GLOBAL ILLUMINATION USING PHOTON MAPS

Fengshi Zheng, Hongyu He, Kehan Xu, Zijun Hui

Department of Computer Science
ETH Zurich, Switzerland

ABSTRACT

With the increasing power of hardware, ray tracing and its variants are becoming more prevalent. They can create images that closely mimics the real world. Stochastic Progressive Photon Mapping (SPPM) is such a variant that improves ray tracing's quality in global illumination, where light bounces around the scene. Although GPUs are often used to accelerate rendering programs, it is not always available. This report accelerates the SPPM algorithm on a single core Skylake CPU and achieved up to a $22.69\times$ speedup over a baseline version. With the help of performance analysis tools, a systematic approach was taken in identifying bottlenecks and optimising them. Notable optimisations include vectoring the entire algorithm with Intel intrinsics, implementing a SIMD random number generator, and minimising cache misses by changing data structure layouts.

1. INTRODUCTION

Motivation. Physically based rendering aims to produce images that closely resembles the real world. Today, this technique is used across the entertainment industry, ranging from video games to movie visual effects. Monte Carlo methods of rendering [1] are powerful techniques that produce physically accurate results. Unfortunately, they are usually very slow even on modern hardware, often taking hours to generate satisfactory noise-free images. Therefore, improving the performance of rendering algorithms has been a vital task in computer graphics. Stochastic progressive photo mapping (SPPM) [2] is one such algorithm that this paper discusses.

Related work. Previous efforts on optimisation have focused on performing generalised ray tracing either on graphics processing units (GPU) or on the same CPU core with single instruction multiple data (SIMD). Nvidia's OptiX [3] is a low level ray tracing API that allows user to define actions on rays to be run in parallel on GPUs. Mitsuba 2 [4], a research-based rendering framework, implements data structures and parallel operations both on GPUs and with SIMD. The user only needs to define the higher-level rendering algorithms without dealing with the underlying logic. Intel's Embree [5] provides ray tracing kernels optimised for Intel processors that support SIMD. Dr.Jit [6] is a just-in-time compiler that turns high-level python/c++ codes into efficient SIMD kernels on GPUs and on CPUs. Mitsuba 3, a newer version of Mitsuba 2, is built on top of Dr.Jit, providing a high-performance forward and inverse rendering framework. The above frameworks only consider parallel programming or general optimization of rendering algorithms, with a specific focus on ray-intersection logic, scheduling, compute framework, instead of specific algorithms such as SPPM.

Contribution. This paper focuses on optimising SPPM on a single core setting. As a proof-of-concept, objects are represented as spheres with three physical materials. Compared to a naive baseline implementation, our optimisation can achieve up to $22.69\times$ times speedup.

2. BACKGROUND

Ray tracing. *Ray tracing* [7] is a family of algorithms that generates photo-realistic images from a 3D scene description and camera location. Rays are shot from the camera center, pass through each individual pixel, and interact with the scene through multiple bounces to obtain a color value. The color is written back to the image pixel where the ray starts from.

Path tracing. The most widely-used ray tracing algorithm is *path tracing* [8], which draws multiple Monte Carlo ray samples for each pixel and scatter the ray in the scene for several times. The energy multiplier contained within each ray is adjusted by physical properties of the surface.

Path tracing can generate accurate global illumination results, but at the cost of increasing number of ray samples. For tricky light transport cases which involve viewing or illuminating through specular or dielectric light paths such as caustics, path tracing renders rather slowly due to the high variances of Monte Carlo estimation under these conditions.

Photon mapping. *Photon mapping* [9] is a two-pass rendering algorithm that aims to solve the issue by searching for light contributions starting from both the camera and the light source.

SPPM [2] is an extension to photon mapping which im-

proves distributive effects such as depth-of-field and motion blur. In the first pass, one ray per pixel shoots from the camera and accumulates direct illuminations. When the ray hits a non-specular surface, it is terminated and the position and energy multiplier is recorded. Such a point in the scene viewed by the camera is called *visible point*. In the second pass, photons shoot from random picked light sources at random angles. Whenever the photon hits a surface, its energy is contributed to nearby visible points whose distance is smaller than the photon's radius. The photon path terminates when its energy decreases below some threshold or number of bounces exceeds a maximum depth.

The two-pass procedure above is repeated for several iterations to produce a high-quality low-variance result. At the end of each iteration, radius of the photons is consolidated according to some update rules to converge the algorithm. The pseudo code of SPPM is illustrated in Algorithm 1 in the appendix.

Querying nearby visible points in the photon pass requires marching through all the visible points, leading to an unacceptable runtime. A spatial hash table is thus used to accelerate this process.

Constraints. Several constraints were imposed on implementation and scene setup. The only type of geometry permitted was spheres. Only three basic types of materials, specular, diffuse, and dielectric, were included. Spatial hashing was used to accelerate the search of nearby visible points. Single-precision floating-points were used throughout the project, since the difference between single- and double- precision is minimal in ray-tracing algorithms [10]. Several static scenes with different complexities were hardcoded instead of considering dynamic scenes. Further details are provided in 3.1.

Input. The algorithm has 5 different input parameters: the number of pixels, p; the number of SPPM iterations, n; the maximum depth of the rays, d; the number of photons in each iteration, l; and the scene S. Let $|S|$ denote the number of objects in the scene.

Cost analysis. Due to the stochastic nature of the algorithm, the exact cost cannot be determined. However, one can calculate the computational complexity for the various stages of the computation. Our brutal force implementation of ray-scene intersection has a complexity of $|S|$. In each iteration, the camera pass has a complexity upper bounded by $O(dp|S|)$, the building of the lookup table has a complexity of $O(p)$, the photon pass has a complexity of $O(ldp|S|)$, and the consolidation pass has a complexity of $O(p)$. Majority of the complexity lies in the computation of the photon pass, since the search of neighbouring visible points is expensive. The use of a spatial hash table only reduces the constant multiplier of the complexity.

The complexity was empirically verified by varying the input parameters and the results are seen in B. This supports the approximate linearity between the runtime and its dependent parameters in our analysis.

3. OPTIMISATION STAGES

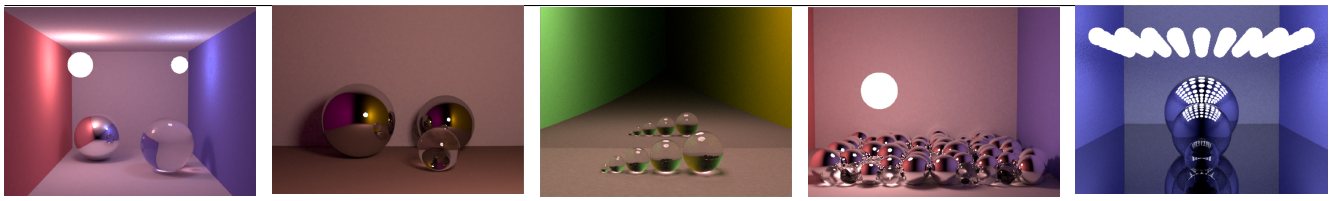
The optimisation is composed of six different stages, excluding two attempts that failed. The first two stages inlined various functions and changed the data structure of the bins of the spatial hash table. Generic arrays were also replaced with type-specific ones. 'rand' was then replaced with a new generator vectorised. An attempt to vectorise vector math operations in a horizontal layout failed, and the next stage overhauled the entire codebase to vectorise every operation in a vertical fashion. The last two stages aimed to optimise data layout and reduce the memory transfers. Attempts to sort or block the hash table failed. An autotuning framework was also created.

To identify bottlenecks and potential optimisations, Intel's Vtune and Cachegrind from Valgrind were used [11, 12]. Vtune performs detailed analysis on microarchitectural pressure. It measures the percentage of pipeline slots that are retired (successfully executed), front-end bound (instruction decoding overhead), and back-end bound. Back-end bound can be further identified as memory-bound and core-bound (computation as bottleneck). Memory and vectorisation utilisation could also be recorded in Vtune, as well as the bad speculation metric (cancelled pipeline slots due to incorrect speculations when branching). Cachegrind performs hotspot and cache analysis. It provides the number of instructions, cache reads, and cache writes on a line-by-line basis. This allows the identification of poorly performing regions of code, which are subsequently improved. Intel's Advisor was also used to create roofline plots [13].

3.1. Baseline

The baseline is implemented naively, following the outline provided in Algorithm 1 in Appendix. We use 32-bit floating point number as the program's data type.Vectors are defined as `structs` containing 3 `floats`. A corresponding vector math library is also implemented. The random function used is C's inbuilt `rand`, which is converted to a floating point uniformly distributed in the range of 0 to 1. All data structures are implemented in array of structures format in an object-oriented fashion.

SPPM implementation composed of a large outer loop controlled by the number of iterations n. Each iteration contains four passes: camera pass, construction of the spatial hashtable, photon pass, and consolidation pass. Each pass is computed on a pixel-by-pixel basis, except for the photon pass, which is photon-by-photon. For a given pixel in the camera pass, associated ray and intersection data are stored in a data array. The spatial hashtable is built upon this data



Cornell Box | Large Box | Mirror Box | Random Box | Surgery Box

Fig. 1. Test scenes rendered with SPPM.

structure. The hash table is implemented as a fixed length array where each bin represents an area in space. The physical size of each bin is calculated as a multiple of the maximum radius of all pixels, as each pixel maintained its own radius for photon-searching and updated it in the consolidate pass of each iteration. Each bin stores a single linked list containing the indices of all intersection points that lie within this area from the aforementioned data array. During the photon pass, when a photon intersects a surface, the corresponding location/bin in the hash table is found and all the stored pixel indices are iterated over to add the photon's contribution to that pixel. The final consolidation pass updates several properties including the search radius and accumulated light radiance based on the mathematical derivation in the algorithm.

To ease testing, 5 different scenes were fabricated: "Cornell Box", "Large Box", "Mirror Box", "Random Box", and "Surgery Box". All of them are inspired by the Cornell Box scene from smallpt. Fig. 2 demonstrates them in detail.

3.2. Stage 1

Hotspot analysis of the baseline implementation is shown in Table 1. A significant portion of time is spent on vector computations. Currently, they are stand-alone functions and are compiled in separated compilation unit. The functions are called potentially millions of times, and the overhead associated with call stack maintenance are unnecessary. Furthermore, analysis of microarchitecture revealed the program is severely front-end bound, taking up 48.7% of all pipeline slots. Close examination revealed that this is caused by high front-end latency, specifically decoding stream buffer switches, in the vector math library. This is likely due to poor code layout since different compilation units were used. Thus, all vector functions are inlined by declaring them as `static inline`. This reduced the front-end bound to 3.8%.

Significant time (22.82 %) was also spent on the photon pass. Recall that traversal of a linked list is needed to iterate over the indices in the hash table bins. The line responsible had a level 1 (L1) cache miss rate of 98.40% and a last level (LL) cache miss rate of 63.71%. It is clear that the incontiguous storage of linked list nodes are extremely unfriendly to cache alignment. Furthermore, node addresses are also stored, increasing memory overhead. To alleviate both issues, a dynamically sized array was stored at each bin instead of a linked list. This ensures that cache coherency is maintained and it has little storage overhead. The cache miss rates for L1 and LL cache were reduced to 6.46% and 4.04% respectively after changing the hashtable data structure.

Percentage runtime	function name
22.82%	sppm_photon_pass_photon
8.78%	mesh_intersect
8.24%	vv_equal
7.93%	v_norm_sqr
7.70%	vv_sub
5.70%	vv_dot

Table 1. Hotspot analysis of baseline. Functions with a runtime of less than 5% are omitted.

One issue still persists: array resizing requires copying all the memory contents. To minimise this overhead, instead of `malloc`ing a hashtable at the start of each iteration and `free`ing it after, the space is only allocated at the start of the program. At the start of each iteration, the size counter of each bin is set to 0 while the capacity does not change. This significantly reduces the number of resizing operations, and subsequently, memory copies.

3.3. Stage 2

Inspired by the effectiveness of inlining vector math operations in the previous stage, short but frequently called functions were also inlined with `static inline`. This includes all the functions related to array access and array setting, as well as ray generation and material-dependent code blocks for evaluating reflectance.

The baseline implementation only implemented generic arrays. To set a value, the address and size of the value is passed in, and `memcpy` was used to insert into the array. Similarly, to get a value, the address within the array is returned, and the user needs to dereference the pointer. This convoluted method prohibits further compiler optimisations and calls `memcpy` for short segments of memory.

Since most of the arrays stores primitive/simple types such as `float` and `Vectors`, type specific arrays that returned the values directly were created.

Static analysis of the code also revealed redundant calculations in branching conditions. Previously, when building the lookup table for pixel data, an if-statement for deciding whether the ray has zero attenuation is performed for each pixel. This is needed twice, once in calculating the bin size, and another in allocating them to the corresponding bin. The boolean result was then stored in an array in the first pass, which was then read in the second pass. This approach aimed to circumvent redundant computation and avoid unnecessary memory reads involved in retrieving and calculating the attenuation property.

With the above optimisations, the proportion of backend bound pipeline slots was reduced from 61.3% to 53.8%.

3.4. Stage 3

A new set of hotspot analysis was performed, and it was found that built-in function `rand` and functions responsible for the generation of random numbers took up 4.28% of the total number of instructions. This was higher than expected, since generating a uniform distribution is often thought of as a "low cost" operation.

Also, as random numbers are relatively independent, it is possible to paralyze their generation with SIMD. Unfortunately, `rand` is inbuilt, and vectorisation requires a reimplementation. Taking the above two factors into consideration, a new generator utilizing the xorshift algorithm was implemented [14]. Xorshift is a family of random number generators that avoid using sparse polynomials to generate uniform random integers. A simple implementation of xorshift maintains a 32-bit state and only requires 3 bitwise shift operations and 3 bitwise xor operations, which is very computationally efficient. Xorshift is robust in being able to pass BigCrush, an empirical test suite that tests for randomness [15]. With a 32-bit state, the generator has a period of $2^{32} - 1$, sufficient for our use case.

After replacing `rand` with a xorshift, vectorisation is now possible. Eight 32-bit random integers is generated each time and then stored. When a random number is needed, one of the eight stored values is read and returned. If all eight values are exhausted, a new group of eight numbers are regenerated. Unfortunately, naively implementing algorithm this way results in lower performance compared to the non-SIMD version, likely due to the high latency of the SIMD instructions. The latency and throughput information of the involved functions on the critical path are shown in Table 2 with information obtained from Intel's website [16]. To perform latency hiding, unrolling is needed, and the optimal number of times of unrolling is $7/0.5 = 14$ times. In isolated experiments, this unrolling factor did produce the greatest speedup. However, when integrated into the SPPM implementation, this is no longer the case. Empirically, it is determined that 8 is the optimal unrolling factor for the SPPM algorithm. This is likely due to the lower pressure on the SIMD registers, since these registers are also needed for floating point instructions. Thus, the lower register use of lower unrolling factors is desirable.

Function	Latency	Throughput (CPI)
load	7	0.5
slli / srli	1	0.5
xor	1	0.33
store	5	1

Table 2. Latency and throughput of functions used in xorshift with Skylake architecture. Note all functions are for 32-bit integers in `__m256i`.

3.5. Failed attempt 1

Further analysis of the program revealed that the vector math library was a major bottleneck. Various functions add up to 32.02% of the total number of instructions. A possible solution is to horizontally vectorise the math library. For each element-wise operation, the entire vector is loaded horizontally to one single `__m128` register. Note that in this implementation each vector includes an unused entry. Element-wise operations are then performed using the corresponding vector instruction to realize vector arithmetic. The result is then stored back to the original vector. Operations that perform dimension reductions, such as dot product or norm computation, cannot be implemented in this way and therefore keep their original implementations.

On Apple M1 processors, a speedup of 15% was achieved with the parallelization of vector operations. Interestingly, on Skylake processors, the program was slowed by 4%. The latter is likely due to the high latency of `store` and `load` instructions on Skylake processors. With 5 and 7 cycles respectively, the total overhead is 12 cycles for every mathematical vector operation. Such increase in overhead does not worth the speedup.

With horizontal layout of vectors, not only vector reductions operations cannot be effectively computed, but also extending parallelization to the rest of SPPM algorithm is hindered. The unused entry in register also leads to a waste of memory. As a result, the attempt on horizontal vectorisations was not further pursued.

3.6. Stage 4

Since the previous optimisation failed, the bottleneck remains to be the vector math library. Another major bottleneck is the various array libraries, the computational cost

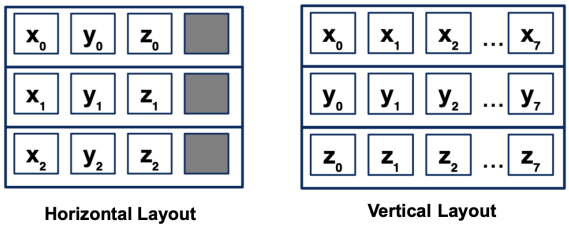


Fig. 2. The left figure displays a horizontal layout with one SIMD `__m128` variable. The right demonstrates a vertical layout with 3 SIMD `__m256` variables packing 8 vectors in total.

of which sum up to 26.16 %. Note that these values include the latency induced by cache misses and memory transfer.

The best option to alleviate these issues is through vectorisation. With the failure of horizontal vectorisation, the only choice is vertical vectorisation. This allows the use of `__m256` instead of `__m128` and can utilise the entire register without wasting space. A comparison of both layouts can be found in Fig. 2. With such representation, the whole SPPM algorithm was rewritten with SIMD instructions.

As a preceding step for whole algorithm vectorisation, the layout of key data structures were changed from array of structures (AoS) to structure of arrays (SoA), simplifying loads and reads with SIMD instructions. This approach alone also brings the benefit of better spatial coherency, as elements of the same property are stored contiguously in memory.

Previously, a pointer to the mesh was stored during the intersection computation to enable retrieval of mesh-related properties in subsequent calculations. Since indirection through pointer is difficult to resolve with SIMD, all the required fields of the mesh are now stored directly in the return value struct. Though this stores redundant copies of properties and consumes more space, it increases coherency of cache.

Adapting the algorithm to SIMD also requires some third party libraries. Native SIMD implementations of trigonometric math functions, unfortunately, are only available in Intel's Short Vector Math Library Operations (SVML). This is partially implemented with gcc, but it is operating system dependent. As a result, avxmathfun's slower implementation was used if SVML is not available [17].

During the camera pass, eights rays are shot at the same time, each represented by one float in the `__m256` register. Since each ray will likely intersect different geometries with different material properties and therefore leading to varying bouncing behaviour, branching is necessary. To resolve this, all possible code paths are executed, and results are blended together afterwards using Boolean masks. Similarly, the hash table is built by processing eight pixels at the same time.

Regarding the photon pass, eight photons are first processed at the same time. Recall that for each photon, the program iterates over the entries stored in the hash table at the corresponding intersection location. Since the number of hash table entries may vary drastically across different photons, batch processing will no longer be efficient. Instead, the program processes the iteration logic on a photon-by-photon basis, but still utilize SIMD to accelerate the spatial hash table entries traversal. For one given photon, eight entries in its iteration are loaded simultaneously.

For each photon and one of its pixel index, the required information is gathered using the inbuilt `i32gather` instruction. The results are scattered back when the computation for each photon finishes. Since our CPU does not support AVX512, the inbuilt scatter operation could not be used, and our own SIMD scatter logic was implemented. The register values were first stored in an array, which were then manually scattered with an unrolled for loop.

3.7. Stage 5

Interestingly, the previous optimisation stage resulted in a slow down rather than a speed up. By inspecting the microarchitecture pipeline and the roofline plot (Fig. 11), we found that this is due to the increase in the proportion of backend-bound pipeline slots. A comparison of pipeline slots percentage across multiple optimization stages can be seen in Table 3. From timing information, while camera pass sped up significantly by a factor of 3.36×, photon pass slowed down by 53.6%. The culprit of slowing down is the gathering and scattering operations at the start and end of iteration for each photon. Since the data structure has previously been converted to the SoA layout for convenience, gathering 16 elements with the same index from different arrays would imply up to 16 cache misses. As each photon gathers value from a group of different pixels, it is highly likely that there is little data reuse and proximity within this process, resulting in almost no temporal or spatial cache locality.

Type	Stage 3	Stage 4	Stage 5	Stage 6
Retiring	29.84%	16.59%	21.00%	31.45%
Front-end	2.48%	4.75%	3.85%	6.13%
Memory	32.94%	55.01%	42.78%	22.50%
Core	22.74%	16.52%	19.12%	22.48%
BS.	11.99%	7.14%	13.26%	17.44%

Table 3. Percentage of pipeline slots spent on each section for optimisation stages 3, 4, 5, and 6. BS: Bad Speculation. Definition are found at the start of Section 3.

To alleviate this issue, reduction in the number of cache misses is necessary. The 16 read elements are all floats, so it is possible to fit all information into one cache line, reducing the number of cache misses by a factor of 16. Thus, instead

of a strict SoA layout, a hybrid layout of SoA and AoS is proposed. The data necessary for each gather operations are grouped together in a struct and stored in a cache aligned array. This way, during the gather operation, only one cache line read is needed. A graphical explanation of this layout change can be found in Fig. 3. As the 16 read elements are previously in a horizontal (SoA) format, to convert them into the vertical layout (AoS) two eight by eight transpositions were performed. Similarly, a hybrid layout is also used for scattering, storing only the four output variables in an AoS layout by transpose and reducing the number of cache misses by a factor of four.

The same idea of changing some data back to AoS was also applied to the scene data structure and benefit scene-related computations including ray intersection logic and estimate direct lighting. The data properties of each mesh object that are accessed together in scene-related functions were store contiguously in memory to increase memory coherency and eliminate cache misses.

3.8. Stage 6

After performing the above optimisations, the bottleneck still lies in the photon pass. Inspecting the pipeline slots proportions presented in Table 3, it is clear that memory bound is still severe in stage 5. Furthermore, transpose operations appear to put heavy pressure on port 0, leading to high core bound as well. It is now necessary to eliminate the number of elements reads for each photon, which also reduces the port pressure from transpose operations. This should reduce the backend bound significantly.

More specifically, out of the 16 elements that are read in stage 5, 3 are only used to compute the sign, a Boolean variable, of the dot product between incident direction and normal. 3 elements of the attenuation vector are also only for checking whether these values are all 0. The former can be replaced by precomputation, and the latter can simply be removed with the correctness of the program guaranteed by the branching optimization done in stage 2. The remaining 11 variables are further reduced by noticing 3 are mesh color attribute. Since the number of meshes is small (no more than 100), it is reasonable to assume that the entire mesh data will fit into cache. Thus, by storing the mesh index instead of mesh color, only 9 elements needs to be gathered. This number is further reduced to 8 by realising the mesh index is always positive, thus the highest bit can be used to store the Boolean variable of dot product's sign. Though the number of cache misses remains the same, the number of memory loads and transposes is reduced, leading to further acceleration. Table 4 provides a clear comparison of the variables gathered in the photon pass before and after the optimization, in accordance with the variable pruning discussed above.

Size (bytes)	Stage 5	Stage 6
3	attenuation	-
1	radius	radius
3	intersect point	intersect point
3	intersect normal	intersect normal
3	incident direction	-
3	mesh color	-
1	-	Boolean + mesh index
Total	16	8

Table 4. The size and values of data gathered for each pixel index in hash table traversal in stage 5 and 6. The last variable stores the mesh index with the Boolean result represented by the highest bit.

3.9. Failed attempt 2

Further analysis revealed that the photon pass is still severely memory bound, so two methods to reduce the number of cache misses are attempted.

The first method aimed to sort within each array in the hash table according to the mesh type. Ideally, this would avoid repeatedly loading the 3 color attributes for the same mesh, which should at least save the 7 cycles each. This does come with the additional overhead of creating multiple hash tables and iterating over them. Unfortunately, such iteration over the hash table caused significantly higher bad branch speculation. Furthermore, since each bin of the hash table is a dynamically sized array, the same bin of different tables are likely stored in different pages. This caused increased translation lookaside buffer misses and the method was not pursued.

The next attempt to reduce cache misses attempted blocking. Recall that the elements required for the gather operations are stored contiguously in the data array. The hash table could be blocked into multiple hash tables such that each hash table only contains entries that are contiguous in the data array and that segment of the data array fits into cache. Ideally, this should reduce the amount of non-compulsory cache misses to zero when gathering. Unfortunately, this requires a significant number of hash tables, which suffered from the same issue encountered above.

3.10. Autotuning

Some hyperparameters exist in our program, such as the size multiplier for the hash table grid size and the initial array size of the spatial hash table bins. As a proof-of-concept, an autotuning framework was created to tune these parameters for a specific scene and a static group of input parameters. This framework utilizes Bayesian optimisation to explore optimal parameter values within a contiguous domain.

Bayesian optimisation was ran with a Matern and white noise kernel. The acquisition function was a weighted av-

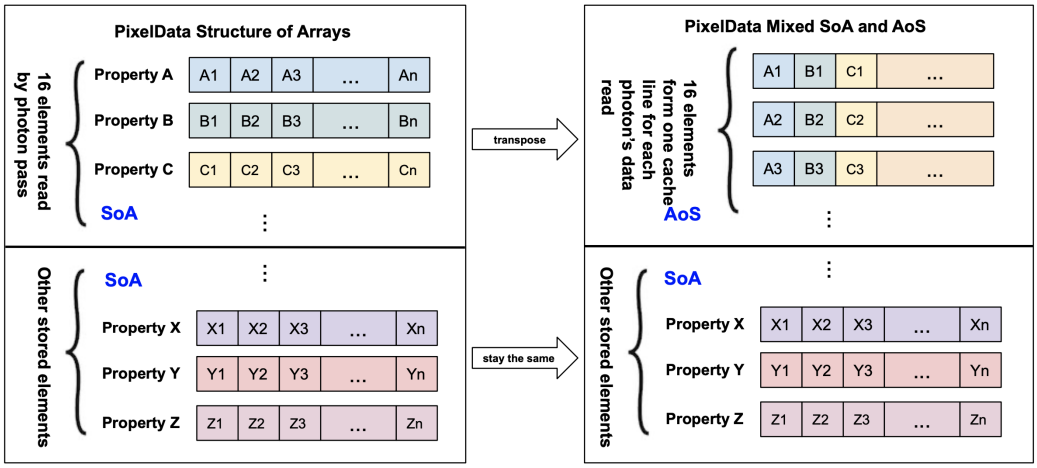


Fig. 3. In stage 5, a hybrid of SoA and AoS replaces our SoA layout. The data loaded in the photon pass was changed from SoA in stage 4 to AoS. By storing the read data together, the number of cache misses in reading 16 elements for each photon is reduced from 16 to 1. A similar transpose was performed on the 4 variables for photon pass scatter.

erage of lower confidence bound, expected improvement, and probability of improvement. Taking into account random noise, an improvement up to 7.42%, scene and input parameters dependent, was observed compared to the default hyperparameters. It may be possible to tune the hyperparameters on the same scene with fewer photons or iterations, but this possibility was not examined. Although the improvement is small, do note that it essentially comes without modifying the code.

Component	Specifications
CPU	Intel Core i9-10850K Processor @ 3.60 GHz
Microarchitecture	Skylake
L1 cache (total)	64 KB
L2 cache	256 KB
L3 cache	20 MB
DRAM	32 GB

Table 5. Specifications of remote testbed. Note that L1 and L2 cache are on a per core basis.

4. EXPERIMENTAL RESULTS

Various experiments were performed to measure the performance gains of each optimisation stage. Different input parameters and scene combinations were tested and the performance of the optimised implemented was compared to the peak theoretical performance. Results were computed without autotuning using default hyperparameters.

Experimental setup. The experiments were conducted with hardware specifications shown in Table 5. Botth for reproducibility. Both turbo-boost and hyperthreading were turned off and caches were cleared prior to each run. The compiler used for clang 14.0.0, with flags `-O3`, `-ffast-math`, and `-march=native`. Without vectorisation, it has a peak performance of 4 flops/cycle. With vectorisation for single-precision floating-points, 32 flops of cycle was reached. Results were obtained using Intel's Vtune and Advisor.

The default parameters are as follows: image size $p = 512\times384$, number of iterations $n = 6$, ray max depth $d = 5$, number of photons per iteration $l = 200000$.

For each experiment, only one parameter was modified. Image sizes ranged from 32×24 to 1024×768 with increments by a power of 2. Number of iterations ranged from 2 to 64 with increments by a power of 2. Number of photons per iteration ranged from 12500 to 800000 with increments by a power of 2. The ray max depth was kept constant.

Results. The performance gain of each optimisation stage was evaluated by iterating over the possible input parameters, shown in 4. For space limitation, only the result of changing the number of iterations is shown. The remaining results are found in Appendix B.

The various optimisation stages gradually increased the relatively speedup compared to the baseline. The largest jumps where caused by optimisation stage 1, stage 5, and stage 6. Note that these stages focused on optimising for memory transfers, suggesting the SPPM algorithm is inherently memory bound. The improvement is greatest in complex scenes "random" and "surgery", where a speedup of $17.06\times$ and $11.03\times$ was observed. This is likely due to the increased complexity of the spatial hash table, which was focused on in the later optimisation stages. As the number of iterations increases, the relative speedup in the last stage increases. In higher iterations, the radius becomes smaller

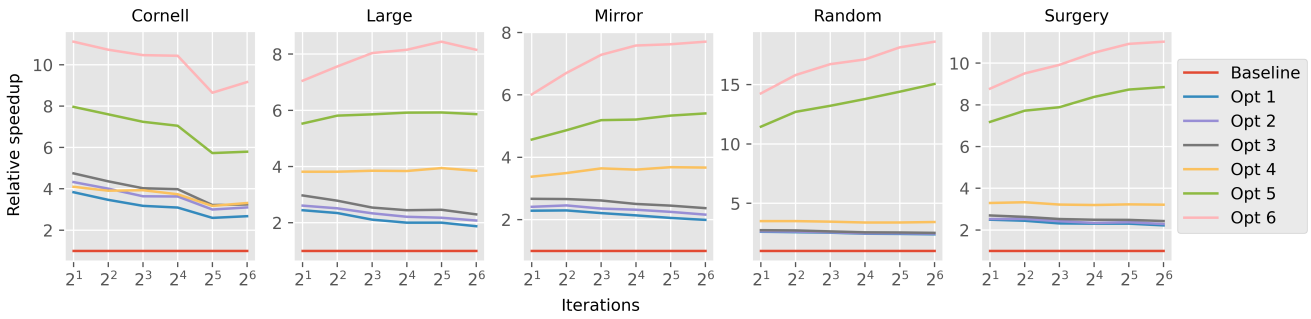


Fig. 4. Performance gain on Intel i9-10850k of each optimisation stage while sweeping the number of iterations.

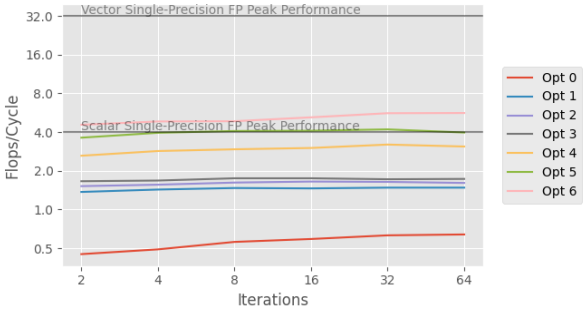


Fig. 5. Flops per Cycle on Intel i9-10850k of each optimisation stage while sweeping the number of iterations. The theoretical maximum performance for vector and scalar single-precision floating-points is also labelled.

and the hash table more sparse, making memory transfers more dominant in affecting program efficiency.

Similarly, the final stage achieved a high speedup when changing the number of photons and the size of the image. Note that the speedup also increased as the number of photons increased, since the spatial hash table is used more heavily. As the size of the image increases, the speedup decreases. This is likely due to the larger proportion spent in the camera pass, which was not identified as the bottleneck and gained significant speed up by parallelization with SIMD. Overall, on the tested scenes and input parameters, a maximum speedup of $22.69\times$ was observed. However, given the trend, it is reasonable to assume that a greater speedup can be observed on complex scenes, high number of photons, and a large number of iterations. This setup is commonly used to obtain images with low noise, suggesting the algorithm generalises well to the commonly used scenarios.

Figure 5 shows the number of Flops per cycle executed for the "Cornell" scene with varying number of iterations. Recall that the first three stages and baseline were implemented without SIMD intrinsics, though the compiler may select to use them. After manually using SIMD intrinsics, there was a 72.02% increase in the number of flops per cycle. Note that this does not translate to reduced runtime, since redundant computations were made during ray masking. The final iteration obtained a maximum flops per cycle of 5.63, which is 17.6% of the theoretical peak maximum. Overall, a $9.56\times$ increase in computation was observed. Most notably, this is higher than the theoretical scalar maximum of 4, suggesting that the final optimised algorithm is faster than any scalar algorithm possible. Increasing the iteration number did not affect the flops per cycle significantly. This is because cache usage is independent of the number of iterations. A small increase is observed when the number of iterations is small. This is likely to due the setup overhead being significant compared to the total computation.

This value is far from the maximum theoretical maximum, and roofline plots were created and shown at 12. All the optimisations stages are still relatively memory bound, since the operational intensity is smaller than the intersection point of memory bound and computation bound.

5. CONCLUSIONS

This report presented an accelerated SPPM algorithm for physically-based rendering. The optimised algorithm was able to achieve up to a $22.69\times$ speedup over the baseline implementation. It is also faster than every scalar implementation possible, with a flops per cycle of 5.63. The increased performance was ubiquitous across a number of scenes and parameters, suggesting from generalisability. The biggest bottleneck was the traversal of the spatial hash table during the photon pass, whose impact was reduced by vectorisation with SIMD intrinsics and clever reorganisation of data structures. An autotuning framework was also developed that could further accelerate the program but is unfortunately scene and input dependent. A potential direction to further accelerate the implementation is to replace the spatial hash table with alternate data structures, e.g., a k-D tree, that can accelerate the neighbour search.

6. CONTRIBUTIONS OF TEAM MEMBERS

Fengshi. Worked on the baseline implementation of the rendering framework, and a path tracer for reference. Worked on bug fixing of the SPPM sequential and SIMD version to ensure their physical correctness with Zijun. Worked on optimization attempt for horizontal layout SIMD 3.5. Worked on cost analysis plots B.

Hongyu. Worked on baseline implementation of vector operations and scene construction with Fengshi. Worked on performance benchmarks and analysis with Kehan. Worked on reducing unnecessary branching in stage 2. Worked on structural similarity, roofline plots, and peak performance analysis.

Kehan. Worked on the material and scene-ray interaction part of the baseline rendering framework. Worked on type-specific arrays and the inlining of its functions in stage 2. Worked on implementing and debugging the SIMD parallelization of SPPM algorithm in stage 4 with Zijun, to an extent that the generated image depicts correct scene geometry and the lighting effect is visually coherent. Discussed with Zijun on improvements to be made in stage 5 and 6.

Zijun. Worked on the baseline implementation of the SPPM algorithm. Worked on optimisation stage 1 fully. Worked on optimisation stage 3 fully, researching random number generators and implemented a xorshift algorithm-based generator both in scalar and SIMD. Computed unrolling factor and experimentally tested optimal factor. Worked on a buggy SIMD implementation of the whole SPPM algorithm in stage 4 together with Kehan, later fixed by Fengshi. Discussed potential of stage 5 with Kehan and implemented fully, changing data layout for gathering and scattering operations in photon pass to reduce cache misses. Discussed potential of stage 6 with Kehan and implemented fully, eliminating the number of variables gathered for each photon from 16 to 8 floats through precomputation. Worked on failed optimisation attempt 2 fully. Worked on the autotuning framework based on Bayesian optimisation fully. Ideas mentioned above were created by discovered by profiling and hotspot analysis, all of which performed independently.

7. REFERENCES

[1] Eric Lafortune, "Mathematical models and monte carlo algorithms for physically based rendering," *Department of Computer Science, Faculty of Engineering, Katholieke Universiteit Leuven*, vol. 20, pp. 74–79, 1996.

[2] Toshiya Hachisuka and Henrik Wann Jensen, "Stochastic progressive photon mapping," in *ACM SIGGRAPH Asia 2009 papers*, pp. 1–8. 2009.

[3] Steven G Parker, James Bigler, Andreas Dietrich, Heiko Friedrich, Jared Hoberock, David Luebke, David McAllister, Morgan McGuire, Keith Morley, Austin Robison, et al., "Optix: a general purpose ray tracing engine," *Acm transactions on graphics (tog)*, vol. 29, no. 4, pp. 1–13, 2010.

[4] Merlin Nimier-David, Delio Vicini, Tizian Zeltner, and Wenzel Jakob, "Mitsuba 2: A retargetable forward and inverse renderer," *ACM Transactions on Graphics (TOG)*, vol. 38, no. 6, pp. 1–17, 2019.

[5] Ingo Wald, Sven Woop, Carsten Benthin, Gregory S Johnson, and Manfred Ernst, "Embree: a kernel framework for efficient cpu ray tracing," *ACM Transactions on Graphics (TOG)*, vol. 33, no. 4, pp. 1–8, 2014.

[6] Wenzel Jakob, Sébastien Speierer, Nicolas Roussel, and Delio Vicini, "Dr. jit: A just-in-time compiler for differentiable rendering," *arXiv preprint arXiv:2202.01284*, 2022.

[7] Arthur Appel, "Some techniques for shading machine renderings of solids," in *Proceedings of the April 30–May 2, 1968, spring joint computer conference*, 1968, pp. 37–45.

[8] James T Kajiya, "The rendering equation," in *Proceedings of the 13th annual conference on Computer graphics and interactive techniques*, 1986, pp. 143–150.

[9] Henrik Wann Jensen, *Realistic image synthesis using photon mapping*, vol. 364, Ak Peters Natick, 2001.

[10] H. Dammertz and A. Keller, "Improving ray tracing precision by object space intersection computation," in *2006 IEEE Symposium on Interactive Ray Tracing*, 2006, pp. 25–31.

[11] Intel, "Vtune," 2022.

[12] Seward et al., "Valgrind," 2022.

[13] Intel, "Advisor," 2022.

[14] George Marsaglia, "Xorshift rngs," *Journal of Statistical Software*, vol. 8, no. 14, pp. 1–6, 2003.

[15] Sebastiano Vigna, "An experimental exploration of marsaglia's xorshift generators, scrambled," *ACM Trans. Math. Softw.*, vol. 42, no. 4, jun 2016.

[16] "Intel intrinsics guide," `https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html`, Accessed: 2022-06-24.

[17] Julien Pommier, "Avx math fun," 2012.

A. SPPM ALGORITHM

Algorithm 1: Overview of SPPM

```
InitializeDataStructure();
for each iteration do
    CameraPass();
    BuildLookUpTable();
    PhotonPass();
    Consolidate();
StorePixelValue();
Function CameraPass() is
    for each pixel do
        generateRay();
        for bounce time < max ray depth do
            hit = intersectRayWithScene();
            if hit surface = DIFFUSE then
                computeDirectLighting();
                record visible point;
                return;
            bounceRay();
Function BuildLookUpTable() is
    for each pixel do
        findGridIndicesByHashing();
        putIntersectionPointIntoEachGrid();
Function PhotonPass() is
    for each photon do
        emitPhotonFromLightSource();
        for bounce time < max ray depth do
            hit = intersectRayWithScene();
            if bounce time ≥ 1 then
                findGridIndexByHashing();
                for each visible point in grid do
                    if distance(visible point, hit position) ≤ photon radius then
                        contributeToVisiblePoint();
```

B. ADDITIONAL FIGURES

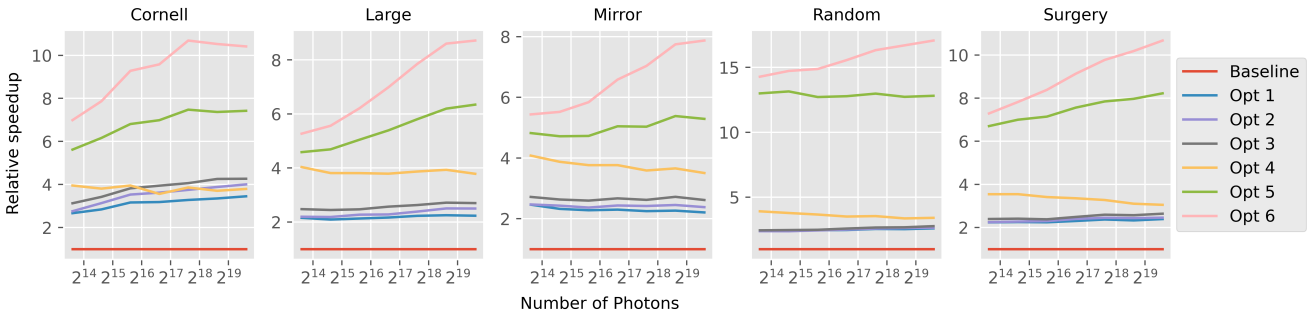


Fig. 6. Performance gain on Intel i9-10850k of each optimisation stage while sweeping the number of photons.

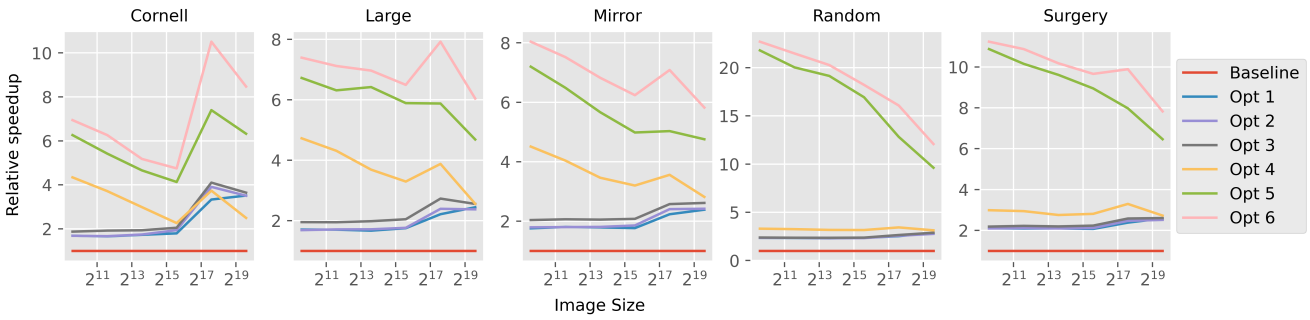


Fig. 7. Performance gain on Intel i9-10850k of each optimisation stage while sweeping the image size.

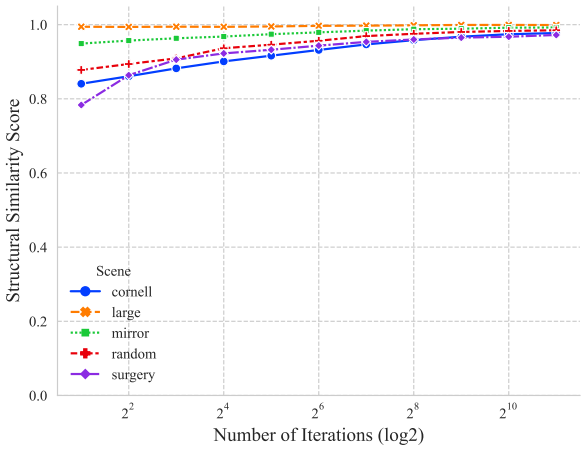


Fig. 8. Structural similarity score of the five test scenes converges as the program iterates more. The similarity between the optimisations and the baseline version converges to 1 across all test scenes as more iterations are applied. This result proves the correctness of our optimisation.

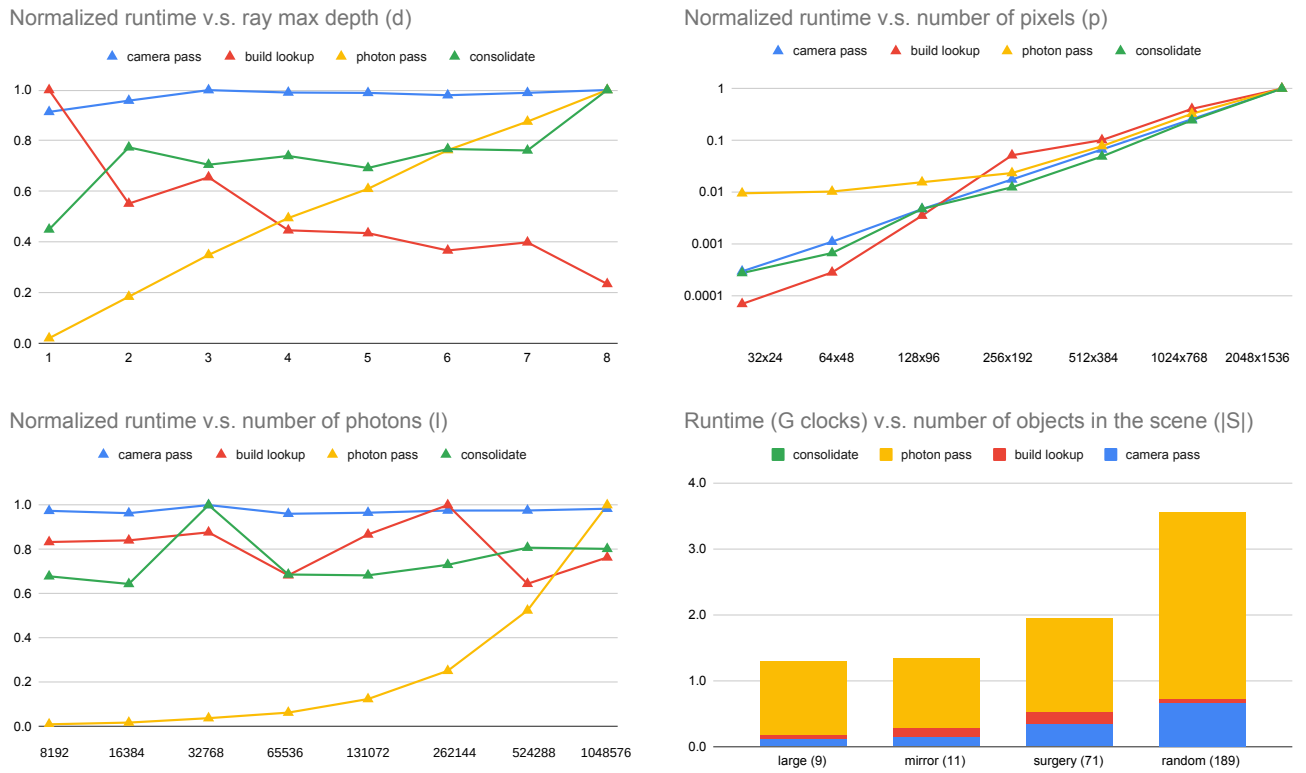


Fig. 9. Verification of our complexity analysis on 4 stages in a iteration. Normalized runtime is defined as the ratio of the runtime over the maximum runtime in that series.

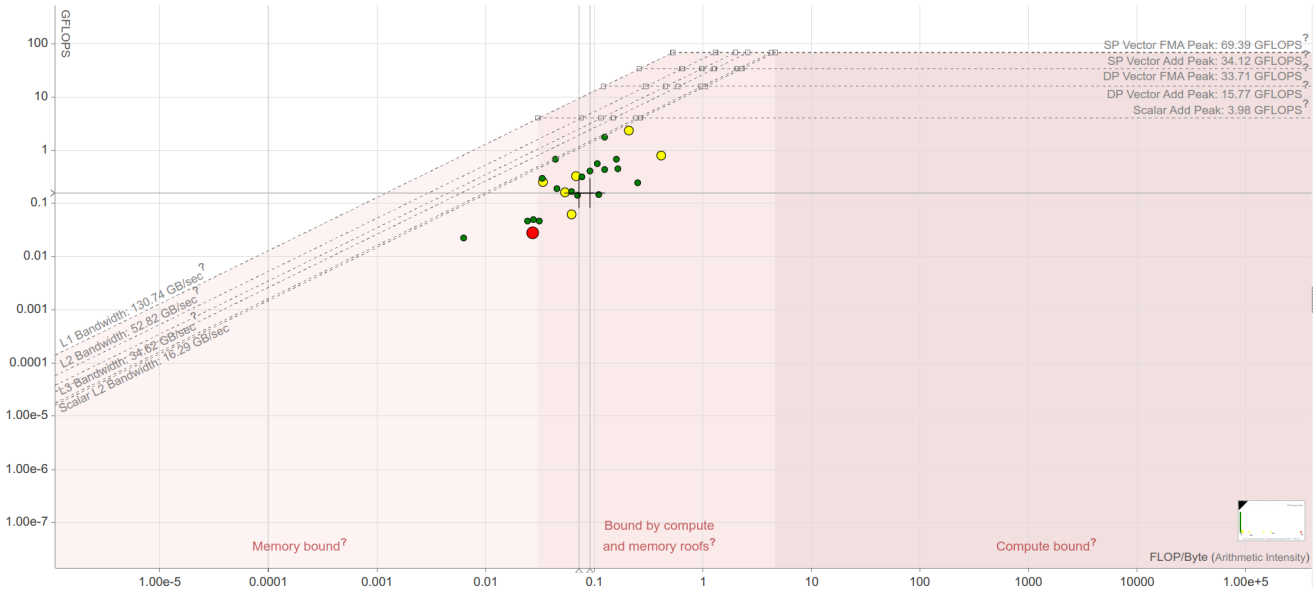


Fig. 10. Roofline plot of Stage 0. Red dot represents the cricital path.

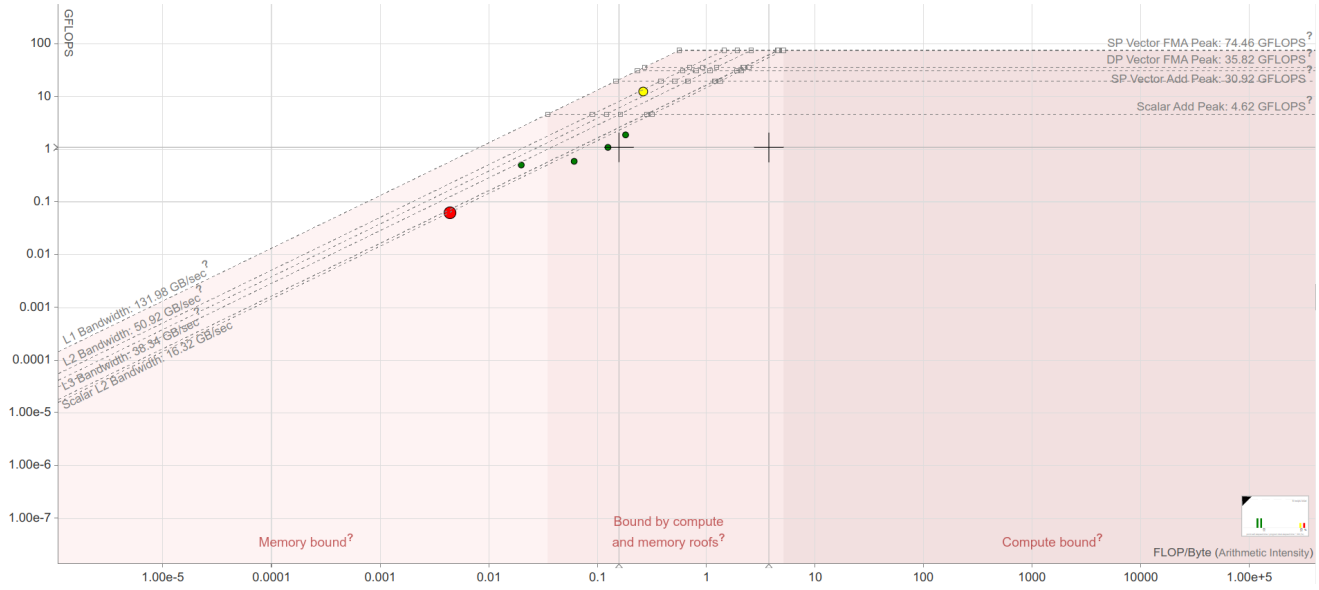


Fig. 11. Roofline plot of Stage 4. Red dot represents the cricital path.

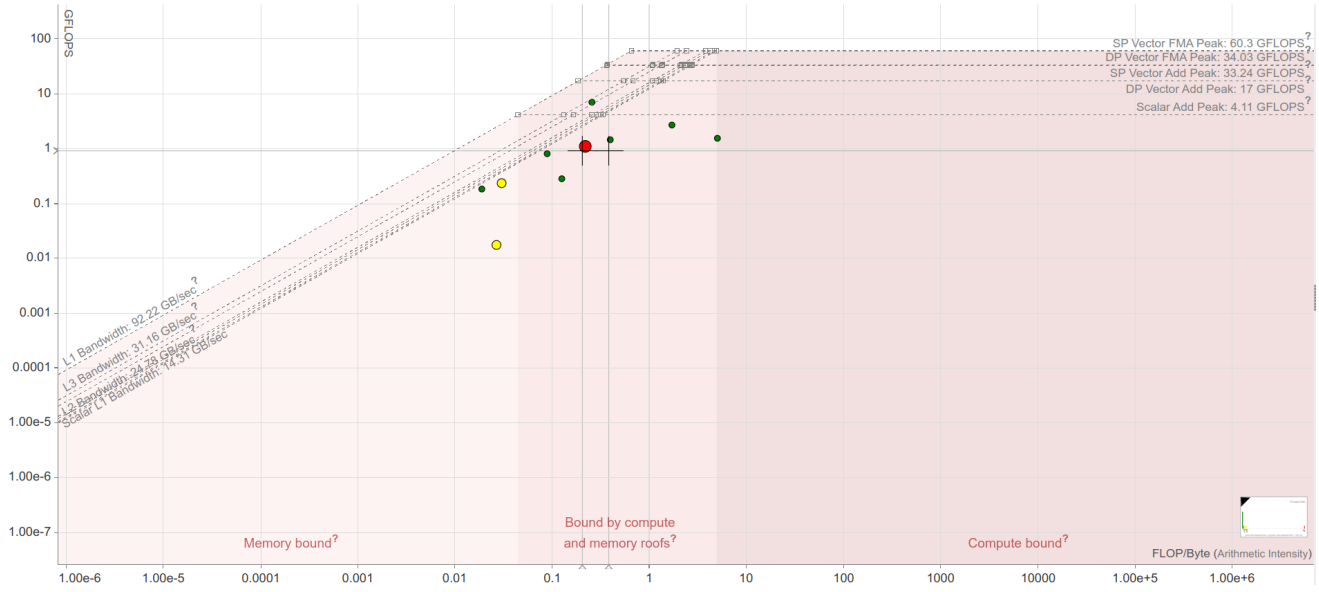


Fig. 12. Roofline plot of Stage 6. Red dot represents the cricital path.