5}2^{-4} - q[j]d_{j+5}2^{-4}$

$q_{j+1} = SELD(\widehat{v}[j])$

$w[j+1] \leftarrow v[j] - q_{j+1}d[j+1]$

$q[j+1] \leftarrow CA(q[j], q_{j+1})$

$Q_{out} \leftarrow q_{j+1}$

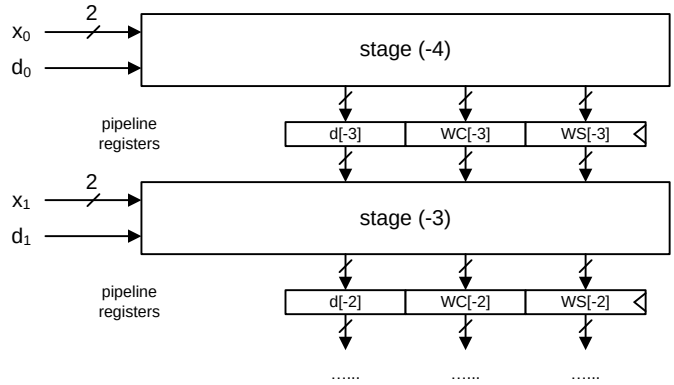
end for

A few notes on the algorithm:

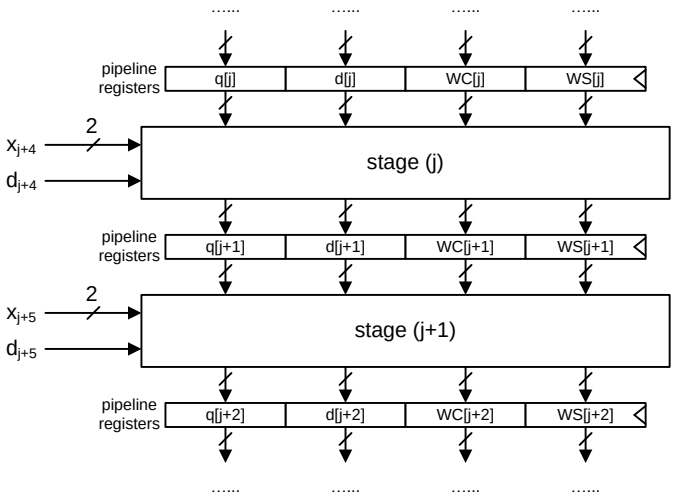
- The residual w is kept in redundant form, where WS is the pseudosum and WC is the stored carry. It is simply shown as $w[j]$ in the above description.
- n is the bit width.
- The online delay $\delta = 4$ for this setup; the estimate $\widehat{v}[j]$ is 3 bits wide.
- CA is on-the-fly conversion, as explained in [16].
- *concat* is simply the concatenation of bits, as the divisor d will not be in redundant form for our calculation.

The diagrams in Figure 3.31 show overviews of the pipeline logic and register construct of the online divider. Each logic stage is entirely combinational, with any required registers all placed between the stages. At each clock, each stage takes inputs from the previous registers and digits from the parallel-to-serial registers, and calculates the register values for the next stage. Each stage is given the x and d digits required at each stage, which enters the Shader Operation module in

parallel form and passes through the parallel-to-serial converter shown in Figure 3.18.



(a) Initial stages



(b) Recurrence stages

Figure 3.31: Online divider pipeline stages overview

The following two figures show the combinational logic of the stages in more detail. Figure 3.32 is the circuit for the initialize stages and 3.33 is the circuit for the recurrence stages.

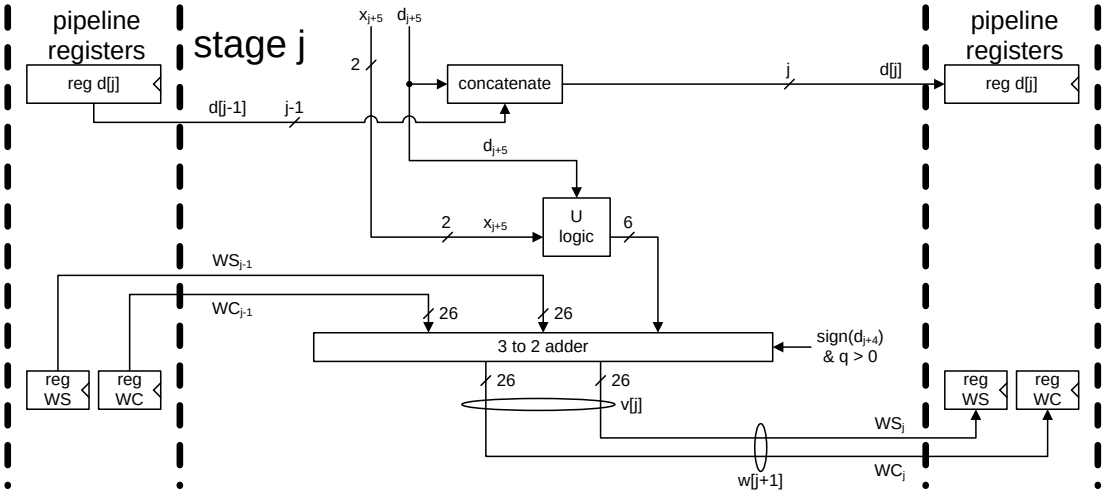


Figure 3.32: Initialize stage details of a pipelined online divider (adapted from [16])

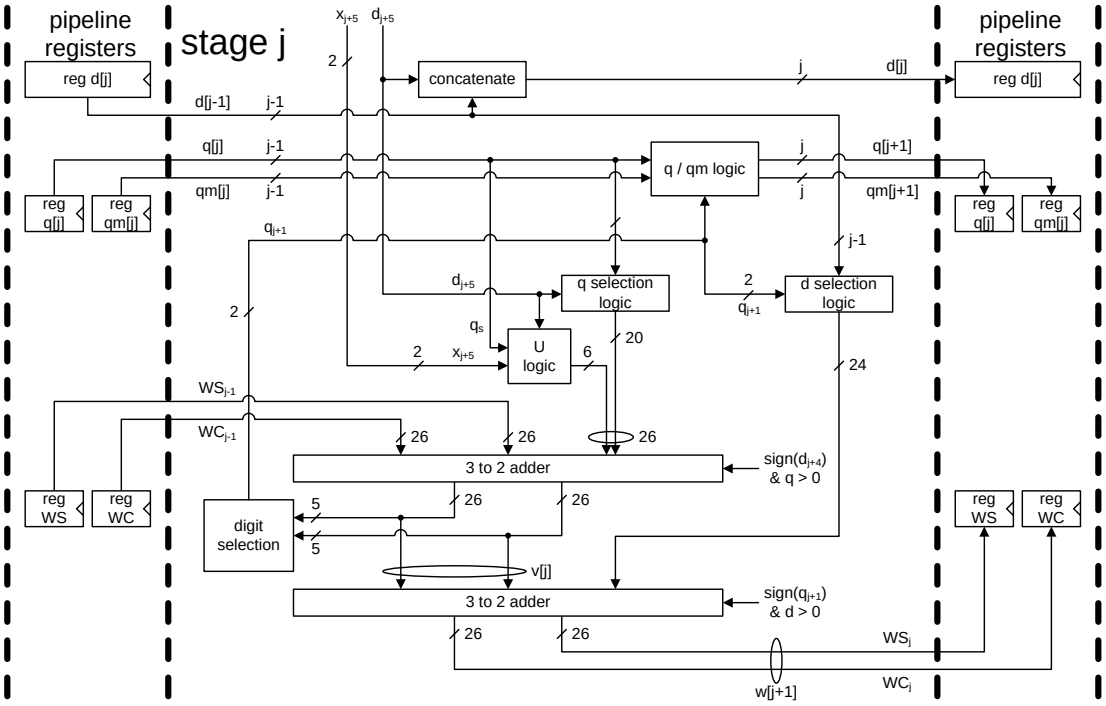
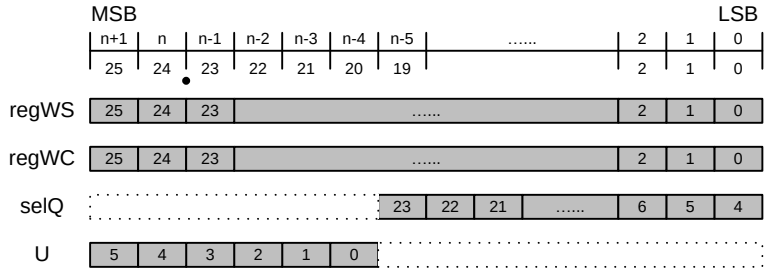


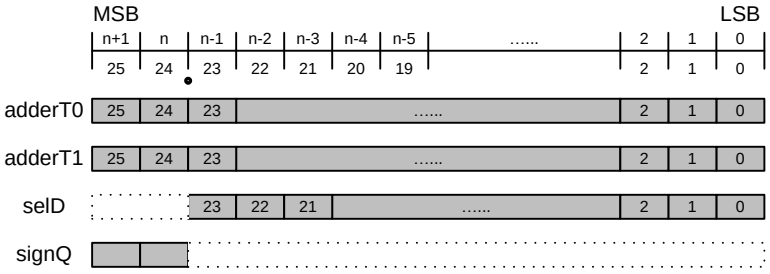
Figure 3.33: Recurrence stage details of a pipelined online divider (adapted from [16])

The top [3:2] adder computes $v[j]$, the next q_{j+1} is chosen from the top bits of $v[j]$, and the next residual value $w[j+1]$ is updated at the bottom [3:2] adder.

The bit alignments of the two adders can be seen in Figures 3.34a and 3.34b.



(a) Top [3:2] adder setup



(b) Bottom [3:2] adder setup

Figure 3.34: [3:2] adder setup for a 24-bit online divider (adapted from [16])

The block U combines the dividend digit $x_{j+5}2^{-4}$ and the sign extension bits of $q[j]d_{j+5}2^{-4}$ in advance, so that the calculation of $v[j]$ can be achieved using a 3 to 2 adder. The full table for an online divider with both x and d in redundant form can be found in [16].

For our implementation, d is in non-redundant form and will never have a value of -1. The simplified implementation table is shown in Table 3.10.

x_{j+5}	$x_p x_n$	d_{j+5}	q_s	-2	-1	0	1	2	3
		1	0	0	0	0	0	0	0
1	10	1	1	0	0	0	0	0	1
		0	–	0	0	0	0	0	1
		1	0	1	1	1	1	1	1
0	00	1	1	0	0	0	0	0	0
		0	–	0	0	0	0	0	0
		1	0	1	1	1	1	1	0
-1	01	1	1	1	1	1	1	1	1
		0	–	1	1	1	1	1	1

Table 3.10: U module output for redundant x and non-redundant d (adapted from [16])

3.6.8 Critial path delay

As with any sequential system, all data paths starting from a register output and ending at a register input needs to be complete before the next clock edge. The clock period needs to be longer than the delay of the critical case path.

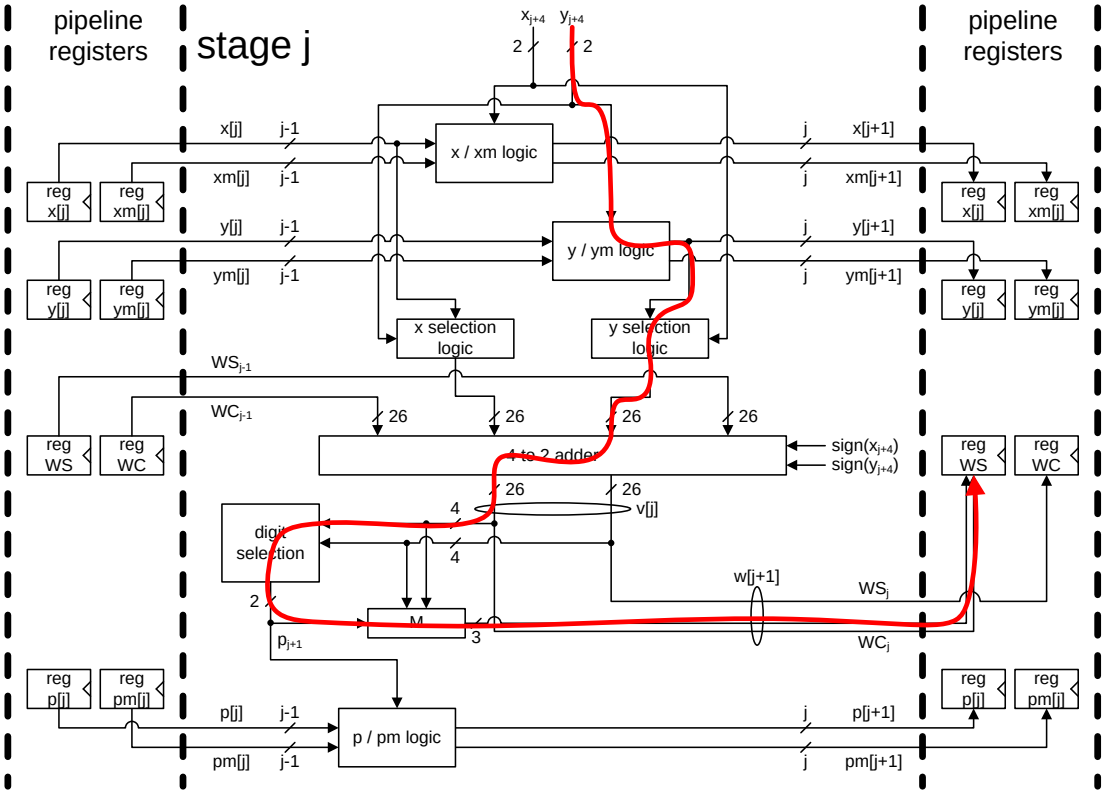


Figure 3.35: Critical path for the online multiplier

Figure 3.35 shows the critical case path for the pipelined online multiplier. The elements on the path in order are: y/ym logic, y selection logic, 4 to 2 adder, digit selection, and M.

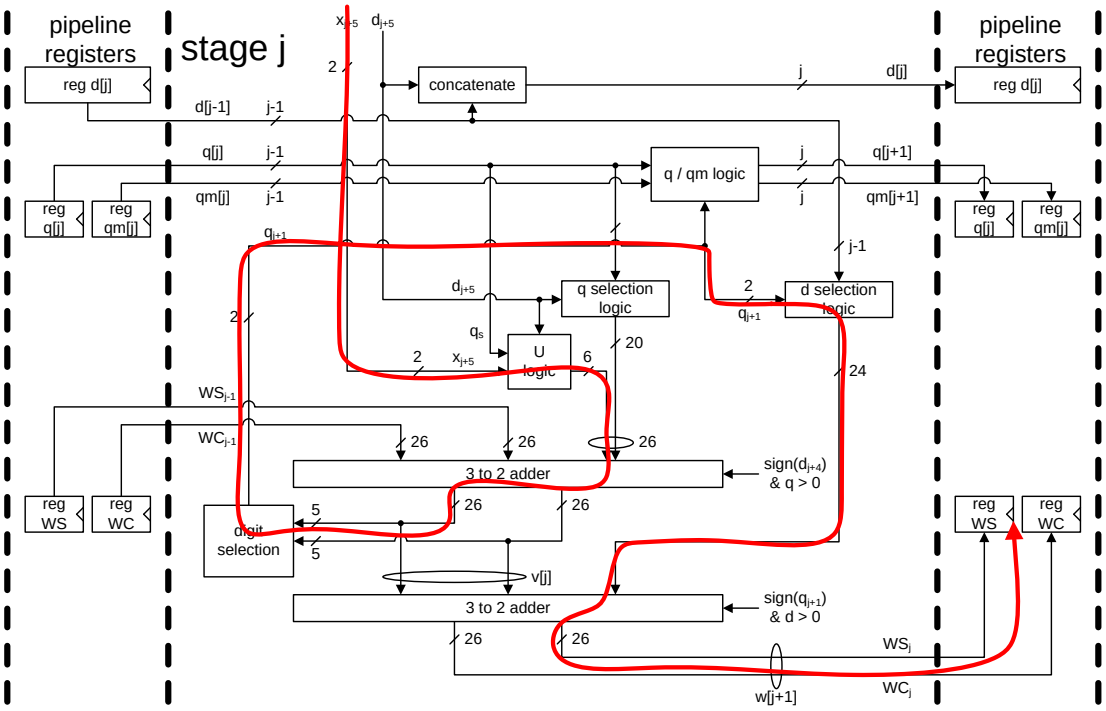


Figure 3.36: Critical path for the online divider

Figure 3.36 shows the critical case path for the pipelined online divider. The elements on the path in order are: U logic, 3 to 2 adder, digit selection, d selection logic, and 3 to 2 adder.

3.7 Cycle stalls

There are two parts in the MAPM that may cause stalls.

1. The first is at Nextnode, which is inside the TS block. Before assigning the results of the TSA to the Register queue (for tree search operations) or the Data bus (for shader operations), it needs to have all necessary node information from the memory. Since the data resides inside the TSA for the number of cycles equal to the pipeline depth of the TSM d_{TSM}, there is some cycle slack for information to arrive, especially if it is already inside

the TS L2 cache. However, on the chance that the information is not here yet, Nextnode will cause a stall for the TS block until the information is fetched.

2. The second is at the Reservation stations, which is inside the SO block. To launch shader operations into the TSAs, we require a number of variables to be ready, as we examined previously in Table 3.8. This causes entries in the Reservation station to be occupied with shader operations that are waiting for required values to arrive from the memory. As the number of slots are limited in the Reservation stations and shader operations tend to happen in bursts, it is possible that new shader operations do not have an empty slot available. When this happens, the Reservation station will cause a stall for the TS block until a shader operation is launched and an empty slot is created.

One thing to note is that any stalls only stop the operation of the TS and not the SO block. Once an instruction is ready in the SO block, nothing else is needed for the calculation to finish, and thus no stalls are necessary.

CHAPTER 4

Evaluation

The evaluation of the architecture is divided into two parts.

First part is the ASIC synthesis and hardware cost. Here, we perform ASIC synthesis on VHDL implementations of the MAPM and conventional parallel circuits. This gives us hardware cost values, including clock speed, area cost, and power dissipation. Using these, we can compare hardware cost of the online implementation used in MAPM to conventional parallel implementations and evaluate the efficiency of online circuits.

The second part is performance evaluation and scalability. By running cycle-accurate simulation of the MAPM for different configurations, we can evaluate the scalability of the architecture and observe perfomance gain in terms of the number of replicated modules. In addition to this, combining the clock speed results with the number of cycles gives us realistic throughput numbers which we can use to compare with other recent photon mapping implementations. Also, the hardware cost numbers will allow us to evaluate the implementation costs of the MAPM, and compare to other chips in the market today.

4.1 Tools used in the evaluation and design

4.1.1 Software environment

In this section, we examine the setup of the evaluation code that was used.

The PM code is adapted from work by collaborator Shawn Singh [47], with minor changes made by the author. The main PM algorithm is coded in C++, using Microsoft Visual Studio 2008 as the main IDE.

All hardware design files are coded in VHDL. Functional verification was done by creating test bench files with the help of Python [6], and running them through ModelSim Altera starter edition. All hardware costs and timing numbers are extracted using Synopsys Design Compiler and the Synopsys 90nm Generic Library [7].

4.1.2 Cycle accurate simulation

The Intel Pin [5, 34, 44] is a binary instrumentation tool that allows the user to insert various useful C or C++ code into arbitrary places in an executable. The insertion and execution of code is processed dynamically while the target program is running. Pin also allows injected code to access context information such as register values and function parameters, while restoring any changes to the original executable's memory space to prevent unintentional errors.

The cycle-accurate simulator is built using the Pin tool libraries and runs on top of the software PM code written by Singh [47]. At the beginning of the execution, the simulator initializes all variable arrays and `structs` in the simulator. These represent registers and memory inside the MAPM, and is used to track the flow of information during execution.

While the PM code is running, the simulator attaches a call inside the Tree Search function, and every time it is executed in the PM code, the simulator calls the function `AdvanceOneClock()` This simulates a Tree Search instruction issue in the MAPM architecture. Once all Tree Searches are issued, the simulator calls the function `AdvanceToEndOfOperation()`. This repeatedly calls `AdvanceOneClock()` without launching a new Tree Search instruction, until all

registers are clear and all calculations are finished. By doing this, the simulator can calculate the exact number of clock cycles that is required for the given workload in the PM code.

The cycle-accurate simulator also takes cache misses into account by utilizing Pin's native cache simulation classes. Any accesses to main memory from the MAPM goes through Pin's cache simulation interface, and is checked for a hit or miss. Data fetched from the memory will wait a predetermined number of cycles before entering the simulator and progressing through the registers of the MAPM, thus imitating the cache delay required for the data to arrive.

4.1.3 Cache sizes

Each render process deals with a single frame, which has a fixed camera and lights, and objects do not move around inside the 3D scene. Therefore, the polygon map (which is derived from the object locations) and the photon map (which is derived from the polygon map and light sources) are also fixed during rendering of a single frame. It follows that once these completed data structures are loaded into the relevant caches, no further accesses to the main memory are necessary, provided that the cache size is large enough to hold the entire structure. This is important for reducing bandwidth requirements and is an important part to consider for the architecture design.

There are two phases to the photon mapping process, the TS phase and the SO phase, where the block dedicated to each has internal caches that store data structures required for each phase.

1. **The Tree Search block** The data structure required in this block is the photon KD-tree, and it is stored in the TS L1/L2 caches. The struct `KDNode`, which holds data of one node in the photon KD-tree, stores two 4 byte unions where data is encoded into different bits, for a total of 8 bytes. The node

count of the tree is equal to the total photon count. Therefore the required space for the TS L2 cache $size_{TS}$ is:

$$\begin{aligned} size_{TS} &= size(\texttt{KDNode}) \times n_{photons} \\ &= 8 \cdot n_{photons} \text{ (bytes)} \end{aligned}$$

Having around 20,000 photons will require a little less than 160 KBs of TS L2 cache.

2. **The Shader Operation block** The cache setup of the Shader Operation block is a bit more complex to figure out, as there is a much larger number of parameters to consider, compared to the Tree Search block.

   Recall the table of values that need to be fetched for a Shader Operation in Table 4.1.

Name	Data type	Defined by
`kernel`	32-bit floating point	photon and query point
`BRDF`	32-bit floating point	photon and query point
h	32-bit floating point	none
`power.color`	3×32-bit floating point	photon
`contribution.color`	3×32-bit floating point	query point

Table 4.1: Required inputs for a single SO

   As was explained in Section 3.5.2, the parameters can be divided by whether the data changes in relation to the photon or to the query point. Each group has a different access pattern, leading to having two separate caches with its own method of fetching and preparing data.

   (a) **Query point parameters** The parameter that is wholly determined by the query point is the `contribution.color` values. This information is created and saved in an array of struct `CameraPoint`.

As there is one for each query point, and query points are in the hundreds of thousands to a few millions, this whole data structure goes up to the MB range and is a rather large data structure to store wholly inside a fast cache. However, after all related shader operations for a query point leave the reservation stations, there is no further need to keep the entry and we can safely overwrite earlier entries.

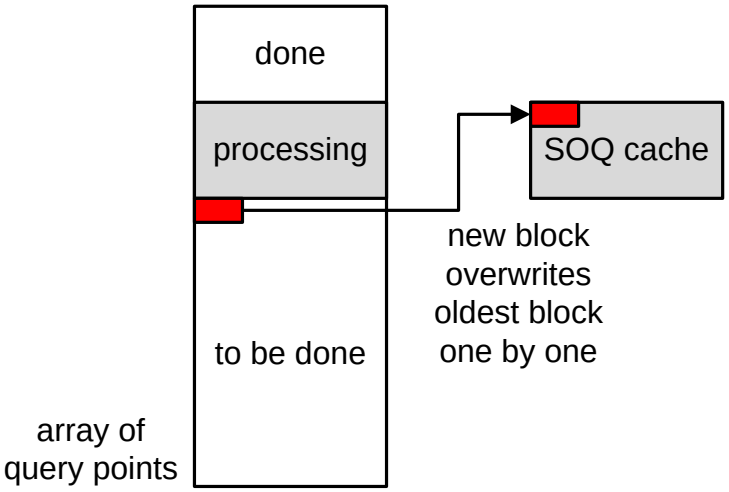


Figure 4.1: Operation of the SOQ cache

The required space for the query point parameter cache $size_{qp}$ is:

$$size_{qp} = size(\text{entry}) \times d_{TSM} \times n_{tsm}$$

where d_{TSM} is the pipeline depth of the TS module and n_{tsm} is the number of TS modules. We have $d_{TSM} = 1(\text{Nextnode}) + 25(\text{TSA})$ so the equation becomes:

$$\begin{aligned} size_{qp} &= size(\text{entry}) \times d_{TSM} \times n_{tsm} \\ &= 32 \cdot 26 \cdot n_{tsm} \\ &= 832 \cdot n_{tsm} \text{ (bytes)} \end{aligned}$$

The required size will be linear to how many TS modules are present,

but as each module will require less than 1 KB each, this will be a manageable size.

(b) **Photon parameters** The parameter that is wholly determined by the photon is the photon's power value. The data structure that holds the power value `power.color`, is stored in an array of struct `PointSample`. Each `PointSample` is 16 bytes, and one exists for each photon. The required space for these two parameters $size_{pp}$ is:

$$\begin{aligned} size_{pp} &= size(\texttt{PointSample}) \times n_{photons} \\ &= 16 \cdot n_{photons} \text{ (bytes)} \end{aligned}$$

This would be roughly twice the size requirement of the TS cache.

(c) **Remaining parameters** The `kernel` and `BRDF` values can be affected by both the query point and photon, depending on which calculation method is chosen for each variable. And because the values depend on both variables, the maximum total count of these parameters become large very quickly, making it infeasible to have the whole data structure stored in the cache at once. Thus, for these variables, we assume they are read through the photon parameter cache, and allocate some additional space to $size_{pp}$ in order to accomodate for the transition of these parameters.

With the above derivations in mind, for our simulation runs, we use cache sizes as shown in Table 4.2.

Name	Size (bytes)		Associativity
	Total	Line	
TS L1	16 K	32	2-way associative
TS L2	256 K	64	4096 (fully associative)
SOQ	16 K	32	1 (direct mapped)
SOP L1	64 K	32	2-way associative
SOP L2	512 K	64	8192 (fully associative)

Table 4.2: Cache setup for simulations

This setup will be able to fully cover up to $256 \times 1024 \div 8 = 32{,}768$ photons in the TS L2 cache, and up to $16 \times 1024 \div 32 = 32$ query points for 16 TSMs.

Due to the Pin simulation code slowing down the actual runtime of the PM code, we had to downsize the simulation input size in order to be able to finish in a reasonable amount of time. The above setup is large enough to cover the simulation sizes that we wish to use.

4.1.4 Benchmark images

These are some benchmark images that are used for the simulations.

The workload that each benchmark represents can be adjusted with different numbers for screen resolution, FGRs per pixel, and total number of photons. Basically, lower number for each variable will result in a smaller workload for the simulation. Lower screen resolution and FGRs will result in a smaller number of query points created, and a lower photon count will result in a smaller photon KD-tree and less search operations overall. And naturally, the quality of the final image will be affected by these numbers too. For instance, a low screen resolution will make the image look blocky, and a low number of FGRs and photons may

result in artifacts or dark patches that should not be there. The exact number required for each variable is not set in stone and may vary according to the scene.

The benchmark in Figure 4.2, as we briefly examined in Section 3.1.2, is the Cornell box, a well known 3D test model that is widely used for testing out various image rendering techniques. The version used here consists of two cuboid blocks placed inside five outer walls as can be seen in the image, and consists of 30 triangles total. This image has been rendered at a screen resolution of 450×450, with 100 FGRs created per primary ray, and 2 million photons scattered in the scene, which is enough to give the image a smooth look without any unintentional dark patches overall.

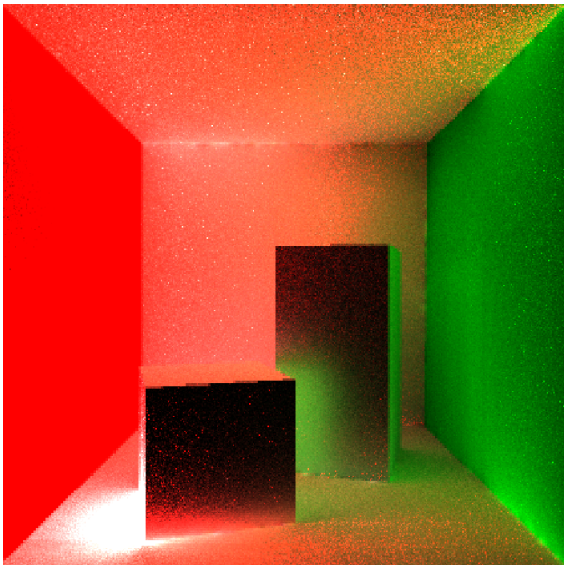


Figure 4.2: Benchmark 1 - Cornell box (adapted from [19])

The benchmark in Figure 4.3 is a view of the corridor inside the Sponza Atrium scene (model by Marko Dabrovic). The scene image is much more complex than the Cornell box benchmark, with the total triangle count being 66,454. This image shown here holds a screen resolution of 320×320, with 100 FGRs created per primary ray, and 2 million photons scattered in the scene.

Figure 4.3: Benchmark 2 - Sponza corridor (model by Marko Davrovic)

The last benchmark image shown in Figure 4.4 is a view from a different location inside the Sponza Atrium scene. It has been rendered with a screen resolution of 320×320, with 100 FGRs created per primary ray, and 2 million photons scattered in the scene. Even though the scenery file is the same as the Sponza corridor benchmark, the different camera location and light source placement creates a whole different set of FGRs and photons.

Figure 4.4: Benchmark 3 - Sponza atrium (model by Marko Davrovic)

Note that the workload for simulation runs has to be adjusted because the simulation code significantly slows down the rendering. Therefore, the images produced will not be at the quality of the images shown here.

4.2 ASIC synthesis and hardware cost

In this section, we examine the hardware cost of the MAPM.

The VHDL design implementation was compiled and analyzed using the Synopsys Design Compiler tool and the Synopsys 90nm Generic component library. Each compile iteration is given a target clock speed, and the tool tries to synthesize a layout that satisfies the timing constraint. This eventually comes to a wall where it is not possible to satisfy the timing constraints for certain high target speeds. Once the timing is achieved, the tool then tries to optimize for better area and power, with an option to emphasize minimizing one over the other. For our designs, more emphasis was given to minimize power, but without completely ignoring area cost.

The synthesis results for a single SOA and TSA block are shown here in Tables 4.3 and 4.4.

		Power (mW)		
Clock speed (GHz)	Slack met?	Total dynamic	Cell leakage	Area (μm^2)
1.00	Yes	21.99	3.26	633,999.06
1.25	Yes	24.14	3.06	599,024.34
1.50	Yes	28.23	3.02	592,129.85
1.75	Yes	34.32	3.05	594,491.91
2.00	No	–	–	–

Table 4.3: Synthesis results for a single SOA

		Power (W)		
Clock speed (GHz)	Slack met?	Total dynamic	Cell leakage	Area (μm^2)
1.00	Yes	2.69m	217.08μ	41,648.95
1.25	Yes	3.37m	222.23μ	41,955.84
1.50	Yes	4.58m	230.93μ	43,007.39
1.75	Yes	6.50m	257.15μ	45,244.11
2.00	No	–	–	–

Table 4.4: Synthesis results for a single TSA

We can see that both modules are synthesizable up to 1.75 GHz, with a general increase in dynamic power for faster circuits. This trend is more visible in the smaller TSA. Also, in the TSA higher clock speed induces a larger area, whereas for the SOA, the clock speed does not have much effect on the overall area of the module. Synthesizing for a clock speed of 2.0 GHz is not possible with the current 90 nm component library.

It is rather difficult to obtain detailed cost values of current CPU architectures, and even if we had the numbers on hand, it would not be a fair comparison as the component libraries are completely different. Therefore, for comparison purposes, we use FloPoCo [2, 10] to generate a circuit which has the same functionality as the MAPM architecture. FloPoCo is an HDL generator specifically designed to create floating-point arithmetic circuits, with optimized data paths and pipelining to reduce worst case paths. In general, FloPoCo circuits achieve performance which is close to that of vendor-supplied operators [10].

The Figures 4.5 and 4.6 show the diagrams of the functionally equivalent circuits implemented with FloPoCo.

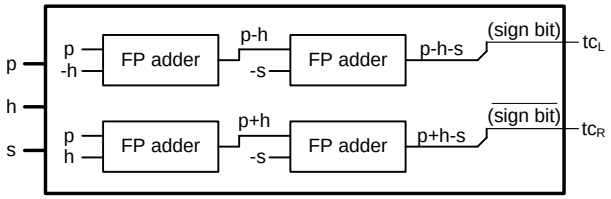


Figure 4.5: FloPoCo implementation of the TSA

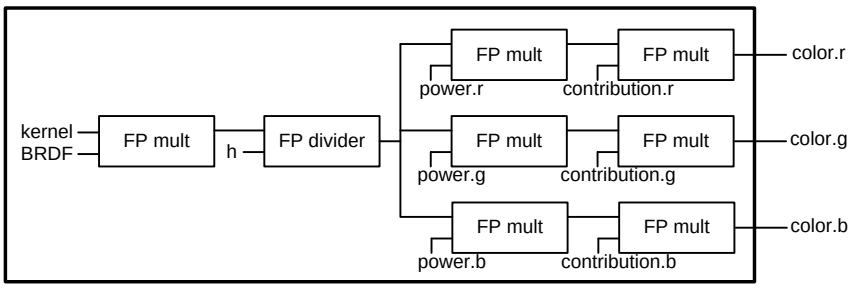


Figure 4.6: FloPoCo implementation of the SOA

From the diagrams, we obtain the following requirements for the FloPoCo setup:

Phase	Operand	Bit width	Count
Tree Search	Floating-point adder	32	4
Shader Operation	Floating-point multiplier	32	7
Shader Operation	Floating-point divider	32	1

Table 4.5: FloPoCo components for Tree Search and Shader Operation

The comparison numbers for our design and FloPoCo are shown in Table 4.6 for the largest module, which is the SOA. Note that the starting clock speed is much lower than the previous tables, as the FloPoCo implementation can only be synthesized at slower clock speeds. In the table, Pwr stands for power, Dyn stands for dynamic, and Lkg stands for leakage.

	MAPM				FloPoCo			
Clock		Pwr (mW)		Area		Pwr (mW)		Area
(GHz)	Slack?	Dyn	Lkg	(μm^2)	Slack?	Dyn	Lkg	(μm^2)
0.50	Yes	11.12	3.27	636,650.50	Yes	106.47	2.32	353,883.34
0.75	Yes	14.67	3.05	598,584.73	No	–	–	–
1.00	Yes	21.99	3.26	633,999.06	No	–	–	–

Table 4.6: Comparison of synthesis results

The first thing that comes to our attention here would be the vastly different clock speed that each implementation can be synthesized for. While the MAPM can be synthesized up to 1.75 GHz, the FloPoCo version is already unsynthesizable at 750 MHz, even though FloPoCo implementations deploy pipeline registers in critical paths to reduce the longest path delay. Area-wise, FloPoCo is a better solution. A lot of the area in the MAPM SOA is due to a large number of

registers required for the serial-to-parallel data conversion and pipelining, which is not required in parallel implementations. The power consumption also shows about a magnitude of difference. As we examined in sections 3.6.6 and 3.6.7, each stage of the online arithmetic architecture is very regular and structured, leading to circuits that can be synthesized to be less power hungry, albeit with some increase in area.

A comparison table with numbers normalized to FloPoCo is shown in Table 4.7.

	MAPM	FloPoCo
Synthesizable clock speed	3.5	1
Dynamic power consumption	0.104	1
Area cost	1.799	1

Table 4.7: Comparison with normalized numbers

Note that even though the two implementations may be functionally equivalent, the MAPM implementation is superior to the FloPoCo version in terms of parallel execution due to its digit-serial nature. This is even before considering the maximum possible clock speed and lower power consumption, an important point when placing multiple instances for increased parallelism. For the FloPoCo version to have similar parallelism to the MAPM, we would need multiple copies of the same implementation, which would likely be too power hungry to be a feasible solution.

4.3 Performance evaluation and scalability

With the insight that the analysis has given us so far, we now look at some simulation performance numbers at various points in the design space, which will also

allow us to examine the scalability of the design. The cycle-accurate simulation, combined with the hardware cost information from the previous section, also gives us concrete numbers on runtime and this in turn allows us to calculate throughput numbers that we can use for comparison with other work.

4.3.1 Target performance numbers

Before looking at the results, let us try to establish some target numbers. We shall be using shader operations per second for measuring the throughput performance.

We take a look back at the numbers we brought up in Section 1.1. We established that for real-time rendering in Full HD, we would need to calculate around 10 billion rays per frame, at around 30 frames per second. This would be equal to 300 billion rays per second. For each ray, depending on the image, an average at least 100 shader operations are required, bringing the total to 30 trillion shader operations per second.

In a recent work on MPSoC implementation of photon mapping [17], a shader operation count is not available, therefore we shall assume the 30 trillion shader operations for Full HD real time rendering holds true here as well and extrapolate numbers to obtain a rough estimate. The best performance from this work is from the combination of the G-KD method along with the PM-DS method with 8×8 mesh size. The total cycles to render a 320×240 frame is 639,238,838, and as this is on a system at 200 MHz, this would equal $639,238,838 \div (200 \times 10^6) = 3.196$ seconds. Assuming we can keep up the same rate, rendering a Full HD frame would require $(1,920 \times 1,080) \div (320 \times 240) = 27$ times many pixels, so $3.196 \times 27 = 86.292$ seconds per frame. For 30 frames, this setup requires $86.292 \times 30 = 2588.76$ seconds. This is the time required to process the same amount of work as Full HD real time, which is equivalent to 30 trillion shader operations. Therefore, by dividing 30 trillion with the time it took, we can get $30 \times 10^{12} \div 2588.76 =$

11.588×10^9, or 11.588 billion shader operations per second.

One thing we do need to consider for the MPSoC platform is that it also processes the tree build and photon scattering, as well as the tree search and shader operation sections that the MAPM is dedicated to accelerating. However, as the tree search and shader operation is the main bulk of the photon mapping process, having the platform be fully dedicated to the latter sections will definitely enhance the numbers, but not by a huge factor.

Another number that we can obtain performance numbers from is a software only work by our collaborator Shawn Singh [48]. Using a purely software approach and various optimizations including SIMD methods and data reordering, a performance number of 10 million shader operations per core per second was obtained. As we cannot replicate the same experiments, we compare the relative CPU performance numbers and try to scale this number up for more recent CPUs. The CPU used in [48] is the Intel Xeon X5355. Using the numbers from [1], we can obtain performance numbers from the PassMark benchmark software [4] between the Intel Xeon X5355 and some current generation CPUs. The numbers can be seen in Figure 4.7.

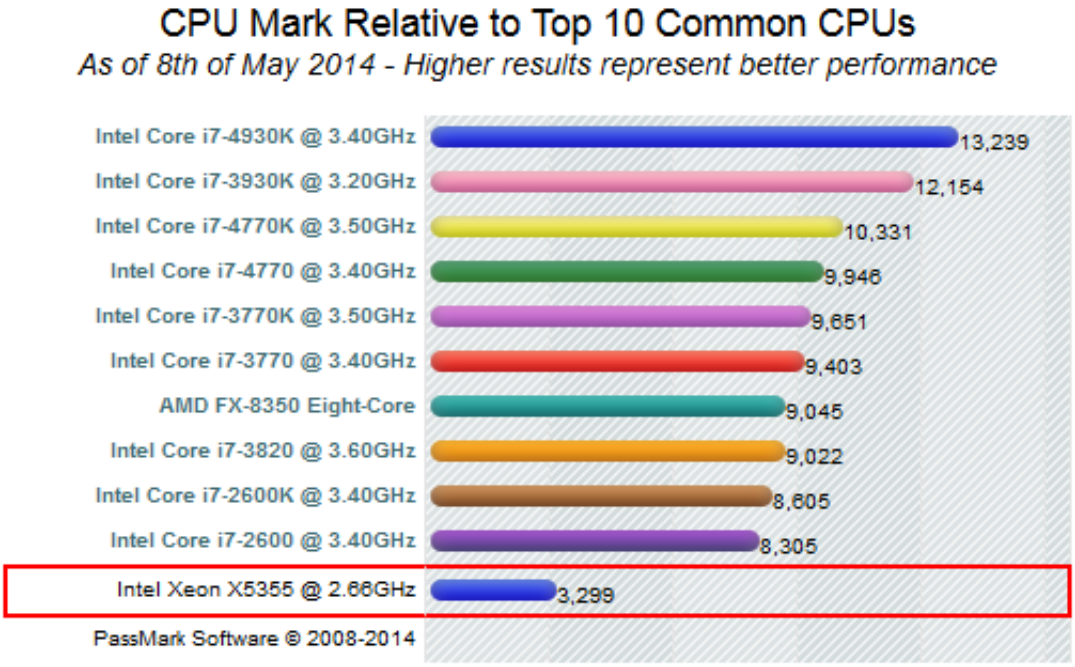


Figure 4.7: CPU performance benchmark number comparison

We shall use numbers from the Intel Core i7-4770K, which is one of the top high-end performance CPUs currently available. The benchmark score is a rough indication of how much work each CPU can process in the same amount of time.

By comparing 3,299 from the Intel Xeon X5355 and 13,239 from the Intel i7-4930K, we will assume that the Intel i7-4930K will be able to process about $13,239 \div 3,299 = 4.013$ times as many shader operations as the Intel Xeon X5355 (although this may be a generous assumption considering the floating-point heavy workload of photon mapping). Using this, we shall assume the current software performance value to be around 40 million shader operations per second.

4.3.2 Design space exploration

In this section, we shall examine and compare throughput for various points in the design space. The goal of this part is to obtain a good balance point for multiple modules inside the MAPM, and also to examine how well the architecture scales

with the increase of modules. We run simulations on various benchmarks using different module configurations for the TSM/SOM/SOAs shown in Figure 4.8.

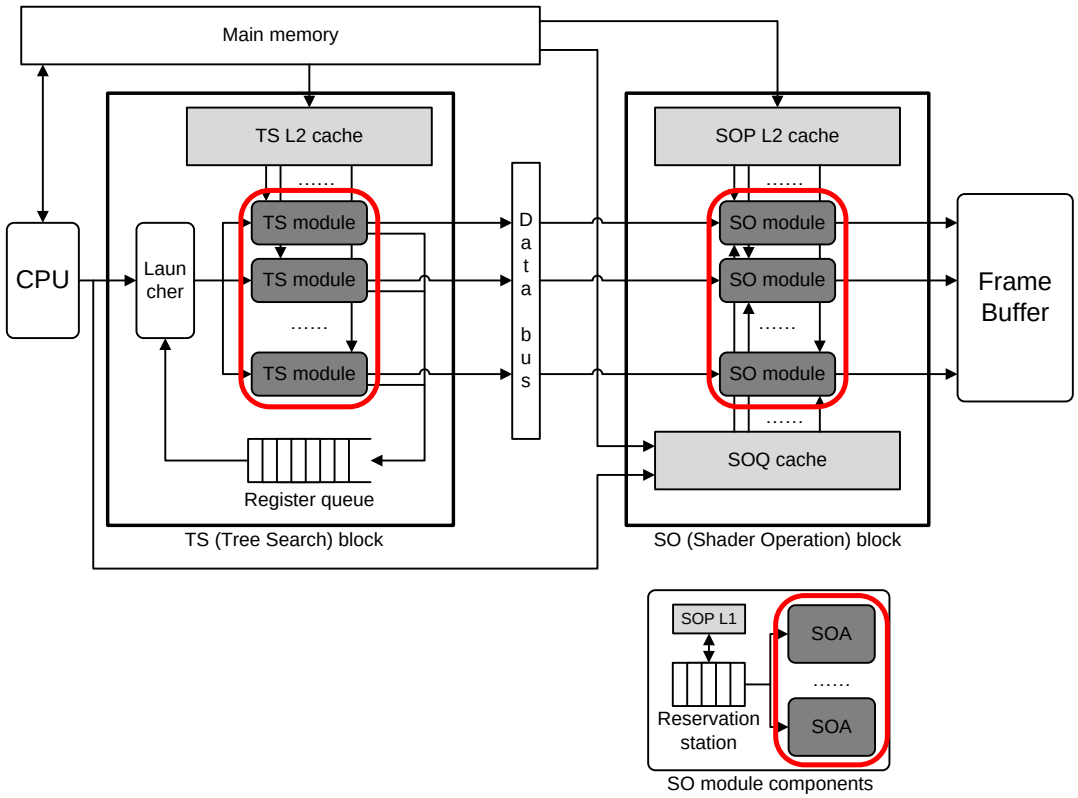


Figure 4.8: Replicable modules in the MAPM

Also, we can plug in the hardware cost values obtained from Synopsys tools to these simulation numbers to evaluate performance per hardware cost. We shall use the setup for 1.5 GHz, which seems to be a good balance between adequate performance and pushing to the hard limit (which is somewhere between 1.75 GHz and 2.0 GHz) of the library. With 1.5 GHz, each clock cycle is $1/1.5\text{G} = 0.667$ ns. Since the TSAs and SOAs will be the dominant parts of the TSM and SOM in terms of area and power, we will use multiples of the area and power of the TSA and SOA to come up with an estimate of the TSM and SOM.

We shall consider the number of shader operations per unit cost as the measuring stick for performance. By obtaining shader operations per cycle, and dividing

by the clock period 0.667 (ns), we can calculate the throughput value at billion shader operations per cycle.

Table 4.8 shows a simulation of the Cornell box, at 75×60 resolution, 36 rays per pixel and 2,000 photons. This comes down to 85,105 queries, 109,881,410 tree search operations, and 8,260,297 shader operations.

The first column shows the number of total TSMs, number of total SOMs, and SOAs per SOM.

The chart below shows the relative throughput for each setup, bundled into different setup ratios and TSM count.

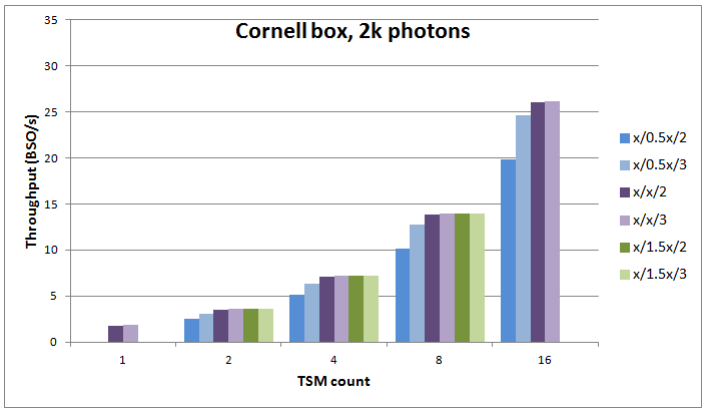


Figure 4.9: Throughput for Cornell box, 2k photons (BSO/sec)

Looking at the raw numbers and the chart, it seems that the performance scales well with the increase of the number of TSMs, which is the main block that produces the shader operations. Each group of bars has twice the number of TSMs as the previous group, and every group shows higher throughput compared to the previous one. The increase in throughput is not quite $2\times$ however, indicating some loss of performance gain at each increase of TSMs.

Inside each group, the bars have a different ratio of SOMs per TSM, and either 2 or 3 SOAs per SOM. They can be classified as one of the following: $x/0.5x/2$, $x/0.5x/2$, $x/x/2$, $x/x/3$, $x/1.5x/2$ or $x/1.5x/3$, where x is the number of TSMs.

TSMs/SOMs/SOAs	Total cycles	BSO/sec
1/1/2	7,116,879	1.741
1/1/3	6,733,815	1.840
2/1/2	4,959,391	2.498
2/1/3	3,986,511	3.108
2/2/2	3,546,205	3.494
2/2/3	3,394,998	3.650
2/3/2	3,394,975	3.650
2/3/3	3,394,972	3.650
4/2/2	2,440,261	5.078
4/2/3	1,945,136	6.370
4/4/2	1,760,068	7.040
4/4/3	1,721,170	7.199
4/6/2	1,721,159	7.199
4/6/3	1,721,154	7.199
8/4/2	1,217,064	10.181
8/4/3	968,693	12.791
8/8/2	897,360	13.808
8/8/3	887,783	13.957
8/12/2	887,776	13.957
8/12/3	887,775	13.957
16/8/2	624,084	19.854
16/8/3	503,494	24.609
16/16/2	476,341	26.012
16/16/3	473,444	26.171

Table 4.8: Total cycles and throughput for different TSM/SOM/SOA configurations

Looking inside each group and comparing between different ratio configurations, we can see $0.5x$ SOM setups show much lower performance compared to x and $1.5x$ SOM setups. For $0.5x$ SOM setups, increasing the SOAs per SOM is beneficial, especially for $x = 16$.

Between x SOMs and $1.5x$ SOMs there is almost no improvement in performance, and it seems that there is hardly any gain by using a larger number of SOMs than TSMs. This is much more evident from the following data, where we also plug in area and power costs (Table 4.9). The unit for throughput per area is million SO/sec/mm^2, and the unit for throughput per power is billion SO/sec/W.

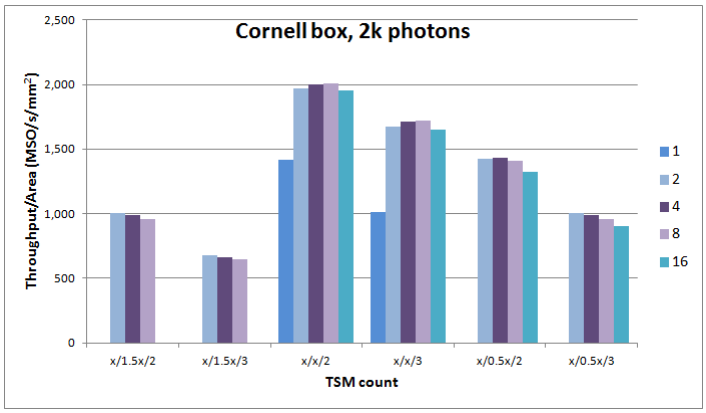


Figure 4.10: Throughput/area for Cornell box, 2k photons (MSO/sec/mm^2)

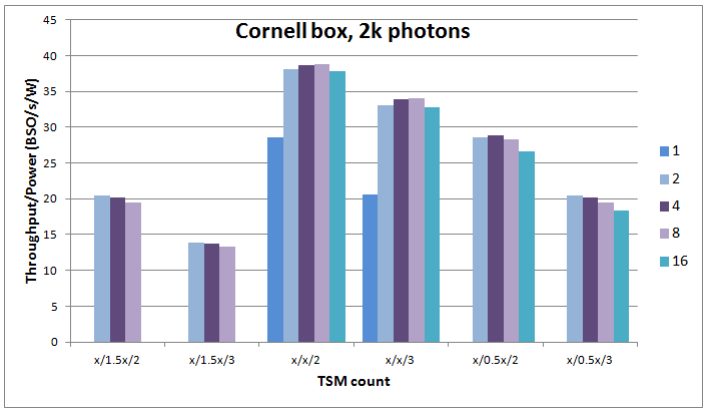


Figure 4.11: Throughput/power for Cornell box, 2k photons (BSO/sec/W)

TSMs/SOMs /SOAs	billion SO/sec	Area (mm^2)	Throughput per Area	Power (mW)	Throughput per Power
1/1/2	1.741	1.226	1,419.811	61.04	28.522
1/1/3	1.840	1.818	1,011.927	89.27	20.612
2/1/2	2.498	1.268	1,970.065	65.62	38.073
2/1/3	3.108	1.860	1,670.747	93.85	33.118
2/2/2	3.494	2.452	1,424.709	122.08	28.621
2/2/3	3.650	3.636	1,003.555	178.54	20.441
2/3/2	3.650	3.636	1,003,561	178.54	20.442
2/3/3	3.650	5.413	674.227	263.23	13.865
4/2/2	5.078	2.536	2,001.901	131.24	38.689
4/2/3	6.370	3.721	1,712.079	187.70	33.937
4/4/2	7.040	4.905	1,435.260	244.16	28.833
4/4/3	7.199	7.273	989.753	357.08	20.160
4/6/2	7.199	7.273	989.759	357.08	20.160
4/6/3	7.199	10.826	664.956	526.46	13.674
8/4/2	10.181	5.073	2,006.945	262.48	38.786
8/4/3	12.791	7.441	1,718.927	375.40	34.073
8/8/2	13.808	9.809	1,407.549	488.32	28.276
8/8/3	13.957	14.547	959.431	714.16	19.543
8/12/2	13.957	14.547	959.439	714.16	19.543
8/12/3	13.957	21.652	644.584	1052.92	13.255
16/8/2	19.854	10.145	1,956.933	524.96	37.820
16/8/3	24.609	14.882	1,653.558	750.80	32.777
16/16/2	26.012	19.619	1,325.813	976.64	26.634
16/16/3	26.171	29.093	899.543	1428.32	18.323

Table 4.9: Performance per hardware cost

With the charts grouped by the ratio of TSM/SOM/SOAs, we can easily see that in terms of throughput per area and throughput per power, the $x/1.5x/2$ and $x/1.5x/3$ configurations are easily outclassed by other TSM/SOM setups. This clearly implies that adding more SOMs past the number of TSMs results in inefficient hardware. The best setup in terms of area and power efficiency is clearly the $x/x/2$ setup, followed by the $x/x/3$ setup.

One more thing to note here is how having only a single TSM is heavily inefficient. Both 1/1/2 and 1/1/3 setups are less area and power efficient by about 20-25%, compared to other setups in the same $x/x/2$ or $x/x/3$ configuration. Combining the fact that this setup showed very low raw throughput numbers as well, we can consider this to be the most inefficient setup, and that in order for the MAPM to show its strength, parallel execution is an essential factor.

To make sure these observations are not an outlier and holds true for other benchmarks, we run simulations on other workloads. The following show three more simulations on different benchmark scenes, at 75×60 resolution, 36 rays per pixel and 8,000 photons. The instruction count for each benchmark is shown in Table 4.10.

Name	Queries	TS ops	Shader ops
Cornell box	85,211	31,973,698	26,478,415
Sponza corridor	151,998	38,941,668	29,953,648
Sponza atrium	196,771	30,548,991	22,046,274

Table 4.10: Operation count for different benchmark scenes

Cycle count and SO/sec is shown in Table 4.11. TP stands for throughput. The unit for throughput is billion SO/sec.

	Cornell box		Sponza corridor		Sponza atrium	
Setup	Cycles	TP	Cycles	TP	Cycles	TP
1/1/2	20,474,828	1.940	25,331,428	1.774	20,320,174	1.627
1/1/3	18,754,570	1.118	23,933,965	1.877	19,478,273	1.698
2/1/2	14,851,484	2.674	17,754,645	2.531	13,731,282	2.408
2/1/3	11,736,254	3.384	14,309,707	3.140	11,225,938	2.946
2/2/2	10,140,375	3.917	12,543,764	3.582	10,065,904	3.285
2/2/3	9,405,470	4.223	12,018,084	3.739	9,792,134	3.377
4/2/2	7,326,185	5.421	8,698,580	5.165	6,713,990	4.925
4/2/3	5,709,090	6.957	6,903,696	6.508	5,448,106	6.070
4/4/2	4,990,699	7.958	6,176,441	7.274	5,014,895	6.594
4/4/3	4,726,300	8.404	6,050,972	7.425	4,952,102	6.678
8/4/2	3,630,749	10.939	4,270,920	10.520	3,317,907	9.967
8/4/3	2,787,675	14.248	3,334,195	13.476	2,693,440	12.278
8/8/2	2,476,073	16.041	3,098,929	14.499	2,552,818	12.954
8/8/3	2,390,627	16.614	3,069,949	14.636	2,538,057	13.029
16/8/2	1,819,489	21.829	2,147,718	20.920	1,682,495	19.655
16/8/3	1,387,157	28.632	1,676,474	26.801	1,384,517	23.885
16/16/2	1,247,988	31.825	1,595,481	28.161	1,338,924	24.698
16/16/3	1,225,668	32.405	1,589,083	28.274	1,335,619	24.760

Table 4.11: Total cycles and throughput for different benchmark scenes

Below, we can see bar charts of each benchmark.

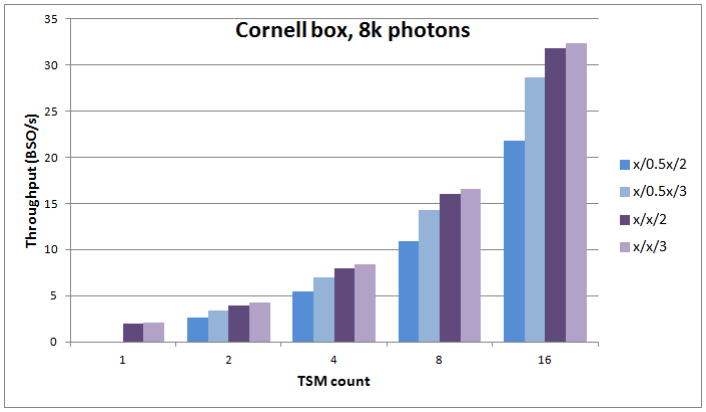


Figure 4.12: Throughput for Cornell box, 8k photons (BSO/sec)

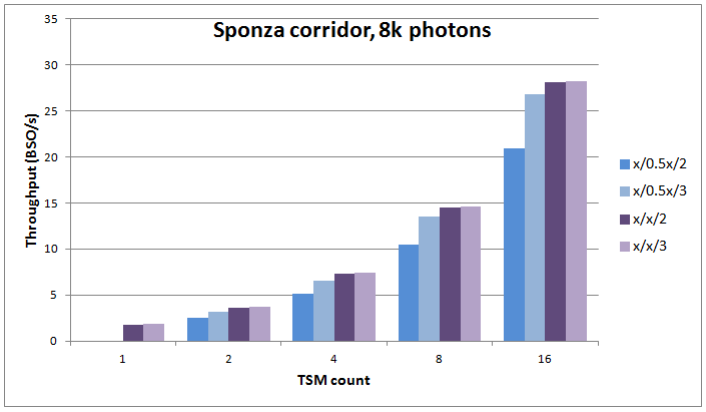


Figure 4.13: Throughput for Sponza corridor, 8k photons (BSO/sec)

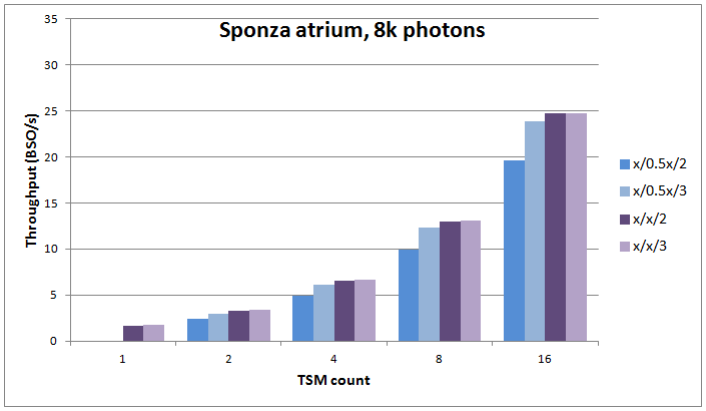


Figure 4.14: Throughput for Sponza atrium, 8k photons (BSO/sec)

Again, we can see that the throughput generally scales up well with the number of TSMs, and $x/0.5x$ setups are less efficient than x/x setups.

We can also see from the numbers between Sponza corridor and Sponza atrium, that with the same number of photons and the same scene file, the location of the camera and light source make a difference in the number of query points. This in turn affects the throughput of the data and overall, performance numbers from Sponza corridor are better than numbers from Sponza atrium.

Area and power efficiency charts are shown below.

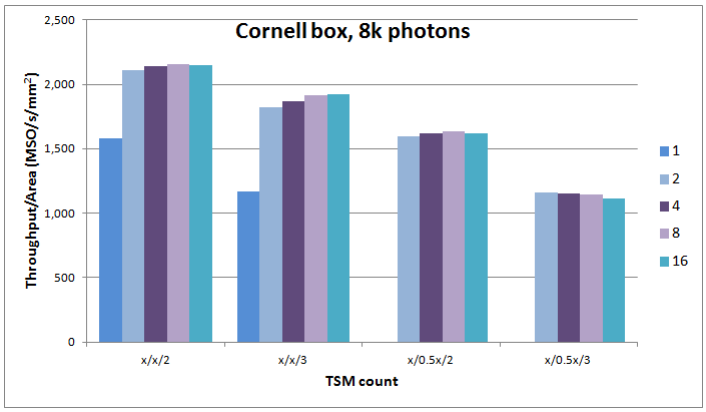


Figure 4.15: Throughput/area for Cornell box, 8k photons (MSO/sec/mm^2)

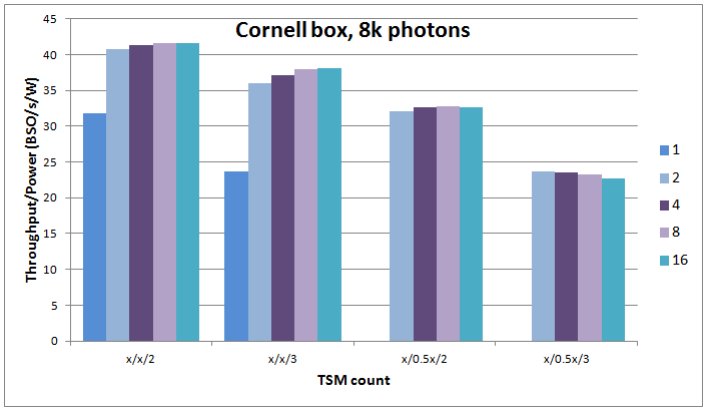


Figure 4.16: Throughput/power for Cornell box, 8k photons (BSO/sec/W)

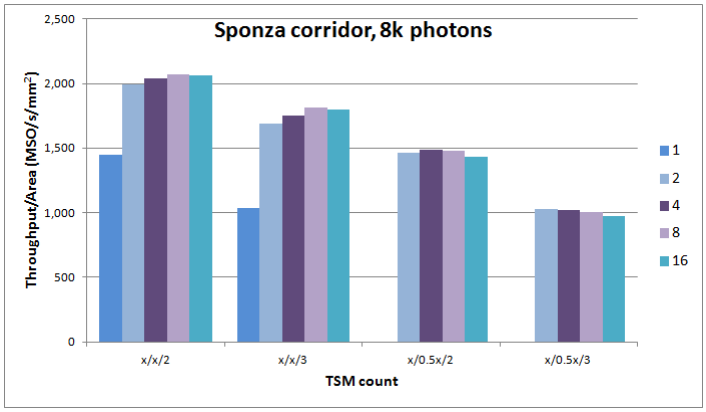


Figure 4.17: Throughput/area for Sponza corridor, 8k photons (MSO/sec/mm^2)

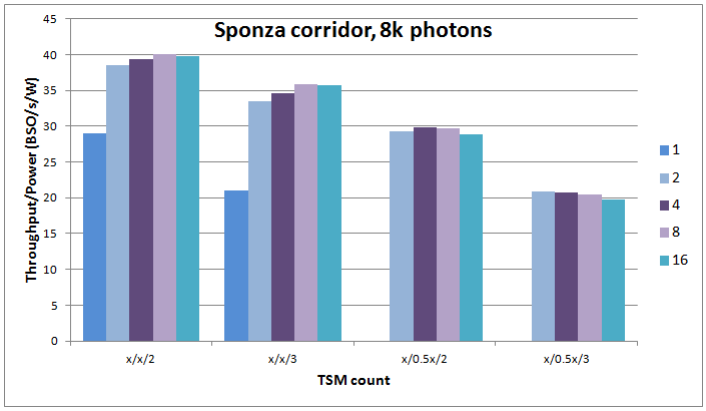


Figure 4.18: Throughput/power for Sponza corridor, 8k photons (BSO/sec/W)

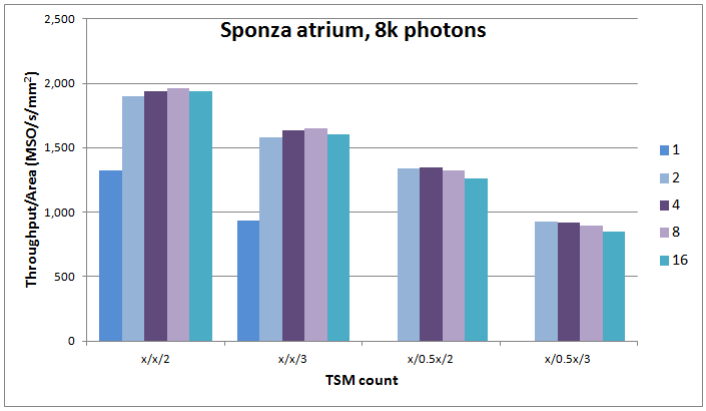


Figure 4.19: Throughput/area for Sponza atrium, 8k photons (MSO/sec/mm^2)

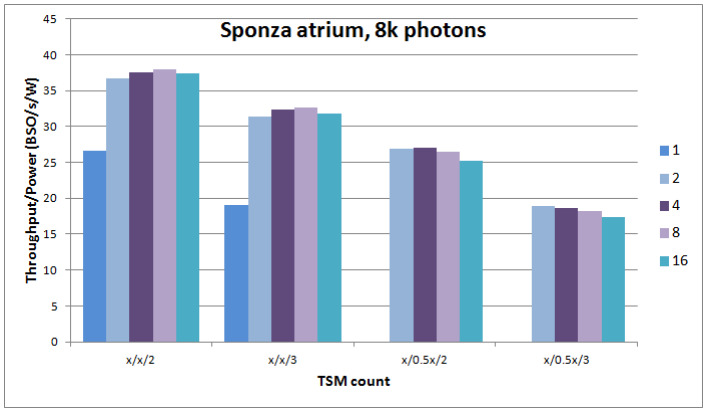


Figure 4.20: Throughput/power for Sponza atrium, 8k photons (BSO/sec/W)

Area and power efficiency graphs also show a similar trend, where the best setup in terms of hardware cost efficiency is the $x/x/2$ setup, followed by the $x/x/3$ setup. And again, having just one TSM is clearly inefficient, as it shows vastly lower numbers in the chart.

4.3.3 Throughput evaluation

The previous section showed us some concrete numbers on what the MAPM is capable of in terms of throughput performance. In this section, we examine the limiting factors to the theoretical throughput we can get, and try to see how much performance we can get before hitting these limits.

Using the derivations from Section 4.3.1, we shall use 40 million SO/s to be the maximum throughput for an optimized software implementation on a top-tier performance CPU, and 11.588 billion SO/s as the performance of the MPSoC implementation.

Regarding hardware costs, we shall look at values from Intel's Haswell chipset CPU specifications. Haswell is the latest chipset from Intel currently in the market, succeeding the Sandy Bridge and Ivy Bridge chipsets. The Haswell desktop CPUs use the 22 nm process, with 1.4 billion transistors and a 177 mm^2 die size.

The Core i7 5930K high-end processor, which is expected to be released later in the year, requests a massive 140 W TDP (Thermal Design Power), whereas less power intensive but still top performing designs like the Core i7 4770K (June 2013) runs at 84 W, and the low end Celeron G1820T (December 2013) has a TDP of 35 W. The numbers in Table 4.12 is a list of a few comparison points for evaluating the throughput limits of the MAPM. All information in the table has been obtained from [3].

Model	TDP (W)	Memory bandwidth (GB/s)
Core i7 5930K	140	(not available yet)
Core i7 4770K	84	25.6
Core i5 4670T	45	25.6
Pentium G3430	53	25.6
Celeron G1820T	35	21.3

Table 4.12: Specifications of select Intel Haswell CPUs

The main comparison point for hardware cost will be the Intel Celeron, which is the low-end processor which sacrifices some performance for a less power-hungry CPU with higher mobility and longer battery life.

Overall, the simulation data shows that a 16/16/2 setup has performance numbers that are close to the highest, which come from 16/16/3 setups, but in terms of hardware cost, is much more efficient than 16/16/3. Therefore, we shall consider the 16/16/2 setup to be our standard setup and use it for comparison purposes. We shall assume the hardware setup to be at the clock speed of 1.5 GHz, with 16 TSMs, 16 SOMs, and 2 SOAs per SOM.

We examine how each resource limits the theoretical throughput we can get, one by one.

1. **Memory bandwidth**

   To be able to sustain the high throughput, the bandwidth between the MAPM and the main memory needs to be able to sustain the data transfer that is necessary to read parameters from the memory. To examine the memory requirements, we look at cache simulation results from four benchmarks from the previous section: Cornell box with 2k photons, Cornell box with 8k photons, Sponza corridor with 8k photons, and Sponza atrium with 8k photons.

   The cache setup for the simulation is unchanged from Table 4.2 in Section 4.1.3. We count all cache misses and multiply them by the block size to obtain the total memory fetched. Dividing by the total time will give us the bandwidth numbers.

	Cycles	Total memory (B)	Bandwidth (GB/s)
Cornell box, 2k	476,341	1,534,784	4.501
Cornell box, 8k	1,247,988	2,055,264	2.301
Sponza corridor, 8k	1,595,481	3,123,904	2.735
Sponza atrium, 8k	1,338,924	3,840,352	4.001
Total	4,658,734	10,554,304	3.165

Table 4.13: Cache simulation results for bandwidth evaluation

   This shows us that in order to sustain throughput of 27.674 billion SO/s, the required memory bandwidth is 3.065 GB/s. This is a much lower value than the memory bandwidth of recent Intel CPUs, which is around 20-25 GB/s. Comparing it to the limit of 21.3 GB/s for the Intel Celeron G1820T, this is about 14.4% of the limit.

2. **Area limit**

For the 16/16/2 MAPM setup at 1.5 GHz, the area cost estimate is 19.619 mm^2. This is about 11.1% of the 177 mm^2 die size of the Intel Celeron G1820T.

3. **Power limit**

   The 16/16/2 MAPM setup has a power requirement of 976.64 Mw. This is around 2.8% of the Intel Celeron G1820T's average power dissipation requirement of 35 W.

In conclusion, memory bandwidth, chip area, and power consumption will each be a factor on limiting the throughput, but our simulation numbers show that even with conservative assumptions, the MAPM is capable of easily achieving 27.674 billion SO/s. If we decide to have two 16/16/2 MAPM accelerator units, it would have a potential possible throughput of 55.348 billion SO/s, at 28.8% of the bandwidth, 22.2% of the area, and 5.6% of the power consumption of the low-performance low-power consuming Intel Celeron G1820T.

55.348 billion SO/s would be an improvement of a factor of 1384× over the software implementation. Compared to the MPSoC implementation, we still have a 4.78× throughput improvement. This is even more impressive when we consider the hardware cost for the MPSoC version, which uses 64 ARMv4 cores. The ARM810 CPU, which uses a single ARMv4 core, has an area of 53.5 mm^2 [31], which is already larger than the size of two 16/16/2 MAPM implementation of 39.24 mm^2. In terms of throughput per area, the MAPM shows a huge improvement over the MPSoC setup.

However, considering the grand goal of real-time rendering in Full HD, the MAPM still comes short of 30 trillion SO/s. Using a somewhat high (for a co-processor) hardware cost limit of the Intel Core i7 4770 at 25.6 GB/s, 177 mm^2 die size, and 84 W TDP, a 16/16/2 MAPM setup would require 12.0% of the memory bandwidth, 11.1% of the area, and 1.2% of the power cost. With 8 duplicate units,

we can obtain a throughput of 221.392 billion SO/s, which is about 0.74% of the processing power necessary for real-time Full HD rendering. With 135 units, we would be able to obtain 30 trillion SO/s, at a cost of 413.775 GB/s, 2648.565 mm^2, and 131.85 W, implying that we still need a couple orders of magnitude of improvement in order to make this feasible.

CHAPTER 5

Summary and future work

5.1 Summary

In this dissertation, we present detailed designs of MAPM, a multi-accelerator architecture for enhancing the throughput of the 3D rendering algorithm, photon mapping. MAPM uses online arithmetic, which allows for extremely parallel processing, high clock speed and low power cost. Implementation of the design in VHDL, and behavioral verification of the design using test benches on ModelSim allowed us to make sure the design is working as intended.

For hardware cost evaluation, we created a conventional parallel circuit with equivalent functionality using the FloPoCo arithmetic circuit generator, and compared it with the cost of the MAPM modules. In order to obtain hardware cost, we used Synopsys Design Compiler with a 90 nm component library. With the same options, MAPM showed a synthesizable clock speed at about 3.5×, dynamic power consumption of 0.104×, and area cost of 1.799× compared to FloPoCo.

The cycle-accurate simulation code was written in C++, using the Intel Pin tool library to interact with the original PM code. The data flow through the MAPM architecture is simulated cycle by cycle, taking into account any stalls created by the memory. By running simulations on various different configurations of the MAPM with various benchmark files, and combining the data with the hardware costs from Synopsys tools, we decided that a 16/16/2 setup of TSMs/SOMs/SOAs at 1.5 GHz shows both good performance and efficiency in

terms of area and power costs.

Using all the information obtained from the evaluations and simulations, we compare the cost and performance of the 16/16/2 MAPM setup with recent hardware and other work. Using two 16/16/2 MAPM configurations, we would require 28.8% of the bandwidth, 22.2% of the area, and 5.6% of the power consumption of the low-end Intel Celeron G1820T, which is an acceptable cost for a potential throughput of 55.348 billion SO/s. This throughput is an improvement of a factor of 1384× over the optimized software implementation of [48], and an improvement of 4.78× compared to the MPSoC setup in [17]. This is an improvement even before we consider the area cost, which is significantly higher for the MPSoC.

From these performance results, along with other evaluation and simulation results, we conclude that the MAPM shows significant throughput improvement while allowing for high clock speed and low power cost circuits. However, for the ultimate goal of real-time photon mapping on Full HD, we still need improvements of a couple orders of magnitude.

5.2 Future work

These are some considerations for futher work on the MAPM and accelerating photon mapping.

The current hardware cost numbers for the MAPM is definitely an improvement over conventional parallel architecture, but it still needs improvement before the ultimate goal of Full HD rendering in real-time can be achieved. One place we can start is by exploring the benefits of more compact hardware. As explored in [25] and [16], for pipelined online arithmetic, a truncation error has limited propagation which can be determined from each circuit. Using this, it is possible to have a truncated circuit with the same precision with the original full circuit, which will likely give us a more compact circuit with even lower power

consumption.

Another point we can consider is reduced precision. The eyesight is one of the human senses which is quite tolerant to errors, and in previous work [49] we have compared the hardware cost with reduced precision bits. Evaluating the throughput gain and hardware cost reduction we can get with a reduced precision, along with any drop in final image quality would be something that may yield interesting results.

An area we can still improve a lot on is memory bandwidth, which is one of the more limiting cost factors of the MAPM. As mentioned in [49], by carefully encoding the data to work on a bulk of data at once, and reordering instructions to benefit from this scheme, photon or query point data can be compressed to 2:1 or up to 3:1 compression ratio. Reduced precision, mentioned above, will also reduce the amount of data transfer. The real gain may come from rearrangement of shader operations, where improvement of data locality will have a huge gain in terms of bandwidth. Previous work [51] has already shown that reordering can reduce the bandwidth requirement of photon mapping by an order of magnitude.

CHAPTER 6

List of abbreviations and commonly used terms

6.1 Abbreviations

The following is a list of all abbreviations used in this dissertation, in alphabetical order.

FGR	Final Gather Ray
FPS	Frames per second
KD-tree	K-Dimensional tree
PM	Photon Mapping
MAPM	Multi-Accelerator for Photon Mapping
RGB	Red, Green, Blue
SIMD	Single Instruction, Multiple Data
SO	Shader Operation
SOA	Shader Operation Accelerator
SOM	Shader Operation Module
SOP cache	Shader Operation – Photon cache
SOQ cache	Shader Operation – Query point cache
TDP	Thermal Design Power
TP	Throughput
TS	Tree Search
TSA	Tree Search Accelerator
TSM	Tree Search Module

6.2 Mathematical expressions

The following is a list of all mathematical expressions used in this dissertation. These are related to the IEEE-754 standard floating-point format.

for the IEEE-754 standard floating-point expression f:

$f.s$	the sign bit. 0 if $f \geq 0$, 1 if $f < 0$
$f.e$	the exponent bits
B_f	the bias of the exponent.
	$B_f = 2^{n-1} - 1$ where n is the bit width of $f.e$
exp_f	the actual exponent value. $exp_f = f.e - B_f$
$f.m$	the mantissa bits
$f.mwh$	the mantissa bits with the hidden bit 1 appended
	before the most significant bit
f_{val}	value of $f = (-1)^{f.s} 2^{exp_f} (f.mwh)$
$f.am$	aligned mantissa with hidden bit,
	used in the Pipelined Comparator in the SOA

References

[1] "Cpu benchmarks." [Online]. Available: http://www.cpubenchmark.net

[2] "Flopoco." [Online]. Available: http://flopoco.gforge.inria.fr

[3] "Intel ark." [Online]. Available: http://ark.intel.com

[4] "Passmark software." [Online]. Available: http://www.passmark.com

[5] "Pin." [Online]. Available: http://www.pintool.org

[6] "Python programming language." [Online]. Available: https://www.python.org

[7] "Synopsys 90nm generic library." [Online]. Available: http://www.synopsys.com/community/universityprogram/pages/library.aspx

[8] "Ieee standard for floating-point arithmetic," *IEEE Std 754-2008*, pp. 1–70, Aug 2008.

[9] J. L. Bentley, "Multidimensional binary search trees used for associative searching," *Commun. ACM*, vol. 18, no. 9, pp. 509–517, Sep. 1975. [Online]. Available: http://doi.acm.org/10.1145/361002.361007

[10] F. de Dinechin and B. Pasca, "Custom arithmetic datapath design for fpgas using the flopoco core generator," *Design & Test of Computers, IEEE*, vol. 28, no. 4, pp. 18–27, 2011.

[11] P. Djeu, W. Hunt, R. Wang, I. Elhassan, G. Stoll, and W. R. Mark, "Razor: An architecture for dynamic multiresolution ray tracing," *ACM Transactions on Graphics (TOG)*, vol. 30, no. 5, p. 115, 2011.

[12] P. Dutre, K. Bala, P. Bekaert, and P. Shirley, *Advanced global illumination*. AK Peters, 2003, vol. 11.

[13] M. D. Ercegovac, "A general hardware-oriented method for evaluation of functions and computations in a digital computer," *Computers, IEEE Transactions on*, vol. 100, no. 7, pp. 667–680, 1977.

[14] M. D. Ercegovac and T. Lang, "On-line scheme for computing rotation factors," *Journal of Parallel and distributed computing*, vol. 5, no. 3, pp. 209–227, 1988.

[15] M. D. Ercegovac, J. H. Moreno, and T. Lang, *Introduction to digital systems*. John Wiley & Sons, Inc., 1998.

[16] M. D. Ercegovac and T. Lang, *Digital Arithmetic*. San Francisco, CA: Morgan Kaufman Publishers, 2004.

[17] M. Fallahpour, M.-B. Lin, and C.-H. Lin, "Parallel photon-mapping rendering on a mesh-noc-based mpsoc platform," *Journal of Parallel and Distributed Computing*, 2014.

[18] K. Fatahalian and M. Houston, "A closer look at gpus," in *Communications of the ACM*, vol. 51, no. 10, New York, NY, October 2008, pp. 50–57.

[19] C. M. Goral, K. E. Torrance, D. P. Greenberg, and B. Battaile, "Modeling the interaction of light between diffuse surfaces," in *ACM SIGGRAPH Computer Graphics*, vol. 18, no. 3. ACM, 1984, pp. 213–222.

[20] A. Grnarov and M. Ercegovac, "On-line multiplicative normalization," *evaluation*, vol. 1, p. 4, 1983.

[21] J. Günther, H. Friedrich, I. Wald, H.-P. Seidel, and P. Slusallek, "Ray tracing animated scenes using motion decomposition," in *Computer Graphics Forum*, vol. 25, no. 3. Wiley Online Library, 2006, pp. 517–525.

[22] T. Hachisuka, S. Ogaki, and H. W. Jensen, "Progressive photon mapping," *ACM Transactions on Graphics (TOG)*, vol. 27, no. 5, p. 130, 2008.

[23] V. Havran, "Heuristic ray shooting algorithms," Ph.D. dissertation, Faculty of Electrical Engineering, Czech Technical University, 2000.

[24] V. Havran, R. Herzog, and H.-P. Seidel, "Fast final gathering via reverse photon mapping," *Computer Graphics Forum*, vol. 24, no. 3, pp. 323–332, August 2005. [Online]. Available: http://dx.doi.org/10.1111/j.1467-8659.2005.00857.x

[25] Z. Huang and M. Ercegovac, "Fpga implementation of pipelined on-line scheme for 3-d vector normalization," in *Field-Programmable Custom Computing Machines, 2001. FCCM '01. The 9th Annual IEEE Symposium on*, 29 2001-april 2 2001, pp. 61 –70.

[26] W. Hunt, W. R. Mark, and G. Stoll, "Fast kd-tree construction with an adaptive error-bounded heuristic," in *Interactive Ray Tracing 2006, IEEE Symposium on*. IEEE, 2006, pp. 81–88.

[27] ITRS, "2013 itrs edition," 2013. [Online]. Available: http://www.itrs.net/Links/2013ITRS/Home2013.htm

[28] H. W. Jensen, *Realistic Image Synthesis Using Photon Mapping*. Natick, MA: A. K. Peters, Ltd., 2001.

[29] G. S. Johnson, J. Lee, C. A. Burns, and W. R. Mark, "The irregular z-buffer: Hardware acceleration for irregular data structures," *ACM Trans. Graph.*, vol. 24, pp. 1462–1482, October 2005. [Online]. Available: http://doi.acm.org/10.1145/1095878.1095889

[30] T. Kim, X. Sun, and S. Yoon, "T-rex: Interactive global illumination of massive models on heterogeneous computing resources," 2013.

[31] G. Larri, "Arm810: Dancing to the beat of a different drum," July 1996. [Online]. Available: http://www.dlhoffman.com/publiclibrary/software/hot_chips_papers/hc96/hc8_pdf/4.1.pdf

[32] B. D. Larsen and N. J. Christensen, "Simulating photon mapping for real-time applications," *Eurographics Symposium on Rendering*, pp. 123–131, June 2004. [Online]. Available: http://www2.imm.dtu.dk/pubdb/p.php?3192

[33] D. Lau, A. Schneider, M. Ercegovac, and J. Villasenor, "Fpga-based structures for on-line fft and dct," in *Field-Programmable Custom Computing Machines, 1999. FCCM '99. Proceedings. Seventh Annual IEEE Symposium on*, 1999, pp. 310 –311.

[34] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: Building customized program analysis tools with dynamic instrumentation," *SIGPLAN Not.*, vol. 40, no. 6, pp. 190–200, Jun. 2005. [Online]. Available: http://doi.acm.org/10.1145/1064978.1065034

[35] J. D. MacDonald and K. S. Booth, "Heuristics for ray tracing using space subdivision," *The Visual Computer*, vol. 6, pp. 153–166, 1990, 10.1007/BF01911006. [Online]. Available: http://dx.doi.org/10.1007/BF01911006

[36] M. McGuire and D. Luebke, "Hardware-accelerated global illumination by image space photon mapping," in *Proceedings of the Conference on High Performance Graphics 2009*, ser. HPG '09. New York, NY, USA: ACM, 2009, pp. 77–89. [Online]. Available: http://doi.acm.org/10.1145/1572769.1572783

[37] E. Mosanya and E. Sanchez, "A fpga-based hardware implementation of generalized profile search using online arithmetic," in *Proceedings of the 1999 ACM/SIGDA seventh international symposium on Field programmable gate arrays*, ser. FPGA '99. New York, NY, USA: ACM, 1999, pp. 101–111. [Online]. Available: http://doi.acm.org/10.1145/296399.296436

[38] V. G. Oklobdzija and M. D. Ercegovac, "An on-line square root algorithm," *IEEE Transactions on Computers*, vol. 31, no. 1, pp. 70–75, 1982.

[39] J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone, and J. C. Phillips, "Gpu computing," *Proceedings of the IEEE*, vol. 96, no. 5, pp. 879–899, May 2008.

[40] S. G. Parker, 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.*, vol. 29, no. 4, pp. 66:1–66:13, Jul. 2010. [Online]. Available: http://doi.acm.org/10.1145/1778765.1778803

[41] M. Pharr and G. Humphreys, *Physically Based Rendering: From Theory to Implementation*, 2nd ed. Burlington, MA: Morgan Kaufmann, 2010.

[42] S. Popov, J. Gunther, H.-P. Seidel, and P. Slusallek, "Experiences with streaming construction of sah kd-trees," in *Interactive Ray Tracing 2006, IEEE Symposium on*. IEEE, 2006, pp. 89–94.

[43] T. J. Purcell, C. Donner, M. Cammarano, H. W. Jensen, and P. Hanrahan, "Photon mapping on programmable graphics hardware," in *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, ser. HWWS '03. Aire-la-Ville, Switzerland, Switzerland: Eurographics Association, 2003, pp. 41–50. [Online]. Available: http://dl.acm.org/citation.cfm?id=844174.844181

[44] V. J. Reddi, A. Settle, D. A. Connors, and R. S. Cohn, "Pin: A binary instrumentation tool for computer architecture research and education," in *Proceedings of the 2004 Workshop on Computer Architecture Education: Held in Conjunction with the 31st International Symposium on Computer Architecture*, ser. WCAE '04. New York, NY, USA: ACM, 2004. [Online]. Available: http://doi.acm.org/10.1145/1275571.1275600

[45] M. Shevtsov, A. Soupikov, and A. Kapustin, "Highly parallel fast kd-tree construction for interactive ray tracing of dynamic scenes," in *Computer Graphics Forum*, vol. 26, no. 3. Wiley Online Library, 2007, pp. 395–404.

[46] B. W. Silverman, *Density estimation for statistics and data analysis*. CRC press, 1986, vol. 26.

[47] S. Singh, "Accelerating photon mapping," Ph.D. dissertation, University of California, Los Angeles, Los Angeles, CA, 2011.

[48] S. Singh and P. Faloutsos, "Simd packet techniques for photon mapping," *Symposium on Interactive Ray Tracing*, vol. 0, pp. 87–94, 2007.

[49] S. Singh, S. h. Pan, and M. Ercegovac, "Accelerating the photon mapping algorithm and its hardware implementation," in *Application-Specific Systems, Architectures and Processors (ASAP), 2011 IEEE International Conference on*, September 2011, pp. 149–157.

[50] G. S. Sohi and M. Franklin, "High-bandwidth data memory systems for superscalar processors," in *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS IV. New York, NY, USA: ACM, 1991, pp. 53–62. [Online]. Available: http://doi.acm.org/10.1145/106972.106980

[51] J. Steinhurst, G. Coombe, and A. Lastra, "Reordering for cache conscious photon mapping," in *Proceedings of Graphics Interface 2005*. Canadian Human-Computer Communications Society, 2005, pp. 97–104.

[52] J. E. Steinhurst, "Practical photon mapping in hardware," Ph.D. dissertation, University of North Carolina, Chapel Hill, Chapel Hill, NC, 2007.

[53] C.-L. Su and C.-W. Jen, "Motion estimation using msd-first processing," *Circuits, Devices and Systems, IEE Proceedings* -, vol. 150, no. 2, pp. 124–133, apr 2003.

[54] R. M. Tomasulo, "An efficient algorithm for exploiting multiple arithmetic units," *IBM Journal of research and Development*, vol. 11, no. 1, pp. 25–33, 1967.

[55] K. S. Trivedi and M. D. Ercegovac, "On-line algorithms for division and multiplication," in *Computer Arithmetic (ARITH), 1975 IEEE 3rd Symposium on*. IEEE, 1975, pp. 161–167.

[56] K. S. Trivedi and J. G. Rusnak, "Higher radix on-line division." in *IEEE Symposium on Computer Arithmetic*, 1978, pp. 164–174.

[57] P.-G. Tu and M. D. Ercegovac, "Application of on-line arithmetic algorithms to the svd computation: Preliminary results," in *Computer Arithmetic, 1991. Proceedings., 10th IEEE Symposium on*. IEEE, 1991, pp. 246–255.

[58] D. M. Tullsen and M. D. Ercegovac, "Design and vlsi implementation of an on-line algorithm," in *30th Annual Technical Symposium*. International Society for Optics and Photonics, 1986, pp. 92–99.

[59] I. Wald, W. R. Mark, J. Günther, S. Boulos, T. Ize, W. Hunt, S. G. Parker, and P. Shirley, "State of the art in ray tracing animated scenes," *Computer Graphics Forum*, vol. 28, no. 6, pp. 1691–1722, 2009.

[60] O. Watanuki and M. D. Ercegovac, "Floating-point on-line arithmetic: Algorithms," in *Computer Arithmetic (ARITH), 1981 IEEE 5th Symposium on*. IEEE, 1981, pp. 81–86.

[61] ——, "Floating-point on-line arithmetic: Error analysis," in *Computer Arithmetic (ARITH), 1981 IEEE 5th Symposium on*. IEEE, 1981, pp. 87–91.

[62] ——, "Error analysis of certain floating-point on-line algorithms," *IEEE transactions on computers*, vol. 32, no. 4, pp. 352–358, 1983.

[63] S. Woop, E. Brunvand, and P. Slusallek, "Estimating performance of a ray-tracing asic design," *Symposium on Interactive Ray Tracing*, vol. 0, pp. 7–14, 2006.

[64] S. Woop, J. Schmittler, and P. Slusallek, "Rpu: a programmable ray processing unit for realtime ray tracing," *ACM Trans. Graph.*, vol. 24, pp. 434–444, July 2005. [Online]. Available: http://doi.acm.org/10.1145/1073204.1073211