Optimization and Vectorization

Assignment 3 Report

Students: Gianmarco Picarella (2713810) and Joel Brieger (1550942)

1. Chosen Project

We decided to work on the SafetyNet project provided by prof. Jacco on the OV course website.

2. Profiling Methodology

All our profiling experiments are carried out on a Windows 11 Laptop equipped with a Core i7-12700H microprocessor and 16GB of DDR4 ram operating at 3200mhz. Dynamic Frequency Scaling is disabled for each profiling by using a software called ThrottleStop [1] mentioned during one of the lectures. This way the CPU cannot adapt the operating frequency based on its workload and heating, thus the variance across multiple samples is minimized.

We use the Intel VTune Profiler as the main profiler for all our measurements. Specifically, we use the Visual Studio VTune plug-in, so that the iteration time (profiling-hotspot location-optimization) is minimized. The profiled exe is compiled in Release mode with Debug info and corresponds to the first project available in the solution "1. Basics". We use the hardware event-based sampling for hotspot lookup in the application: this method is the most efficient and introduces the lowest overhead possible to the application. Nonetheless, we noticed a 10FPS degradation between the profiled and non-profiled version of the application.

We standardize the duration of each run by limiting the amount of Renderer::Tick updates to a constant value called PROFILING_TICKS and set to 3000. This specific value was chosen so that the profiler has sufficient time to collect enough data. For each run, we compute the average FPS counter and its standard deviation. Also, we compute the average MRPS (million rays per second) and its standard deviation. These two metrics are used as main macro indicators of overall improvement of the application.

We also perform various micro-benchmarking measurements by enclosing the code fragment within two high resolution timers and sampling until the standard deviation of all our measurements is within our target range, that is [-25ns, +25ns]. We use boost::chrono::process_cpu_clock [2] for all the measurements because it allows us to capture only the aggregate real, user-CPU and system-CPU time in nanoseconds.

3. Our Optimizations

As first step, we profiled the entire application. As can be easily seen, functions from multiple modules are listed. The modules ntoskrnl.exe and ntdll.dll are OS related modules and are out of this optimization's scope. We started by checking each function in the hope for possible optimizations across the first 13 listed functions.

Hotspots

Analysis Configuration | Collection Log | Summary | Bottom-up | Caller/Callee | Top-down Tree | Platform

Grouping: Function / Call Stack

Function / Call Stack	CPU Time ▼	Instructions Retired	Microarchitecture Usage	Module
Tmpl8::Scene::FindNearest	68.712s	311,240,832,000	47.9%	1. basics.exe
func@0x140440d3f	51.441s	56,485,632,000	20.4%	ntoskrnl.exe
Tmpl8::Torus::Intersect	43.151s	174,720,000,000	43.2%	1. basics.exe
func@0x1800043a4	33.063s	10,964,352,000	12.3%	vcomp140.dll
Tmpl8::Cube::Intersect	28.380s	175,932,288,000	55.3%	1. basics.exe
Tmpl8::Plane::GetAlbedo	25.239s	125,018,880,000	45.6%	1. basics.exe
NtDelayExecution	22.943s	80,650,752,000	45.8%	ntdll.dll
Tmpl8::Quad::Intersect	22.023s	37,855,104,000	29.8%	1. basics.exe
func@0x14023fc59	18.865s	14,568,960,000	14.9%	ntoskrnl.exe
Tmpl8::Renderer::Tickomp1	15.430s	31,613,568,000	25.6%	1. basics.exe
Tmpl8::Scene::GetAlbedo	14.124s	48,050,688,000	41.5%	1. basics.exe
cbrt	12.836s	22,708,224,000	21.1%	ucrtbase.dll
Tmpl8::Scene::GetNormal	9.084s	43,723,008,000	48.3%	1. basics.exe
ORD_6	9.072s	19,455,744,000	24.8%	ntoskrnl.exe
func@0x140246230	7.014s	2,067,072,000	9.1%	ntoskrnl.exe
TransformPosition_SSE	6.293s	29,328,768,000	56.3%	1. basics.exe
std::max	6.024s	44,145,024,000	75.7%	1. basics.exe
RtlDelayExecution	6.019s	1,204,224,000	13.0%	ntdll.dll
Tmpl8::Ray::{ctor}	5.284s	32,307,072,000	64.5%	1. basics.exe
RGBF32_to_RGB8	4.336s	12,690,048,000	34.0%	1. basics.exe

We immediately spotted some interesting properties in the Scene::GetAlbedo and Plane::GetAlbedo functions:

1. Plane::GetAlbedo accounts for more than 90% of Scene::GetAlbedo execution time.
2. Plane::GetAlbedo uses a series of if-then-else statements to select the appropriate Albedo implementation. This portion of code is very expensive because of two reasons: first, the CPU has to predict the correct branch every time; second, the majority of the rays usually result in a hit with one of the six planes, thus the function is used heavily.

```
784         float3 GetAlbedo( int objIdx, float3 I ) const
785         {
786             if (objIdx == -1) return float3( 0 ); // or perhaps we should just crash      0.295s   1,424,640,000
787         #ifdef FOURLIGHTS
788             if (objIdx == 0) return quad[0].GetAlbedo( I ); // they're all the same      0.278s   1,284,864,000
789         #else
790             if (objIdx == 0) return quad.GetAlbedo( I );
791         #endif
792             if (objIdx == 1) return sphere.GetAlbedo( I );                               0.665s   3,198,720,000
793             if (objIdx == 2) return sphere2.GetAlbedo( I );
794             if (objIdx == 3) return cube.GetAlbedo( I );                                 0.485s   2,583,168,000
795             if (objIdx == 10) return torus.GetAlbedo( I );                               0.305s   1,231,104,000
796             return plane[objIdx - 4].GetAlbedo( I );                                    12.045s  38,148,096,000
797             // once we have triangle support, we should pass objIdx and the bary-
798             // centric coordinates of the hit, instead of the intersection location.
799         }
```

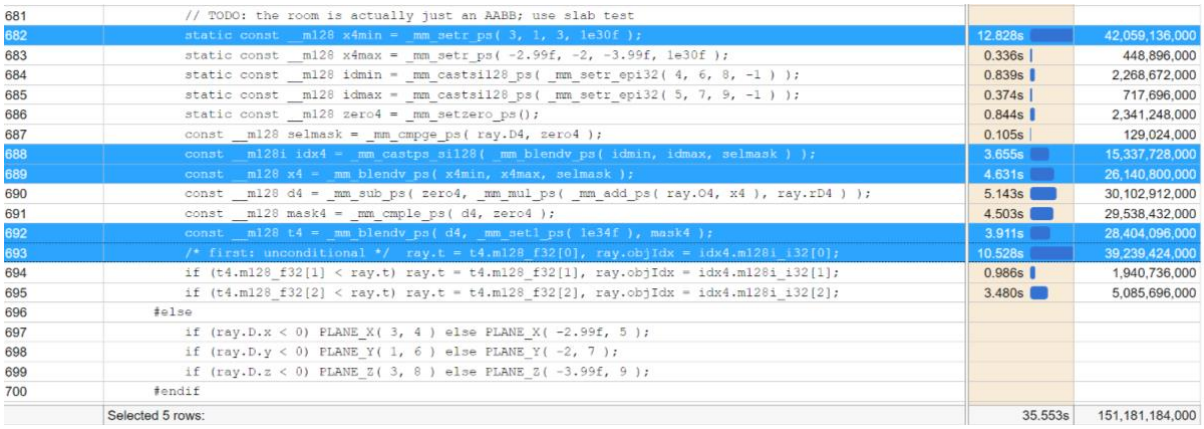
Consequently, we propose the following optimizations:

1. A new Albedo enum class and PlaneCompileTime<Albedo A> are defined in scene.h. PlaneCompileTime exploits the compile-time knowledge to produce a branch-less GetAlbedo function. Each new plane is initialized in the Scene constructor.
2. The if-then-else instructions in Scene::GetAlbedo are replaced by a switch statement.

We executed a new profiling and noticed that our changes have a huge positive impact: Plane::GetAlbedo (previously requiring 25 seconds circa) takes now less than four second and Scene::GetAlbedo (previously requiring 14 seconds circa) takes now around two seconds. We experienced an increase of 13 FPS and 9 MRPS.

Then, we focused on Scene::FindNearest (and noticed the following:

1. The static variables within lines 682, 687 account for 15 seconds of execution.
2. The conditional code within lines 693, 695 takes roughly another 15 seconds to execute. The main problem is that each SIMD lane value is accessed individually thus heavily slowing down the code. Microsoft explicitly suggests to avoid doing that in release builds.
3. Only 75% of SIMD potential is exploited.



681	// TODO: the room is actually just an AABB; use slab test		
682	static const __m128 x4min = _mm_setr_ps(3, 1, 3, 1e30f);	12.828s	42,059,136,000
683	static const __m128 x4max = _mm_setr_ps(-2.99f, -2, -3.99f, 1e30f);	0.336s	448,896,000
684	static const __m128 idmin = _mm_castsi128_ps(_mm_setr_epi32(4, 6, 8, -1));	0.839s	2,268,672,000
685	static const __m128 idmax = _mm_castsi128_ps(_mm_setr_epi32(5, 7, 9, -1));	0.374s	717,696,000
686	static const __m128 zero4 = _mm_setzero_ps();	0.844s	2,341,248,000
687	const __m128 selmask = _mm_cmpge_ps(ray.D4, zero4);	0.105s	129,024,000
688	const __m128i idx4 = _mm_castps_si128(_mm_blendv_ps(idmin, idmax, selmask));	3.655s	15,337,728,000
689	const __m128 x4 = _mm_blendv_ps(x4min, x4max, selmask);	4.631s	26,140,800,000
690	const __m128 d4 = _mm_sub_ps(zero4, _mm_mul_ps(_mm_add_ps(ray.O4, x4), ray.rD4));	5.143s	30,102,912,000
691	const __m128 mask4 = _mm_cmple_ps(d4, zero4);	4.503s	29,538,432,000
692	const __m128 t4 = _mm_blendv_ps(d4, _mm_set1_ps(1e34f), mask4);	3.911s	28,404,096,000
693	/* first: unconditional */ ray.t = t4.m128_f32[0], ray.objIdx = idx4.m128i_i32[0];	10.528s	39,239,424,000
694	if (t4.m128_f32[1] < ray.t) ray.t = t4.m128_f32[1], ray.objIdx = idx4.m128i_i32[1];	0.986s	1,940,736,000
695	if (t4.m128_f32[2] < ray.t) ray.t = t4.m128_f32[2], ray.objIdx = idx4.m128i_i32[2];	3.480s	5,085,696,000
696	#else		
697	if (ray.D.x < 0) PLANE_X(3, 4) else PLANE_X(-2.99f, 5);		
698	if (ray.D.y < 0) PLANE_Y(1, 6) else PLANE_Y(-2, 7);		
699	if (ray.D.z < 0) PLANE_Z(3, 8) else PLANE_Z(-3.99f, 9);		
700	#endif		
	Selected 5 rows:	35.553s	151,181,184,000

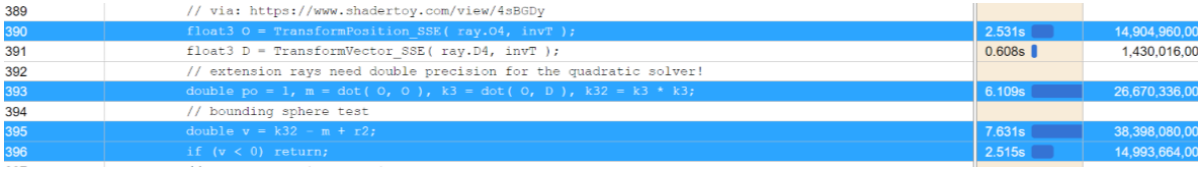
Consequently, we propose the following optimizations:

1. The SIMD implementation is replaced with the macros PLANE_X, PLANE_Y and PLANE_Z. These macros are faster because of the lower number of operations per plane and minimal cache usage for the data.

We executed a new profiling and noticed that our changes have a huge positive impact: Scene::FindIntersection (previously requiring 68 seconds circa) takes now 41 seconds circa, a 27 seconds improvement on the baseline version. We experienced an increase of 13 FPS and 7.6 MRPS.

Then, we focused on Torus::Intersect and noticed the following:

1. 6.5 seconds circa are related to the execution of two dot product operations.
2. TransformPositionSSE and TransformVectorSSE account for 3.1 seconds circa of execution. This happens because the returned float3 is constructed by inefficiently decomposing a SIMD variable into each single lane value.



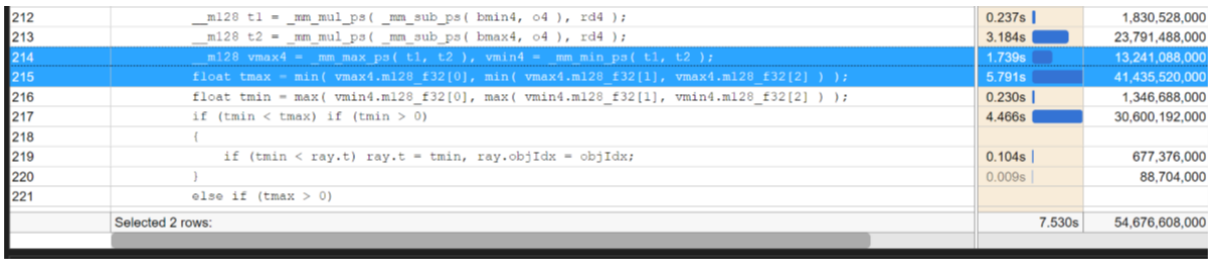
Consequently, we propose the following optimization:

1. A fast dot product function (Dot3SSE) exploting SIMD is implemented in fastmath.h and used.
2. Given that O and D are only used for the dot operations, two variations of TransformPositionSSE and TransformVectorSSE (TransformPositionSSE_128 and TransformVectorSSE_128) returning a __m128 value are implemented in fastmath.h and used in the implementation, so that each single lane value access is avoided.

We executed a new profiling and noticed that our changes have a modest positive impact: Torus::FindIntersection (previously requiring 43 seconds circa) is now taking 35.8 seconds circa to execute, a 7 seconds improvement on the baseline version. We experienced an increase of 4 FPS and 3.2 MRPS.

Then, we looked up for other optimization opportunities, this time focusing on the Cube class. We noticed the following:

1. Both Cube::Intersect and Cube::IsOccluded compute the horizontal minimum and maximum float in a __m128 variable inefficiently. This is because each lane value is accessed and compared sequentially.



Consequently, we propose the following optimization:

1. Two efficient horizontal minimum and maximum functions (HorizontalMin2SSE and HorizontalMaxSSE3) are implemented in

fastmath.h. Each tmin, tmax computation is then adapted with our new functions.

We executed a new profiling and noticed that our changes have a modest positive impact: Cube::Intersect and Cube::IsOccluded (previously requiring 28 and 9 seconds circa) are now taking 22.2 and 4 seconds circa to execute, a 6 and 35seconds improvement on the baseline versions. We experienced an increase of 8 FPS and 5.4 MRPS.

Then, we noticed that a non-negligible amount of time was spent executing trigonometric functions (cosine, arccosine), square roots and cubic roots. These functions are known to be expensive, especially the cubic root one (cbrt). Alone, cbrt accounts for almost 13 seconds of execution time.

After some days of research and experimentation, we selected the following articles [3, 4, 5, 6, 7, 8, 9] as guidelines for the implementation of four fast cosine, arccosine, square and cubic roots functions. We provide these implementations (FastCos, FastAcos, FastSqrt, FastCbrt respectively) in fastmath.h.

Then, we replaced the old math functions in scene.h with our new ones and run a test. Unfortunately, two main issues arised:

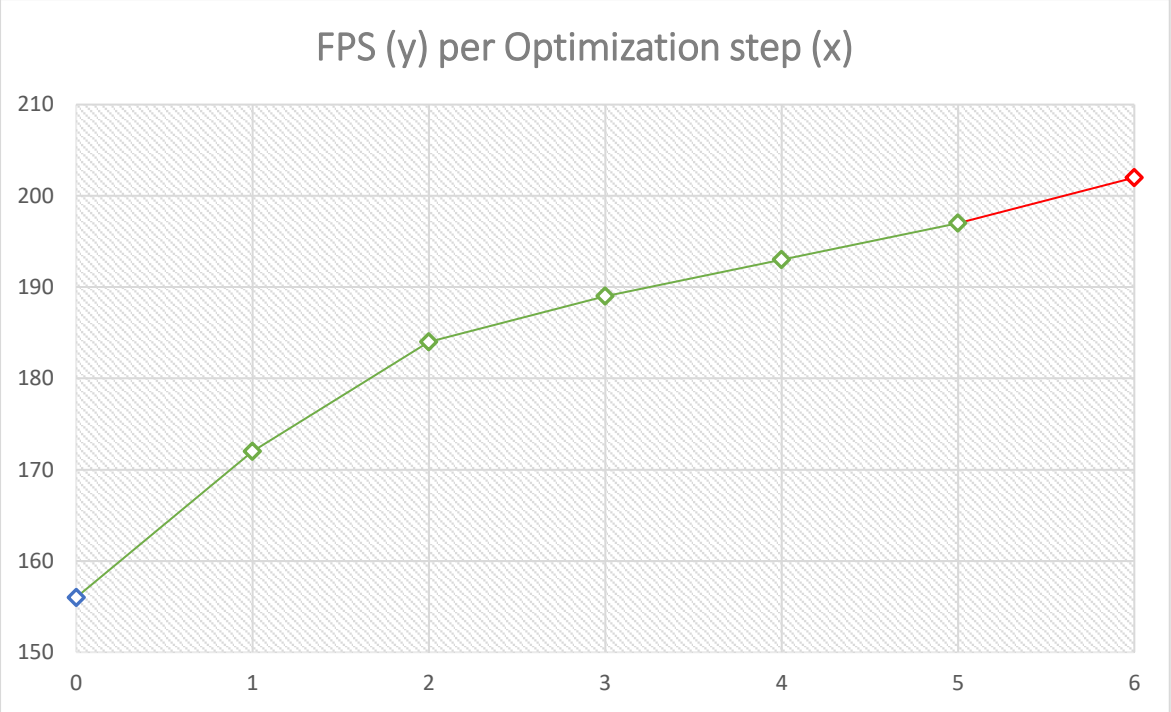
1. No significant speed-up was achieved for cosine, arcosine and sqrt. We think the main bottleneck is related to memory transfers, so the benefits of our implementations cannot be noticed.
2. Even though cbrt was running 70% faster (improved by 5 FPS and 3.2 MRPS), the precision of our implementation was not sufficiently high (within the range $[-2.6 \cdot 10^{-7}, 2.7947 \cdot 10^{-7}]$) not to introduce minor artifacts in the "2. Whitted" project. Adding the required number of newton's iterations to achieve a good level of precision made it slower than the baseline cbrt.

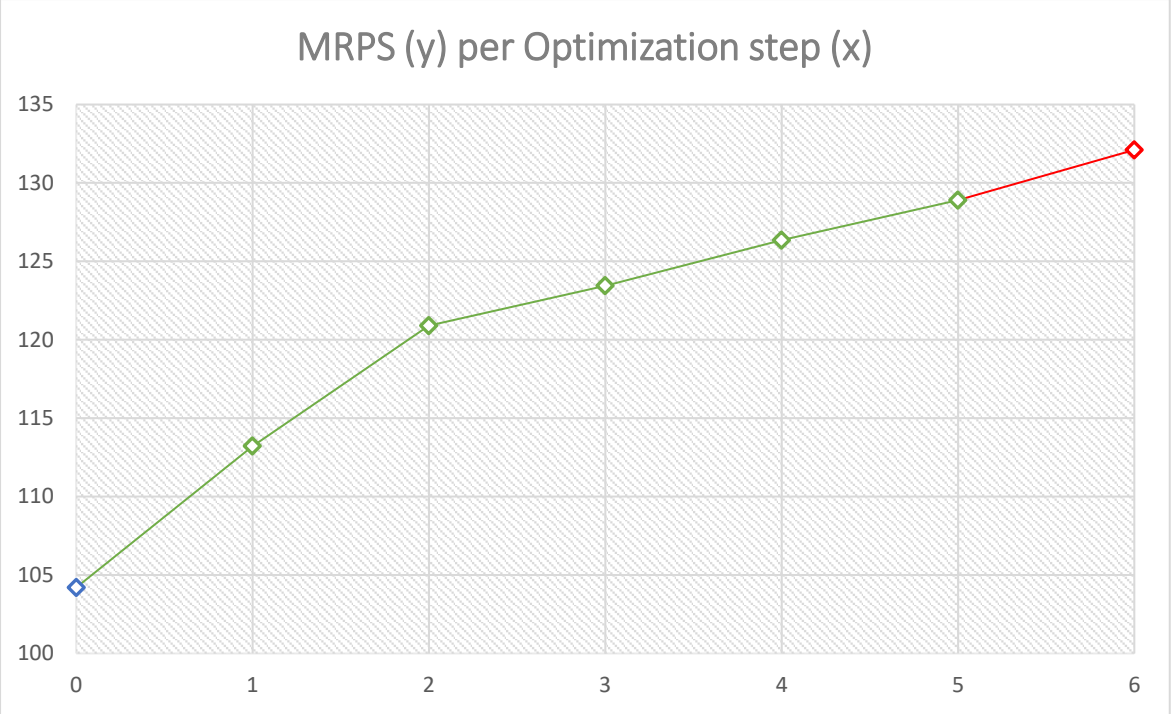
Because of the very strict deadlines for this project we decided not to include this optimization. Nonetheless, we believe that with further improvements our functions can contribute to an overall speed-up of the application. We provide the results obtained by comparing our functions with the ones implemented in the C++ standard library. Each value has been sampled 1000 times and the standard deviation is always within the range [-25ns, 25ns].

/	10^3	10^4	10^5	10^6	10^7
sqrtf	300ns	21.100ns	284.900ns	1.590.700ns	14.888.500ns
fastSqrtf	200ns	400ns	18.900	332.400ns	2.991.300ns
cbrt	39.800ns	2.368.400ns	6.897.400ns	35.450.300ns	388.228.900ns
fastCbrt	8400ns	99.200ns	676.500ns	6.613.600ns	83.868.200ns
cosf	6000ns	35.800ns	2.190.800ns	6.618.200ns	19.697.200ns
fastCosf	1300ns	1800ns	15.500ns	144.200ns	2.912.500ns
acosf	29.300ns	26.600ns	1.146.500ns	6.638.900ns	23.579.500ns
fastAcosf	600ns	1600ns	15.900ns	393.300ns	3.712.200ns

4. Achieved speed-up

Our optimizations achieve overall an increase of 23% in FPS and 24% in MRPS when compared to the baseline application. In the following graph we show the increase in FPS and MRPS for all our optimizations. The blue point represents the baseline performance while the red point represents the performances using our fast math functions which cannot be considered a real improvement because of the visual artifacts introduced.





3. Notes

No particular action should be taken in order to run the project. We modified a bit renderer.cpp in the project "1. Basics" just to record profiling data.

5. References

- [1] ThrottleStop, https://www.techpowerup.com/download/techpowerup-throttlestop
- [2] Boost.Chrono, https://www.boost.org/doc/libs/1_74_0/doc/html/chrono.html
- [3] Fast Sign Flip, http://fastcpp.blogspot.fr/2011/03/changing-sign-of-float-values-using-sse.html
- [4] SSE, http://www.songho.ca/misc/sse/sse.html
- [5] Fast SSE Select, http://markplusplus.wordpress.com/2007/03/14/fast-sse-select-operation
- [6] Function approximation, http://cbloomrants.blogspot.fr/2010/11/11-20-10-function-approximation-by_20.html
- [7] Fast Arctan/Atan, http://nghiaho.com/?p=997
- [8] Fast Arctan, http://www.researchgate.net/publication/3321724_Efficient_approximations_for_the_arctangent_function
- [9] Fast Cubic Root, http://www.researchgate.net/publication/349411371_Fast_Calculation_of_Cub

e_and_Inverse_Cube_Roots_Using_a_Magic_Constant_and_Its_Implementation_on_Microcontrollers