Mass Parallelization of *n*-Dimensional Perlin-Like Noise with CUDA

Sav Wheeler & Enbai Kuang

I - Introduction

Perlin noise is a procedural noisy texture generation algorithm designed by Ken Perlin, originally developed as the Pixel Stream Editor in 1985 [1][2]. Perlin later developed Simplex Noise, which improved this to utilize a simpler space-filling grid in 2001 [3]. Traditionally, this algorithm is applied to terrain and particle generation programs, particularly in game development and landscape ecology [4][5][6], though it has extensive applications in other fields such as microbiology and material sciences [7][8][9][10]. Figures 1 and 2 show examples of the Perlin noise algorithm applied to terrain generation, with Figure 2 using a variant known as value noise.

Figure 1: Procedural Terrain Generation Using Perlin Noise

Figure 2: Procedural Terrain Generation Using Value Noise [11]

With the rising popularity of machine learning and deep learning algorithms for various applications, noise generation algorithms are essential in adversarial machine learning [12][13][14], with potential applications in defending against model poisoning attacks [15]. In areas of machine learning, Perlin noise has already seen some applications, particularly in data augmentation [16][17][18]. However, an overlooked factor for scaling this algorithm to modern machine learning algorithms is the dimensionality of the noise in relation to the data, which in many ML systems may exist in much higher dimensionality than 2D or 3D.

Few studies have investigated the parallelization of Perlin noise generation in CUDA [19][20], none of which have scaled the Perlin noise algorithm to *n* dimensionality in their implementations. As an unexplored avenue of research, creating a parallelized *n*D implementation of Perlin noise could lay a foundation for improved noise generation applications in data augmentation and adversarial machine learning, where data dimensionality may be much higher than the current three-dimensional limit.

II - Perlin Noise Algorithm

The Perlin Noise algorithm can be divided into these major sections for each point in the n-dimensional matrix:

(1) Generate distance vectors from each point to its grid space's edges.

(2) Generate the appropriate pseudo-random gradient vectors for each edge.

(3) Calculate the dot product of each distance vector with each gradient vector.

(4) Repeatedly perform interpolation in each dimension between dot products until only one value remains.

In subsections here, we explain the theoretical implementation, pseudocode, and *n*-dimensional scaling behind each step, including the additional fade function applied during step 4. In section III, we explain in depth the constraints of a CUDA implementation of this algorithm.

II.A. Distance Vector Generation

For a given 2-dimensional *m* x *m* grid, a given point located at (x, y) has four edges: (x_0, y_0), (x_1, y_0), (x_0, y_1), and (x_1, y_1), where $x_0 = |x|$, $y_0 = |y|$, $x_1 = (x_0+1)\%m$, and $y_1 = (y_0+1)\%m$. Generalizing this to *n* dimensions, we have (x_0, y_0, z_0, …, $p_{n,0}$), (x_1, y_0, z_0, …, $p_{n,0}$), … (x_1, y_1, z_1, …, $p_{n,1}$). Each *n*-dimensional point has 2^n edges and, therefore, 2^n distance vectors.

We save the effort of calculating and storing all possible combinations of edges by treating our point's coordinates in the grid space as a fraction of the grid limits- that is, for a grid of size m^n, we treat our coordinate set as (x/m, y/m, z/m, ... i_n/m). When treated as such, to calculate distances, our edge vectors are (0, 0, 0, … 0), (0, 0, 0, … 1), … (1, 1, 1, … 1); the digits of the set of all binary numbers of length *n*.

Our pseudocode for *n*-dimensional distance vector generation is as follows:

```
| distances = empty list
| for i in [0, 2^n): // loop through edges
|   for j in [0, n): // loop through dimensions
|     edge = ((i & (1 << (j - 1))) >> (j - 1)); // get the jth digit of binary
i
|     distances[i*n+j] = point[j] - edge // take the distance
```

II.B. Gradient Vector Generation

Expanding gradient vector generation to *n* dimensions proves a more difficult task, as the original and improved Perlin Noise algorithms use a hard-coded list of gradients, the selection of which is determined by a hash function. Generating these lists of gradients in higher dimensionality requires an exponentially increasing space complexity, just as with the distance vectors.

The gradient vectors generally consist of pseudo-random components determined by a hash function. We encountered difficulty understanding and implementing the gradient function in a fashion consistent with other writings, as many implementations offer different gradient generation methods. However, our final implementation uses the hash function's pseudo-randomness to map block and thread offsets to random seeding. It uses the hashed seed to randomly generate a set of gradient values from -1 to 1. Once generated, we divide these values by the Euclidean distance of the resulting vector to normalize the vector to a length of 1, mapping it to a *n*-dimensional unit sphere.

Our pseudocode for *n*-dimensional gradient generation is as follows:

```
|   gradients = empty list
|   srand with hash[(blockDim.x * blockIdx.x + threadIdx.x)%len(hash)]
|   for i in [0,2^n):
|     euclidean_distance = 0
|     for i in [0, n): // loop through dimensions
|        gradients[i*n+j] = rand in (-1, 1)
|        euclidean_distance += gradients[i*n+j]^2
|     euclidean_distance = sqrt(euclidean_distance)
|     for i in [0, n):
|        gradients[i*n+j] = gradients[i*n+j] / euclidean_distance
```

II.C. Dot Product

In this step, we take the dot products of the gradient and distance vectors of each edge to acquire an "influence" vector for the whole matrix point, showing the "influences" of each edge on the point with pseudo-randomization from the gradients. This is a fast step, taking O(n) operations for each point. For any two *n*-dimensional vectors $\vec{a}$ and $\vec{b}$, we define the dot product as:

$$\vec{a} \cdot \vec{b} = a_1b_1+a_2b_2+a_3b_3+\ldots+a_nb_n$$

Or,

$$\vec{a} \cdot \vec{b} = \sum_{i=1}^{n} a_ib_i$$

Our pseudocode for the dot product of the distance and gradient vectors is as follows:

```
| products = empty list
| for i in [0, 2^n):
```

```
|   dot = 0
|   for j in [0, n):
|     dot += distances[i*n+j] * gradients[i*n+j]
|   products[i] = dot
```

II.D. Interpolation

Two different types of interpolation are commonly used to generate Perlin noise: linear interpolation and cosine interpolation. Cosine interpolation is typically preferred as it provides a smoother fade of different values across the grid. However, we will use linear interpolation for our implementation, as we are primarily interested in performance. Our implementation can be easily modified to use cosine interpolation instead of linear interpolation.

Given that we do not have a set dimensionality, our literal number of iterations for interpolation is unknown but can be modeled as $\sum_{i=0}^{n-1} 2^i$. Thus, for $n = 1$, we have one interpolation, for $n = 2$, we have three interpolations, for $n = 3$, we have 7, etc. For a given vector, we can start with interpolations of values at a step of 1, saving the results into the former, and increase our step by a factor of 2 each time until it exceeds the length of the vector. At this point, we return the resulting value at index 0.

Our pseudocode for the interpolation is as follows:

```
| step = 1, dimension = 0
| while step < 2^n:
|   for i in [0, 2^n] incrementing by 2*step:
|     fraction = point[dimension]
|     products[i] = products[i] + fraction * (products[i+step] - products[i])
|   step *= 2
|   dimension++ // move to the next dimension
```

II.E. Fade Function

The fade function defined by Perlin, $\psi(t) = 6t^5 - 15t^4 + 10t^3$, allows the gradient to "fade" as the displacement grows further from the edges, preventing disproportionate influences of the edges [21]. This function was originally defined as the Hermite blending function $\psi(t) = 3t^2 - 2t^3$ but later revised to the fifth-degree polynomial to ensure a continuous second derivative [3]. We can modify our interpolation function to incorporate the fade function implicitly as such:

```
| step = 1, dimension = 0
| while step < 2^n:
|   for i in [0, 2^n] incrementing by 2*step:
|     f = point[dimension] // our fractional displacement
|     f = f*f*f*(f*(f*6-15)+10) // apply fade function
|     prods[i] = prods[i] + f * (prods[i+step] - prods[i])
|   step *= 2
|   dimension++ // move to the next dimension
```

II.F. Full Pseudocode Without Optimizations

Combining all parts of the pseudocode thus far, our pseudocode for calculating the Perlin noise value for a given point is as follows:

```
| perlin(point):
|     // generate gradients and distances
|     distances = empty list
|     gradients = empty list
|     srand with hash[(blockDim.x * blockIdx.x + threadIdx.x)%256]
|     for i in [0, 2^n): // loop through edges
|        euclidean distance = 0
|        for j in [0, n): // loop through dimensions
|           index = i*n+j
|           gradients[index] = rand in (-1, 1)
|           euclidean distance += gradients[index]^2
|           edge = (i & (1 << (j - 1))) >> (j - 1)); // get the jth digit of
binary i
|           distances[index] = edge - point[j] // take the distance
|        euclidean distance = sqrt(euclidean distance)
|        for j in [0, n):
|           gradients[index] = gradient[index] / euclidean distance
|     // calculate dot product
|     prods = empty list
|     for i in [0, 2^n):
|        dot = 0
|        for j in [0, n):
|           dot += edges[i*n+j] * gradients[i*n+j] // take dot product
|        products[i] = dot
|     step = 1, dimension = 0
```

```
|     while step < 2^n:
|        for i in [0, 2^n] incrementing by 2*step:
|           f = point[dimension] // our fractional displacement
|           f = f*f*f*(f*(f*6-15)+10) // apply fade function
|           products[i] = products[i] + f * (products[i+step] - products[i])
|        step *= 2
|        dimension++ // move to the next dimension
|     return products[0] // return the collapsed dot product
```

II.G. Trivial Optimizations and Improved Pseudocode

As is, this is a very inefficient implementation of the noise function in terms of memory, with each point requiring two $n*2^n$ length vectors (gradients and edges) and two n-length vectors (point and dot products). We can eliminate much of the memory overhead by reusing registers and vector space for multiple operations. For example, the gradients vector can be used to hold the dot products while a set-length vector loads the current gradient per each edge. In the linear interpolation step, we can collapse each interpolation along the steps in the gradient vector, holding the dot products until the final interpolation rests at gradients[0]. Additionally, we can forgo creating a distances vector as distances are only used once per edge, and instead, we can use their formula implicitly while calculating dot products. The following shows an improved version of the pseudocode using trivial memory optimizations.

```
| perlin(point):
|   gradients = empty list
|   srand with hash[(blockDim.x * blockIdx.x + threadIdx.x)%256]
|   // calculate gradients
|   for i in [0, 2^n): // loop through edges
|     current_gradient = empty list
|     euclidean_distance = 0
|     for j in [0, n):
|        current_gradent[j] = rand in (-1, 1)
|     // calculate distances and dot products
|     edge = ((i & (1 << -1)) >> - 1);
|     gradients[i] = (current_gradient[0] / euclidean_distance) *
(point[0]-edge); // unroll first addition operation into initialization
|     for j in [1, n):
|        edge = ((i & (1 << (j - 1))) >> (j - 1)); // get the jth digit of
binary i
|        gradients[i] += (point[j]- edge) *
(current_gradient[j]/euclidean_distance)
|     // linear interpolation
|     step = 1, dimension = 0
|     while step < 2^n:
|        for i in [0, 2^n) incrementing by 2*step:
|           f = point[dimension] // our fractional displacement
|           f = f*f*f*(f*(f*6-15)+10) // apply fade function
|           gradients[i] = gradients[i] + f * (gradients[i+step] -
gradients[i])
|        step *= 2
|        dimension++ // move to the next dimension
|   return gradients[0]
```

III - Hypotheses and System Limitations

III.A. System Specifications

For measuring runtimes, we ran our code with the following system specifications:

- CPU: AMD Ryzen 7 4800H
- GPU: NVIDIA GeForce RTX 3060 Laptop
- Compute Capability [22]: 8.6

- Architecture [23]: Ampere
- 30 Streaming Multiprocessors
- Software: Visual Studio 2022 v. 17.9.3, CUDA v. 12.3
- CPU Memory: 16 GB RAM, 475 GB HDD
- GPU Memory [3060 specs]: 6 GB, 192-bit Memory Interface Width

For running the profiler, we ran our code on a separate system with the following specifications:

- CPU: Intel® Core™ i9-10920X -12 Core -3.5GHz Processor
- GPU: NVIDIA GeForce RTX 3070
- Compute Capability [22]: 8.6
- Architecture: Ampere
- 46 Streaming Multiprocessors
- Software: Nsight Compute v. 2024.1, CUDA v. 12.3
- CPU Memory: 32 GB RAM, 1 TB HDD
- GPU Memory: 8 GB, 256-bit Memory Interface Width

With Compute Capability 8.6 on both systems, we also observe the following technical specifications [24]:

- Maximum 128 resident grids per device.
- Maximum 16 resident blocks, 1536 resident threads, and 48 resident warps per SM.
- Maximum 64K registers per thread block, 255 registers per thread.
- For 1-D grids/blocks:
    - 2^{31}-1 maximum blocks per grid.
    - 1024 maximum threads per block.
- Warp size 32.

With this in mind, our system should be able to host over $2.19*10^{12}$ total threads on a 1-dimensional grid and execute 1536 threads concurrently per SM. For a given matrix size $1024 < m^n < 2.19*10^{12}$, we need $m^n/1{,}572{,}864$ warps to fully execute our noise generation on the device.

III.B. Memory & Register Use

One of the primary limitations on the feasibility of parallelizing Perlin noise is the space complexity needed to hold all relevant data in thread registers. While the baseline matrix of size m and dimensionality n will only generate $O(m^n)$ floating-point noise values, the intermediate spaces needed to hold the points on the matrix ($O(n*m^n)$ floating-point values) and the influence vectors ($O(2^n*m^n)$ floating-point values) result in significant memory usage. Table 1 shows the memory space needed for different grid sizes and dimensionality, where row R/T is the registers per thread necessary for the coordinates, influences, and intermediate gradient vector ($2*n+2^n$).

Table 1: Matrix Size (m) & Dimensionality (n) vs. Total Memory Space and Registers per Thread Needed

m	***n***				
	1	2	3	4	5
10	40 B (10 floats)	800 B (200 floats)	12 KB (3000 floats)	160 KB (40,000 floats)	2 MB (500,000 floats)
100	400 B (100 floats)	80 KB (20,000 floats)	12 MB ($3*10^6$ floats)	1.6 GB ($4*10^8$ floats)	200 GB ($5*10^{10}$ floats)
500	2 KB (500 floats)	320 KB (80,000 floats)	1.5 GB ($375*10^6$ floats)	8 GB ($2*10^9$ floats)	625 TB ($1.5625*10^{14}$ floats)
1000	4 KB (1000 floats)	8 MB ($2*10^6$ floats)	12 GB ($3*10^9$ floats)	16 TB ($4*10^{12}$ floats)	20 PB ($5*10^{15}$ floats)
R/T:	4	8	14	20	42

III.C. Hypotheses

With the given optimizations shown in pseudocode, in addition to parallelizing matrix operations, we aim to produce simple yet effective code that operates on large matrix sizes in small dimensionalities and medium matrix sizes in higher dimensionalities. In regard to runtime, we hypothesize that our runtimes of the parallelized noise generation will be much faster than sequential runtimes and fall close to or within $O(n*2^n)$.

IV - Experimental Setup

IV.A. Testing Targets

As this is a runtime- and memory-heavy algorithm at higher dimensionalities, we limited our testing to a matrix size of a maximum side length of 10,000 and a maximum dimensionality of 5. We aimed to test the side lengths 10, 250, 500, 1000, 5000, and 10,000, and the dimensionalities 1, 2, 3, 4, and 5.

IV.B. Parallelization Target

The time complexity of the Perlin noise algorithm for a grid of m^n points is $O(n*2^n*m^n)$, with each point taking $O(n*2^n)$ operations (utilizing an outer loop of 2^n operations- the number of edges- and an inner loop of n operations- the dimensionality). The interpolation step additionally takes $O(2^n\log(n))$ operations per point. The most significant contributing factor to overall runtime is m^n, the matrix space, so this will be the target for our parallelization. This means that for a size-m^n matrix, we need a total of m^n threads, with each thread handling one point.

We originally intended to utilize 1024-block threads (or a number of threads equal to the matrix size, whichever is smaller), but we encountered issues getting CUDA to run kernels with configurations of more than 512 threads per block despite our compute capability allowing for a maximum of 1024 threads per block. Therefore, we will be using a maximum number of threads per block of 512 or the matrix side length (*m*), whichever is smaller.

With an effective maximum threads per block of 512 and a maximum x-dimensionality of thread blocks of 2^{31}-1 (2,147,483,647, the highest possible integer value), certain matrix sizes are infeasible to run within our compute capability on 1-dimensional grids. Table 2 shows the different thread totals and 512-thread blocks needed for various matrix sizes and dimensionalities, with infeasible sizes indicated in bold. We can be certain a given matrix size *m* and dimensionality *n* is infeasible if the *n*th root of the integer maximum is less than *m* (in code: `pow(INT_MAX, 1.0/m) < n`), as the math functions necessary to lay out the matrix size will cause overflow.

Table 2: Total Threads (T) and 512-Thread Blocks (B) Needed for Noise Generation at Different Matrix Sizes and Dimensionalities

m	***n***				
	1	2	3	4	5
10	T: 10 B: 1	T: 100 B: 1	T: 1000 B: 2	T: 10,000 B: 20	T: 100,000 B: 196
100	T: 100 B: 1	T: 10,000 B: 20	T: 1,000,000 B: 1954	T: 10^8 B: 195,312	**T: $>2^{31}$-1** **B: 19531250**
250	T: 250 B: 1	T: 62,500 B: 123	T: 15,625,000 B: 30,518	**T: $>2^{31}$-1** **B: 7629395**	**T: $>2^{31}$-1** **B: 1,907,348,633**
500	T: 500 B: 1	T: 250,000 B: 489	T: 125,000,000 B: 244,141	**T: $>2^{31}$-1** **B: 122,070,313**	**T: $>2^{31}$-1** **B: $>2^{31}$-1**
1000	T: 1000 B: 2	T: 1,000,000 B: 1954	**T: 10^9** **B: 1,953,125** **Matrix overflow due to byte count ($>2^{31}$-1).**	**T: $>2^{31}$-1** **B: 1,953,125,000**	**T: $>2^{31}$-1** **B: $>2^{31}$-1**
5000	T: 5000 B: 10	T: 25,000,000 B: 48,829	**T: $>2^{31}$-1** **B: 244,140,625**	**T: $>2^{31}$-1** **B: $>2^{31}$-1**	**T: $>2^{31}$-1** **B: $>2^{31}$-1**
10,000	T: 10,000 B: 20	T: 10^8 B: 195,312	**T: $>2^{31}$-1** **B: 1,953,125,000**	**T: $>2^{31}$-1** **B: $>2^{31}$-1**	**T: $>2^{31}$-1** **B: $>2^{31}$-1**

IV.C. Algorithm Validation

To act as a proper generation of Perlin noise, our algorithm needs to satisfy three conditions:

(1) Noise values must be uniformly distributed within the range of -1.0 to 1.0.

(2) Noise values must have overall similarity to neighbors.

(3) Noise must appear random on a small scale, but follow similar patterns on larger scales.

To preserve our noise points and ensure our algorithm is valid, our CUDA code outputs to a .txt file, “perlin_out.txt”. This file is formatted so that the first line shows the matrix's size (*m*) and dimensionality (*n*), while each subsequent line holds a single matrix point of noise.

IV.D. Study Limitations

While our gradient vector generation derives from a C implementation of the algorithm, different implementations of Perlin noise use different gradient generation algorithms [3][21], and our use of the n-dimensional unit sphere is a new approach. It is unlikely that our generated noise will be consistent with existing implementations' generations, both as a result of this unique approach to the pseudo-random generation and parallelized random seeding.

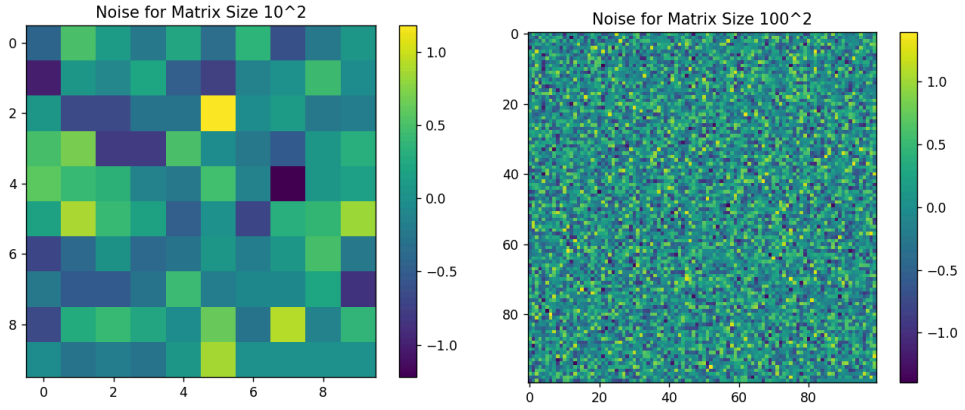
For the feasibility of development time, our visual outputs of perlin noise will be 2-dimensional only. We can represent higher-dimensional noise matrices in the 2-dimensional space as collapsed grids. Still, these grids will not accurately show the adjacency of noise points in higher dimensions and will exist solely for verification purposes of algorithmic integrity.

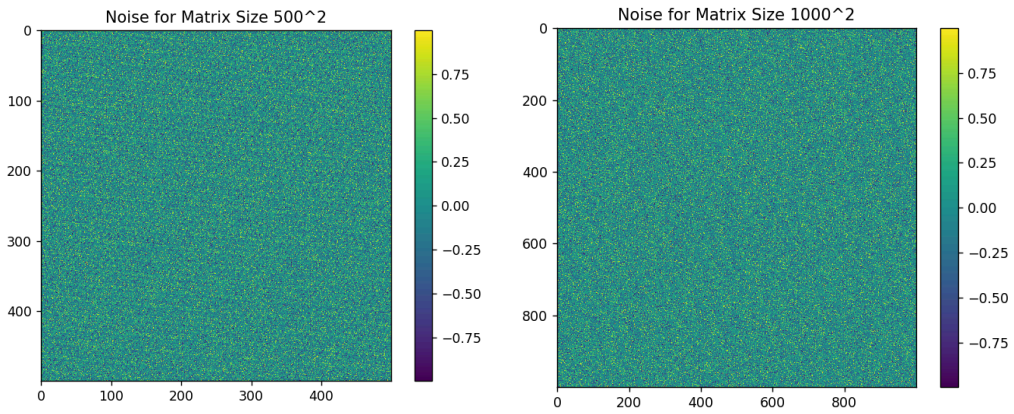
V - Data

V.A. Example Noise Generations

Figures 3, 4, 5, and 6 show the pixel plots of noise generations for grid sizes 10^2, 100^2, 500^2, and 1000^2, using matplotlib's pixel graph function.

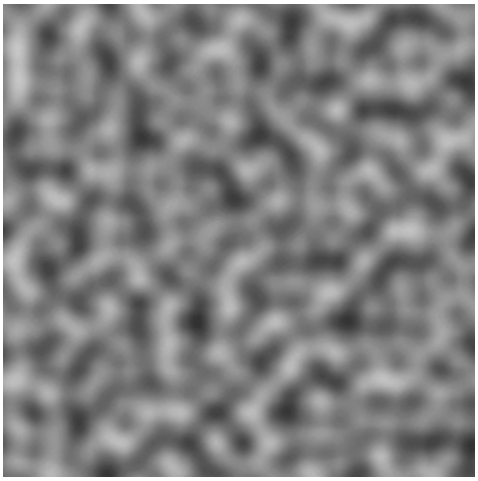
Figures 3, 4, 5, & 6: Noise matrices for sizes 10^2, 100^2, 500^2, and 1000^2





Noise values do appear to be uniformly distributed in the range of -1.0 to 1.0. At m = 500 it becomes clear that, while the generated noise is random, there is some regularity to it, fulfilling the requirement of Perlin noise appearing random on small scales but following notable patterns at larger scales. There is, however, certainly room for improvement in respect to points having overall similarity to neighbors; comparing these noise distributions to an external example of Perlin noise seen in Figure, the noise generated in this algorithm is much rougher. We believe this is a result of the method of gradient vector generation, which we will discuss in section VI.

Figure 7: Example of ideal generated Perlin noise



V.B. Kernel Configurations

Table 3 shows the actual kernel configuration of the CUDA kernels at different matrix sizes and dimensionalities. As previously mentioned, we encountered unexpected errors when running at thread counts of over 512, despite having a maximum of 1024 threads per block on our systems. It is unknown why these errors occurred.

Table 3: Kernel Configurations at Different Matrix Sizes + Dimensionalities

<table>
<tr><td>m</td><td colspan="5">n</td></tr>
<tr><td></td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td></tr>
<tr><td>10</td><td><<<1, 10>>></td><td><<<10, 10>>></td><td><<<100, 10>>></td><td><<<1000, 10>>></td><td><<<10,000, 10>>></td></tr>
<tr><td>100</td><td><<<1, 100>>></td><td><<<100, 100>>></td><td><<<10,000, 100>>></td><td><<<1,000,000, 100>>></td><td rowspan="2">Infeasible (Size)</td></tr>
<tr><td>250</td><td><<<1, 250>>></td><td><<<250, 250>>></td><td><<<62,500, 250>>></td><td rowspan="3">Infeasible (Size)</td></tr>
<tr><td>500</td><td><<<1, 500>>></td><td><<<500, 500>>></td><td><<<250,000, 500>>></td><td rowspan="4">Infeasible (Configuration)</td></tr>
<tr><td>1000</td><td><<<2, 512>>>*</td><td><<<1954, 512>>>*</td><td rowspan="3">Infeasible (Size)</td></tr>
<tr><td>5000</td><td><<<10, 512>>>*</td><td><<<48,829, 512>>>*</td><td>Infeasible (Configuration)</td></tr>
<tr><td>10000</td><td><<<20, 512>>>*</td><td><<<195,313, 512>>>*</td></tr>
</table>

* Indicates a configuration with some idle/skipped threads.

V.C. Runtimes

Table 4 shows the runtimes, in seconds, of the noise generation algorithm as implemented in a sequential rather than parallel fashion, not including the memory operations required for initializing and freeing the noise vector.

Table 4: Sequential runtime (secs.) of Perlin noise algorithm

m	***n***				
	1	2	3	4	5
10	0.000	0.0002	0.00349	0.0682	1.60

100	0.000	0.0125	2.84	671	Infeasible (Size)
250	0.0001	0.0751	44.1	Infeasible (Size)	
500	0.0003	0.293	358.		
1000	0.0003	1.253	Infeasible (Size)		
5000	0.0023	28.6			
10000	0.0028	116.			

Table 5 shows the runtimes, in seconds, of the parallelized noise generation algorithm without memory or file-write operations to 3 significant digits. Table 6 shows the runtimes of the same algorithm with the memory and file-write operations to 3 significant digits. We cannot proceed with performance tests on layouts that are infeasible in the scope of block and thread limitations or that produce errors due to memory space limitations; these are indicated in Tables 4 and 5.

Table 5: Parallel runtime (secs.) of Perlin noise algorithm

(no memory/file-write operations)

m	***n***				
	1	2	3	4	5
10	0.0130	0.199	0.465	Infeasible (Memory Errors)	0.129
100	0.0139	0.203	0.429		Infeasible (Size)
250	0.190	0.211	3.63	Infeasible (Size)	
500	0.190	0.275	30.3		Infeasible (Configuration)
1000	0.187	0.448	Infeasible (Size)		
5000	0.201	5.83		Infeasible (Configuration)	
10000	0.203	22.4			

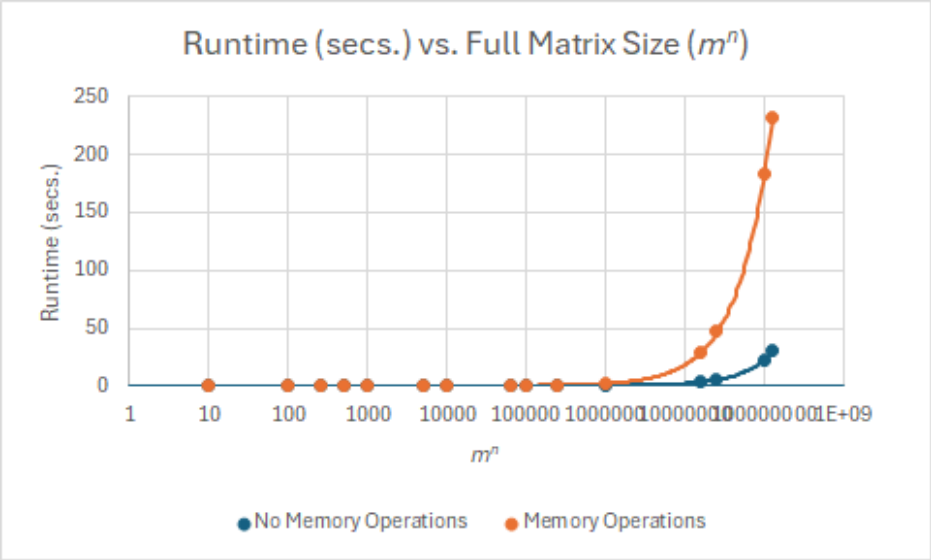
Table 6: Parallel runtime (secs.) of Perlin noise algorithm (including memory/file-write operations)

<table>
<tr><th>m</th><th colspan="5">n</th></tr>
<tr><td></td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td></tr>
<tr><td>10</td><td>0.0148</td><td>0.201</td><td>0.477</td><td>Infeasible (Memory Errors)</td><td>0.292</td></tr>
<tr><td>100</td><td>0.0158</td><td>0.224</td><td>2.06</td><td rowspan="4">Infeasible (Size)</td><td rowspan="2">Infeasible (Size)</td></tr>
<tr><td>250</td><td>0.192</td><td>0.324</td><td>28.9</td></tr>
<tr><td>500</td><td>0.193</td><td>0.714</td><td>232.</td><td rowspan="4">Infeasible (Configuration)</td></tr>
<tr><td>1000</td><td>0.190</td><td>2.06</td><td rowspan="3">Infeasible (Size)</td></tr>
<tr><td>5000</td><td>0.211</td><td>47.5</td><td rowspan="2">Infeasible (Configuration)</td></tr>
<tr><td>10000</td><td>0.222</td><td>183.</td></tr>
</table>

In 4 dimensions, our code always produced runtime errors in copying memory between device and host code despite vector sizes not being particularly extreme. The noise generation worked in 5 dimensions within the bounds of viable matrix sizes, and as such, it is challenging to determine the cause of these errors. The errors do appear to be specific to the cudaMemcpy operation, apparently as an issue with the arguments provided, despite no such errors occurring with the same arguments in different dimensionalities.

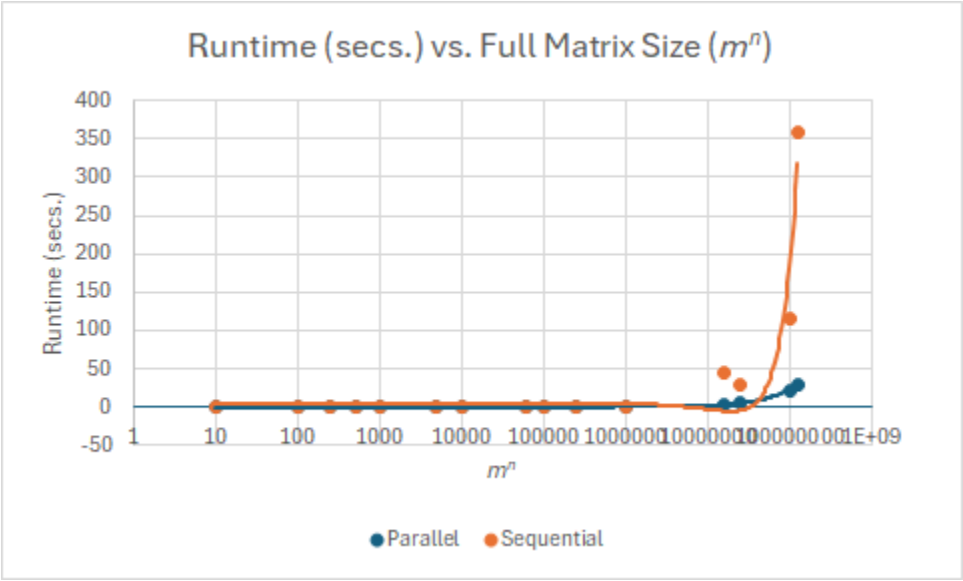
Memory operations and file-writing incurred a very significant runtime overhead in our parallelized implementation. Figure 8 shows the difference in performance between the runtimes with memory operations not included versus included.

Figure 8: Total matrix size (m^n) vs runtime (secs.) of parallel noise generation with and without memory operations



Overall, the CUDA implementation is much faster than the sequential implementation at large dimensionalities and matrix sizes. Figure 9 shows the runtimes of sequential and parallel executions (without memory operations) at different total matrix sizes (m^n).

Figure 9: Sequential vs. CUDA runtimes (secs.) at different total matrix sizes (m^n).



V.D. Profiler

Profiling was performed on a computer with an NVIDIA RTX 3070 GPU. Though we were able to gather data for $n = 2$, the profiler crashes at matrix sizes of 500 and greater for $n = 3$ due to memory constraints, so we could only see trends for matrices of size 100 to 400.

Figure 10 & 11: Memory and DRAM Throughput at different matrix sizes

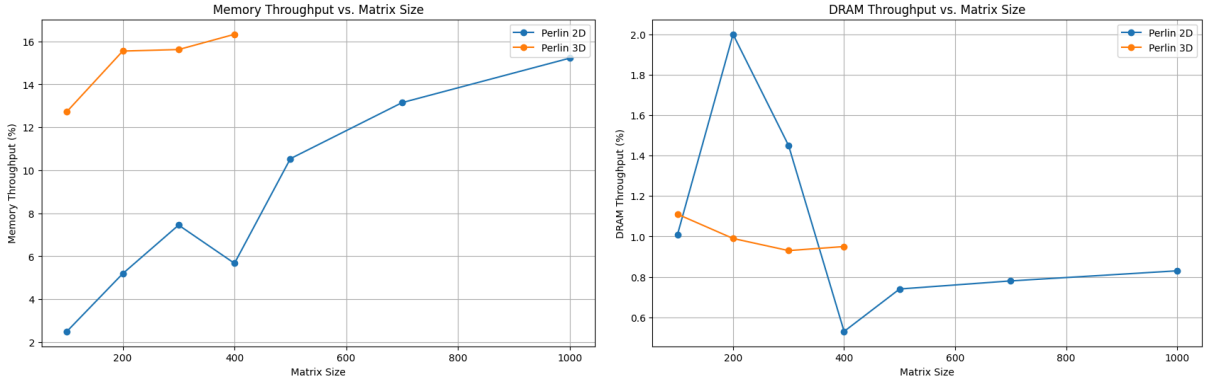
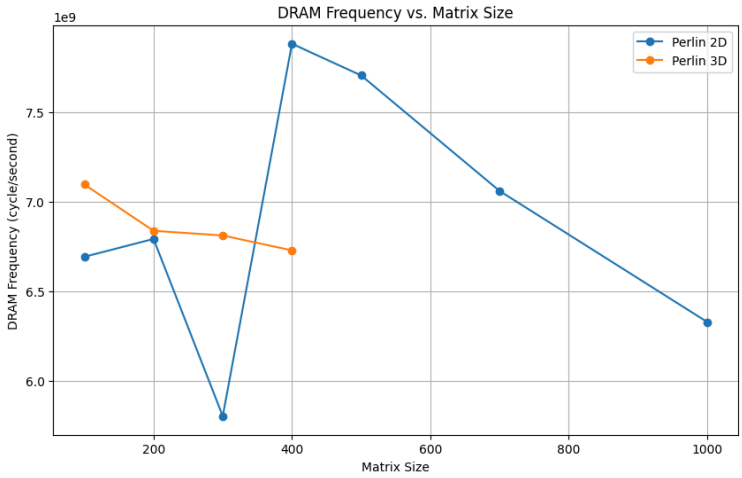


Figure 12: DRAM Frequency at different matrix sizes



Memory and DRAM throughput followed similar patterns, as DRAM throughput contributes to memory. Meanwhile, DRAM frequency significantly increases at matrix size 400 for $n = 2$, while throughput dips. This may be related to the kernel configurations and possible issues with memory access as block and grid sizes increase with matrix size, though the peaking at $m = 400$ is notable for its later implications; it may be symptomatic of a behavior that occurs specifically when we have "full" blocks (that is, with 512 threads, as we were encountering errors at higher than 512 threads per block as previously mentioned). Both throughputs return to

an increasing trend as matrix sizes increase, while DRAM frequency decreases with the increase of matrix, block, and grid size. This pattern is also consistent with Figure 12 with SM Frequency.

Figure 13: Streaming Multiprocessor Frequency at different matrix sizes

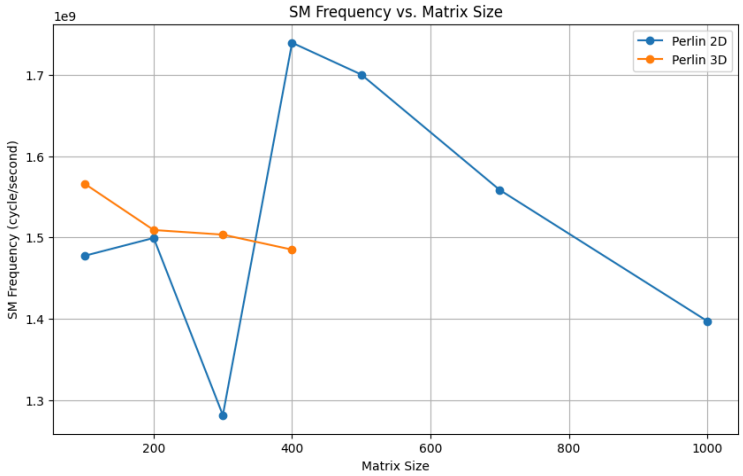
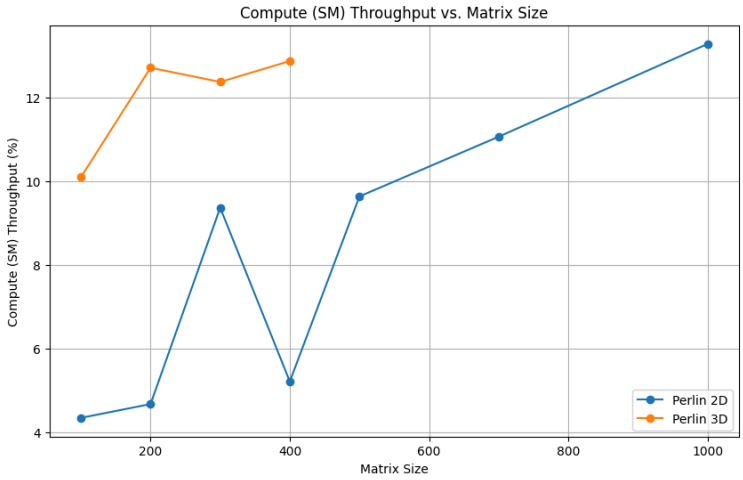
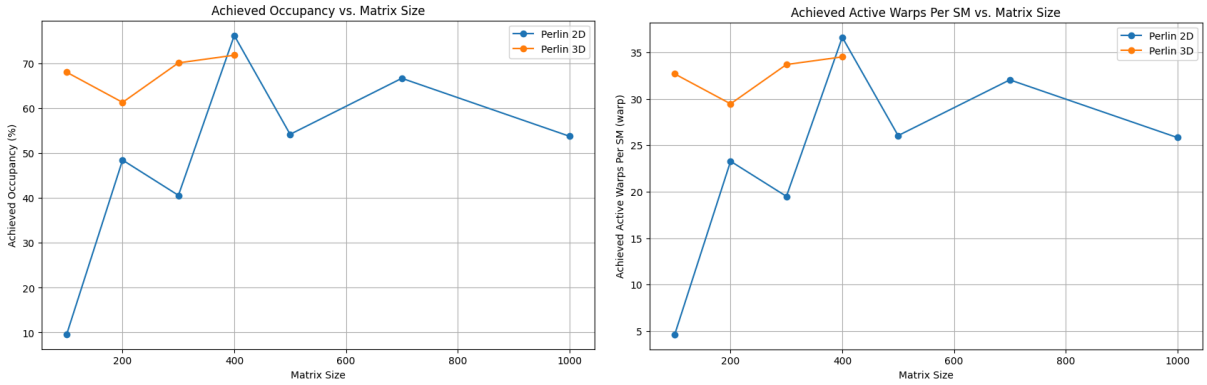


Figure 14: Compute (SM) Throughput at different matrix sizes

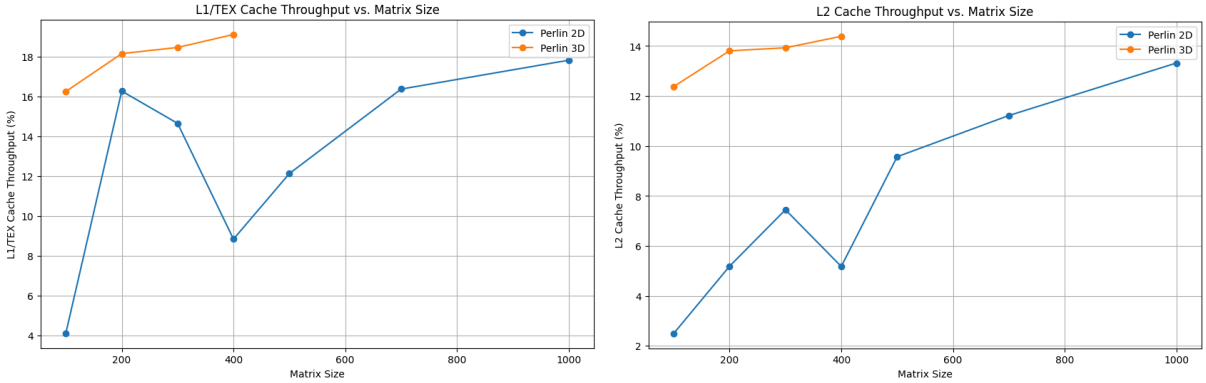


In most scenarios, Streaming Multiprocessor (SM) Frequency and Compute (SM) Throughput should have a direct relationship, but as seen in Figures 13 and 14, these two metrics have an inverse relationship where we can see reversed trends from one as the other increases or decreases. Each kernel's task and computation algorithm remains the same regardless of matrix, grid, or block size. This information helps narrow down possible causes for this issue, since block and grid size were calculated based on matrix size, each increasing incrementally by 100 from matrix sizes 100 to 500 for $n = 2$. While there are evident fluctuations at points 300 and 400 similar to DRAM frequency, resource conflict between threads is the likely factor contributing to the inverse relation we are seeing between SM Frequency and Throughput.

Figure 15 & 16: Achieved occupancy and warps at different matrix sizes

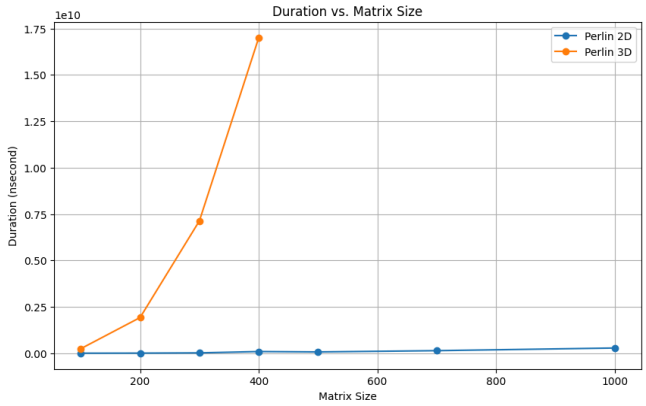


Figures 17 & 18: L1 and L2 Cache Throughput with Increasing Matrix Size



In terms of occupancy and cache throughput shown in Figures 15, 16, 17, and 18, we can see a general increasing trend between these metrics for $n = 3$ and $n = 2$ after the initial spikes when m <= 400. Increasing occupancy and throughput for these is a sensible pattern for increased matrix size (and thereby threads), but eventual resource conflict between threads should lead to a dent in occupancy, which we see occurring around $m = 700$. Several other metrics appear to follow this pattern is well; while not all are increasing with matrix size, the memory and DRAM throughput (Figures 10 & 11), DRAM Frequency (Figure 12), SM Frequency (Figure 13), and SM throughput (Figure 14), all display this tendency after $m > 400$.

Figure 19: Runtime comparison between n = 2 and n = 3, with table



m	n	
	2	3
100	0.001897472	0.243,316,736
200	0.006072960	1.923235072
300	0.019346880	7.135014016
400	0.090550912	17.007651456
500	0.069939328	
700	0.138358720	
1000	0.281416800	

Lastly, for runtime in the profiler, we notice that duration for n = 3 follows the typical growth pattern as the size of the matrix increases, but for n = 2, we can see a small spike at m = 400. This pattern concurs with the previous profiling results, where $m = 400$ induces a spike in the pattern. Due to the scaling of the graph, the jump in runtime may not be evident. Still, from the table to the right of Figure 11, we can see that the runtime increased by over 400% from $m = 300$, whereas previous increments only increased duration by a little over 300%. This increase is also seen to fall off and decrease for increments of matrix size for $m > 400$.

VI - Analysis & Conclusion

With the limitations encountered in this study, it appears infeasible to generate n-dimensional Perlin noise sequentially and in parallel at large grid sizes and high dimensionality. While mitigated by parallelization, the limitations of memory, device grid sizes, and time constraints still pose such a barrier to generation in higher dimensions that there is little scalability to these functions, even with the most significant contributing factor to runtime eliminated. However, many of these barriers may still be overcome with adequate modifications.

VI.A. Runtime Analysis

Our runtimes show poorer performance of the parallelized algorithm than sequential at low m and n values but highly improved parallel performance over sequential at larger m and n values. The poorer performance in lower m and n is likely due to the large memory overhead of copying between the device and host and initializing the kernel, which has a disproportionate impact when the device code and sequential implementation run quickly. Memory and file-writing operations had exponentially increased the impact of performance at higher matrix sizes, showing a significant barrier to scaling. However, with the significant speedup at larger

sizes and dimensionalities (up to 1140% faster at $m = 10$, $n = 5$), parallelization of this algorithm shows incredible promise if the issue of memory thresholds can be mitigated.

VI.B. Profiler Analysis

Though results from the profiler were fairly much consistent with the expected patterns for increasing matrix size (and therefore overall thread count), there are two notable anomalies. First is the spikes occurring at $m = 400$. These spikes are consistent across multiple metrics and often indicate a reversal of an existing pattern of increase or decrease; we suspect this is where the GPU does not have enough resources to support full concurrency for all threads, and memory and runtime must be yielded depending on which thread needs to run. This may also be symptomatic of thrashing; both cache throughputs take a noticeable hit, and it may be the case that the caches are entirely occupied, leading to constant paging and page faults. There is also a case to be made that, given that this value is close approaching the threshold of 512 threads per block wherein we came up against serious errors, the issues of potential memory throttling occurring at $m = 400$ directly influence or cause the thread limitation of 512.

The second anomaly is the inverse relationship between SM Frequency and Compute Throughput. Under normal circumstances, these two variables should share a direct relationship. Our standing theory as to why we observe this inverse relationship is that it ties into the same issue behind the 512 thread limit and $m = 400$ spiking, a memory conflict that reduces the overall efficacy of the GPU in quickly accessing the memory needed to perform operations.

VI.C. Potential Future Optimization - Mitigating the Memory Issue

While our current priority is scoping out the source of the cudaMemcpy errors for n=4 parallel runtimes, there are several memory optimizations we could implement to potentially reduce the space constraints in our code. Three optimizations we could perform are globalizing gradients, using shared memory and tiling, and creating a gradient stencil that can be applied in increments in 256. Unfortunately, meaningful loop unrolling is not a feasible optimization, as the size and divisibility of our loops are unpredictable with respect to changing n and m factors.

VI.C.i. Global Gradient Vector.

One easy optimization that would improve both per-thread memory usage and smoothing of the noise matrix would be to globalize the gradient vector. Each thread would still load one

gradient to the global memory, but by using global memory, we can use the registers used per thread and prevent conflict over resources that might result in reduced occupancy.

Vi.C.ii. Tiled Implementation.

Another optimization could be a tiled approach, with threads handling multiple points and utilizing shared memory to load the current gradients. Grid points could also be moved to shared memory, but they contribute much less towards the overall memory usage than the gradient vectors. Using shared memory would be a tradeoff between the number of potential cache loads and the register usage, but with our memory constraints, it may be a worthwhile tradeoff.

VI.C.iii. Stencil Gradients.

One final optimization- perhaps the most prudent- would be loading a stencil of 256 gradients to global memory, which can be reused at given increments of points. Because gradients are determined by a repeating hash function and, therefore, repeat every 256 values, there is little point in having a full vector of $n*2^n*m^n$ gradients when a vector of 256 would suffice. If this causes noise to become too regular despite grid point randomization, there is additionally the option of scaling this to size $256n$, with each random seeding resulting in n gradient value generations.

References

[1] K. Perlin. “An image synthesizer.” *ACM SIGGRAPH Computer Graphics*, vol. 9, no. 3, 1 Jul. 1985, pp. 287-296, doi: 10.1145/325165.325247.

[2] K. Perlin. “Chapter 4: In the beginning: The Pixel Stream Editor.” In *SIGGRAPH 2002 Course 36 Notes*, 2001.

[3] S. Gustavson. “Simplex noise demystified.” [Online]. Available: https://web.archive.org/web/20230310204125/https://webstaff.itn.liu.se/~stegu/simplexnoise/simplexnoise.pdf.

[4] T. R. Etherington. “Perlin noise as a hierarchical neutral landscape model.” *Web Ecology*, vol. 22, no. 1, 2022, pp. 1-6, doi: https://doi.org/10.5194/we-22-1-2022.

[5] F. Gürler and E. Onbaşioğlu, "Applying Perlin Noise on 3D Hexagonal Tiled Maps," *2022 International Symposium on Multidisciplinary Studies and Innovative Technologies (ISMSIT)*, Ankara, Turkey, 2022, pp. 670-673, doi: 10.1109/ISMSIT56059.2022.9932712.

[6] S. Ahmed and B. Pandey. “Procedural Terrain Generation by Sampling a 2D Monochrom Perlin Noise Map in Unity.” *Asian Journal of Research in Computer Science,* vol. 16, no. 1, 2023, pp. 37-42, doi: 10.9734/ajrcos/2023/v16i1333.

[7] D. Jakes, K. Burrage, C. C. Drovandi, P. Burrage, A. Bueno-Orovio, R. W. dos Santos, B. Rodriguez, and B. A. J. Lawson. “Perlin Noise Generation of Physiologically Realistic Patterns of Fibrosis.” *bioRxiv Preprint,* Jun. 2019, doi: https://doi.org/10.1101/668848.

[8] A. Alreni, G. Momcheva, and S. Pavlov. “Voronoi Diagrams and Perlin Noise for Simulation of Irregular Artefacts in Microscope Scans.” In *Proceedings of the 15th International Joint Conference on Biomedical Engineering Systems and Technologies (BIOSTEC 2022) - BIOIMAGING,* SciTePress, 2022, pp. 117-122, doi: 10.5220/0010833000003123.

[9] S. Michot-Roberto, A. Garcia-Hernández, S. Dopazo-Hilario, and A. Dawson. “The spherical primitive and perlin noise method to recreate realistic aggregate shapes.” *Granular Matter,* vol. 23, no. 41, 2021, doi: https://doi.org/10.1007/s10035-021-01105-6.

[10] F. Conde-Rodríguez, Á. L. García-Fernández, and J. C. Torres. “Modelling Material Microstructure Using the Perlin Noise Function.” *Computer Graphics Forum*, vol. 40, no. 1, 2021, pp. 195-208, doi: https://doi.org/10.1111/cgf.14182.

[11] I. Parberry. “Designer worlds: Procedural generation of infinite terrain from real-world elevation data.” *Journal of Computer Graphics Techniques*, vol. 3, no. 1, 2014.

[12] T. Kaneko and T. Harada. “Noise Robust Generative Adversarial Networks.” In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, Jun. 2020, pp. 8404-8414

[13] A. Kurakin, I. Goodfellow, and S. Bengio. “Adversarial Machine Learning at Scale.” *arXiv preprint*, 2016, doi: https://doi.org/10.48550/arXiv.1611.01236.

[14] A. S. Hashemi and S. Mozaffari. “Secure deep neural networks using adversarial image generation and training with Noise-GAN.” *Computers & Security,* vol. 86, 2019, pp. 372-387, doi: https://doi.org/10.1016/j.cose.2019.06.012.

[15] T. Y. Liu, Y. Yang, and B. Mirzasoleiman. “Friendly Noise against Adversarial Noise: A Powerful Defense against Data Poisoning Attack.” In *Advances in Neural Information Processing Systems 35 (NeurIPS 2022),* 2022, pp.11947-11959.

[16] N. Inoue, E. Yamagata and H. Kataoka, "Initialization Using Perlin Noise for Training Networks with a Limited Amount of Data," *2020 25th International Conference on Pattern Recognition (ICPR)*, Milan, Italy, 2021, pp. 1023-1028, doi: 10.1109/ICPR48806.2021.9412955.

[17] W. Bazuhair and W. Lee, "Detecting Malign Encrypted Network Traffic Using Perlin Noise and Convolutional Neural Network," *2020 10th Annual Computing and Communication Workshop and Conference (CCWC)*, Las Vegas, NV, USA, 2020, pp. 0200-0206, doi: 10.1109/CCWC47524.2020.9031116.

[18] H. Bae, C. Kim, N. Kim, B. Park, N. Kim, J. B. Seo, & S. M. Lee. “A Perlin Noise-Based Augmentation Strategy for Deep Learning with Small Data Samples of HRCT Images.” *Sci Rep* vol. 8, no. 17687, 2018, doi: https://doi.org/10.1038/s41598-018-36047-2.

[19] H. Li, X. Tuo, Y. Liu, and X. Jiang. “A Parallel Algorithm Using Perlin Noise Superposition Method for Terrain Generation Based on CUDA architecture.” In *Proceedings of the 2015 International Conference on Materials Engineering and Information Technology Applications*, Aug. 2015, pp. 967-974, doi: 10.2991/meita-15.2015.183.

[20] E. Skejić, D. Demirović, and D. Begić. “Evaluation of Perlin Noise using NVIDIA CUDA Platform.” *Elektrotehniski Vestnik*, vol.. 87, no. 5, 2020, pp. 260-266.

[21] K. Perlin “Chapter 2: Noise Hardware.” In *SIGGRAPH 2002 Course 36 Notes*, 2002.

[22] “Your GPU Compute Capability.” *NVIDIA Developer.* [Online]. Available: https://developer.nvidia.com/cuda-gpus.

[23] “NVIDIA GeForce RTX 3060 Mobile.” *TechPowerUp.* [Online]. Available: https://www.techpowerup.com/gpu-specs/geforce-rtx-3060-mobile.c3757.

[24] “CUDA C++ Programming Guide.” *CUDA,* 2 Mar. 2024. [Online]. Available: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#features-and-technical-specifications-technical-specifications-per-compute-capability.