ELEKTROTEHNIŠKI VESTNIK 87(5): 260-266, 2020
ORIGINAL SCIENTIFIC PAPER

Evaluation of Perlin Noise using NVIDIA CUDA Platform

Emir Skejić, Damir Demirović, Dino Begić[1]

University of Tuzla, Faculty of Electrical Engineering, Tuzla, Bosnia and Herzegovina
[1] Platogo, Inc., Vienna, Austria
E-mail: damir.demirovic@untz.ba

Abstract. Nvidia Compute Unified Device Architecture (CUDA) is a platform for parallel programming and application programming interface which allows developers and engineers to drastically accelerate the calculation of common parallel algorithms using the power of a graphical processing unit (GPU). One of the easily parallelized image-generation algorithms is the Perlin noise. The paper evaluates parallel implementations of the Perlin noise on a desktop central processing unit (CPU) and GPU. The obtained speedup is about 45 times for GPU compared to a single CPU thread. Differences between the CPU and GPU results are evaluated and are found significant. The performances like image differences and profiling performances are evaluated with the CUDA profiler.

Keywords: Perlin noise, CUDA, performance evaluation

Ocena zmogljivosti platforme NVIDIA CUDA za izračun gradientnega šuma Perlin

Nvidia CUDA (Compute Unified Device Architecture) je platforma za vzporedno programiranje, ki temelji na grafičnih procesnih enotah (GPU). Aplikacijski vmesnik platforme omogoča razvojnim inženirjem drastično pospeševanje računanja vzporednih algoritmov. Algoritem za ustvarjanje slik z gradientnim šumom Perlin je enostaven za vzporedno izvajanje. V prispevku smo ovrednotili izvajanje vzporednega algoritma Perlin na CPU in GPU. Algoritem se izvede na GPU 45-krat hitreje kot na posamezni niti CPU. Ugotovili smo signifikantne razlike med CPU in GPU ter s profiliranjem ocenili zmogljivost platforme CUDA.

1 INTRODUCTION

The procedural texture primitive [1] is often used to increase the appearance of realism in the computer graphics. A procedural texture is a texture created using a mathematical description rather than stored data.

The Perlin noise [2] is an extremely powerful algorithm that is often used in the procedural content generation. The Perlin noise is probably the most well-known procedural noise function [3].

The procedural noise has many advantages, like a very low memory footprint, fast computation, etc. Its function assigns pseudo-random gradient vectors to each point of the vector lattice and obtains the resulting value by using a cubic polynomial to interpolate between the closest gradient vectors. The pseudo-random gradient is obtained by hashing the lattice point and using the result to choose a gradient. There are several improvements to the original algorithm.

The improved Perlin noise [4], [5] reduces visual artifacts in the noise derivatives and improves the overall noise appearance.

The lattice gradient noise generates noises by interpolating or convolving random values and/or gradients defined at the points of the integer lattice [3]. The Perlin noise is a representative example of the lattice gradient noise.

In [6], the authors present a general data-augmentation strategy using the Perlin noise which generates a random mixture of class-labeled ROI patches.

In [7], the authors propose the Gaussian brightness models with the Perlin noise model to simulate aged banknotes.

In [8], the author proposes a modification of the Perlin improved noise that makes it much suitable for implementation on GPU. The modified noise function is presented without table lookups.

In [9], a Pixel shader on GPU to generate the Perlin noise function is proposed.

The paper evaluates, a parallel implementation of the improved Perlin noise [4] in a CPU and GPU execution environment using CUDA. The Perlin noise implemented by using CPU and GPU on a Google Cloud. The obtained results and speedup are analyzed. Other profiling performances are analyzed using the CUDA profiler.

2 PERLIN NOISE ALGORITHM

The Perlin noise algorithm is usually implemented as a two- or three- dimensional function but can be defined for any number of dimensions. The algorithm consists of three steps: grid definition with random gradient vectors,

Received 24 July 2020
Accepted 22 October 2020

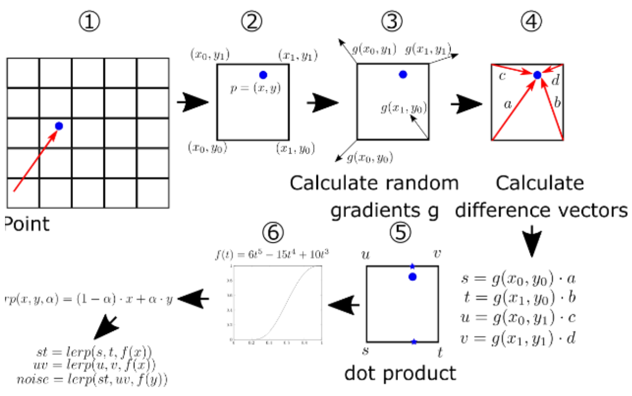
①
②
③
④
Point
(x_0, y_1)
(x_1, y_1)
$p = (x, y)$
(x_0, y_0)
(x_1, y_0)
$g(x_0, y_1)$
$g(x_1, y_1)$
$g(x_1, y_0)$
$g(x_0, y_0)$
Calculate random gradients g
Calculate difference vectors
⑥
⑤
$f(t) = 6t^5 - 15t^4 + 10t^3$
$s = g(x_0, y_0) \cdot a$
$t = g(x_1, y_0) \cdot b$
$u = g(x_0, y_1) \cdot c$
$v = g(x_1, y_1) \cdot d$
$rp(x, y, \alpha) = (1 - \alpha) \cdot x + \alpha \cdot y$
$st = lerp(s, t, f(x))$
$uv = lerp(u, v, f(x))$
$noise = lerp(st, uv, f(y))$
dot product

Figure 1. Phases of the Perlin noise generation.

computation of the dot-product between the distance-gradient vectors, and interpolation between these values. Due its nature, the Perlin noise can be relatively easily parallelized. This set of problems is called embarrassingly parallel, which is a problem where little effort is needed to separate the problem into many parallel tasks.

The first step is to define an n-dimensional grid by subdividing the domain into unit cells where each point is an n-dimensional unit-length gradient vector (see Figure 1). The random gradients (g) at the cell corners are then calculated. The random gradients are assigned by using a pseudo-random number generator. As this process can be computably expensive hash and lookup tables can be used for precomputed gradient vectors. In the fourth step, the difference vectors from cell corners for point p are calculated (Figure 1).

The dot product between the gradient vector at the node and the distance vector is now computed. In the last step, a smooth interpolation is made between the computed dot products at the nodes of the cell containing the point. The resulting pixel, given with a variable noise, is obtained using a linear interpolation.

2.1 CUDA

CUDA [10] is a platform for parallel processing and Application Programming Interface (API) created by Nvidia for General Purpose computing on Graphical Processing Units (GPGPU). Dramatical improvements in processing can be obtained by harnessing the power of modern GPUs.

The CUDA application works together with CPU, the sequential workload runs on CPU and the computer- intensive portion runs on thousands of GPU cores in parallel. CUDA can be used with C [11], C++, Fortran, and Python code. It supports heterogeneous computing with a host which is CPU and its associated memory, and a device with the GPU and its associated memory.

The processing flow in CUDA is the following. The data is copied from the CPU memory to the GPU memory, execute the code on GPU and copy the results from the GPU memory to the CPU memory. The advantage of CUDA is the usage of the standard C program that runs on the host, and the NVIDIA compiler (nvcc) used to compile the programs without the device code. CUDA extends the standard C code with some new possibilities.

The compiler nvcc separates the source code on the host and device functions. The device functions are processed with the nvcc and host functions by a standard host compiler.

The host code is executed in a kernel. In the CUDA terminology, the parallel invocation of the code is referred to as a block, and the set of blocks is referred to as a grid. Each block is executed on the device in parallel. Each block can be separated into parallel threads as shown in Figure 2.

The CUDA programming model assumes a system composed of a host and a device, each with its separate memory. Kernels operate out of the device memory. For a full control and optimal performance, the CUDA runtime provides functions to allocate the device memory, release the device memory, and transfer the data between the host memory and the device memory. Upon transferring the data to the GPU global memory, the kernel function can be invoked from the host side to perform, for example, the array summation on GPU. As soon as the kernel is called, the control is immediately returned to the host. At this point, the host is able to perform other functions while the kernel is running on GPU. Thus, the kernel is asynchronous with regard to the host. When the kernel has finished processing all array elements on GPU, the result is stored on the GPU global memory in the array. The result is copied from the GPU memory back to the host array.

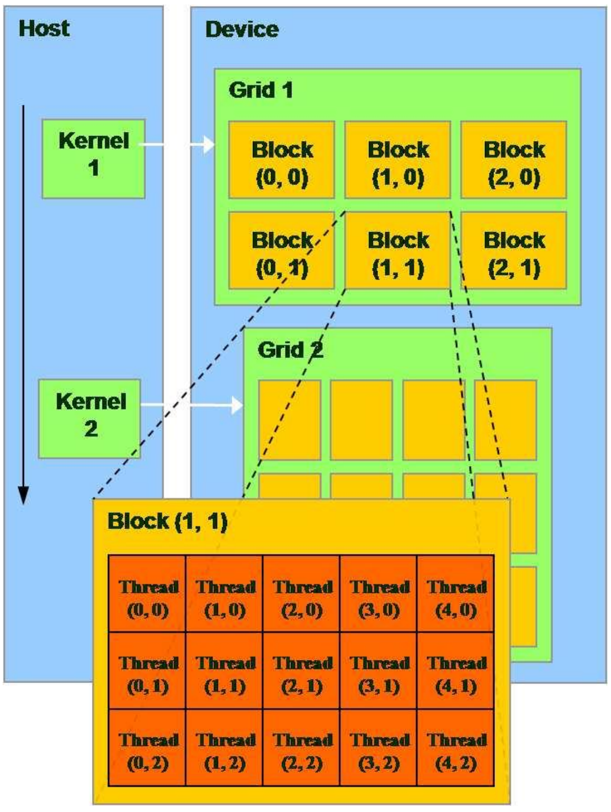

Figure 2. CUDA architecture.

3 MEASURING PERFORMANCE

The improved Perlin noise is implemented in a single-threaded version, used as a baseline for comparison and is parallel with the multiple-thread CPU and CUDA version for GPU. To have it evaluated, several generated noise images are used. The algorithm evaluation is made for several resolutions, from 8x8 to 10000x10000 pixels.

The CPU and GPU specifications used in the process of comparison are given in Table 1. All experiments are made on a Google Colaboratory on a Linux system and CUDA version 10.1 with the 418.67 graphics card driver version for GPU and CPU1 which is on a Google Cloud. The desktop CPU2 is used.

Table 1. Hardware specifications for the experiments.

Component	Model
CPU1	Intel® Xeon® CPU @ 2.30GHz 2 CPUs
CPU2	Intel® Core™ i5-2500 @ 3.30Ghz
GPU	Tesla K80 2x12 GB RAM
RAM	12 GB
OS	Ubuntu 18.04.3 LTS

The GPU implementation is compiled with an nvcc compiler using a O3 optimization flag. The CPU version is compiled with a gcc compiler using a O2 optimization flag and a pthreads library. No other optimizations are applied for either of the versions.

Each test run is made 100 times. Thus obtained mean value is used for the display in the following charts. An example of a generated Perlin image is shown in Figure 3.

In the first experiment, the running times for three versions of the same algorithm are measured. To simplify the display, the image resolutions are grouped into a small, medium and large number of the image elements. The Y-axis shows the time in microseconds needed to complete the algorithm. The resolution of the image and the type of the implementation are shown on the X-axis.

As expected, one CPU thread achieves the best performance for a relatively small number of elements (Figure 4). The performance of two threads is worse because of the time lost between the switching threads. Here, CUDA here performs almost consistently.

There are no lost CPU cycles for thread creation and synchronization. The only raw power of one CPU core/thread is enough to get the job done and there is no time lost for copying the results from GPU to the main memory for the CUDA comparison. For all charts, the blue bar represents a single CPU thread, the orange two threads, and the yellow CUDA threads. The speedup of one thread against two threads, for a small number of elements ranges from 2.1 to 23 times for a 32x32 and 8x8 resolution, respectively. Figure 5 shows the results for a medium number of elements.

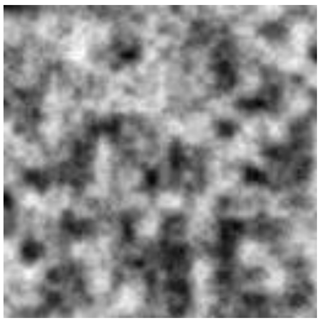

Figure 3. Example of a generated Perlin noise image.

Table 2. GPU speedup for small and medium resolutions.

Resolution	8x8	8x16	16x8	16x16	32x32	128x128	512x512
x	5.9	2.0	2.6	1.2	**0.3**	25.1	43.2

Table 3. GPU speedup for large resolutions.

Resolution	1280x720	1920x1080	2048x1024	2560x1440	3840x2160	4096x4096	8192x8192	10000x10000
X	44.0	43.9	43.5	43.9	43.8	44.1	43.6	**44.7**

As seen in each experiment made, for this image group, CUDA outperforms CPU, and performances of one and two threads are similar. The speedup of CUDA against one thread is from 25 to 44 times for the 128x128 and 1280x720 images, respectively.

Finally, when applied to large resolutions, CUDA performs best, and its performance between one and two CPU threads is similar. Speedup of one CUDA against one thread is from 43 to 44 times for the 2084x1024 and 10000x10000 images, respectively (Figure 6).

Tables 2 and 3 show the speedup for a single thread implementation. Except for only one case, GPU gives a significant speedup, i.e. between 1.2 and 44.7.

The CPU1 speedup is negligible in almost any case. For a small number of elements, the threaded version is significantly slower, and for a medium and a large number of elements it is a little faster.

3.1 Comparison with the desktop CPU

The performance on a Google Colaboratory CPU, denoted as CPU1, is low. This particularly applies to the

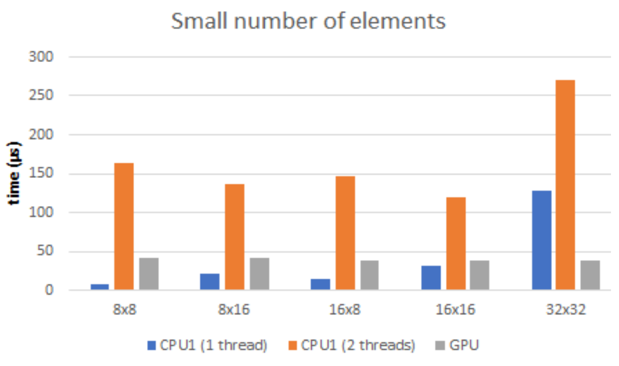

Figure 4. Timing performances for a small number of elements.

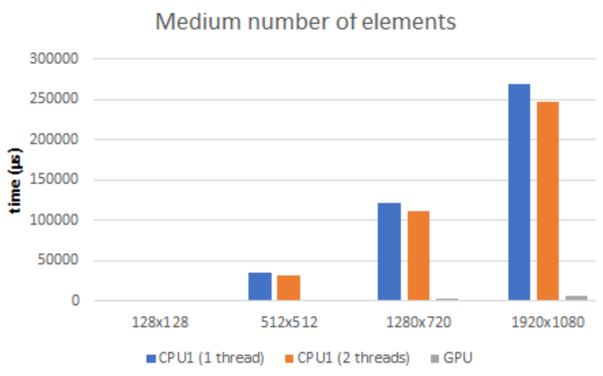

Figure 5. Timing performances for a medium number of elements.

threaded implementations. Figures 7, 8, and 9 show a comparison with a rather old CPU2. The performance of both CPUs is similar for almost all experiments. This is probably due to the high virtualization overhead in the Google Cloud.

3.2 *Image-quality differences between CPU and GPU*

Due the differences between the CPU and GPU architecture calculation results, the two architectures are compared and the trade-off is calculated. The results obtained with CPU are used as a baseline and a comparison is made with the GPU implementation results using floating-point operations. Table 4 shows the maximum absolute difference between the pixel values and the number of different pixels after scaling to [0,255].
As these errors are relatively small, their impact on the GPU version is minor. Figure 10 shows the differences obtained when using GPU and their distribution on the image. The pixels calculated with GPU are marked with crosses.

3.3 *Profiling measurements*

The effectiveness of the GPU implementation is evaluated with a profiler. The CUDA profiler [12] is a providing a vital feedback for optimizing the CUDA applications.

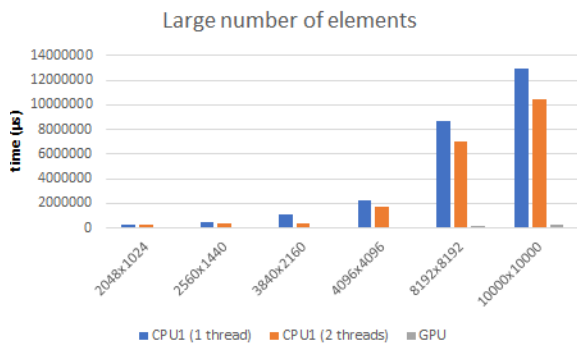

Figure 6. Timing performances for a large number of elements.

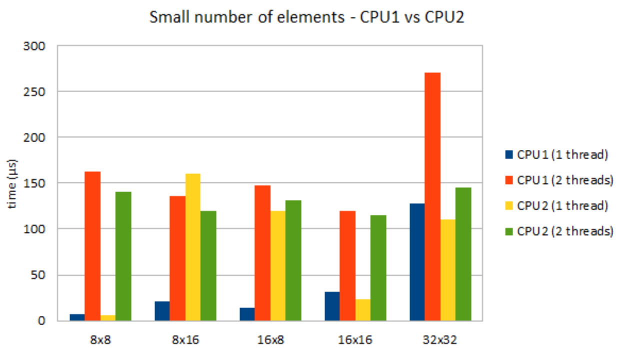

Figure 7. CPU1 and CPU2 performance comparison.

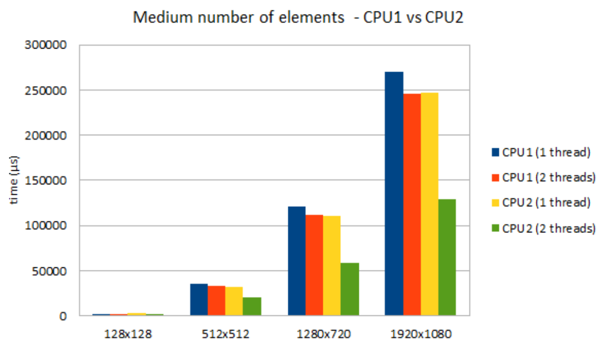

Figure 8. CPU1 and CPU2 performance comparison.

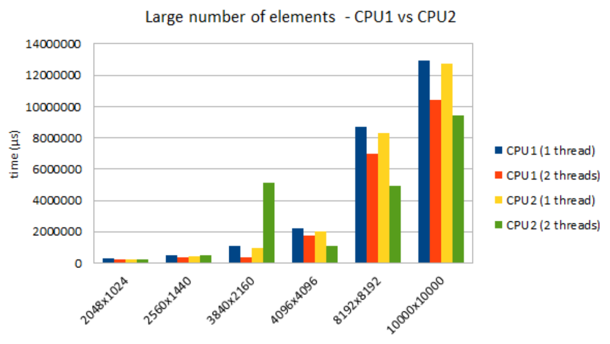

Figure 9. CPU1 and CPU2 performance comparison.

In our experiments, a command-line application nvprof and Visual Profiler from CUDA SDK are used. The profiler carries information about different performance measures of the whole application but the user usually needs information about the performance-critical code that can be later on improved. For profiling, the following performance measures are analyzed: achieved occupancy, instruction replay overhead, requested global-load throughput, L1 throughput (L1 reads), eligible warps per active cycle, multiprocessor activity, issue-slot utilization and FLOP efficiency (Peak single) and compute utilization. Short descriptions of the used measures are given below.

3.4 *Achieved occupancy*

The achieved occupancy is measured during the execution of the kernel and can be compared with the theoretical occupancy. The theoretical occupancy is an upper limit for active warps, compile options for the kernel and device capabilities.
The upper limit for active warps is the product of the upper limit for active blocks and the number of warps per block. The achieved occupancy of a kernel is defined as the ratio of the average active warps per cycle to the maximum of the warps supported on a streaming multiprocessor (SM). The occupancy varies over the time during the execution of the warps and can be different for each SM. A low occupancy results in poor instruction efficiency because there are not enough eligible warps to hide the latency between dependent instructions.
When running at different occupancy levels the effects on a kernel execution time are usually observed.

Table 4. Differences between the CPU and GPU implementation for different resolutions.

Image resolution	**Max absolute difference ($x10^{-7}$)**	**Number of different pixels**
8x8	0.0596	0
8x16	0.1788	0
16x8	0.0596	0
16x16	0.1788	0
32x32	0.1788	0
128x128	0.1788	0
512x512	0.2533	0
1280x720	0.2980	0
1920x1080	0.2980	2
2048x1024	0.2980	3
2560x1440	0.2980	8
3840x2160	0.2980	22
4096x4096	0.2980	35
8192x8192	0.2980	139
10000x10000	0.2980	191

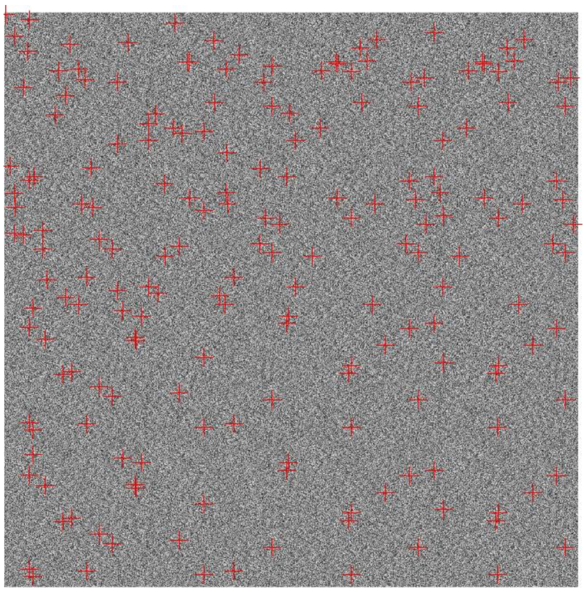
Figure 10. Example of errors distribution in a GPU-generated image of 10000x10000; red crosses mark the pixels with errors.

3.5 *Instruction Replay Overhead*

The measure indicates the number of times an instruction is issued without being completed. There can be various reasons for this, such as constant cache miss on an immediate constant, load cache misses, etc. The instruction replays use an instruction for slot reducing to compute the throughput.

Figure 11 shows the performances for an achieved occupancy and instruction-replay overhead. For the medium and large resolutions, the occupancy is over 90% which is a good result and can be further improved by optimization. The instruction replay overhead rises with higher resolutions and stays at about 45%, and the achieved occupancy for bigger images is above 90%.

3.6 *Multiprocessor Activity*

This measure reports the microprocessor activity. It indicates the percentage of the time SM has one or more warps that are active. So, a low value indicates that much of the time a microprocessor is in an idle state i.e. not issuing instructions.

3.7 *Issue Slot Utilization*

The issue-slot utilization measure indicates the percentage of the issue slots issuing at least one instruction. This performance is an indication of how busy the kernel keeps the device.

3.8 *FLOP Efficiency (Peak single)*

The FLOP Efficiency (Peak single) measure defines the rate of the achieved peak single-precision floating-point operations. The CUDA profilers calculate FLOPs twice by replying the kernel. In the first step, the time and SM-elapsed cycles are collected. In the second step, the profiler modifies the kernel to calculate the total number of FLOPS.

The multiprocessor activity, issue-slot utilization and FLOP efficiency performances are given in Figure 12. The issue slot utilization is almost constant for all behavior similar to that of the achieved occupancy.

3.9 Requested Global-Load Throughput

This metric marked with the gld_efficiency is defined as 100*gld_requested_throughput/gld_throughput.
The matrix is at its maximum when all accesses to memory have perfectly coalesced. When an application requests the values from the memory and if the values are scattered several transactions can be done to load all the data.

3.10 L1 throughput (L1 reads)

The architectures that employ an L1 cache combined with a shared memory exhibit a higher L1 bandwidth than the architectures employing an L1 cache and a shared memory.

3.11 Eligible Warps per Active Cycle

An actively executing warp is called a selected warp. If an active warp is ready to execute but not executing, it is an eligible warp. If a warp is not ready to execute, it is a stalled warp. The requested Global-Load Throughput, L1 throughput (L1 reads) and Eligible Warps per Active Cycle performances are given in Figure 13.

Figure 14 shows the kernel and multiprocessor activity execution vs the memory copy. The time taken for the kernel to run and the rest of the time is time, when, the data is copied from the device to the host memory. L1 throughput is directly associated with the performance and shows a behavior similar to that of achieved occupancy. Figures 11, 12 and 13 show the achieved occupancy and the multiprocessor activity and the L2 throughput is almost constant at a 512x512 resolution and higher, and for lower resolutions, the multiprocessor activity is low.

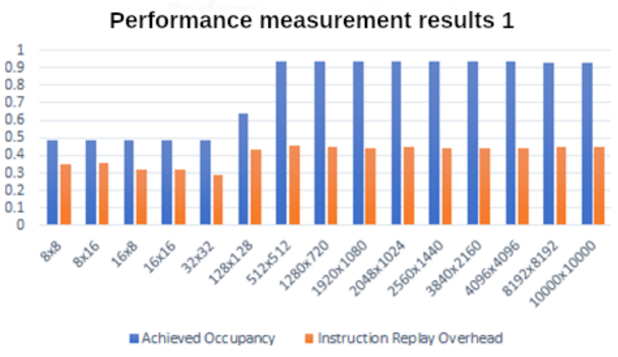

Figure 11. Performances for the achieved occupancy and instruction replay overhead.

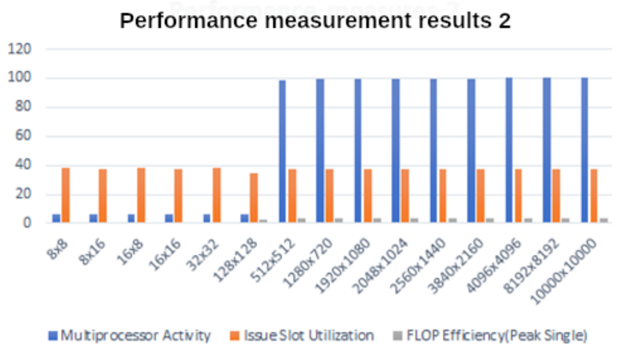

Figure 12. Performances for the multiprocessor activity, issue-slot utilization and FLOP efficiency (Peak single).

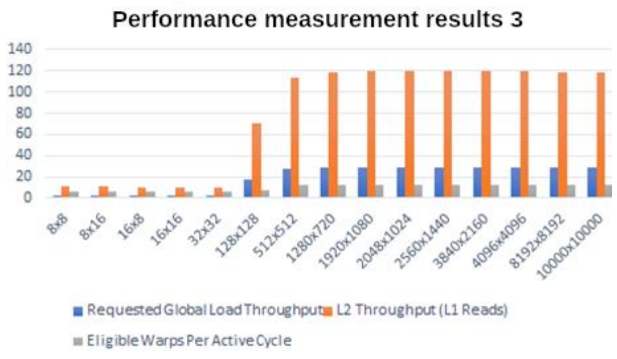

Figure 13. Performances measurement results for the requested global-load throughput, L2 throughput (L1 reads) and Eligible Warps per active cycle.

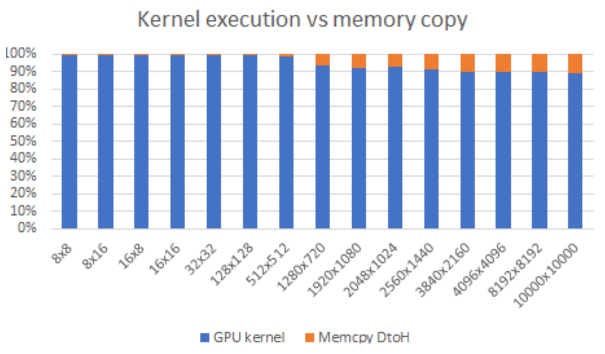

Figure 14. Obtained performance measurement results for the GPU kernel running time and the time copy taken to the data from the device to the host memory.

As seen, to copy a larger resolution from GPU to CPU takes about 10% of the whole execution time, so, the cost of moving data across the bus should be taken into account. Profiling the performance result shown in Figures 11, 12, 13 and 14 is given in Table 5.

Table 5. Profiling the performance measurement results.

	8x8	8x16	16x8	16x16	32x32	128x128	512x512	1280x720	1920x1080	2048x1024	2560x1440	3840x2160	4096x4096	8192x8192	10000x10000
Achieved Occupancy	0.484953	0.486051	0.485055	0.487151	0.488877	0.640281	0.936983	0.934747	0.933168	0.933929	0.93349	0.932242	0.932073	0.931453	0.93135
Requested Global Load Throughput	2.4867	2.4935	2.4249	2.4231	2.3173	17.125	27.551	28.629	28.872	28.815	28.904	28.956	28.922	28.792	28.736
Multiprocessor Activity	6.46	6.49	6.44	6.44	6.38	6.209	98.16	99.49	99.78	99.69	99.82	99.92	99.96	99.99	99.99
Instruction Replay Overhead	0.353357	0.354329	0.316003	0.316697	0.291724	0.429945	0.459406	0.44739	0.444033	0.445238	0.443992	0.44377	0.444058	0.44586	0.448984
L2 Throughput (L1 Reads)	11.284	11.315	10.23	10.223	9.5677	70.722	113.9	118.33	119.34	119.1	119.47	119.68	119.54	119.01	118.78
Issue Slot Utilization	38.27	37.34	38.27	37.53	38.5	34.27	37.07	37.51	37.61	37.54	37.58	37.54	37.5	37.39	37.18
FLOP Efficiency(Peak Single)	0.28	0.28	0.3	0.29	0.3	2.31	3.88	4	4.05	4.03	4.05	4.04	4.04	4.02	3.98
Eligible Warps Per Active Cycle	6.465453	6.570727	6.440702	6.343696	6.42666	7.186495	12.667829	12.880582	12.914691	12.868511	12.880582	12.840956	12.766661	12.65356	12.463822

4 CONCLUSION

This paper presents a speedup and profiling performance of the CPU and GPU version of the Perlin noise. The measurement results, the Perlin noise and the other image processing algorithms significant for the performance improvement are obtained by using parallel programming techniques. To enable parallelism within an application, the capacity assesses of the main processor using threads should be sufficient It is shown, that if calculations are not done for very large arrays, the threads of the main processor should be used. The threads are even slower than the serial code execution unless the resolution is to be processed is the order of 1000 pixels or higher.

The performance of the Perlin noise algorithm is evaluated for several image resolutions used in practice. It is shown that for a very large number of the data to be processed, the obtained speedup for a GPU implementation is about 45 times of that of non-optimized version. A profiling provides an additional insight in the algorithm performance and enables taking further optimization steps.

The focus of the future work will be on the GPU-code optimization based on the profiling data presented in this work.

References

[1] Ebert et al: Texturing and Modeling: A Procedural Approach, page 10. Morgan Kaufmann, 2003.

[2] Perlin, Ken. 1985. "An Image Synthesizer." In Computer Graphics (Proceedings of ACM SIGGRAPH 85) 19(3), pp. 287-296.

[3] Lagae, A., Lefebvre, S., Cook, R., DeRose, T., Drettakis, G., Ebert, D. Zwicker, M. (2010). A survey of procedural noise functions. Computer Graphics Forum, 29(8), 2579-2600. https://doi.org/10.1111/j.1467-8659.2010.01827.x

[4] Perlin, Ken. 2002. "Improving Noise." ACM Transactions on Graphics (Proceedings of SIGGRAPH 2002) 21(3), pp. 681-682. Updated version available online at http://mrl.nyu.edu/~perlin/noise/

[5] Perlin, Ken. 2004. "Implementing Improved Perlin Noise." In GPU Gems, edited by Randima Fernando, pp. 73-85. Addison-Wesley.

[6] Bae HJ, Kim CW, Kim N, et al. A Perlin Noise-Based Augmentation Strategy for Deep Learning with Small Data Samples of HRCT Images. Sci Rep. 2018;8(1):17687. Published 2018, Dec 6. doi:10.1038/s41598-018-36047-2

[7] Baek, S., Le, S., Choi, E., Baek, Y., & Lee, C. (2017). Banknote simulator for agng and soiling banknotes using Gaussian models and Perlin noise. In ICPRAM 2017 – Proceedings of the 6th International Conference on Pattern Recognition Applications and Methods (Vol. 2017-January, pp. 289-292). SciTePress.

[8] Marc Olano. 2005. Modified noise for evaluation on graphics hardware. In Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware (HWWS ’05). Association for Computing Machinery, New York, NY,USA,105-110. DOI:https://doi.org/10.1145/1071866.1071883

[9] John C. Hart. 2001. Perlin noise pixel shaders. In Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware (HWWS ’01). Association for Computing Machinery, NewYork,NY,USA,87–94. DOI:https://doi.org/10.1145/383507.383531

[10] Rob Farber, CUDA Application Design and Development, 2011.

[11] John Cheng, Max Grossman, Ty McKercher, Professional CUDA C Programming, 2014.

[12] Duane Storti and Mete Yurtoglu. 2015. CUDA for Engineers: An Introduction to High-Performance Parallel Computing (1st Ed.). Addison-Wesley Professional.

Emir Skejić received his B.Eng, M.Sc. and Ph.D. degrees in Electrical Engineering from the Faculty of Electrical Engineering, University of Tuzla, Bosnia and Herzegovina, in 2000, 2003 and 2007, respectively. Since 2001, he has been employed with the same faculty, where he is currently an associate professor in the field of Computer and Information Science.

Damir Demirović received his B.Eng, M.Sc. and Ph.D. degrees in Electrical Engineering from the Faculty of Electrical Engineering, University of Tuzla, Bosnia and Herzegovina, in 2003, 2006 and 2011, respectively. Currently, he is an associate professor at the same faculty. His research interests are in computer science and include pattern recognition, image processing and analysis.

Dino Begić received his B.Sc. and M.Sc. in Electrical Engineering from Faculty of Electrical Engineering, University of Tuzla, Bosnia and Herzegovina, in 2013 and 2018, respectively. Currently, he works as a software developer for the Platogo Interactive Entertainment GmbH in Vienna, Austria.