Fingerprinting With Atomic Counters in Upcoming Web Graphics Compute APIs

Sabrina Jiang
Supervised by Dr. Hovav Shacham

A thesis presented for the Turing Scholar honors distinction.
This document has been approved by my advisor for distribution to my committee.

The University of Texas At Austin
May 2020

Note: Because this paper is so data-heavy, it comes with nearly 60 pages of detailed results. These can be found after page 18, and corresponding figures will be referenced from this file.
These charts after page 18 can be printed separately if the reader chooses to. We recommend that if the reader does so, to print them single-sided so that the graphs may be compared side-by-side.

1 Abstract

As modern browsers move towards adding compute features to graphics APIs, evidence shows that the attack surface for graphics card fingerprinting increases, most notably due to support for atomic operations within the GPU.

Browser Fingerprinting is a widely used technique to track users accessing a website, without the users signing in or authenticating their identity in any way. This is a violation of the user's privacy, who may not know and may not have consented to being tracked. In particular, WebGL, a browser graphics library, has been shown to be a high-entropy fingerprinting factor.

Recently, browsers have moved towards adding new graphics compute features. One of these endeavors is WebGL 2.0 Compute, which includes support for atomic counters. In this research, we give evidence that adding atomic counters to browser graphics libraries increases the fingerprinting attack surface. To do so, we set up a dummy website, `sabrinaj.me`, with three sets of tests that leverage the atomic counters. Instructions for how to enable WebGL 2.0 Compute features and run the tests are enumerated in section 3.1.

The first test involves discovering the order with which the graphics card renders pixels in an image. The second and third tests involve using the atomic counter as a homebrewed implicit timer, which we show can be constructed by creating high contention for the counter. The second test involves timing an arbitrary task, and the third test involves probing the graphics card's caching behavior by timing walks of varying sizes through a random buffer.

The preliminary results from these three tests indicate that the atomic counters in web graphics libraries have potential to become flexible fingerprinting tools.

2 Introduction

2.1 Browser Fingerprinting and WebGL

Browser Fingerprinting [3, 4, 5, 7] is a widely used technique to track users accessing a website. The website fingerprints the user by collecting information about the user's unique browser, or running tests on the user's unique hardware. Then, it stores this information, and repeats these actions each time the website is loaded. If two accesses have the same unique fingerprint, then it is likely that the same user accessed the site. Thus, the website can identify each user without using cookies or other more obvious methods of tracking - all without the user signing in or authenticating their identity in any way. This is a violation of the user's privacy, who may not know and may not have consented to being tracked.

A popular fingerprinting technique is the rendering behavior of WebGL[6], a Javascript API for rendering graphics in the browser. WebGL allows websites to leverage the capabilities of a user's graphics card (GPU), and thus is used in many sites that have special 2D scenes or 3D models.

WebGL has been shown to be a very high-entropy fingerprint, meaning its behavior greatly improves the distinguishability of fingerprints. For example, in Figure 1, the same simple triangle was rendered on two different machines. The image on the left was rendered by a 2016 Macbook Pro; the one on the right was rendered by a desktop computer in the Computer Science lab at UT Austin. The image below them shows pixels in which the two differed in black.

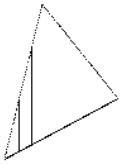
Recent research shows that the culprit behind this entropy could be differences in how graphics cards perform floating-point operations. As a result, some have suggested reducing the amount of floating point operations done by WebGL, and instead replacing as many as possible with integer operations - an approach taken in UniGL [1]. However, in this research, we show that as compute capabilities are added to browser graphics libraries, the attack surface could increase without using floats at all, making mitigations like UniGL insufficient.

2.2 WebGL Semantics

Next, to give a brief overview of WebGL semantics, we will describe its overall three-step rendering process, and how a programmer would go about adding WebGL to their website.

(a) The triangle, rendered by a Macbook Pro.

(b) The triangle, rendered by a lab machine in the UT Austin Computer Science Lab.



(c) Pixels where they differ, highlighted in black.

Figure 1: A pixel-by-pixel comparison of two machines rendering the same triangle

When a programmer adds graphics to their website with WebGL, they would begin by preparing input to tell a graphics card what to render.

1. For the first stage, vertex rendering, they would write an OpenGL Shading Language (GLSL) program called a vertex shader. The vertex shader calculates coordinates in 3D space describing the structure of the object(s) to render. It is called once for each vertex - each vertex runs the same shader code.

   Additionally, the programmer can buffer arrays with input data to the GPU, typically through an Attribute buffer. The input data might be the coordinates of the object(s) to render.

   There are other ways that data can be buffered in, such as using Uniform buffers and Textures, which are read-only inputs. However, we used Attribute buffers in this work, which are read-write. All buffers are shared across all running threads of execution.

2. The second stage, rasterization and interpolation, is handled under the hood by WebGL. This stages involves turning the coordinates calculated

in the vertex shading stages a 2D image with pixels.

3. For the third stage, fragment rendering, the programmer would write another GLSL program called a fragment shader. Similar to the vertex shader running once per vertex, the fragment shader runs once per pixel, or fragment, of the final image. For example, it might change the color of the output pixel with a programmer-specified formula.

   Again, the programmer can buffer in arrays with input data. The graphics card may read or write from these buffers during the rendering process. Afterward, the programmer can also read output data from these buffers. These buffers are shared among all running threads of execution.

2.3 Browser GPU Compute Libraries and Atomic Counters

In addition to the rendering functionality above, developers are moving towards adding new features for GPU compute. Projects like WebGL 2.0 Compute [8] and WebGPU [10] are hoping to allow browsers to perform a wider range of GPU workloads, such as those used in machine learning, gaming, image processing, and cryptocurrency, and more. These upcoming features can be run in the "beta" versions of popular browsers by enabling a feature flag - including the Chrome Canary and Safari Technology Preview.

Among the features WebGL 2.0 Compute implements is atomic counters. For an operation to be atomic, it is mutually exclusive, and appears instant to all threads of execution. If one thread is reading or writing to a memory location, others cannot read or write to that location halfway through - instead, they must wait or retry their operation in turn. Such atomic primitives, like counters, are very useful for implementing synchronous behavior on GPUs, a new capability for browsers.

To use these atomic counters, the programmer must allocate some memory for them by buffering an empty array in as input, similar to how they would buffer in input data. Next, in the fragment shader, they must declare the aforementioned buffer as an atomic counter. Then, they can read, increment, or decrement the counter freely in the shader program. Like the buffers for input and output used in the shaders, these atomic counters are shared among all running threads of execution.

In this research, we explore using WebGL 2.0 Compute atomic counters

as a fingerprinting mechanism, through simple but revealing tests. First, we create a fingerprint using atomic counters to discover the rendering order of the pixels in an image. Next, we explore creating homebrewed timers with atomic counters, and create more fingerprints by leveraging the timer to reveal private and uniqute information about graphic cards' performance and caching behavior.

3 Design and Evaluation

3.1 Overarching Design

Next, we discuss the overarching design of all experiments laid out in this paper.

We used the same vertex shader for all tests, since most experiments were run in the fragment shader. The vertex shader draws a square in space, with coordinates (1, 1), (1, -1), (-1, -,1), (-1, 1). These are simply here to make something to render onto later in the fragment rendering stage.

The fingerprinting experiments were conducted in the fragment shader, since it comprises the final stage of the rendering process, and its results are the final output of the rendering process. Therefore, a variety of fragment shaders were written, all performing different tasks on different inputs.

To simulate how these tests might work in practice, we set up a website, `sabrinaj.me`, that runs and records three sets of fingerprinting tests on a site user's graphics card. Anyone can run these tests, if they follow these steps:

1. Be on a Windows machine. WebGL 2.0 Compute only works in windows.
2. Download the Google Chrome Canary browser.
3. Set two flags in the Canary to enable WebGL 2.0 Compute: "WebGL 2.0 Compute" option to "Enabled", and the "Choose ANGLE graphics backend" to "OpenGL". This can be done from teh `chrome://flags` page.
4. Restart the Canary.
5. Go to `sabrinaj.me` in the Canary. There are three buttons that run three sets of tests.

The three sets of tests we chose involve leveraging the WebGL 2.0 Compute atomic counters to:

1. Discover the fragment, or pixel, rendering order.
2. "Time" a simple calculation workload, using the atomic counter as a homebrewed timer.
3. Probe for caching behavior, using the atomic counter as a homebrewed timer.

Thanks to our friends who spared their valuable time to run these tests, we have data for several graphics cards, both integrated graphics processors and dedicated graphics cards:

- Intel RHD Graphics 520 OpenGL 44 Core
- Intel RHD Graphics 520 OpenGL 45 Core
- Intel RHD Graphics 620 OpenGL 45 Core
- Intel RUHD Graphics 620 OpenGL 45 Core
- NVIDIA GeForce GTX 770 PCIe SSE2 OpenGL 45 core
- NVIDIA GeForce GTX 1060 6GB PCIe SSE2 OpenGL 45 core
- NVIDIA GeForce GTX 1070 PCIe SSE2 OpenGL 45 core
- NVIDIA GeForce GTX 1080 PCIe SSE2 OpenGL 45 core
- NVIDIA GeForce RTX 2080 SUPER PCIe SSE2 OpenGL 45 core
- NVIDIA Quadro P620 PCIe SSE2 OpenGL 45 core

Note that the Intel graphics cards are integrated graphics cards, while the NVIDIA cards are dedicated GPUs.

3.2 Test Set 1: Render Order

The first fingerprinting test conducted discovers the order that the graphics card renders fragments, or pixels, in.

To do so, we incremented a single atomic counter in the fragment shader program. Recall that because the atomic counter is shared among all running fragments, as each fragment get rendered, it can read the value on the shared counter and increment it. It can then set its final color based on the value it read - if the value it read is very low, it can set its RGB value low. If the value

it read is very high, it can set its RGB value high. Because the counter was atomic, no increments will be lost, and no two fragments will read the same count. This process creates a black-and-white image where pixels that render first are blacker, while pixels that render later are whiter.

To get a more complete idea of the order in which graphics cards render fragments, we varied the dimensions of the image being rendered. The dimensions tested were: 100 by 100, 256 by 256, 256 by 512, 512 by 256, 512 by 512, 400 by 400, and 800 by 800.

The reader is highly recommended to reference pages 1-7 of the Tables and Charts file to view the collected images for each graphics card.

Some cards rendered in the same pattern no matter the size, while some broke up larger tasks into smaller, set-size chunks in a predictable way. All tested integrated graphics processors rendered the same pattern. They started at the bottom of the image, rendered up to the top left corner, then returned to the bottom right corner, and rendered up to the top. On the other hand, dedicated GPUs had vastly different rendering patterns. The 1060, 1070, 1080, and 2080 all generally split up larger-sized images into smaller chunks:

- The 1060 broke up large sizes into 256 by 256 squares, and rendered those from bottom left to top right.
- The 1070 and 1080 broke up large sizes into 256 by 512 squares, and rendered those from bottom left to top right.
- The 2080 broke up large sizes into 512 by 512 squares, and rendered those from bottom left to top right.
- The P620 and 770 both rendered in a similar overall manner as the integrated graphics processors.

Additionally, some of the GPUs also had underlying patterns in addition to breaking up larger images into chunks.

- The 1080, 770, P620, created a checkerboard-like patterns of 16 by 16 squares all over the images.
- The 1060 and 1070 created vertical stripes all over the images.
- The 2080 created an almost polka-dot-like speckled pattern all over the images.

The reader is left to observe pages 1-7 of the Tables and Charts file and draw more qualitative conclusions themselves.

Taking all these results into account, it is clear that this test could serve as a fingerprint. More data should be taken with a much wider range of graphics cards in order to discover how different cards render, but these preliminary tests spark concern, given that it all the dedicated GPUs created distinct and unique patterns, and that their patterns can be easily distinguished from the integrated graphics patterns.

Additionally, while the same, predictable patterns are rendered by each given graphics card, their results are not exactly the same from render to render. For this fingerprint to be more reliable, this technique should be further refined to account for these small differences - perhaps by running the experiment multiple times and averaging the result, and/or running basic pattern-matching code on the images.

Observe that these rendering order tests do not rely on differences in floating-point operations in any way, since the atomic counter is an unsigned integer. Therefore, WebGL floating-point fingerprinting mitigations such as UniGL are, well, rendered ineffective against these attacks, and new mitigations will need to be taken.

3.3 Timers From Atomic Counters

The next tests and corresponding fingerprints involve using an atomic counter as a homebrewed timer. Timers in WebGL have already been studied by the security research community, with several timing mechanisms already disabled in WebGL due to serious security vulnerabilities [2] they cause, leaving WebGL practically devoid of any timing features at all. However, our results indicate that the the addition of atomic counters into the browser's graphics library could reintroduce fine-grain timers - useful for exploring the internal behavior of graphics cards, and potentially reviving all the attacks previously discovered.

In our methodology, the counter itself serves as an implicit timer. Firstly, in the fragment shader, all fragments except for one are tasked with incrementing the single shared atomic counter several thousand times each, creating extreme contention for the counter. Note that both the size of the image and the number of times each counting fragment increments the counter matter here, since if either is too low, the counter will not be incremented high enough - and the counting will no continue for long enough. Similar to how a stopwatch that can

only count up to one second is not exactly a useful stopwatch. For convenience, these fragments will be called "counting fragments."

The sole fragment that does not increment the counter instead reads from the counter, performs a task, then reads it again. Finally, if the working fragment takes the difference between the two counter values it read and outputs the result to a buffer, the resulting value serves as benchmark of the time the task took compared to how quickly the counter was incremented. The result does not directly equate a wall time, but it serves as an implicit time taken by the task. This fragment will be called the "working fragment."

To choose which fragment to designate as the working fragment, observe from the first set of experiments that the bottom-left pixels are generally rendered first. Since the working fragment should begin is task as the shared counter starts incrementing, and finish the task before all the counting fragments stop counting, we chose the working fragment to be in the bottom-left corner. The task selected for the working fragment should be a calculation with a difficult-to-predict result, to avoid any compiler optimizations of the shader code that may interfere with the calculation speed.

If the counter is incremented at a consistent pace, we can expect the difference between the two observed counts to increase linearly with the task size - that is, if some task is done twice, the counter difference should increase twofold. If the task is done ten times, the counter difference should increase tenfold.

3.4 Test Set 2: Timing Tasks with Atomic Counters

The second set of tests used the technique from the previous subsection to time an arbitrary task - in this case, calculating sine values.

The tests were all conducted on images of 300 by 300 pixels in size for a total of 89,999 counting pixels. In practice, this was roughly the smallest size image before results began deviating downwards from the consistent results from larger sizes, perhaps indicating that there was not enough contention for the counter, or that the counting fragments finished counting too quickly.

Each counting pixel was asked to increment the shared atomic counter 3,000 times, for a total count of 269,997,000. Notably, some cards were able to handle upwards of 100,000 increments per fragment, while others, once tasked with incrementing more than 5,000 times each, would cause WebGL to throw errors, stop rendering, freeze the screen, or flash the entire screen black. This is in and of itself a fingerprint - how high each fragment in a card can increment

a counter under high contention - but it negatively impacts user experience, making it suboptimal.

The fragment chosen to be the working fragment was the pixel at index (20, 20), the bottom-left of the image. Since calculating the sine is complex and therefore unlikely to be optimized by a compiler, the working fragment's task was to calculate the sine of a value, then calculate the sine of the result, then the sine of that, repeatedly. We recorded the difference in counter value at 0 sine calculations in a row, 10,000 calculations, 20,000 calculations, and so on, up to 100,000 calculations.

The reader is highly recommended to reference pages 8-17 to view the results of this experiment. Each graphics card has a single page with its unique results.

Firstly, the tests show that the size of the task, or number of times the sine was calculated, generally increases linearly with the counter difference read in all graphics cards that were tested, as expected. This makes a case that the atomic counter as a homebrewed timer behaves consistently, and makes for a predictable and reliable timer.

Secondly, many of the tested cards had differing counter differences per sine calculation - that is, the slopes of their resulting graphs differed, as listed in the table below. This indicates that there is potential for this test to be a fingerprint, in addition to the rendering order test from before.

Increments per Sine Calculation Comparison	
Graphics Card Name	Increments per Sine Calculation
Intel RHD Graphics 520 OpenGL 44 Core	15
Intel RHD Graphics 520 OpenGL 45 Core	1.5
Intel RHD Graphics 620 OpenGL 45 Core	1.5
Intel RUHD Graphics 620 OpenGL 45 Core	1.5
NVIDIA GeForce GTX 770 ...	1,600
NVIDIA GeForce GTX 1060 ...	1,600
NVIDIA GeForce GTX 1070 ...	880
NVIDIA GeForce GTX 1080 ...	1,200
NVIDIA GeForce RTX 2080 ...	760
NVIDIA Quadro P620 PCIe ...	800

3.5 Test Set 3: Revealing Caching Behavior with Timers

For the last set of tests, we attempt to observe a graphics card's caching behavior with the technique described above that uses a shared atomic counter as a timer.

Again, we create a 300 by 300 image with counting fragments incrementing the atomic counter 3,000 times each, and the working fragment remains at (20, 20). This time, instead of calculating sines in a loop, the working fragment performs a different task.

Recall that the programmer can buffer in input data to the fragment shader, and later read output from the same buffer after the shader has finished rendering. We will buffer in a large array of 32-bit unsigned integers with length 1,000,000. This totals to 4 megabytes of data, and is likely to be far too large to fit in a first-level cache. The contents of the array will be the values 0 through 999,999; in other words, indexes into the array. The values might be shuffled around in the array.

The working fragment's task will be to walk, or step through, this buffer. It will first start at its pixel index - in this case, 20 - and read the buffer at index 20. The value of the buffer at index 20 is a second index into the buffer, and the working fragment will read the buffer at this second index, getting a third index. It will read the buffer at this third index, getting a fourth index, and so on and so forth. It will continue stepping through the buffer for some number of steps. Because the values are shuffled in the buffer, predicting where in the buffer will be accessed next cannot be done without actually reading the buffer.

Next, we can shuffle the values around in interesting ways that reveal information about the graphics card's behavior. First, we can randomly arrange the values in the array, but make it so that walking through the buffer creates a cycle. If the cycle length is 10, no matter where the working fragment starts, it will return to where it started every 10 steps.

Here is an example of an array of length 6, where after every three steps, the working fragment would always return to where it began:

$$[3, 4, 1, 5, 2, 0]$$

Thus, the cycle length in this example array is length three.

Next, we will discuss how the graphics card's cache might behave given this task. Let the cache size be S, and the cycle length be L.

If L is less than or equal to S, we can expect to see that the time needed to step the first L steps is greater than the next L steps. For the first L steps, the

traversed parts of the buffer need to be cached. But after the first L steps, all parts of the buffer that will ever been traversed have been already been brought into the cache, making these steps take less time. The working set fits inside the cache.

As L increases beyond S, we can still expect to see the time needed to step the first L steps to be high. However, the time to step the next L steps may still stay high, because the working set is too big to fit in the cache. Eventually, the cache will run out of room, evict something, and take extra time to cache the new value - stalling execution.

This means that running this test with different cycle lengths L may shed light a graphics card's cache size, S.

For this final set of tests, we tested each graphics card with cycle lengths of 50, 100 150, and 200. For each cycle length, we used the atomic counter as a timer with the methodology described before, and timed walk lengths of 0, 20, 40, 60, and so on, up to 500 steps.

The reader is highly recommended to reference pages 18-57 to view the results of this experiment. Each graphics card has four pages of its unique results of this experiment.

In dedicated GPUs, we observed behavior in line with the scenarios described above. For each cycle length, the slope, or increments per step, started very high, until the cycle length was reached. After which, the slope decreased. The inflection points were clearly at the cycle lengths. This is a strong indication that these atomic counter timers are able to detect caching behavior.

In integrated graphics processors, however, no inflection points were seen. The increments per step ratio stay constant, indicating that integrated graphics processor caches may not be as performant or advanced as dedicated GPU caches for such read/write data buffers - perhaps it is much smaller, or has a less efficient caching policy for such a workload. This is very plausible, given that dedicated GPUs are typically much more powerful than integrated graphics processors.

To get to the bottom of the differences in performance, more in-depth experimentation should be done here, with varying hardware, cycle lengths, and working fragment tasks. However, even with these preliminary tests, it is clear that these could serve as a fingerprinting mechanism. As with the sine-calculating test, the increments per step metric differed between each card, and even between each cycle length.

3.6 Extended Testing of Caching Behavior

Finally, we will discuss extended testing on the NVIDIA 1070 card that seeks to take a closer look at its caching behavior.

The extended testing consisted of collecting data for more cycle lengths in between 150 and 200. This was chosen because in this particular card, the slope after the inflection point stayed low in the cycles of length 50, 100, and 150 at about 59,000 increments per step. However, when the cycle length was 200, the slope after the inflection point jumped to 90,000 increments per step - possibly indicating that the cache can simultaneously hold between 150 and 200 32-bit unsigned integers.

Thus, more tests for cycle lengths of 150, 160, 170, 180, 190, and 200 were taken, at more frequent intervals of 10, rather than 20, steps.

The reader is highly recommended to view page 58 of the Tables and Charts file to view the results from this test, where the resulting graphs have been arranged on a single page so they can be compared side-by-side. There are number of interesting observations here:

- The data points, each representing a single test, generally create lines of certain reoccurring slopes. The graphs have been annotated with the lines, and the lines have been labelled with their approximate slopes in increments per step, so that the graphs can be better compared.

- Beginning at cycle length 150, the slope beyond 150 steps is almost always 59,000 increments per step. There are a couple outliers, which form a line with a slope of about 76,000. These particular tasks may have taken longer because the buffer happened to be shuffled in a way that, combined with the working set of size 150, made it difficult for the GPU to cache the entire working set as effectively.

- Next, at a cycle length of 160, the slope beyond 160 steps is usually 59,000, but a few more tests jump up the 76,000 increments per step line. At a cycle length of 170, even more tests end up on the 76,000 line. This could be because as the working set increases and approaches the cache size, the GPU is less likely to be able to cache the entire working set.

- At a cycle length of 180, we observed a couple more tests beginning to form a new line, this one with slope of about 94,000 increments per cycle.

- At a cycle length of 190, yet another line seems to appear, with slope 160,000. At this point, the first line with slope 59,000 has disappeared - indicating that with an arbitrary working set of about 190, the cache is no longer able to operate at its peak efficiency - or at least, what we have observed its peak efficiency to be.
- At cycle length of 200, only a few tests remained on the 76,000 increments per step line.

It is unclear why the tests create such predictable linear patterns with such consistent slopes. Again, more carefully constructed tests using these atomic counter timers must be done before more conclusions about the 1070's cache behavior can be made. If such experiments are run for each unique graphics card, and each card's caching behavior is profiled to great extent, there is potential for a high-entropy fingerprint, since each card's caching behavior is likely to be unique.

The final set of tests seeks to discover the cache line size. To do so, the cycle-walking experiment will be altered slightly. Recall that because the programmer controls the buffer, they can control where in the buffer the working fragment will visit next. In previous tests, the cycle always had a set length, but the steps in the cycle were randomly distributed around the buffer. This time, the steps in the cycle will be linear; that is, they will be consecutive. The working fragment might start at index i, then step to $i + 1$, then step to $i + 2$, until the number of steps it has taken equates the cycle length, in which it would return to i and repeat. One might think of it as having a stride length of 1, except when returning to the starting point of the cycle.

Here is an example of an array of length 6, where the cycle length is 3, and the steps have stride length 1. For example, starting at index 0, the working fragment would proceed to index 1, then index 2, and then return to 0 and repeat.

[1, 2, 0, 4, 5, 3]

Next, consider if the stride length is instead 2. The working fragment might start at index i, then step to $i + 2$, then step to $i + 4$, until the number of steps it has taken equates the cycle length, in which it would return to i and repeat. Here is an example of an array of length 6, where the cycle length is still 3, and the steps have a stride length of 2. For example, starting at index 0, the

working fragment would proceed to index 2, then index 4, and then return to 0 and repeat.

$$[2, 3, 4, 5, 0, 1]$$

The buffer consists of 4-byte unsigned integers, so the values in the buffer are likely 4-byte aligned. Consider the expected behavior if the cache line size is 8 bytes, and the stride length is 1. When the first index i is accessed by the working fragment, the 8 bytes at index i are cached. Since the cache line size is 8 bytes, both the indices i and $i + 1$ are cached. Then, then the working fragment proceeds to index $i+1$, its value is conveniently already cached, making accessing it faster. This means that, up until the cycle length, the increments per step slope would be low. Next, consider the same scenario, except where stride length is 2. When the first index i is accessed by the working fragment, the 8 bytes at index i are cached. Since the cache line size is 8 bytes, both the indices i and $i+1$ are cached. However, the working fragment proceeds to index $i + 2$, rather than $i + 1$. Because the stride length exceeded the cache line size, accessing the buffer at $i + 2$ will take longer, since it has not yet been cached. This means that, up until the cycle length, the increments per step slope would be much higher.

Using this technique, by testing different stride lengths and observing the increments per step slope of the results, we might be able to discover information about the cache line size. In our tests, we used a cycle length of 150 steps, and tested stride lengths of 1, 2, 4, and 8.

The reader is highly recommended to view page 59 of the Tables and Charts file to view the results from this test, where the resulting graphs have been arranged on a single page so they can be compared side-by-side.

- When the stride length is 1, as expected, the increments per step before 150 steps stays low, at about 68,000.
- When the stride length is 2, the increments per step before 150 steps goes up to 77,000.
- When the stride length is 4, it increases to 93,000.
- When the stride length is 8, it increases to 126,000.

Beyond a stride length of 8, the increments per step stops increasing. This indicates that the cache line size might be 8 4-byte integers long, for a total of

32 bytes. Unfortunately, we were not able to find clear documentation online that the 1070 graphics card has a cache line size of 32 bytes to back up this experimental data.

These results also raise new questions. Observe that the increments per step values from this experiment correlate with those from the previous cycle-walking test. We have already seen slope values of about 76,000, 93,000, and 126,000, they reappear in these tests. The reason why is unclear, and there is great potential for more experiments to be done to find out why.

4 Future Work

The future of WebGL 2.0 Compute is uncertain - it is on track to never be added to future browsers [9]. Instead, the graphics compute library currently on track to be integrated into browsers is WebGPU, a different graphics library from WebGL 2.0 Compute.

The currently WebGPU specification does not include atomic counters. Attempting to instantiate an atomic counter in WebGPU results in an error saying Vulkan, a graphics library that WebGPU is based off, does not support atomic counters. However, in Vulkan, atomic operations are supported in the form of atomic exchanges - WebGPU therefore, may implement atomics in this way instead. The tests and fingerprints outlined in this paper should be attempted in WebGPU with that approach, to see if similar behavior persists.

5 Conclusion

In this paper, we show preliminary testing results indicating that as modern browsers move towards adding compute features to graphics APIs, the attack surface for hardware fingerprinting increases, most notably due to support for atomic operations within the GPU. Using the in-development WebGL 2.0 Compute features enabled in the Windows Google Chrome Canary browser, we construct three categories of tests that show potential to be fingerprints.

The first test involves discovering the order with which the graphics card renders pixels in an image, and all tested dedicated GPUs rendered in a unique pattern. This refutes claims by previous research that by reducing the number of floating-point operations in WebGL, fingerprinting ability is hampered.

The second and third tests involve using the atomic counter as a homebrewed

implicit timer, which we show can be constructed by creating high contention for the counter.

The second test involves timing an arbitrary task; in this work, we used repeated sine calculations. The implicit time recorded by the atomic counter appears to grow linearly with the task size, indicating the timer is reliable. This may also serve as another fingerprinting strategy.

Finally, the third test involves probing the graphics card's caching behavior by timing walks of varying sizes through a random buffer. The results in dedicated graphics cards suggest caching behavior, which implies that these atomic counters could be leveraged to glean insight into graphics card architecture, all from the browser. It serves as another potential fingerprinting technique.

Taking all these results into account, the argument against including atomics in browser graphics libraries strengthens. More in-depth tests should be done, and reproduced in WebGPU if possible, but even the preliminary data shown in this paper cause concern over how these new features could be used for fingerprinting in the wild.

References

[1] Shujiang Wu, Song Li, Yinzhi Cao, and Ningfei Wang. *Rendered Private: Making GLSL Execution Uniform to Prevent WebGL-based Browser Fingerprinting*. 2019.

[2] Pietro Frigo, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. *Grand Pwning Unit: Accelerating Microarchitectural Attacks with the GPU*. 2018.

[3] Gunes Acar, Marc Juarez, Nick Nikiforakis, Claudia Diaz, Seda Gürses, Frank Piessens, and Bart Preneel. *FPDetective: Dusting The Web for Fingerprinters*. 2013.

[4] David Fifield and Serge Egelman. *Fingerprinting web users through font metrics.*. 2015.

[5] Pierre Laperdrix, Walter Rudametkin, and Benoit Baudry. *Beauty and the beast: Diverting modern web browsers to build unique browser fingerprints*. 2016.

[6] Keaton Mowery and Hovav Shacham. *Pixel perfect: Fingerprinting canvas in html5*. 2012.

[7] Yinzhi Cao, Song Li, and Erik Wijmans. *(cross-)browser fingerprinting via os and hardware level features*. 2017.

[8] WebGL 2.0 Compute Specification. https://www.khronos.org/registry/webgl/specs/latest/2.0-compute/

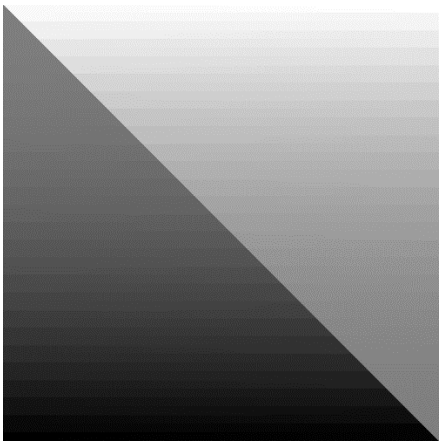
[9] WebGL 2.0 Compute Intent to Implement. https://groups.google.com/a/chromium.org/forum/#!msg/blink-dev/bPD47wqY-r8/5DzgvEwFBAAJ

[10] WebGPU Specification. https://gpuweb.github.io/gpuweb/

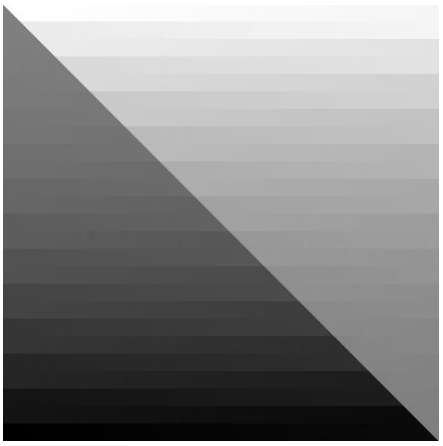
Render Order Tests Results

Integrated Cards:

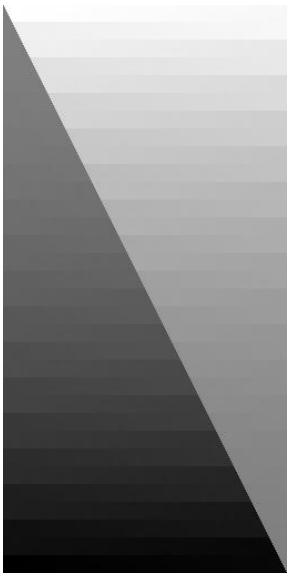
Intel RHD Graphics 520 OpenGL 44 Core
Intel RHD Graphics 520 OpenGL 45 Core
Intel RHD Graphics 620 OpenGL 45 Core
Intel RUHD Graphics 620 OpenGL 45 Core



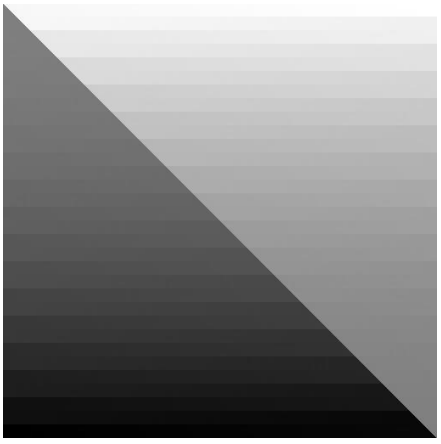
800 x 800 (sized down for fit)



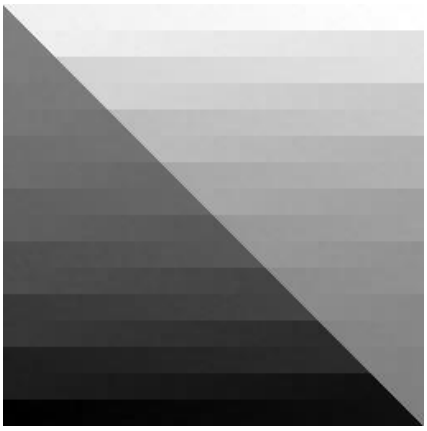
400 x 400 (sized down for fit)



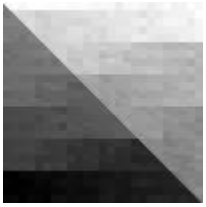
256 x 512 (sized down for fit)



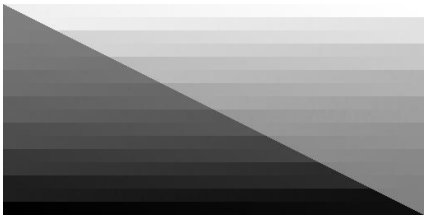
512 x 512 (sized down for fit)



256 x 256 (sized down for fit)



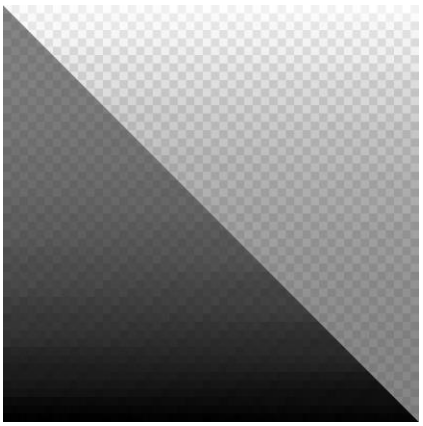
100 x 100



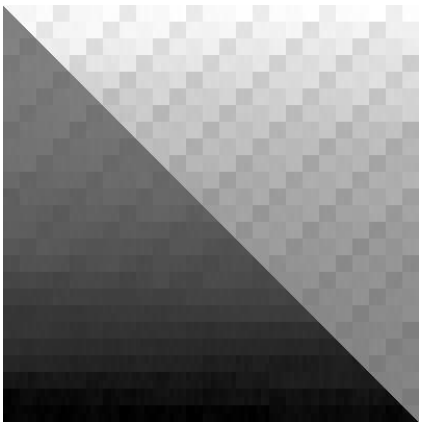
512 x 256 (sized down for fit)

Render Order Tests Results

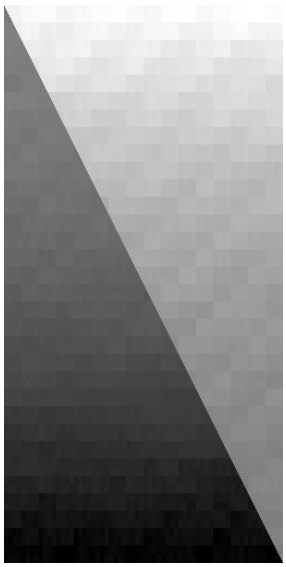
NVIDIA GeForce GTX 770 PCIe SSE2 OpenGL 45 core



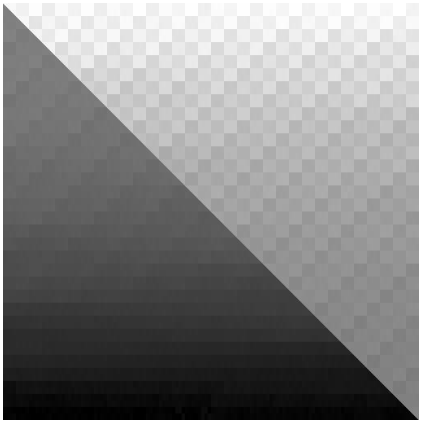
800 x 800 (sized down for fit)



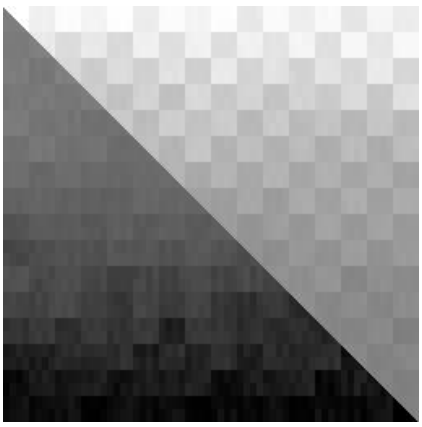
400 x 400 (sized down for fit)



256 x 512 (sized down for fit)

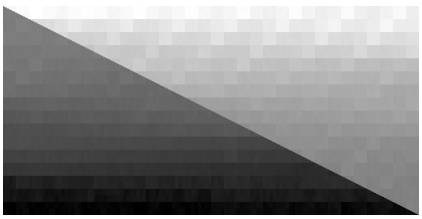


512 x 512 (sized down for fit)



256 x 256 (sized down for fit)

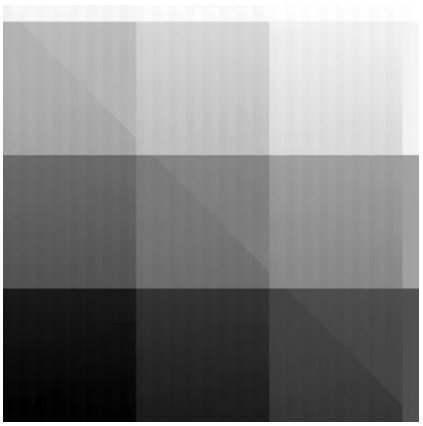
100 x 100



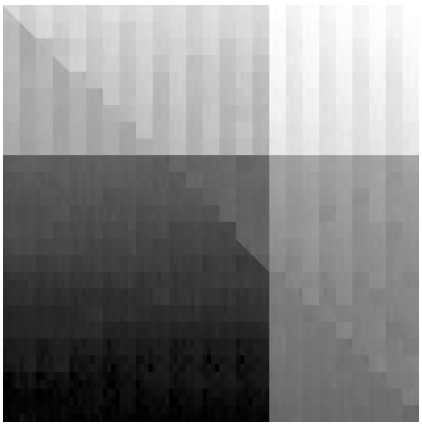
512 x 256 (sized down for fit)

Render Order Tests Results

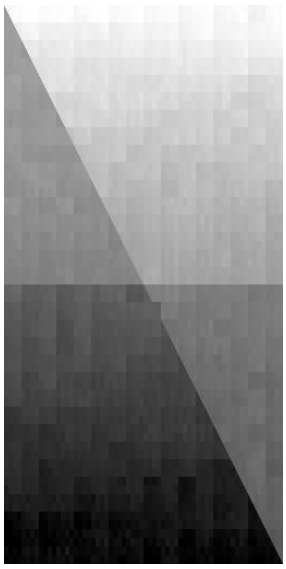
NVIDIA GeForce GTX 1060 6GB PCIe SSE2 OpenGL 45 core



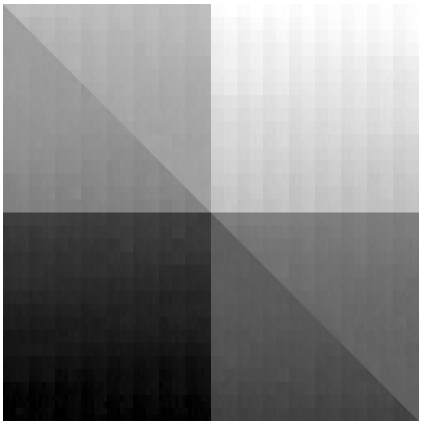
800 x 800 (sized down for fit)



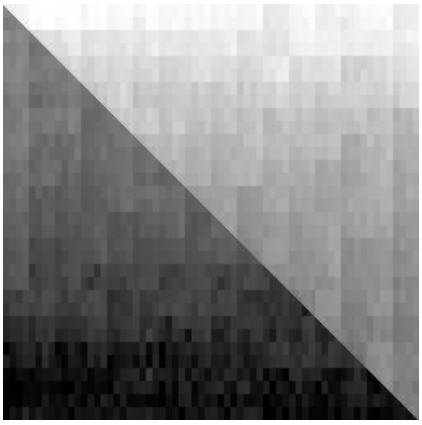
400 x 400 (sized down for fit)



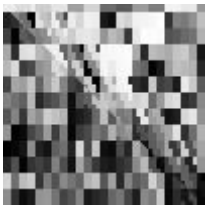
256 x 512 (sized down for fit)



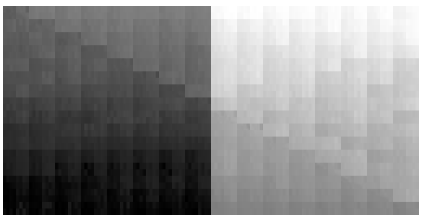
512 x 512 (sized down for fit)



256 x 256 (sized down for fit)



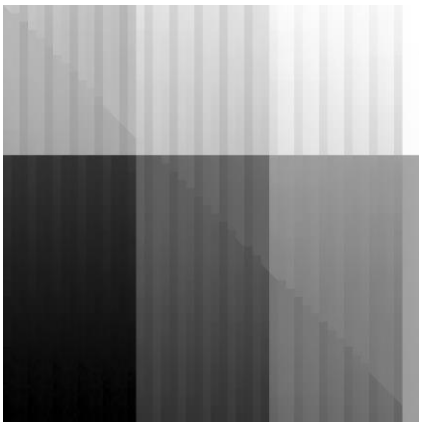
100 x 100



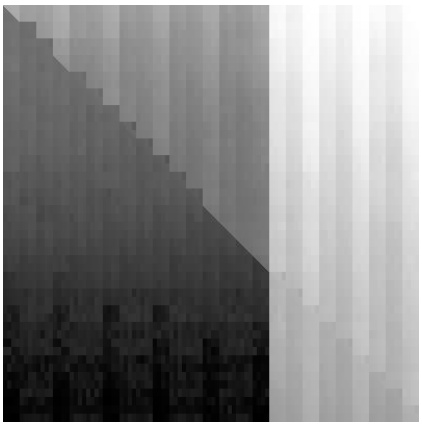
512 x 256 (sized down for fit)

Render Order Tests Results

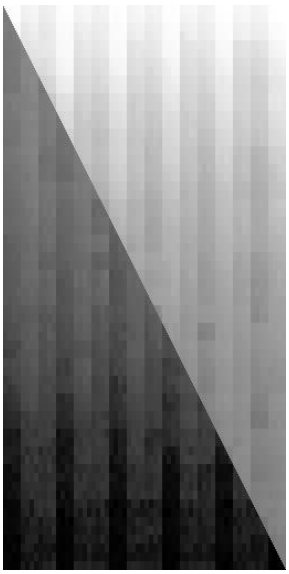
NVIDIA GeForce GTX 1070 PCIe SSE2 OpenGL 45 core



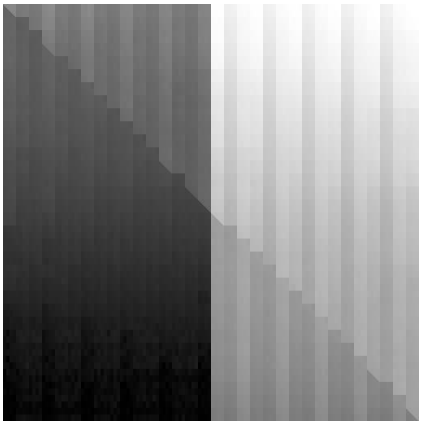
800 x 800 (sized down for fit)



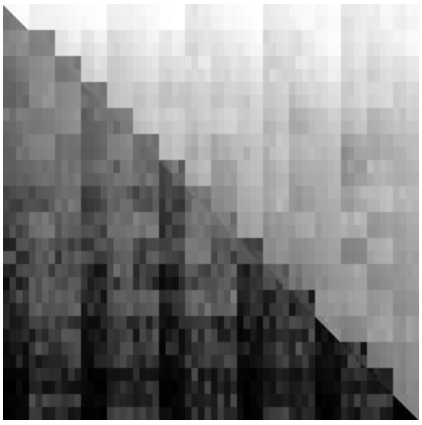
400 x 400 (sized down for fit)



256 x 512 (sized down for fit)

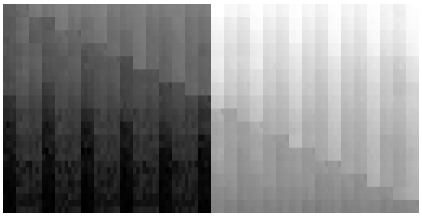


512 x 512 (sized down for fit)



256 x 256 (sized down for fit)

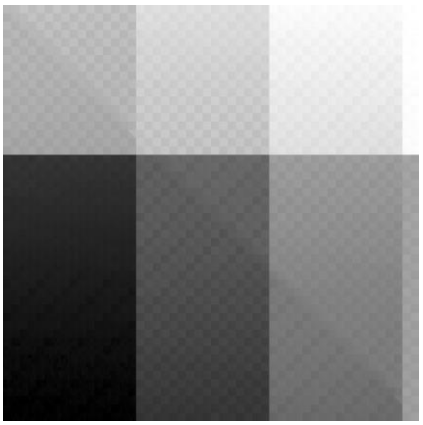
100 x 100



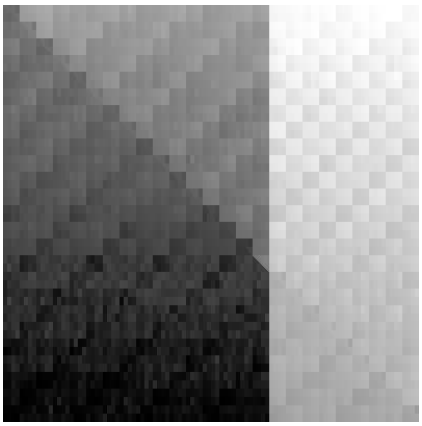
512 x 256 (sized down for fit)

Render Order Tests Results

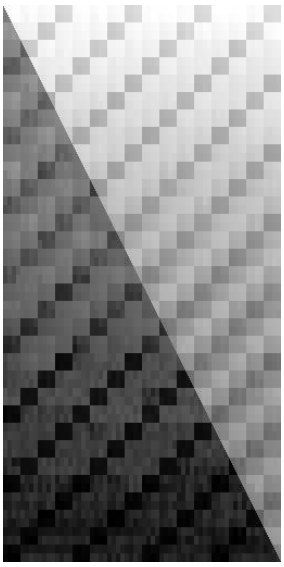
NVIDIA GeForce GTX 1080 PCIe SSE2 OpenGL 45 core



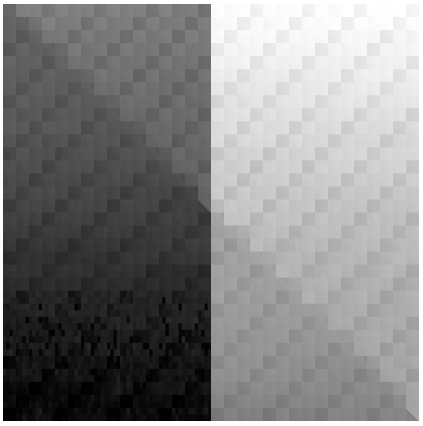
800 x 800 (sized down for fit)



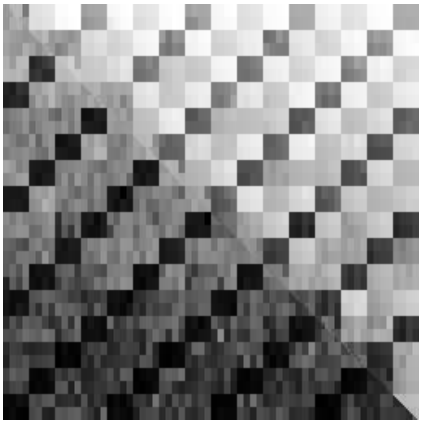
400 x 400 (sized down for fit)



256 x 512 (sized down for fit)

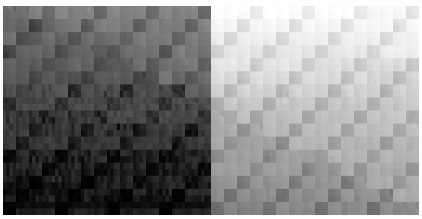


512 x 512 (sized down for fit)



256 x 256 (sized down for fit)

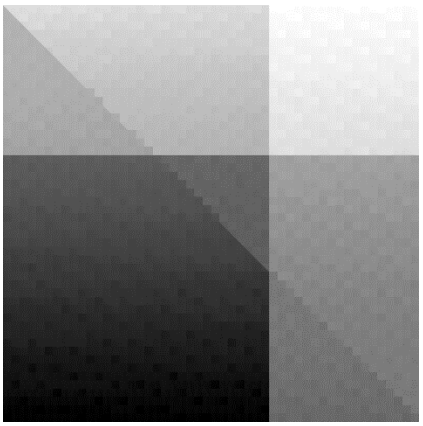
100 x 100



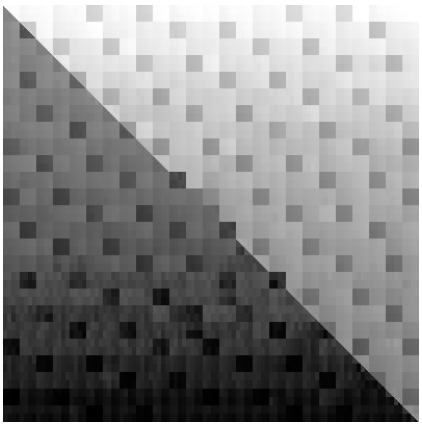
512 x 256 (sized down for fit)

Render Order Tests Results

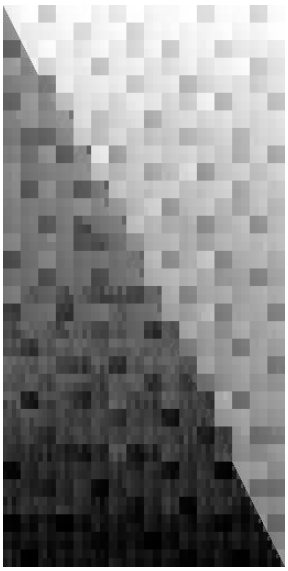
NVIDIA GeForce RTX 2080 SUPER PCIe SSE2 OpenGL 45 core



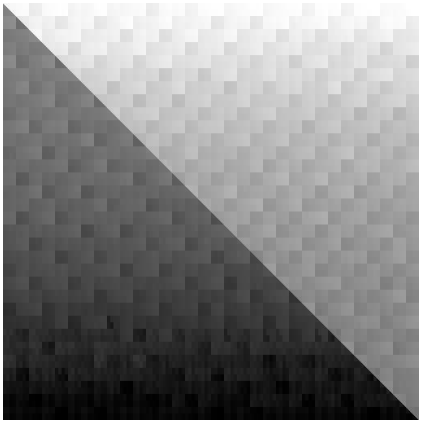
800 x 800 (sized down for fit)



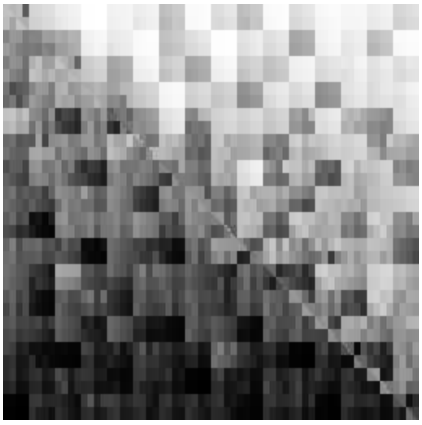
400 x 400 (sized down for fit)



256 x 512 (sized down for fit)

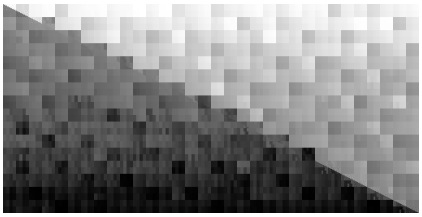


512 x 512 (sized down for fit)



256 x 256 (sized down for fit)

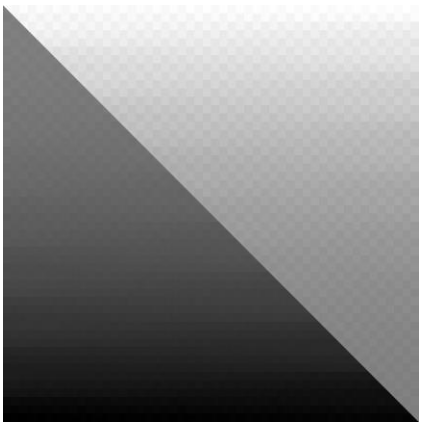
100 x 100



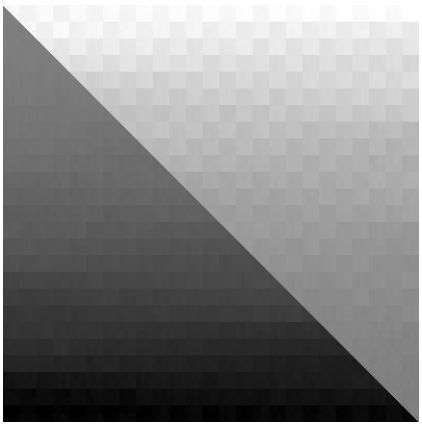
512 x 256 (sized down for fit)

Render Order Tests Results

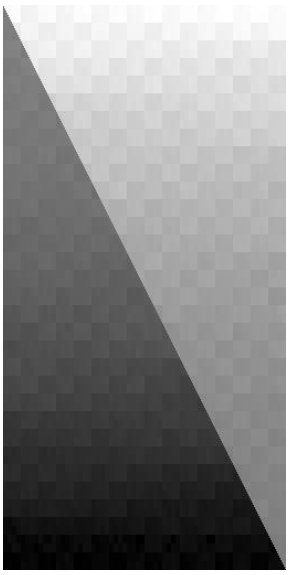
NVIDIA Quadro P620 PCIe SSE2 OpenGL 45 core



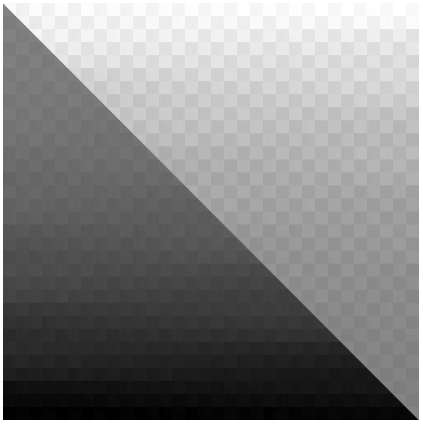
800 x 800 (sized down for fit)



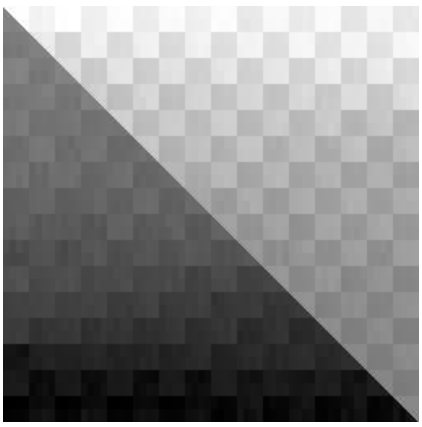
400 x 400 (sized down for fit)



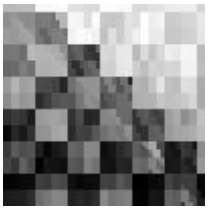
256 x 512 (sized down for fit)



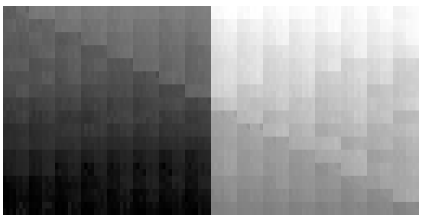
512 x 512 (sized down for fit)



256 x 256 (sized down for fit)



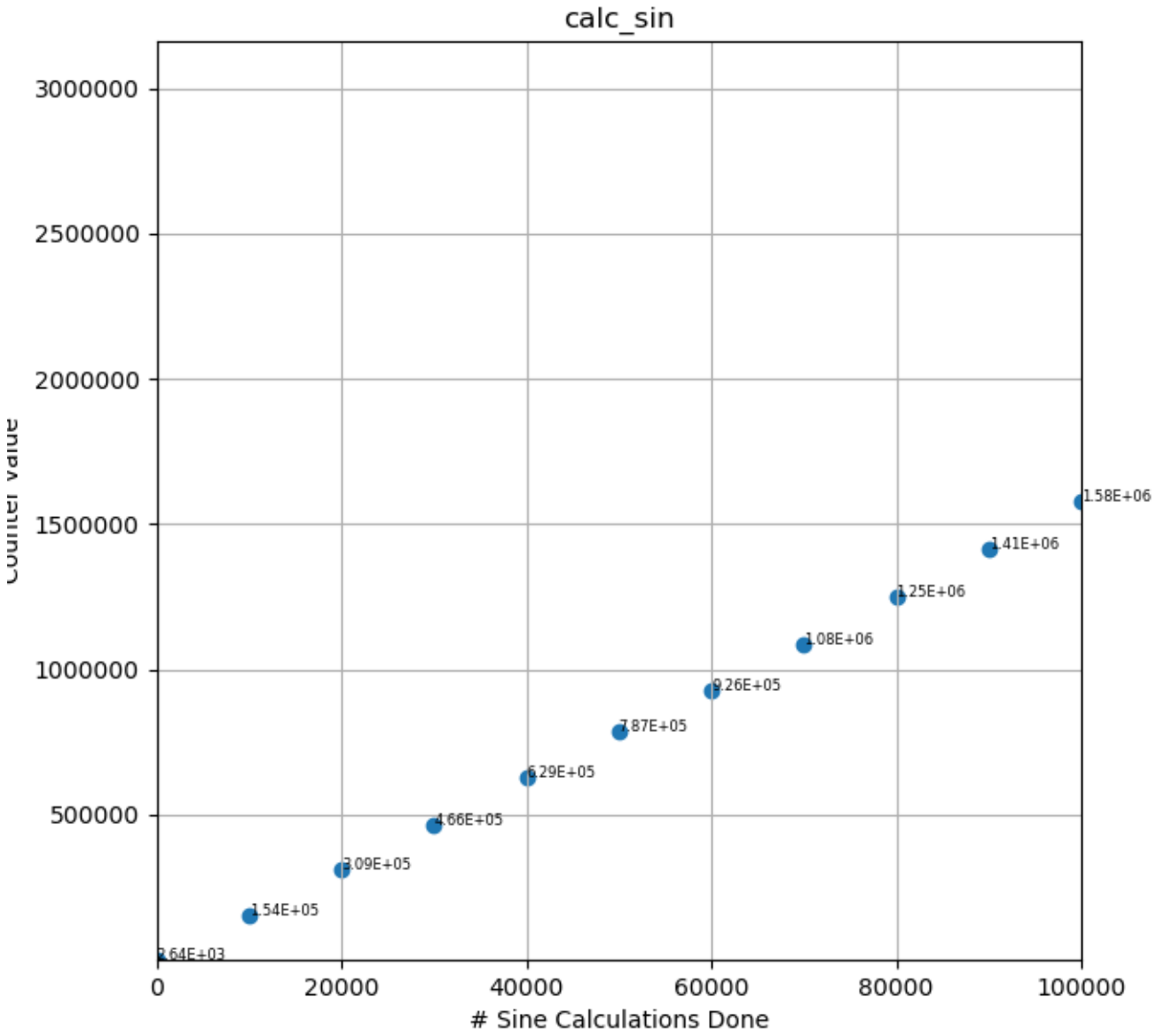
100 x 100



512 x 256 (sized down for fit)

Sine Calculation Test Results

Intel RHD Graphics 520 OpenGL 44 Core

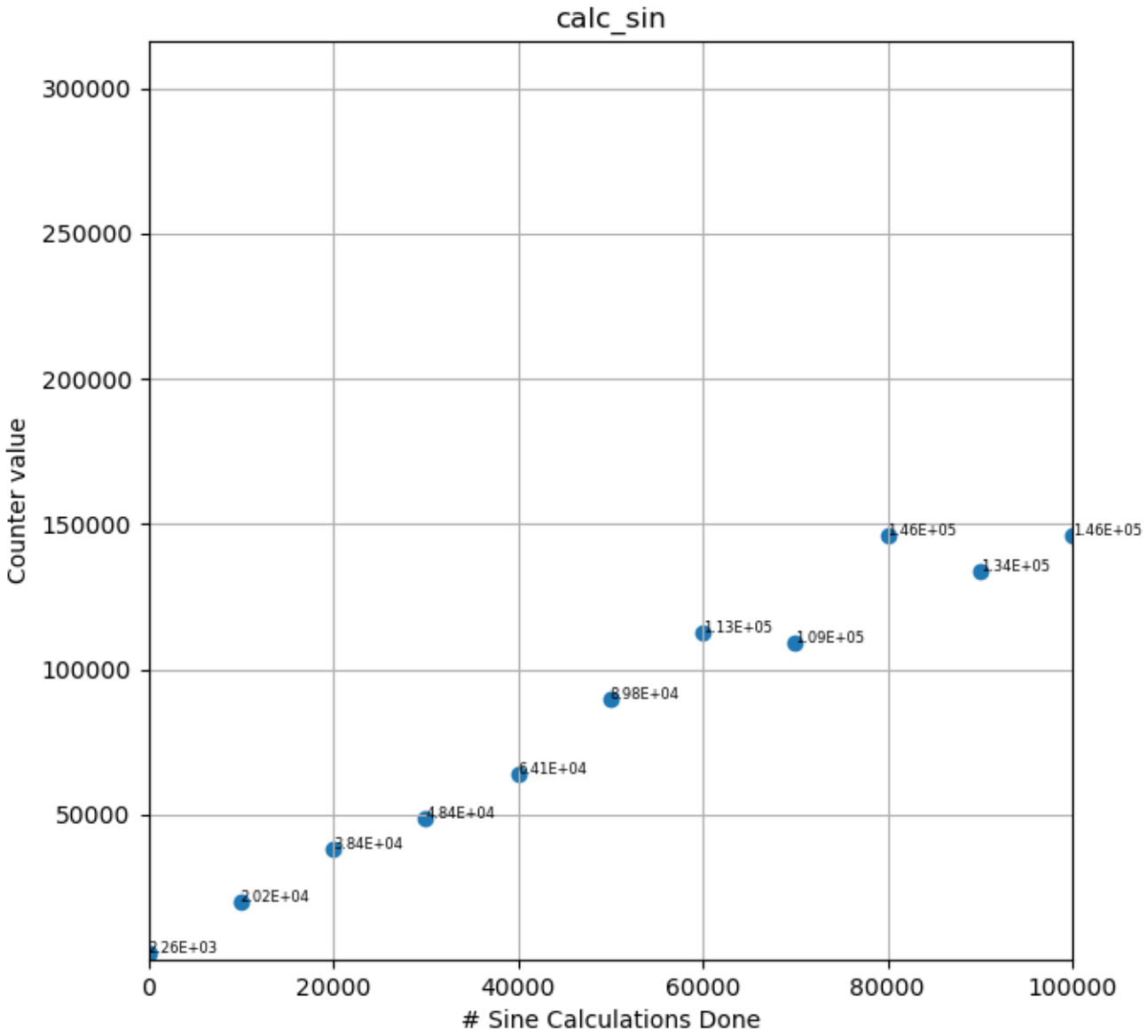


Results of the repeated Sine Calculations.

SLOPE= ~15 Counter Increments per Sine Calculation

Sine Calculation Test Results

Intel RHD Graphics 520 OpenGL 45 Core

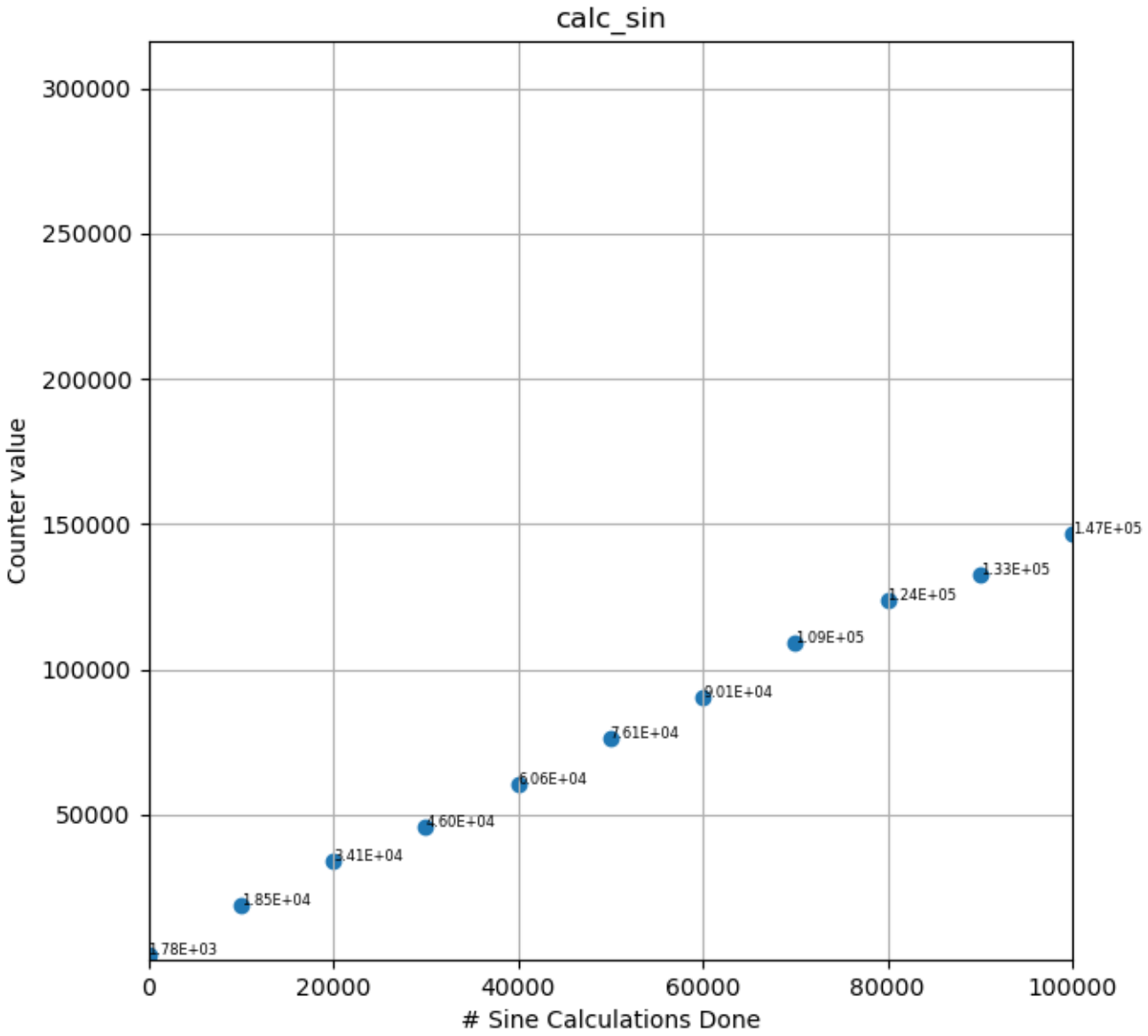


Results of the repeated Sine Calculations.

SLOPE= ~1.6 Counter Increments per Sine Calculation

Sine Calculation Test Results

Intel RHD Graphics 620 OpenGL 45 Core

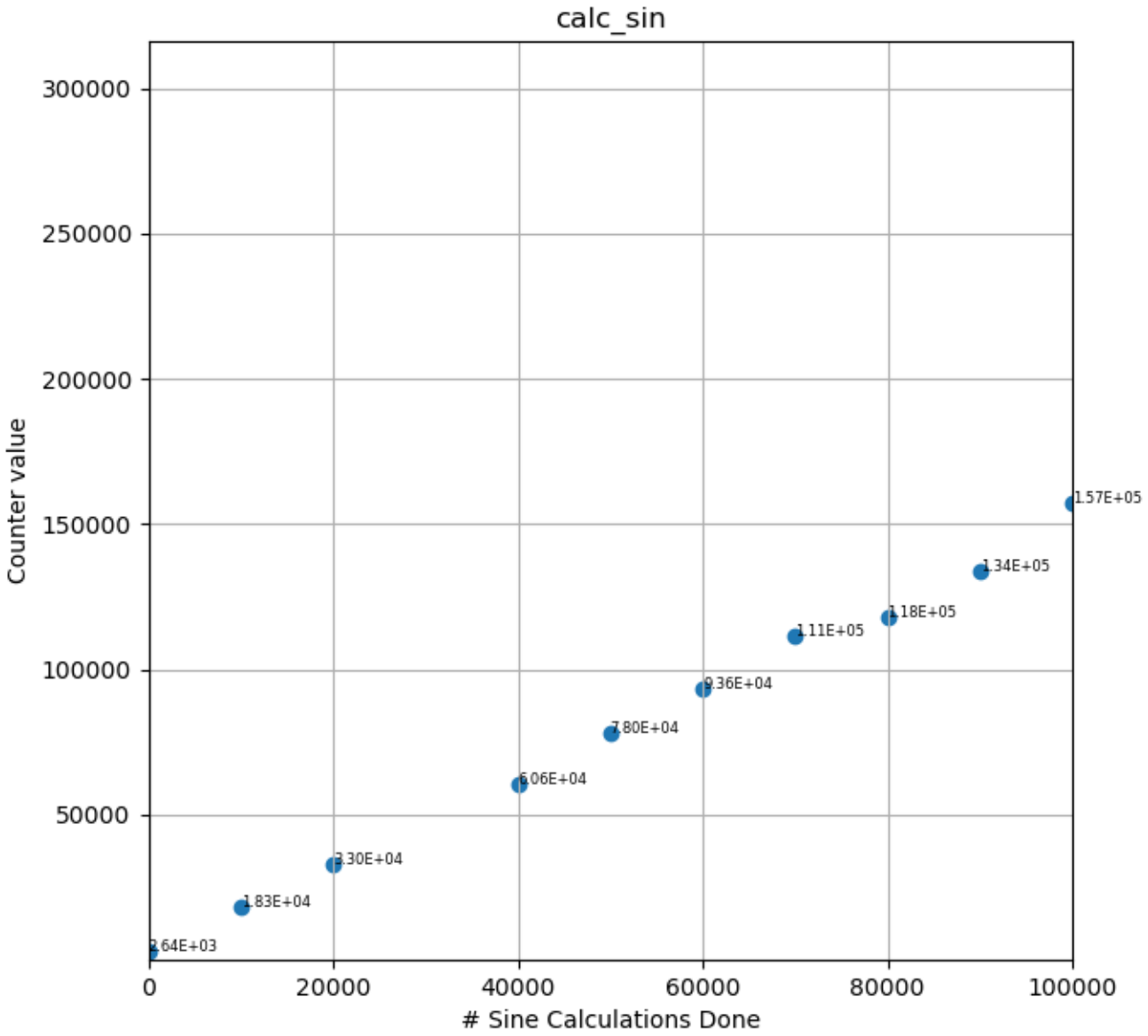


Results of the repeated Sine Calculations.

SLOPE= ~1.5 Counter Increments per Sine Calculation

Sine Calculation Test Results

Intel RUHD Graphics 620 OpenGL 45 Core

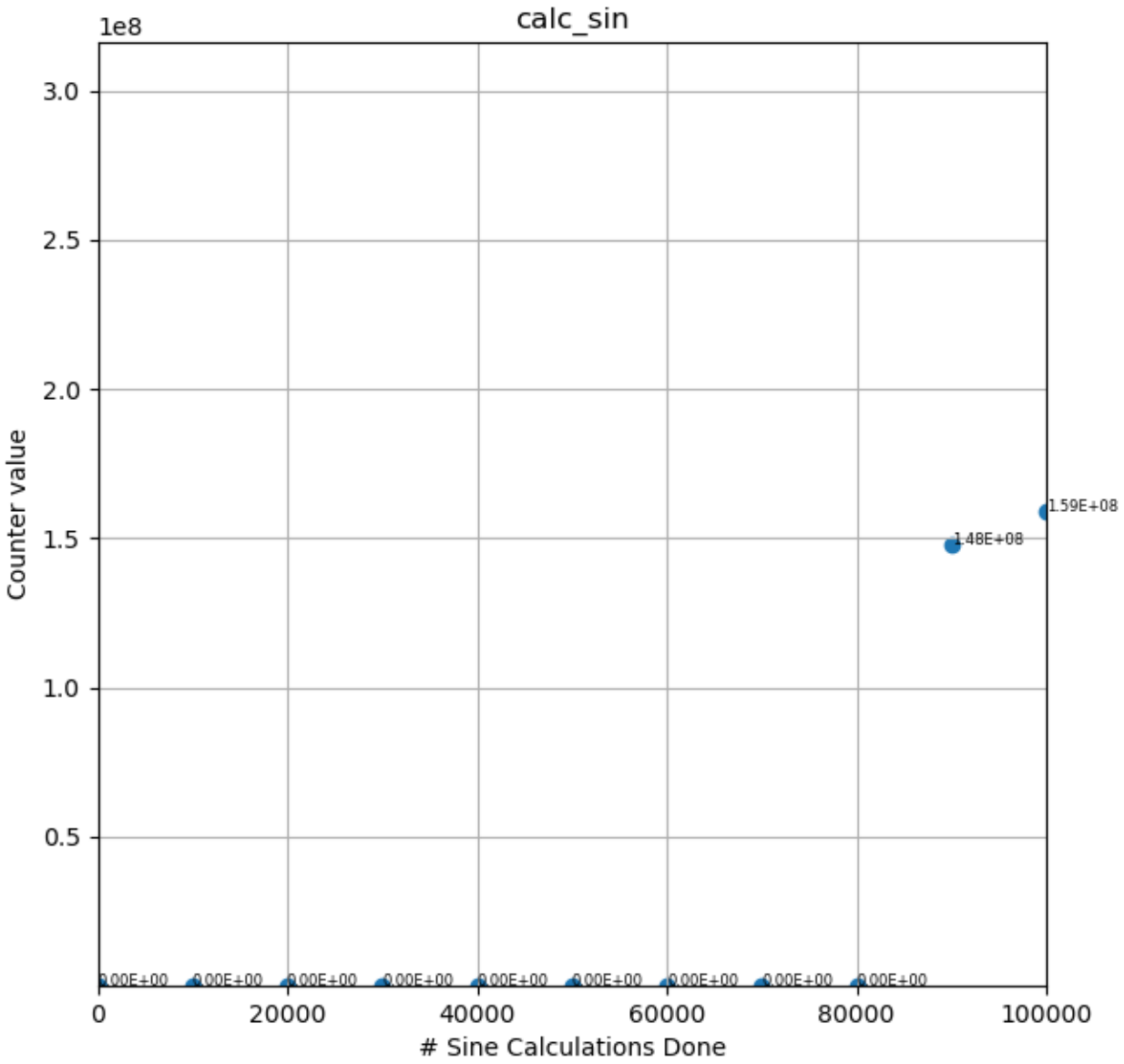


Results of the repeated Sine Calculations.

SLOPE= ~1.5 Counter Increments per Sine Calculation

Sine Calculation Test Results

NVIDIA GeForce GTX 770 PCIe SSE2 OpenGL 45 core

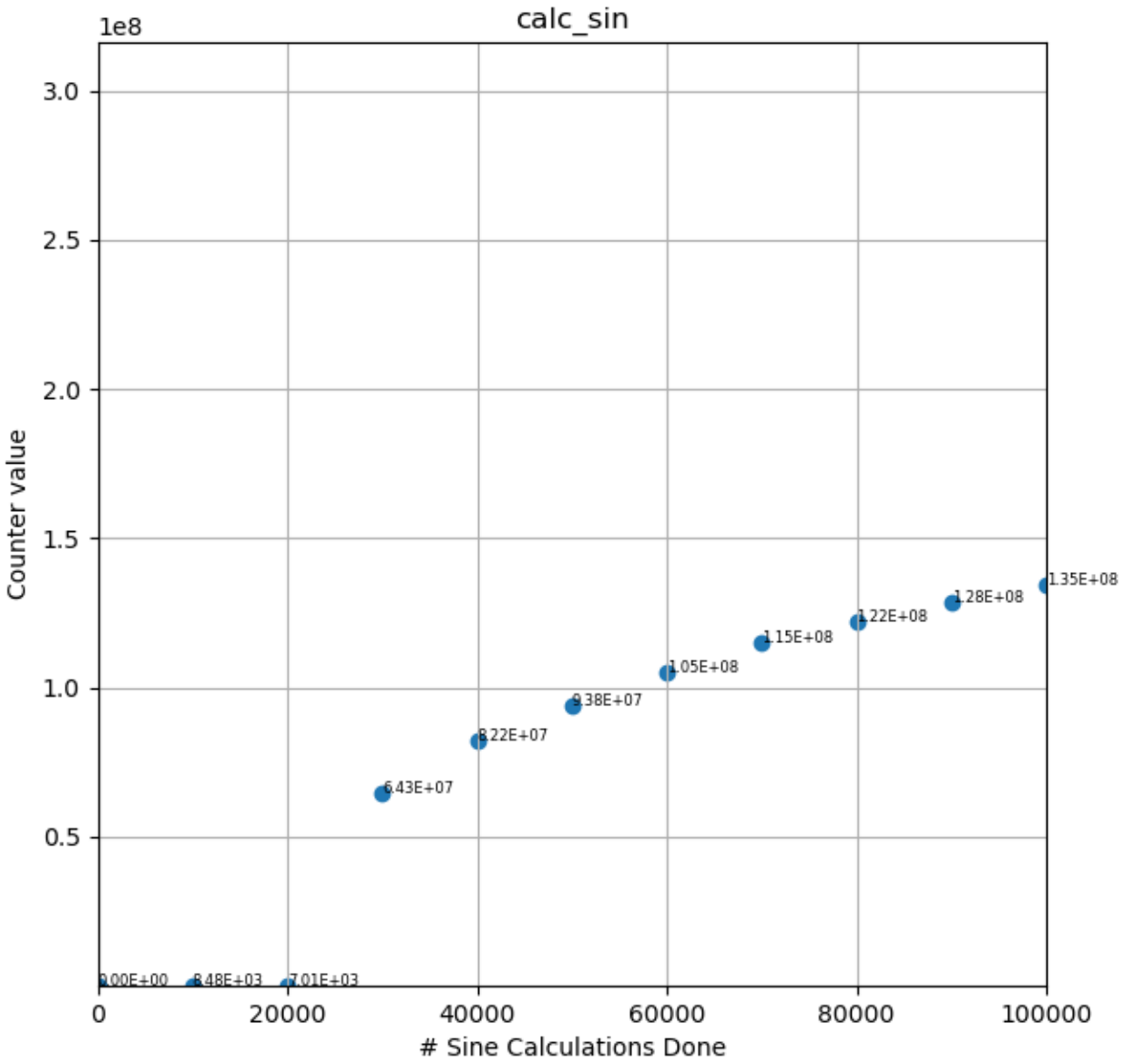


Results of the repeated Sine Calculations.

SLOPE= ~1600 Counter Increments per Sine Calculation,
only takes data points at 90,000 and 100,000 sine calculations into account.

Sine Calculation Test Results

NVIDIA GeForce GTX 1060 6GB PCIe SSE2 OpenGL 45 core

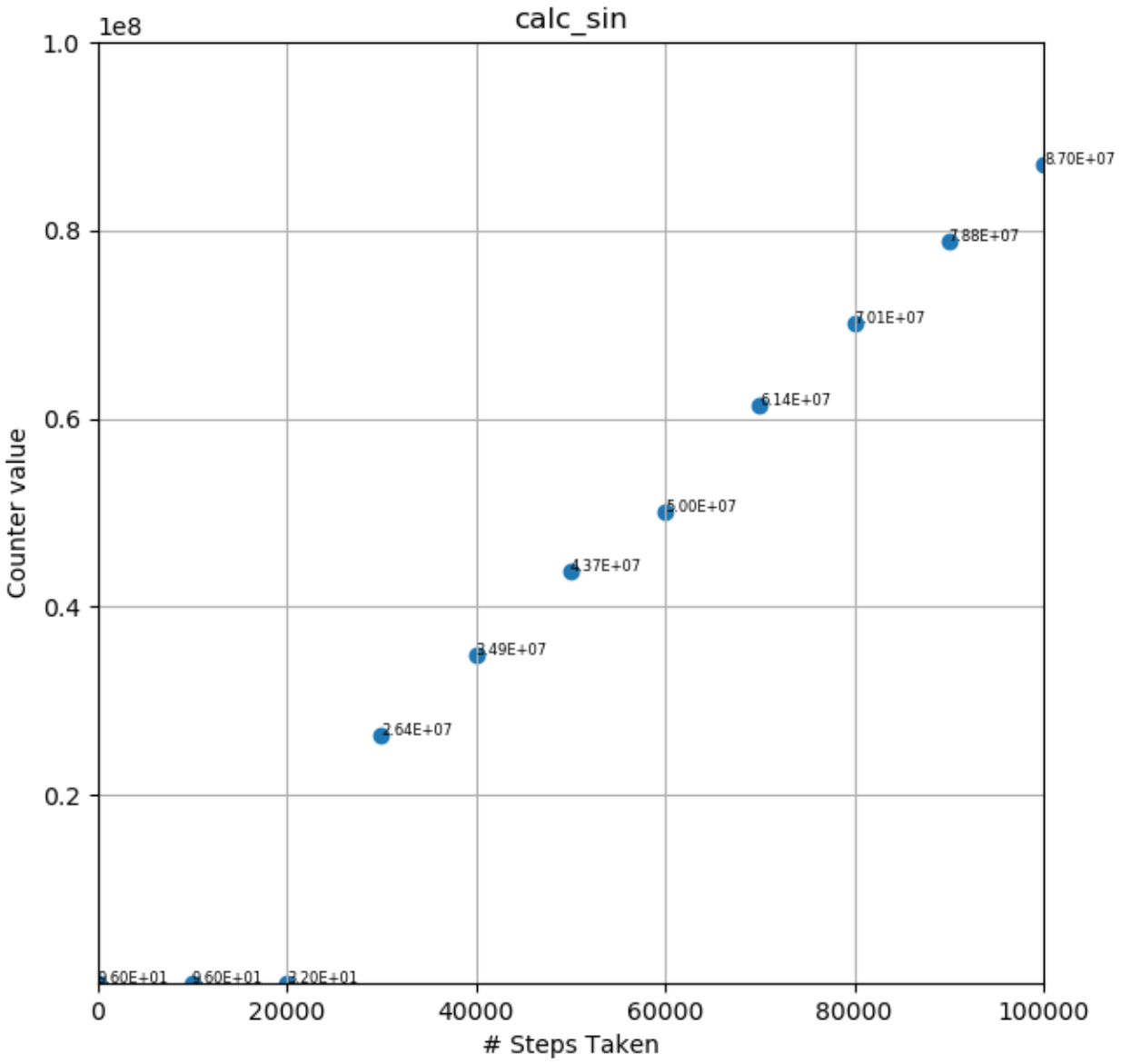


Results of the repeated Sine Calculations.

SLOPE= ~1600 Counter Increments per Sine Calculation,
only takes data points at 30,000 through 100,000 sine calculations into account.

Sine Calculation Test Results

NVIDIA GeForce GTX 1070 PCIe SSE2 OpenGL 45 core

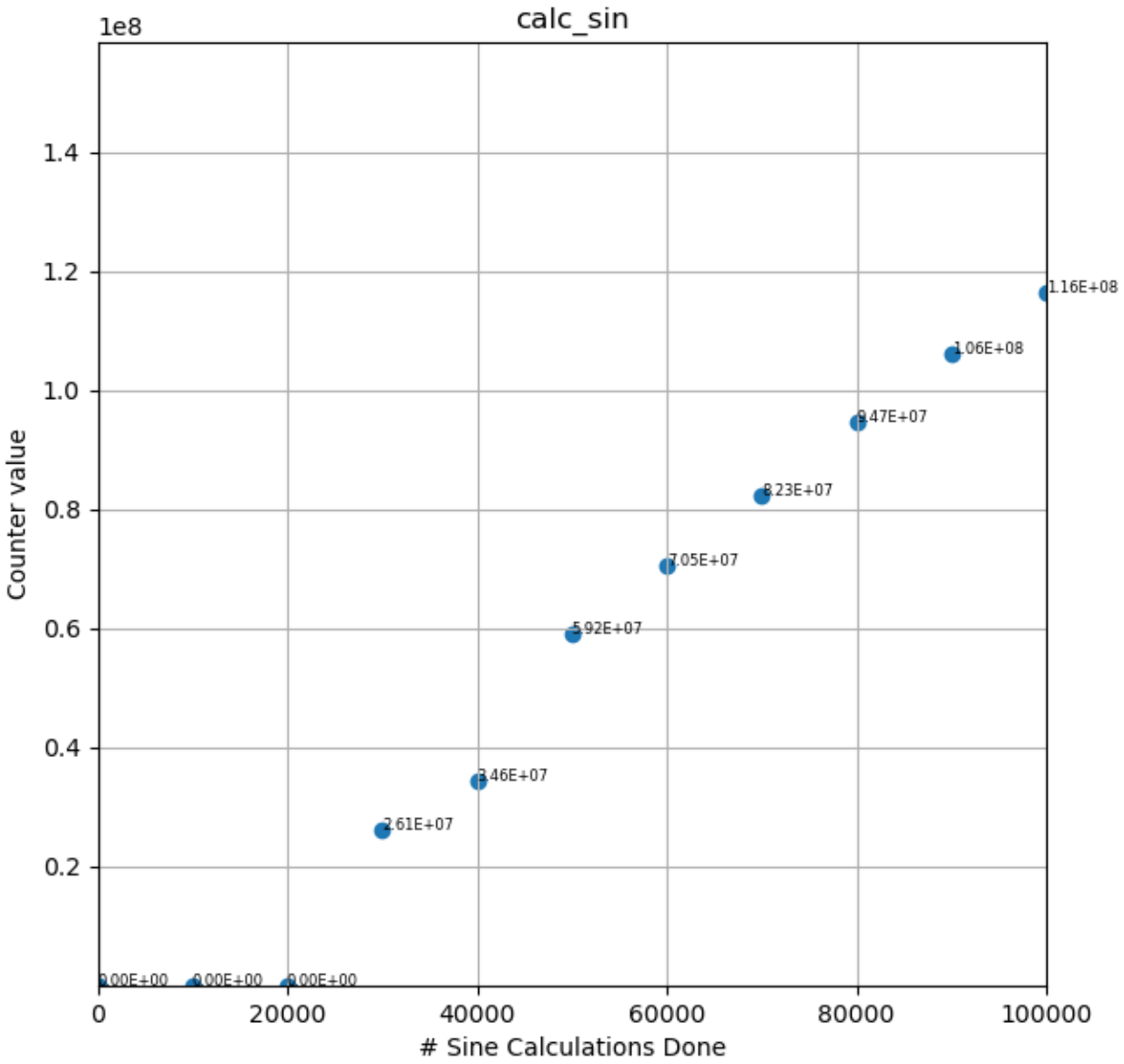


Results of the repeated Sine Calculations.

SLOPE= ~875 Counter Increments per Sine Calculation,
only takes data points at 30,000 through 100,000 sine calculations into account.

Sine Calculation Test Results

NVIDIA GeForce GTX 1080 PCIe SSE2 OpenGL 45 core

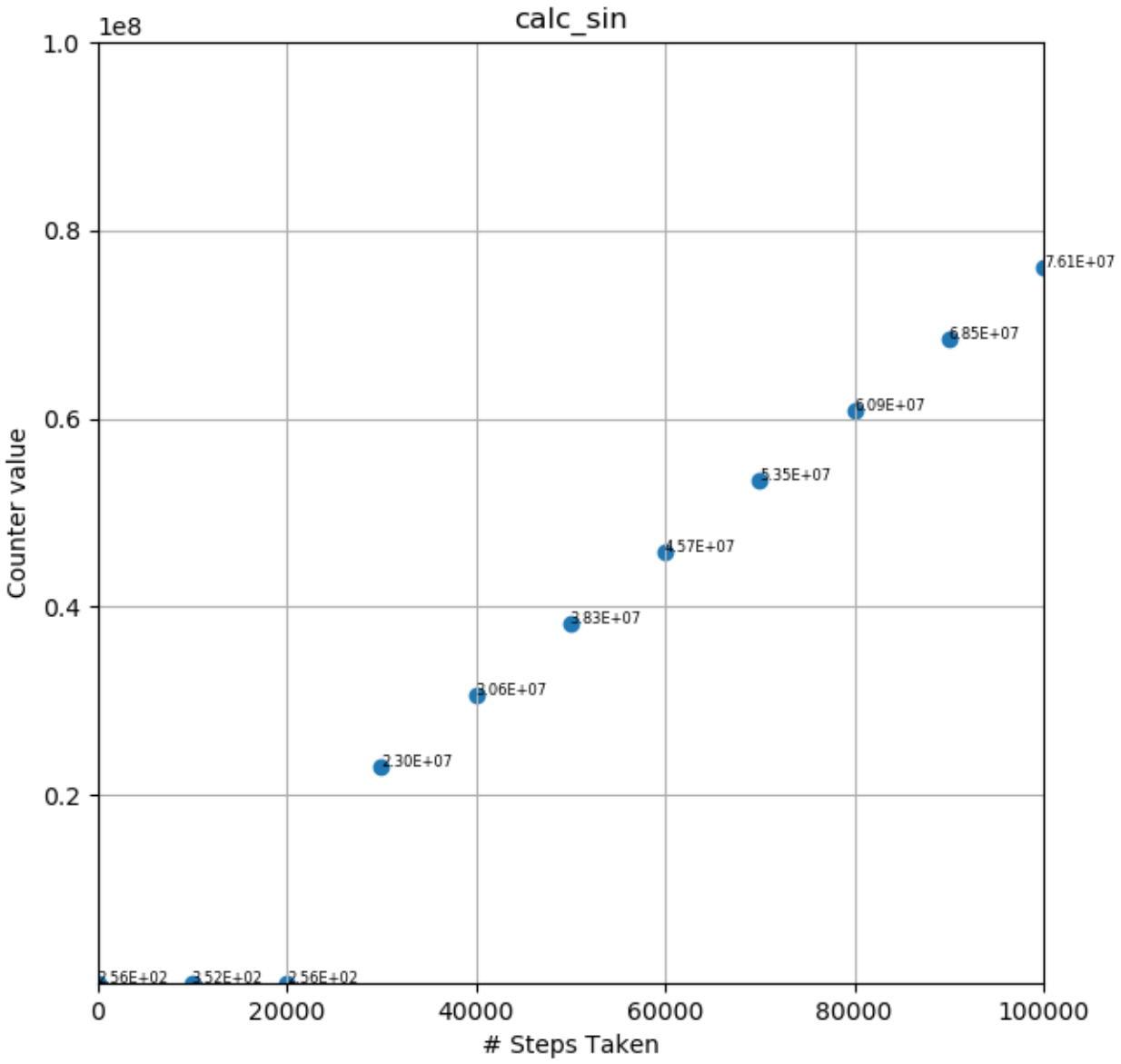


Results of the repeated Sine Calculations.

SLOPE= ~1200 Counter Increments per Sine Calculation,
only takes data points at 30,000 through 100,000 sine calculations into account.

Sine Calculation Test Results

NVIDIA GeForce RTX 2080 SUPER PCIe SSE2 OpenGL 45 core

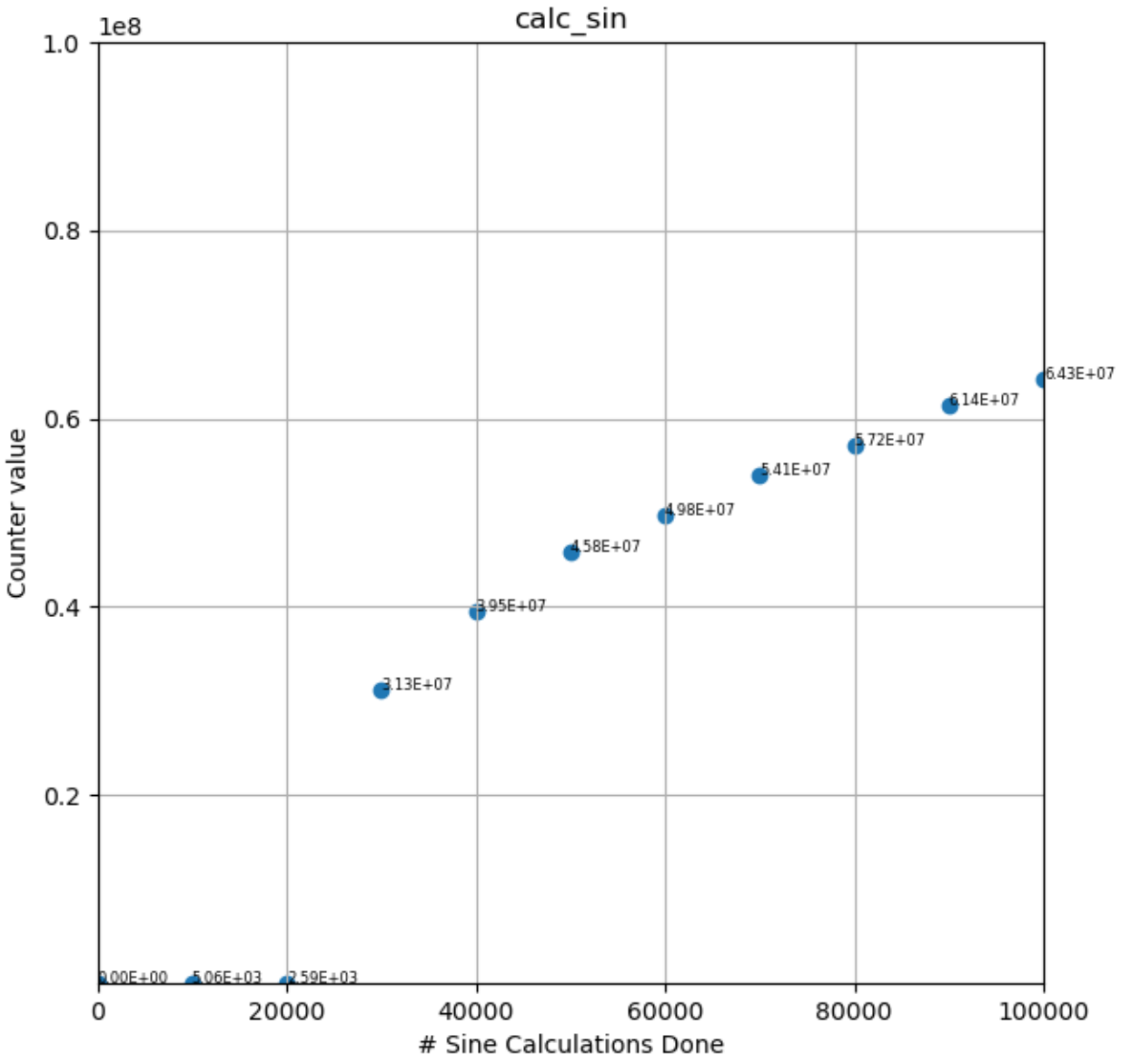


Results of the repeated Sine Calculations.

SLOPE= ~760 Counter Increments per Sine Calculation,
only takes data points at 30,000 through 100,000 sine calculations into account.

Sine Calculation Test Results

NVIDIA Quadro P620 PCIe SSE2 OpenGL 45 core

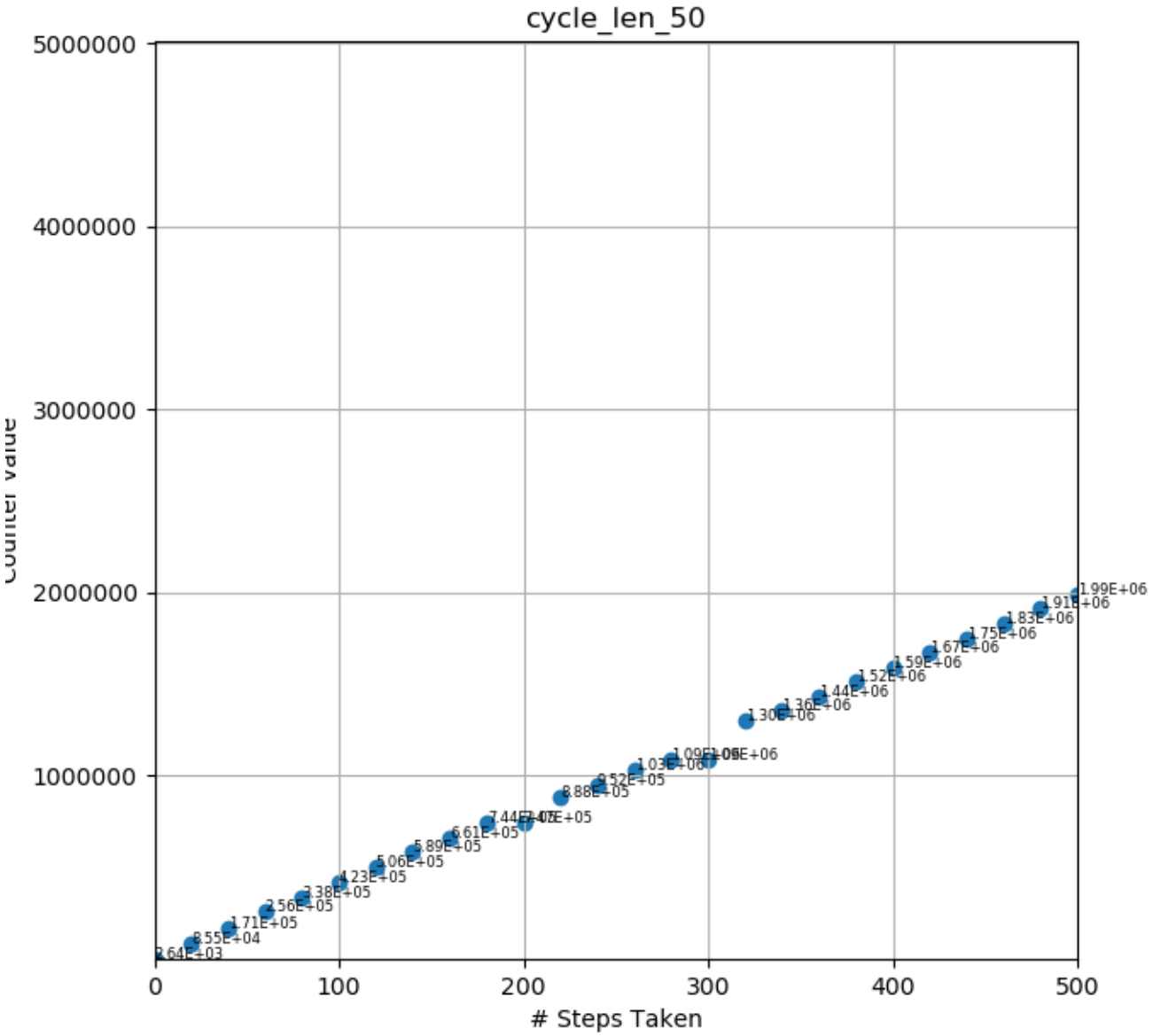


Results of the repeated Sine Calculations.

SLOPE= ~800 Counter Increments per Sine Calculation,
only takes data points at 30,000 through 100,000 sine calculations into account.

Cycle-Walking Test Results

Intel RHD Graphics 520 OpenGL 44 Core

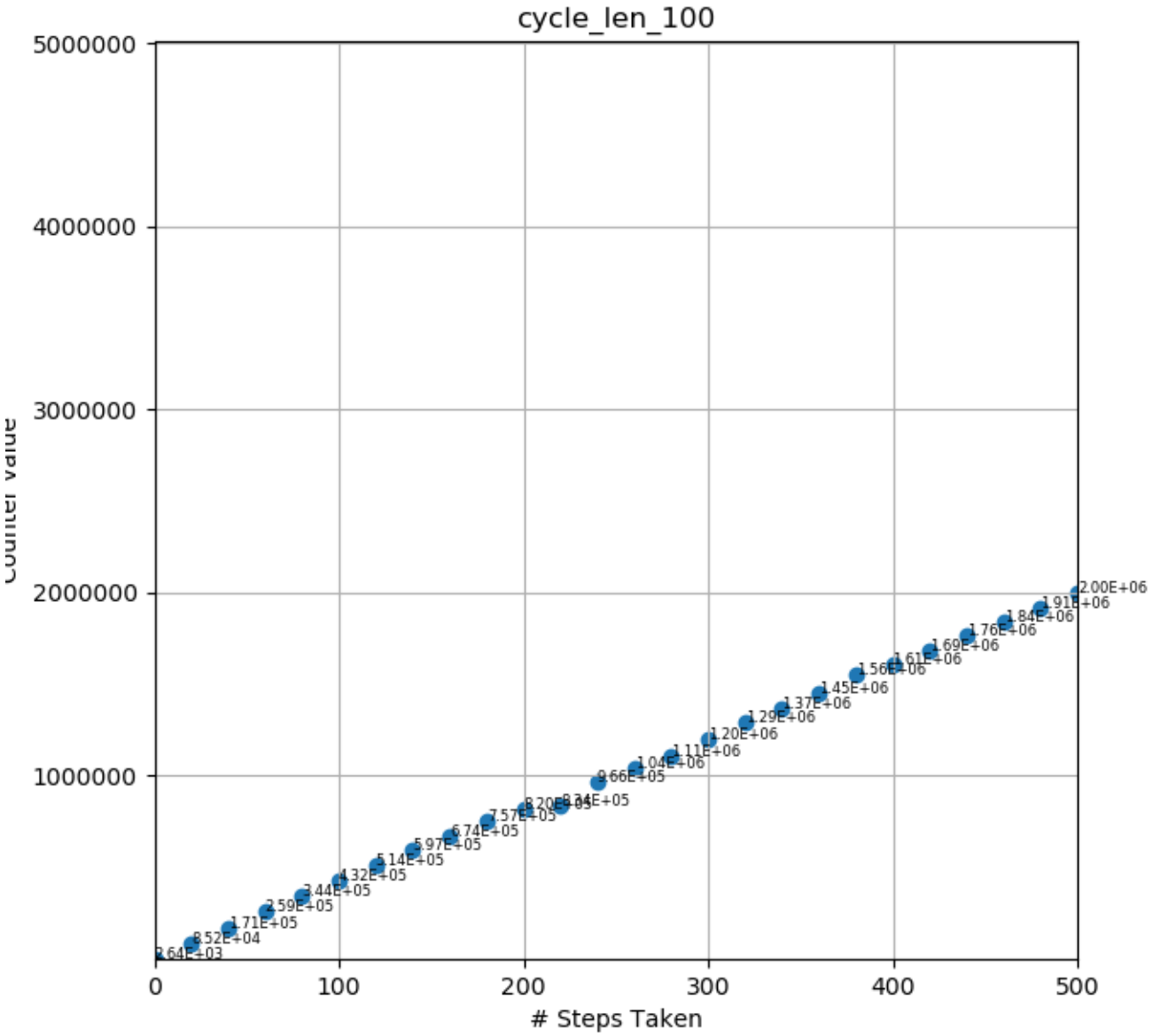


Results from the cycle-walking test, where the cycle has length 50.

SLOPE: ~4,000 Counter Increments per Step

Cycle-Walking Test Results

Intel RHD Graphics 520 OpenGL 44 Core

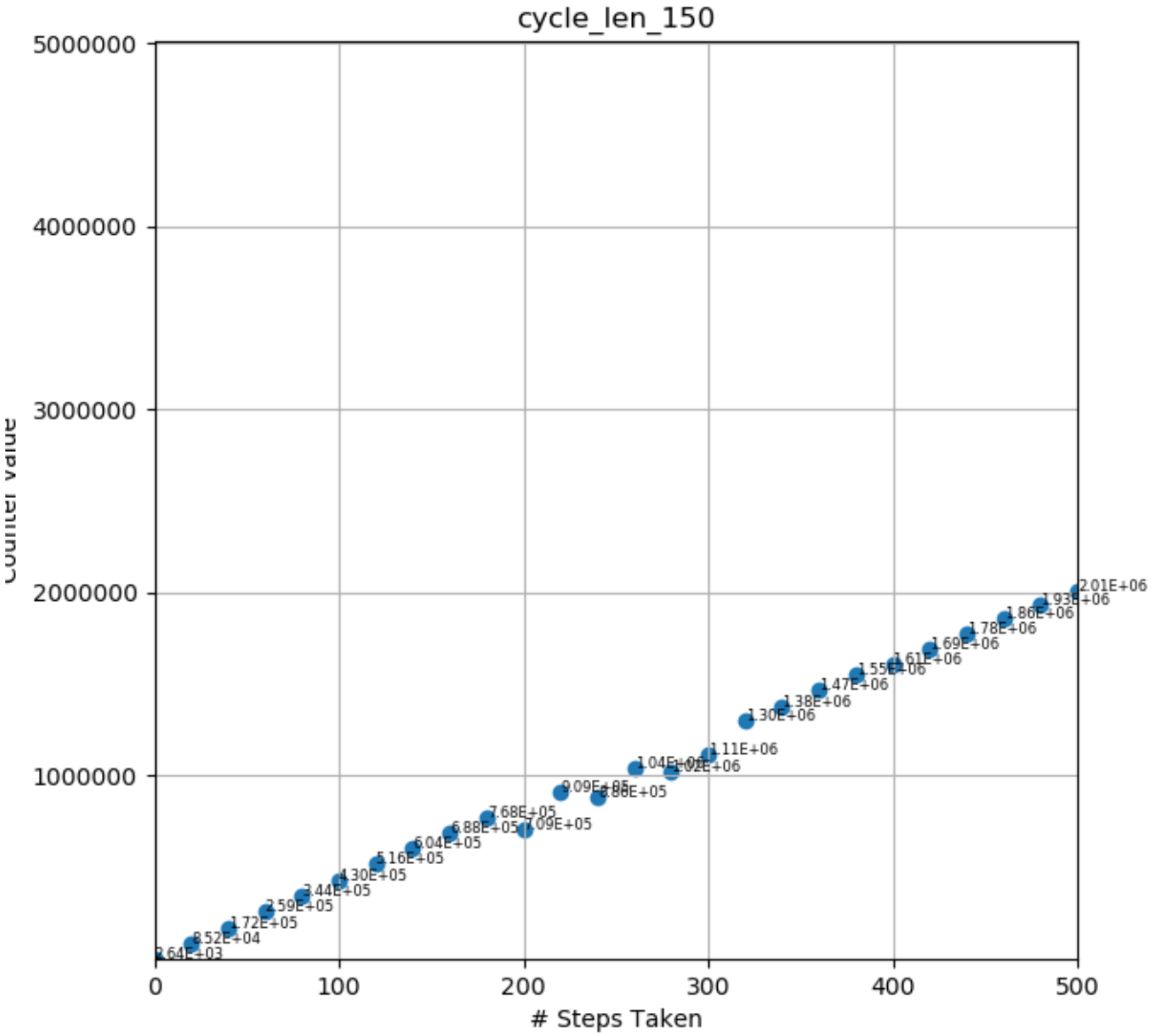


Results from the cycle-walking test, where the cycle has length 100.

SLOPE: ~4,000 Counter Increments per Step

Cycle-Walking Test Results

Intel RHD Graphics 520 OpenGL 44 Core

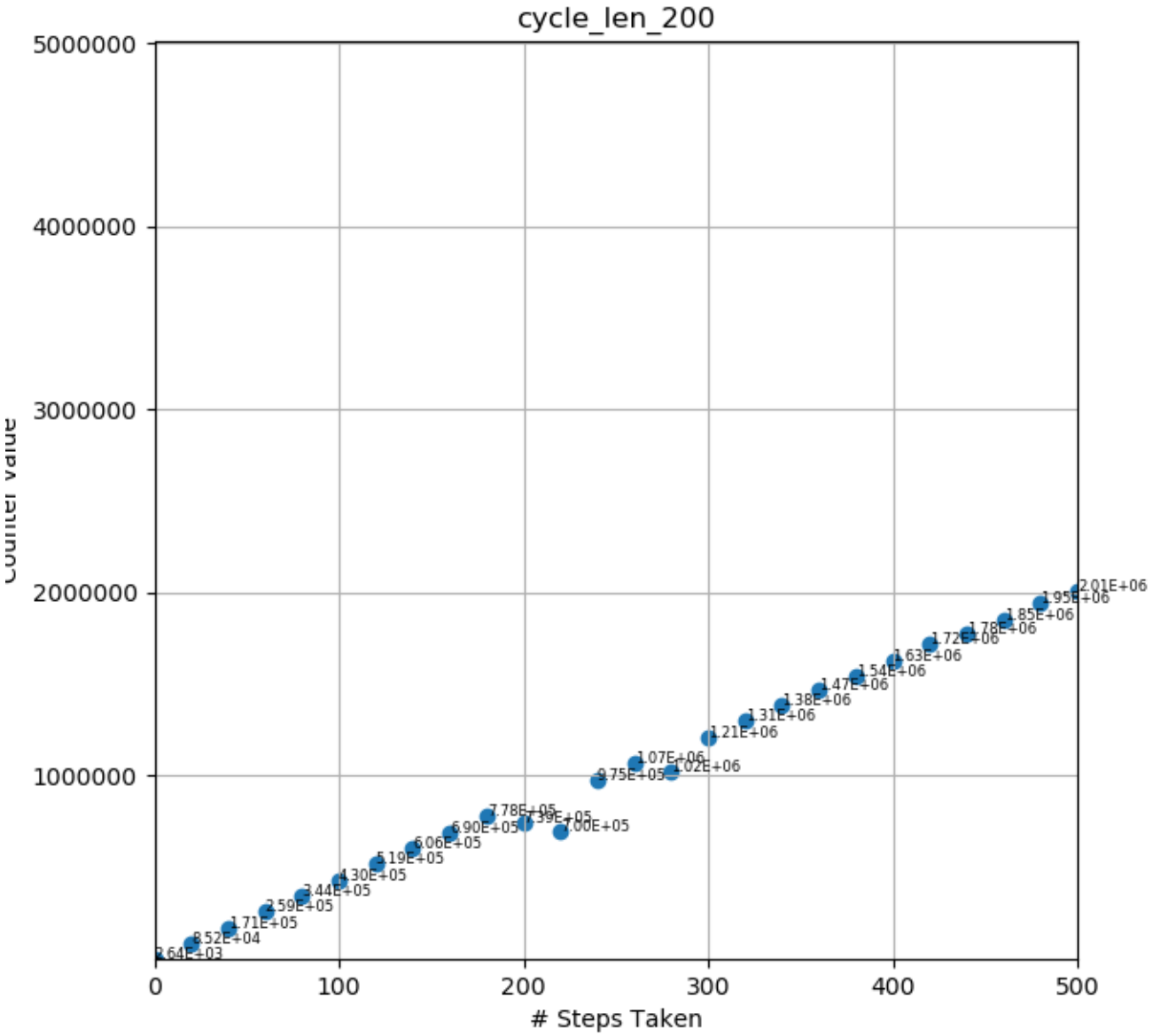


Results from the cycle-walking test, where the cycle has length 150.

SLOPE: ~4,000 Counter Increments per Step

Cycle-Walking Test Results

Intel RHD Graphics 520 OpenGL 44 Core

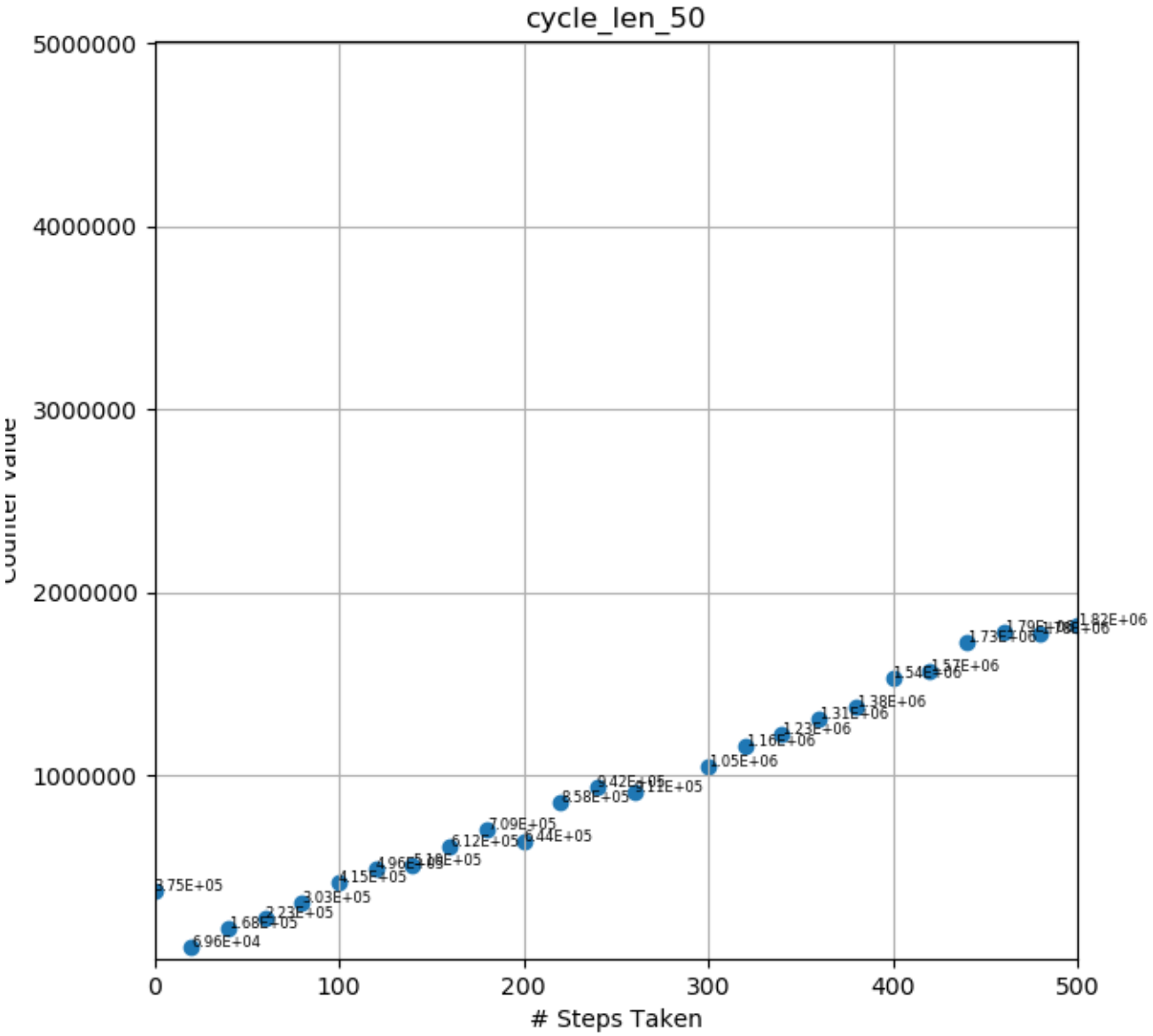


Results from the cycle-walking test, where the cycle has length 200.

SLOPE: ~4,000 Counter Increments per Step

Cycle-Walking Test Results

Intel RHD Graphics 520 OpenGL 45 Core

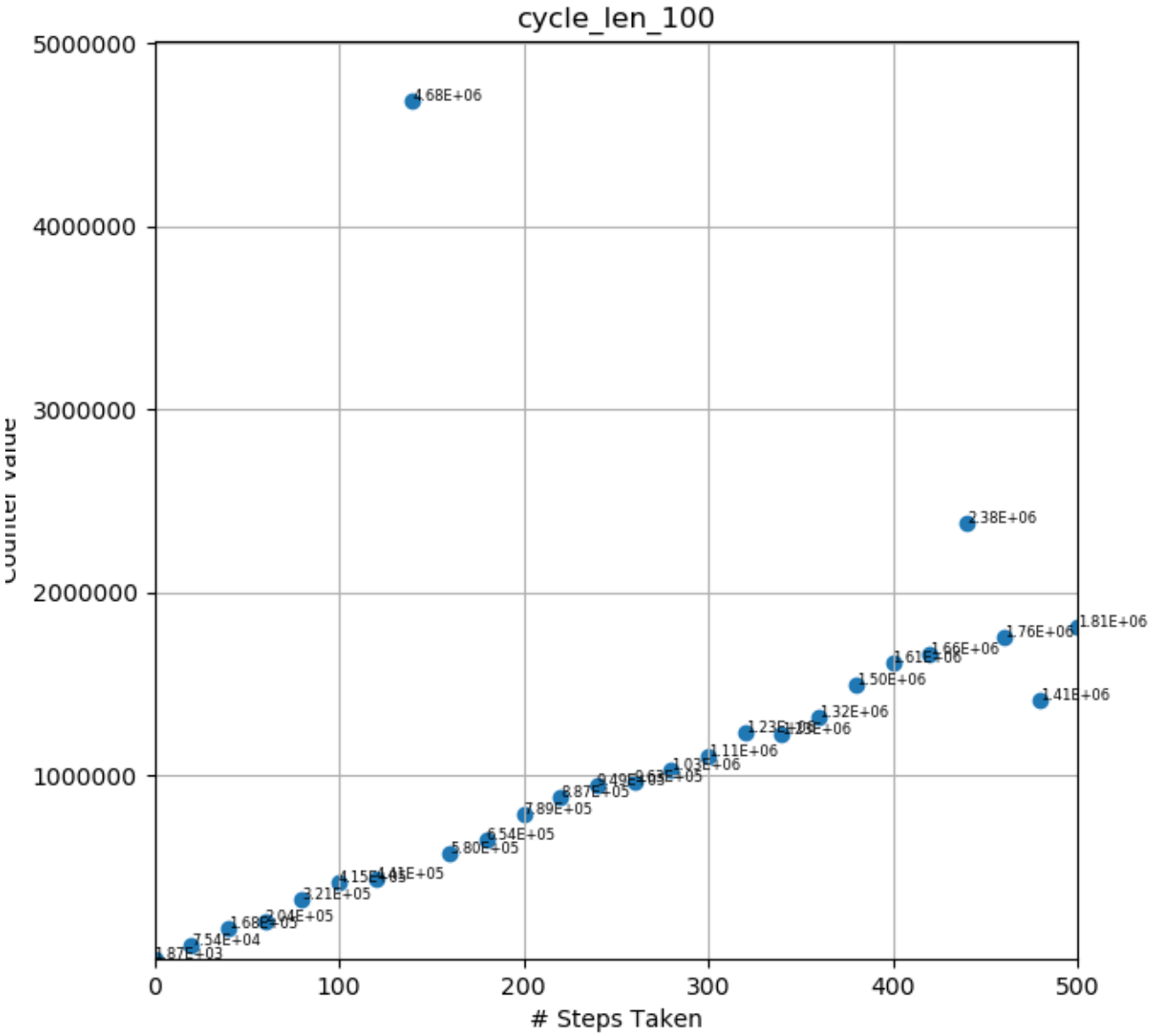


Results from the cycle-walking test, where the cycle has length 50.

SLOPE: ~4,000 Counter Increments per Step

Cycle-Walking Test Results

Intel RHD Graphics 520 OpenGL 45 Core

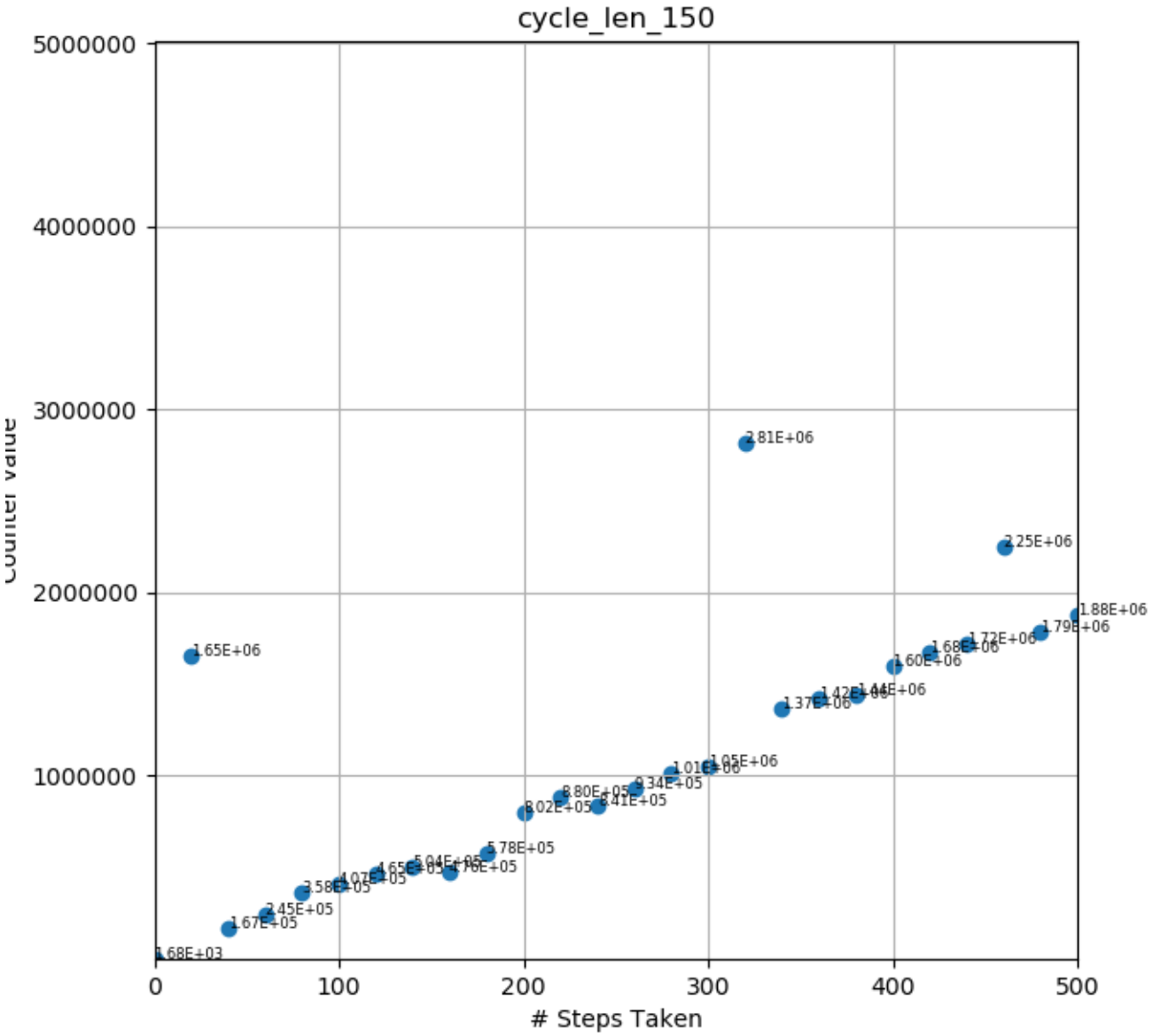


Results from the cycle-walking test, where the cycle has length 100.

SLOPE: ~4,000 Counter Increments per Step

Cycle-Walking Test Results

Intel RHD Graphics 520 OpenGL 45 Core

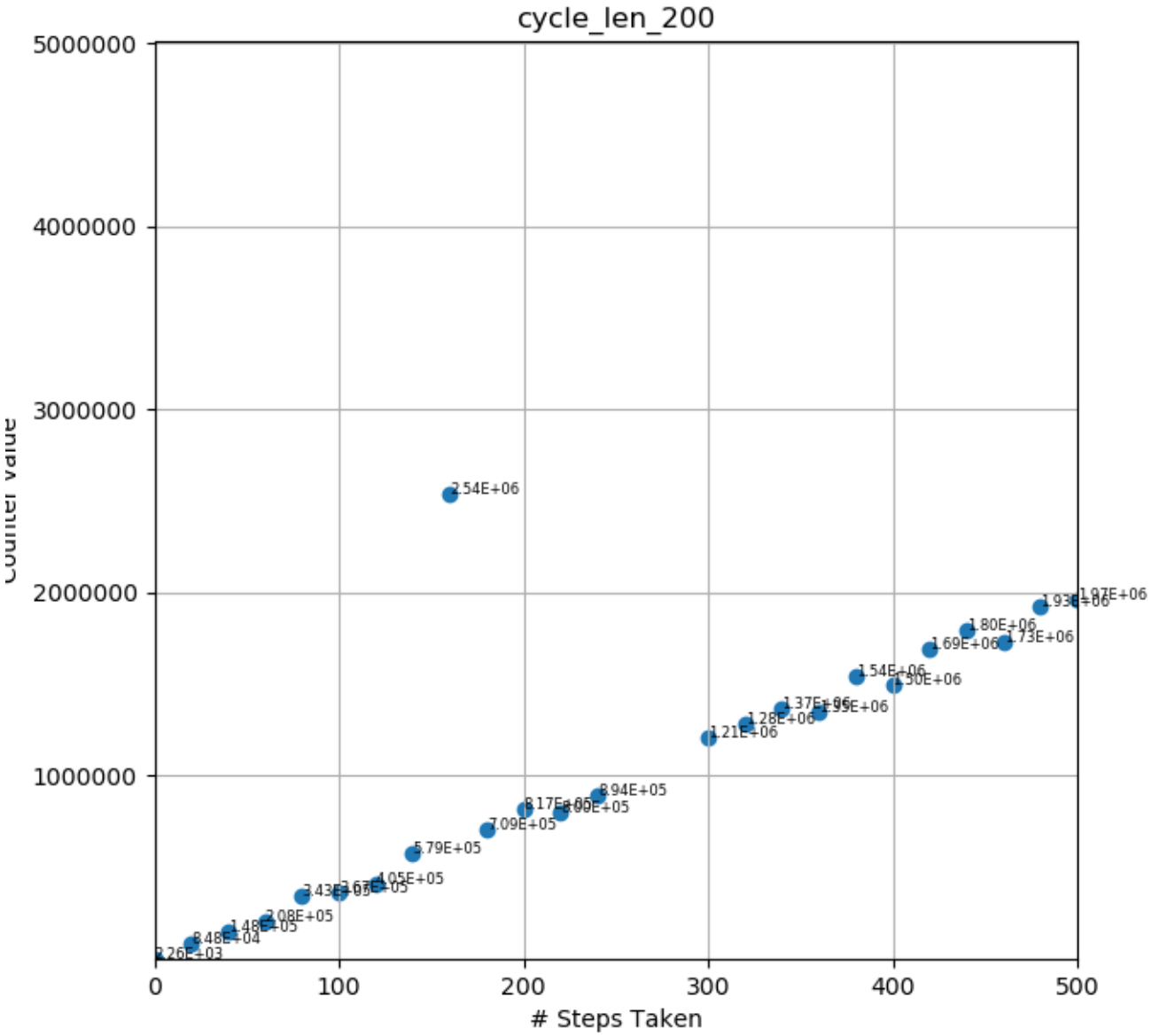


Results from the cycle-walking test, where the cycle has length 150.

SLOPE: ~4,000 Counter Increments per Step

Cycle-Walking Test Results

Intel RHD Graphics 520 OpenGL 45 Core

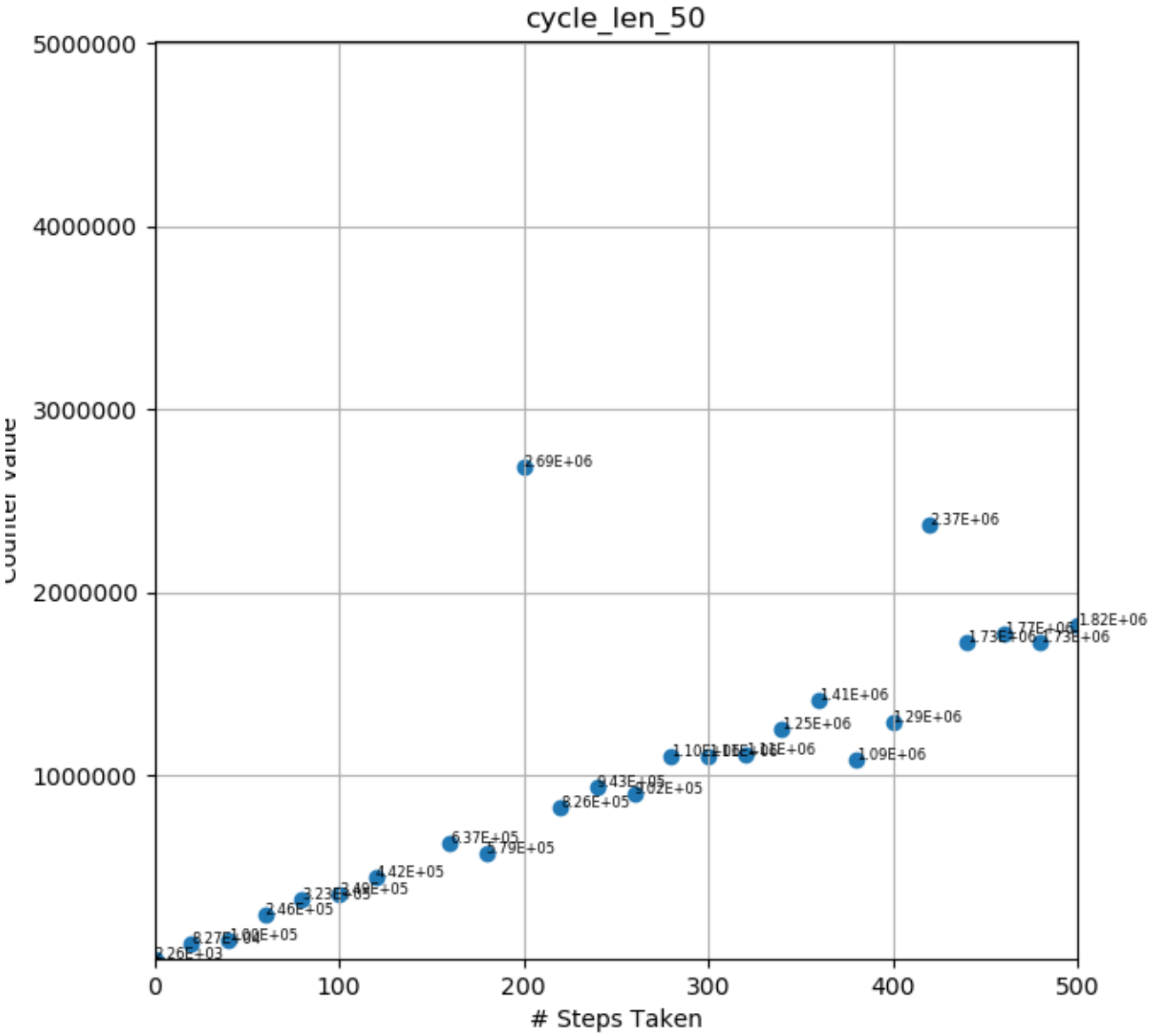


Results from the cycle-walking test, where the cycle has length 200.

SLOPE: ~4,000 Counter Increments per Step

Cycle-Walking Test Results

Intel RHD Graphics 620 OpenGL 45 Core

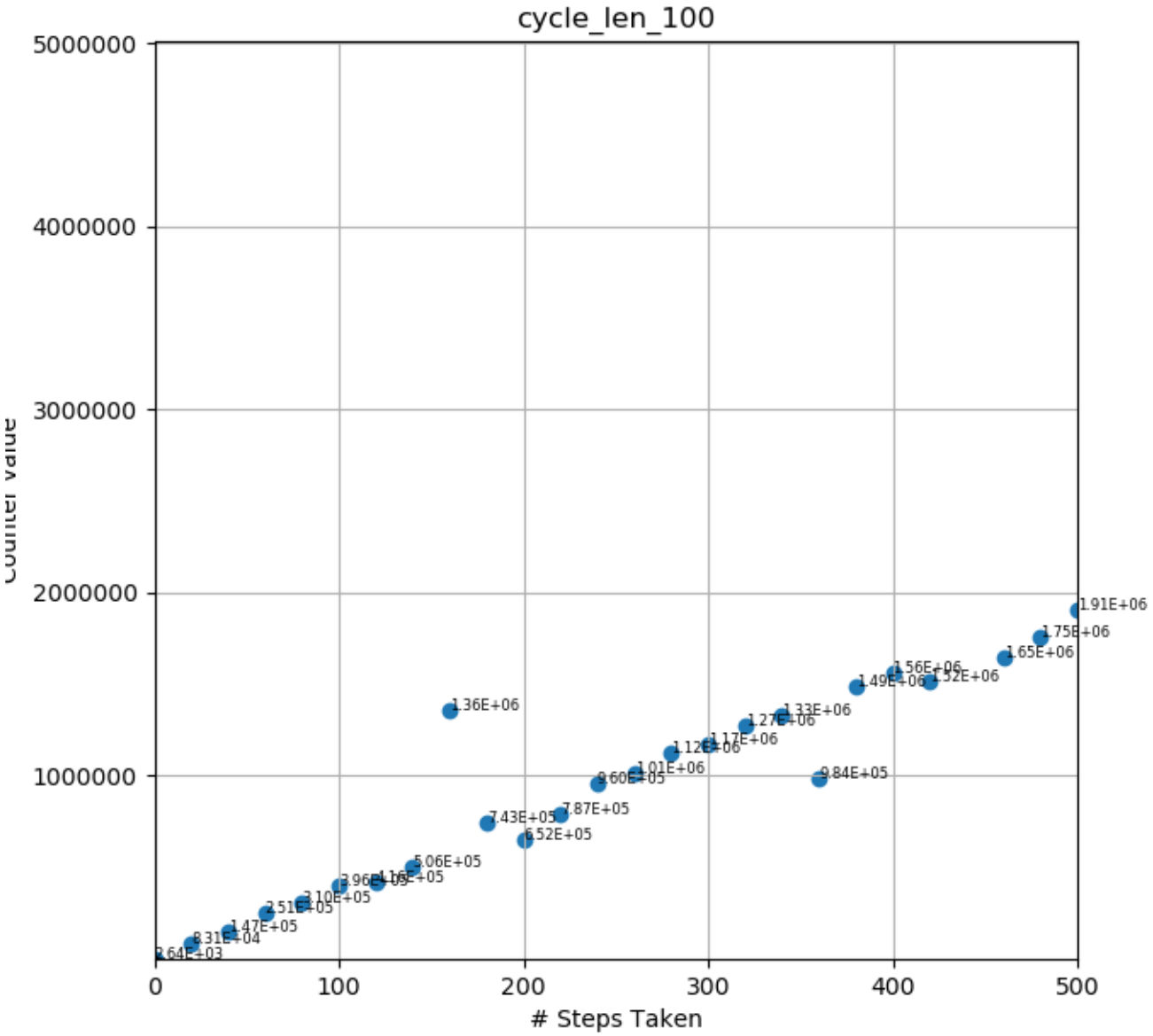


Results from the cycle-walking test, where the cycle has length 50.

SLOPE: ~3,800 Counter Increments per Step

Cycle-Walking Test Results

Intel RHD Graphics 620 OpenGL 45 Core

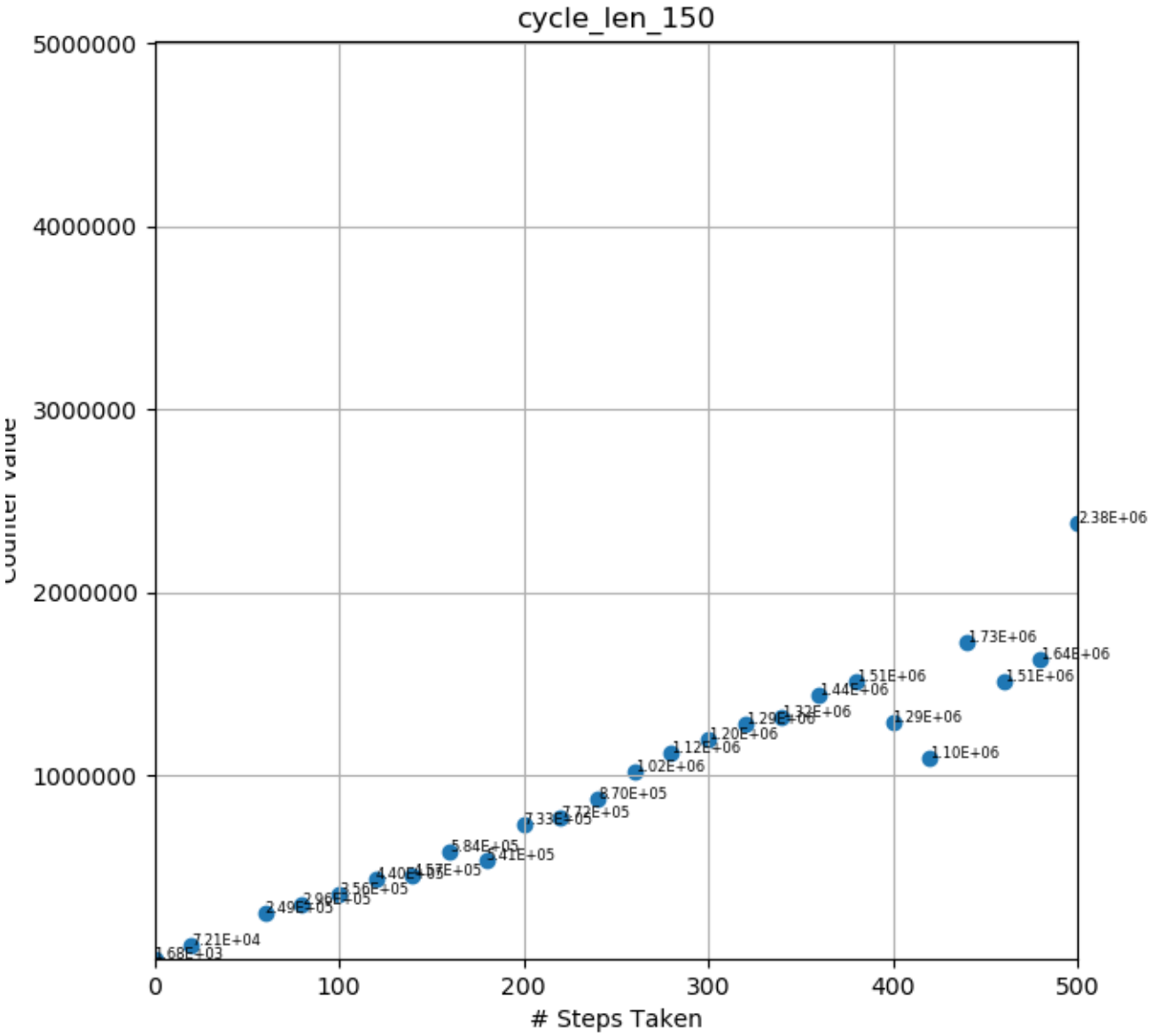


Results from the cycle-walking test, where the cycle has length 100.

SLOPE: ~3,900 Counter Increments per Step

Cycle-Walking Test Results

Intel RHD Graphics 620 OpenGL 45 Core

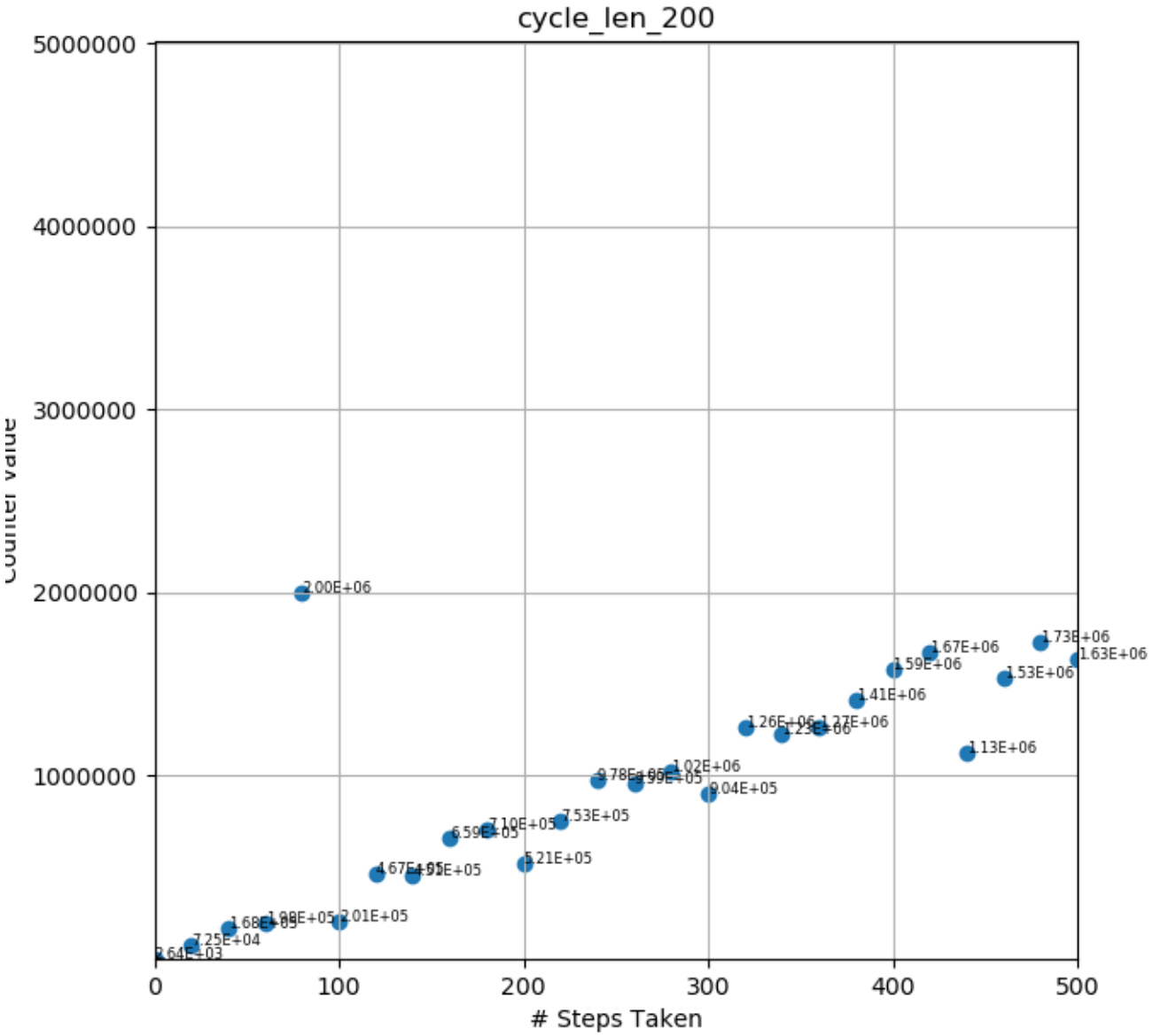


Results from the cycle-walking test, where the cycle has length 150.

SLOPE: ~3,900 Counter Increments per Step

Cycle-Walking Test Results

Intel RHD Graphics 620 OpenGL 45 Core

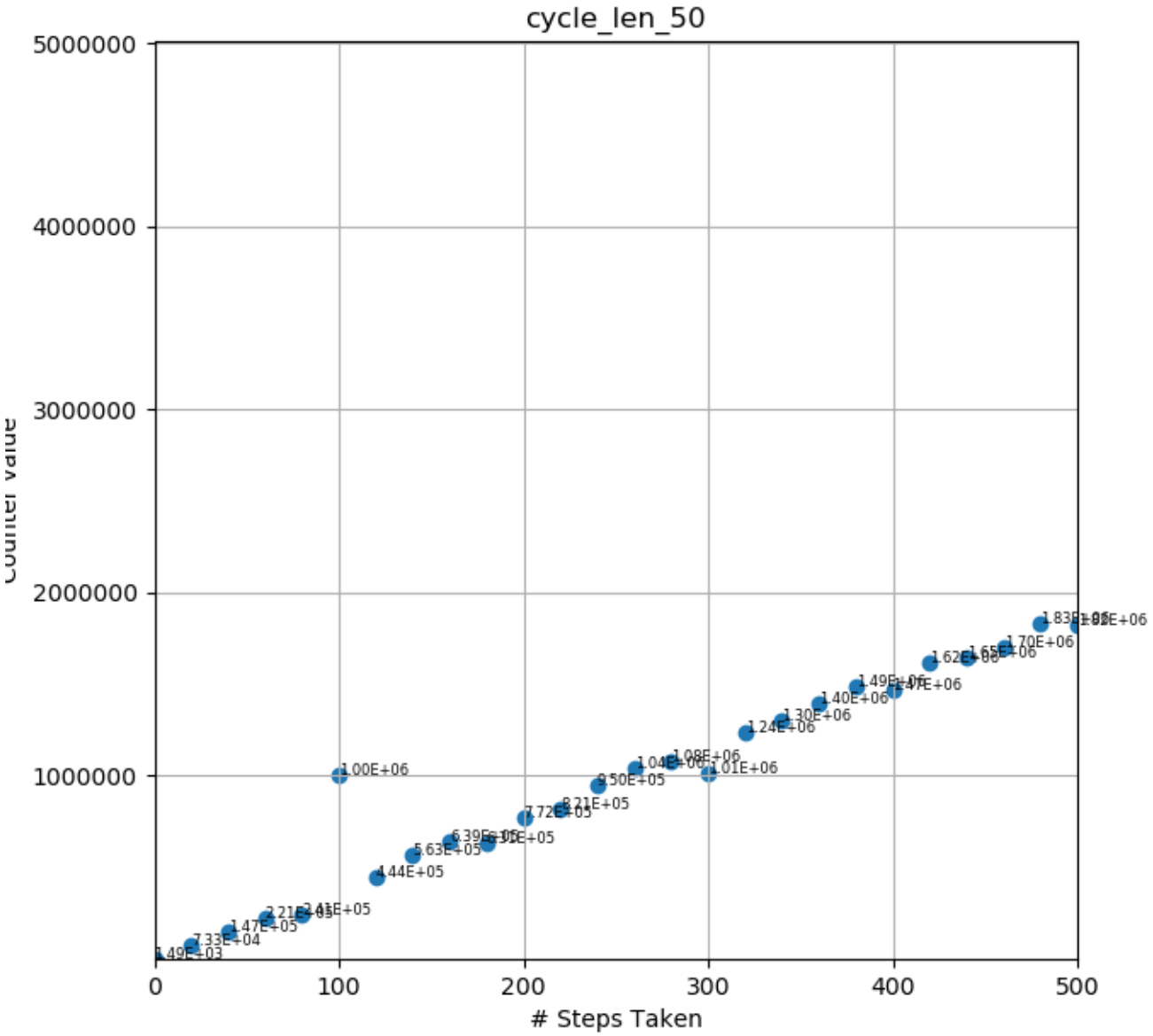


Results from the cycle-walking test, where the cycle has length 200.

SLOPE: ~3,900 Counter Increments per Step

Cycle-Walking Test Results

Intel RUHD Graphics 620 OpenGL 45 Core

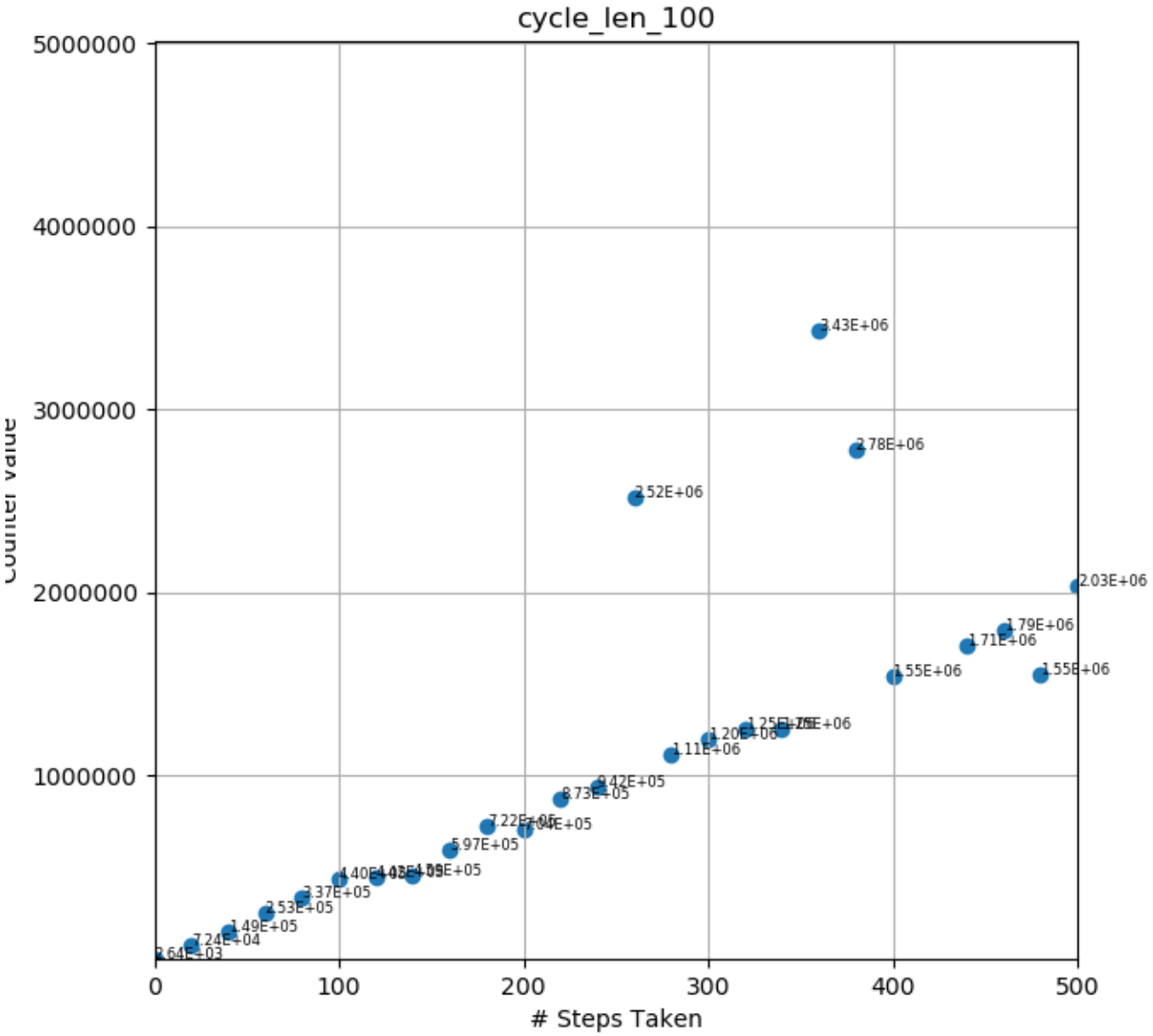


Results from the cycle-walking test, where the cycle has length 50.

SLOPE: ~3,800 Counter Increments per Step

Cycle-Walking Test Results

Intel RUHD Graphics 620 OpenGL 45 Core

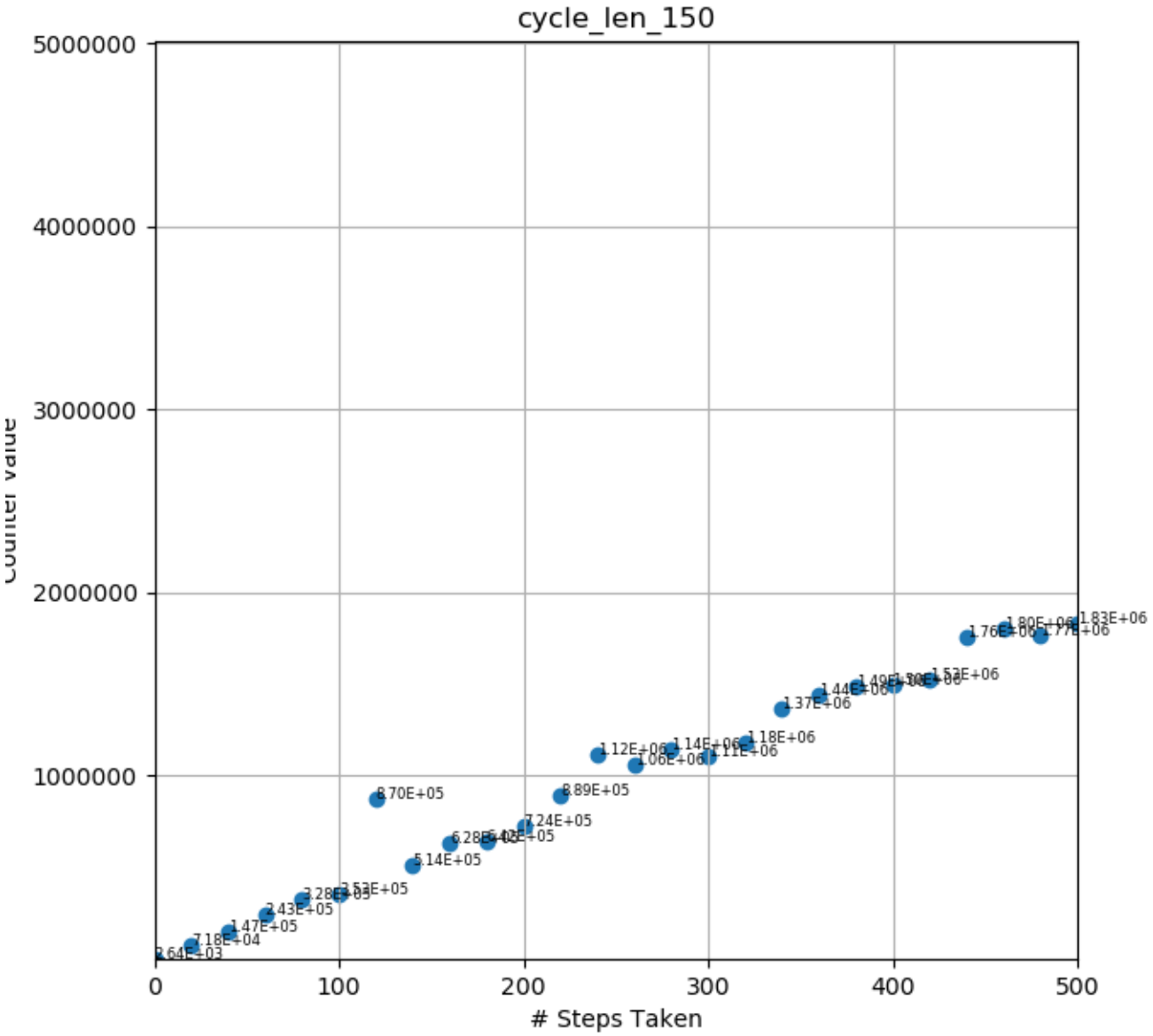


Results from the cycle-walking test, where the cycle has length 100.

SLOPE: ~3,900 Counter Increments per Step

Cycle-Walking Test Results

Intel RUHD Graphics 620 OpenGL 45 Core

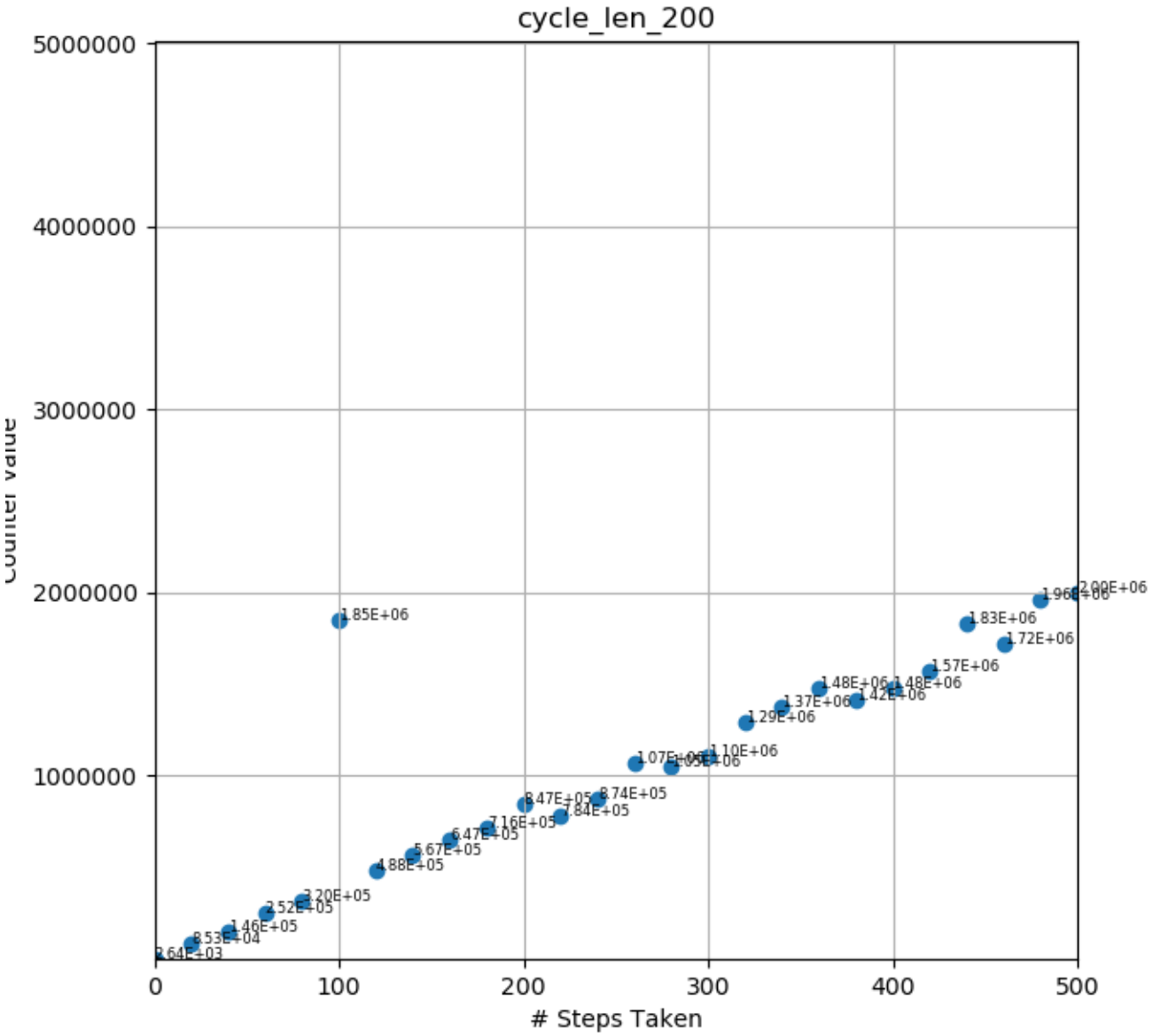


Results from the cycle-walking test, where the cycle has length 150.

SLOPE: ~3,900 Counter Increments per Step

Cycle-Walking Test Results

Intel RUHD Graphics 620 OpenGL 45 Core

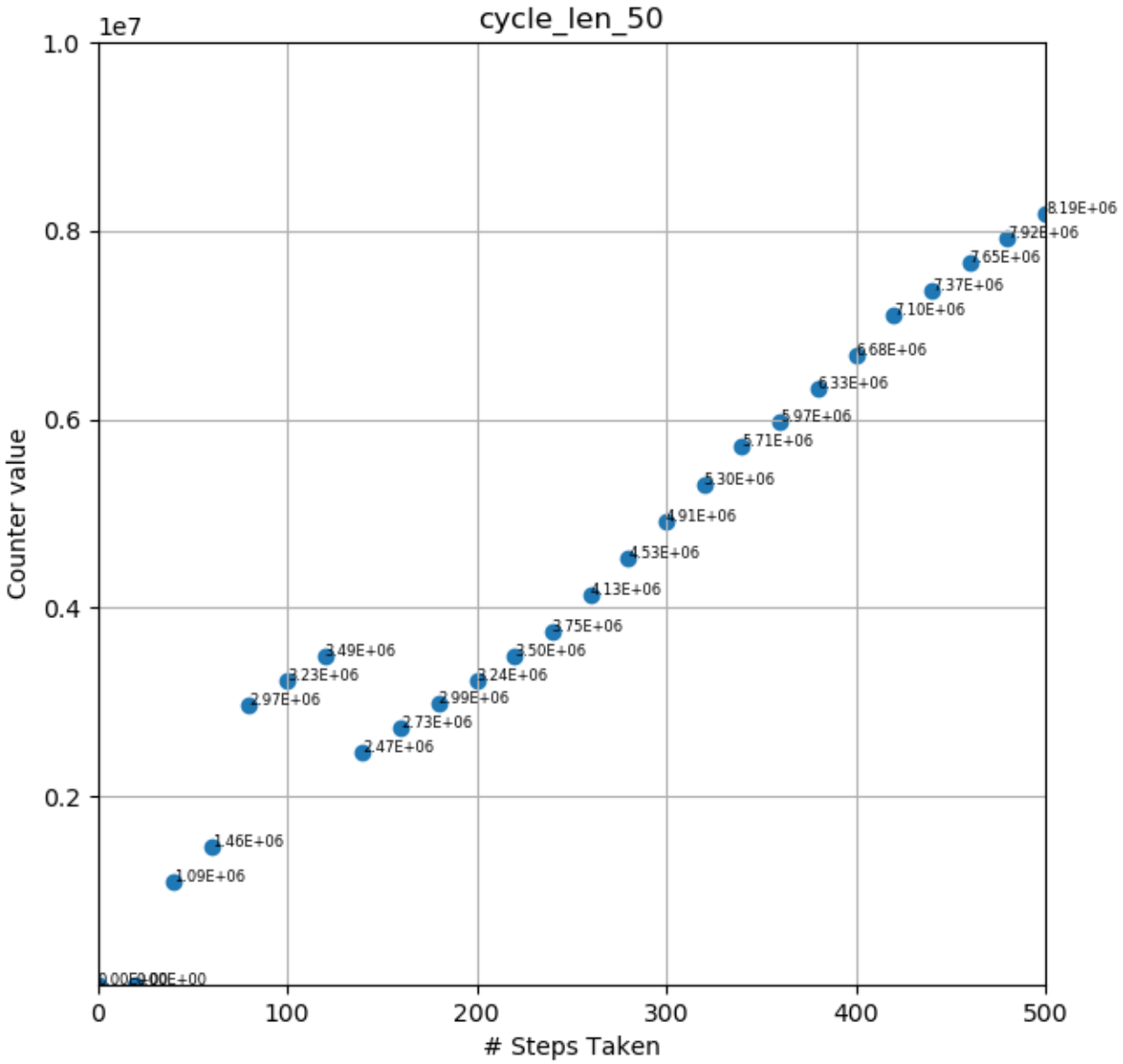


Results from the cycle-walking test, where the cycle has length 200.

SLOPE: ~3,900 Counter Increments per Step

Cycle-Walking Test Results

NVIDIA GeForce GTX 770 PCIe SSE2 OpenGL 45 core

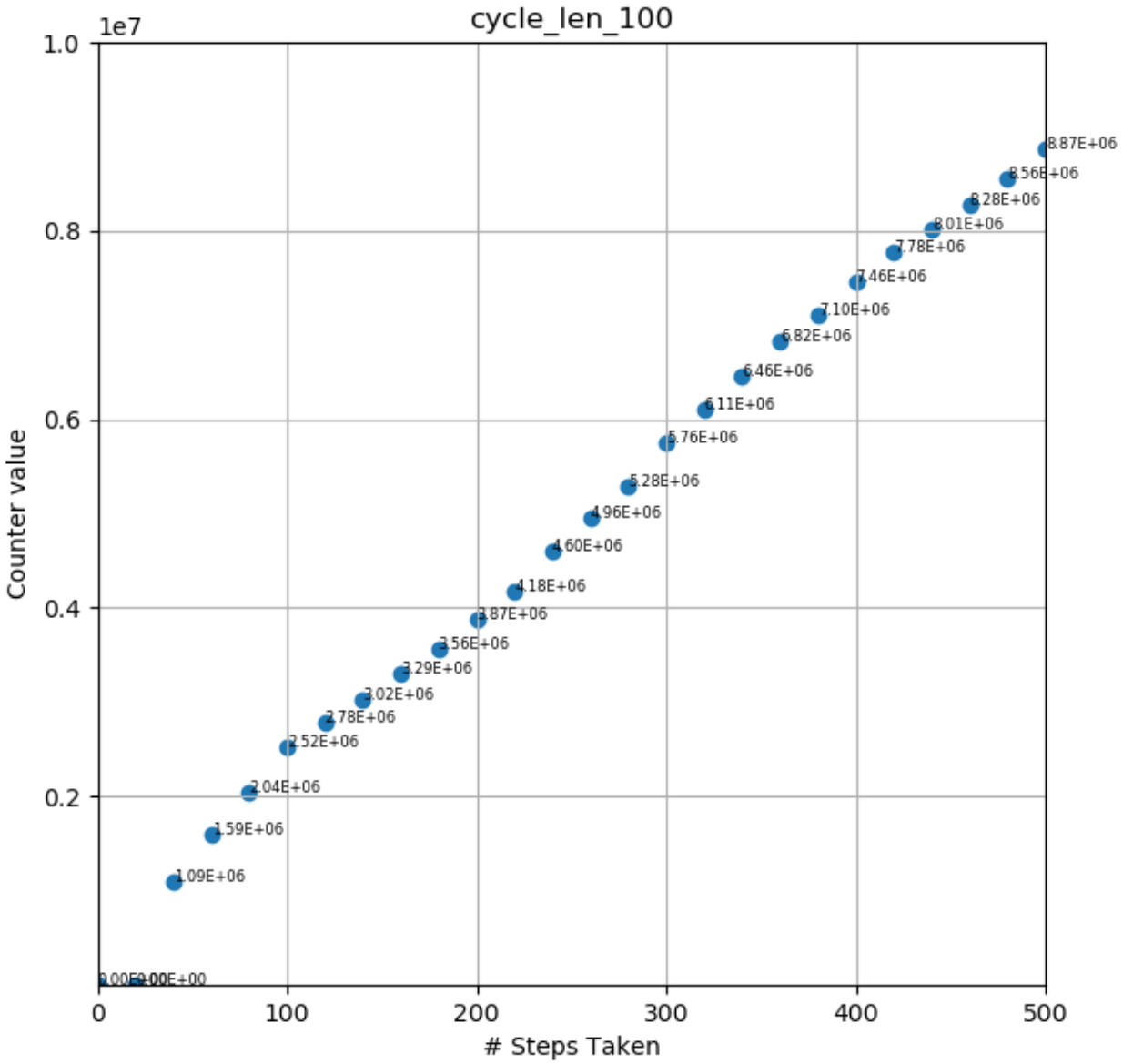


Results from the cycle-walking test, where the cycle has length 50.

SLOPE: ~16,000 Counter Increments per Step

Cycle-Walking Test Results

NVIDIA GeForce GTX 770 PCIe SSE2 OpenGL 45 core

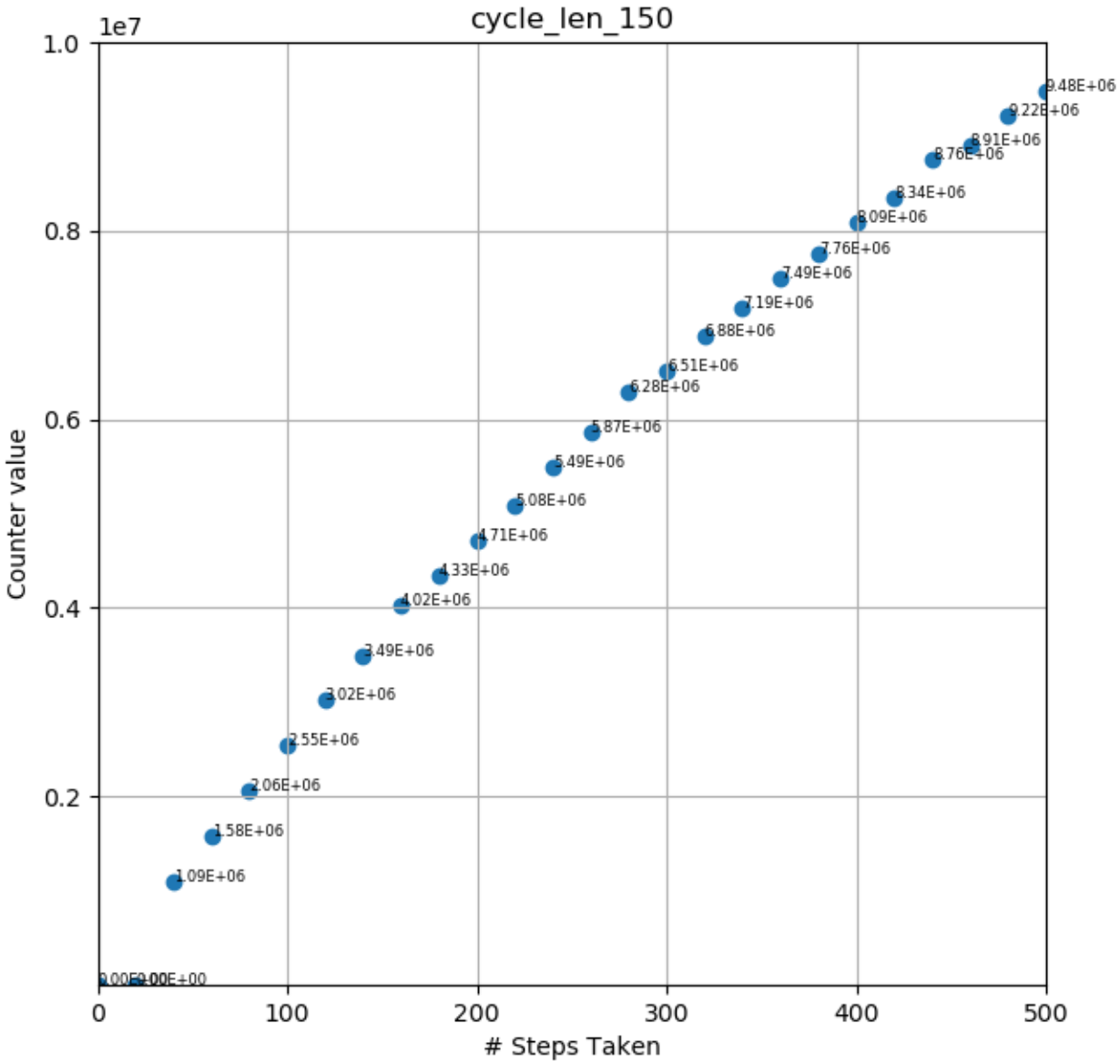


Results from the cycle-walking test, where the cycle has length 100.

SLOPE: ~25,000 Counter Increments per Step at 100 steps and below
~16,000 Counter Increments per Step at 100 steps and above

Cycle-Walking Test Results

NVIDIA GeForce GTX 770 PCIe SSE2 OpenGL 45 core

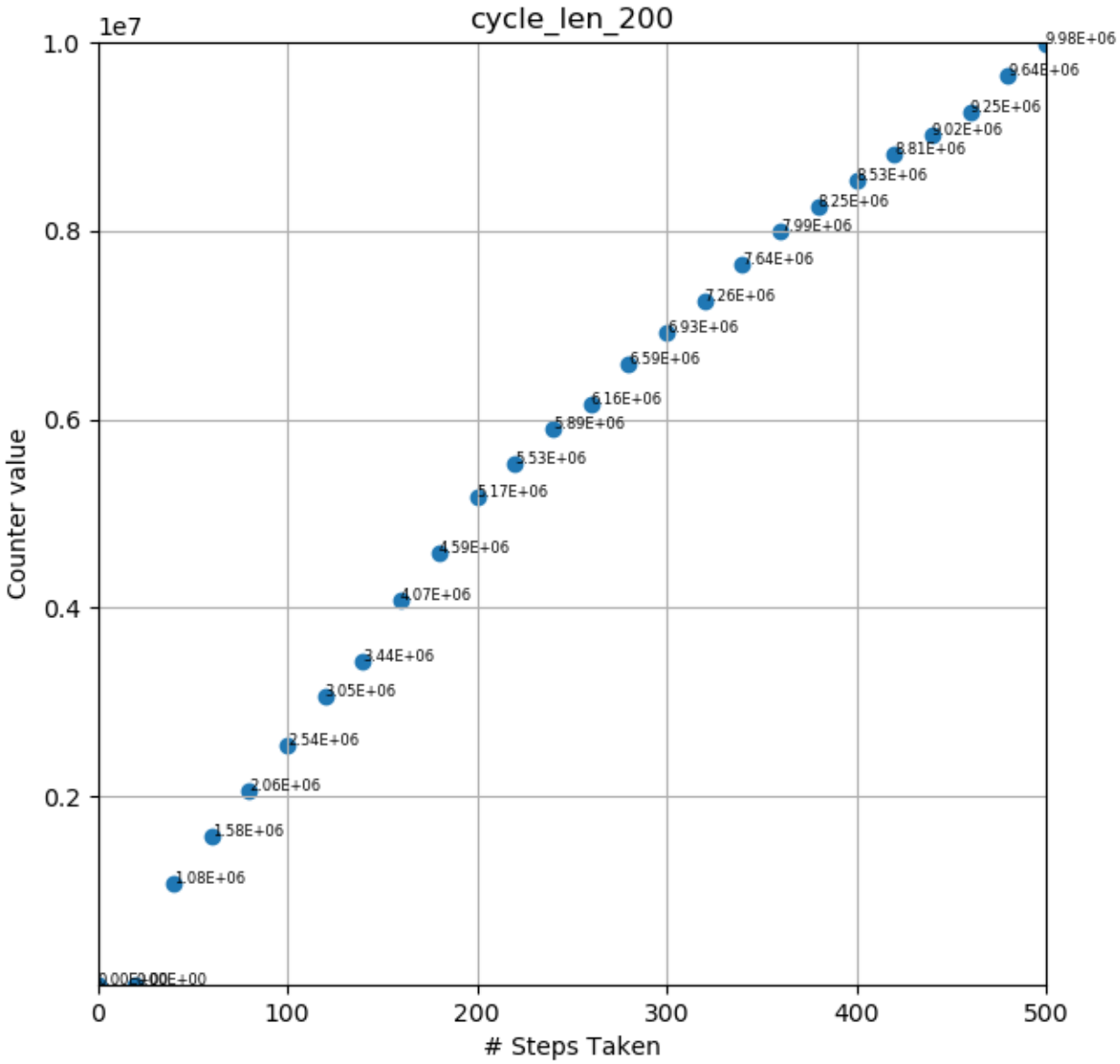


Results from the cycle-walking test, where the cycle has length 150.

SLOPE: ~25,000 Counter Increments per Step at 150 steps and below
~16,000 Counter Increments per Step at 150 steps and above

Cycle-Walking Test Results

NVIDIA GeForce GTX 770 PCIe SSE2 OpenGL 45 core

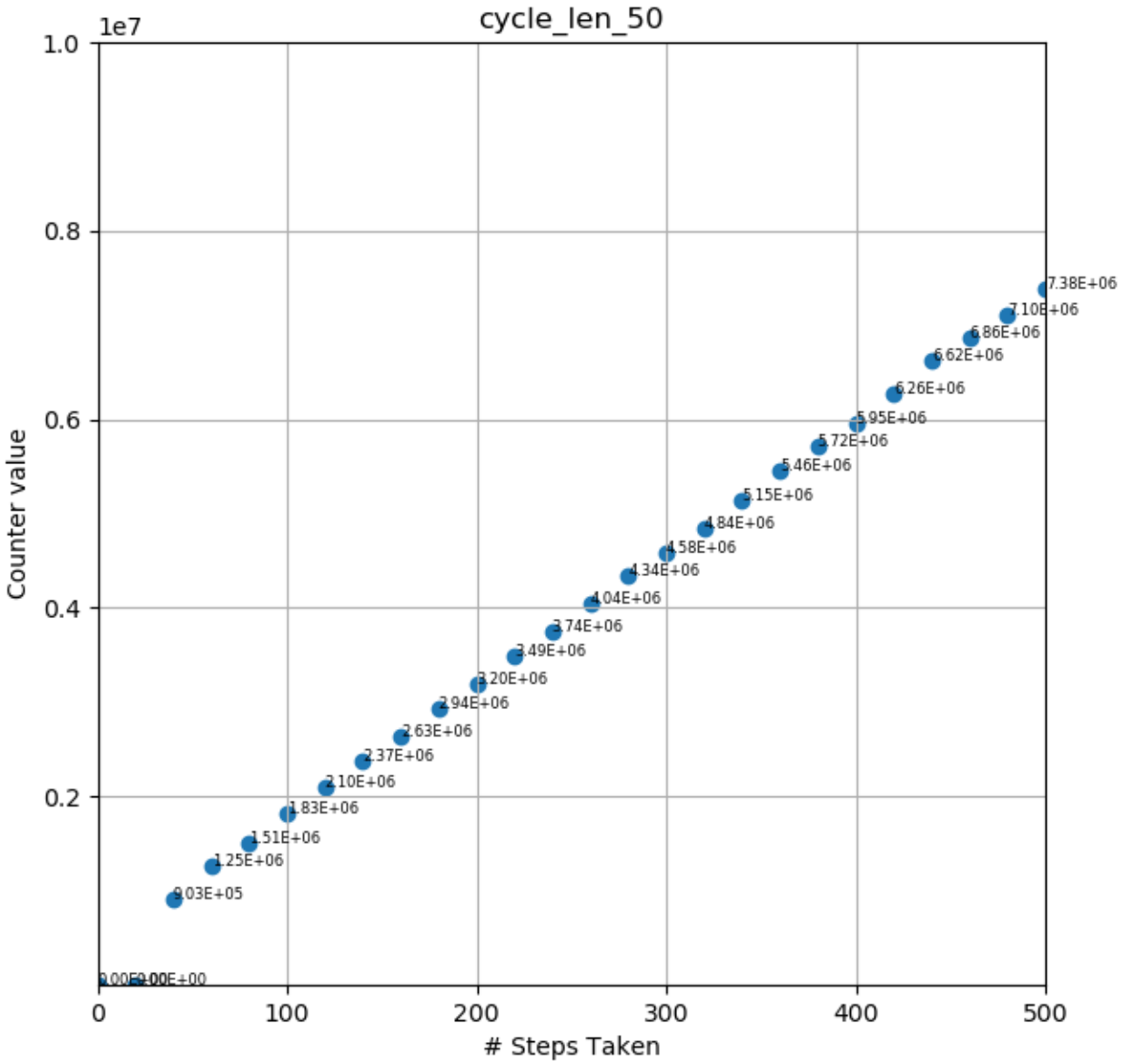


Results from the cycle-walking test, where the cycle has length 200.

SLOPE: ~25,000 Counter Increments per Step at 200 steps and below
~16,000 Counter Increments per Step at 200 steps and above

Cycle-Walking Test Results

NVIDIA GeForce GTX 1060 6GB PCIe SSE2 OpenGL 45 core

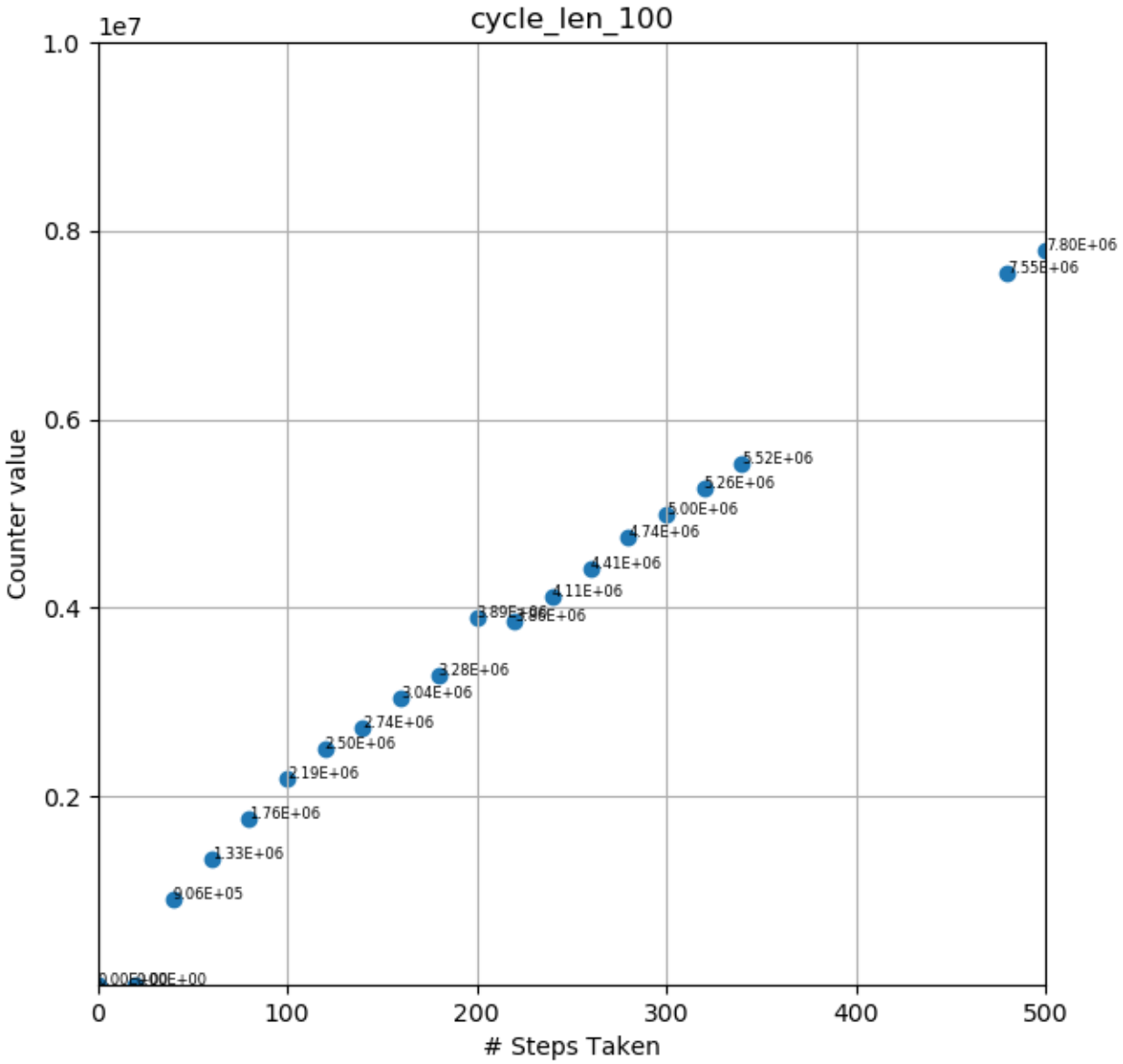


Results from the cycle-walking test, where the cycle has length 50.

SLOPE: ~15,000 Counter Increments per Step

Cycle-Walking Test Results

NVIDIA GeForce GTX 1060 6GB PCIe SSE2 OpenGL 45 core

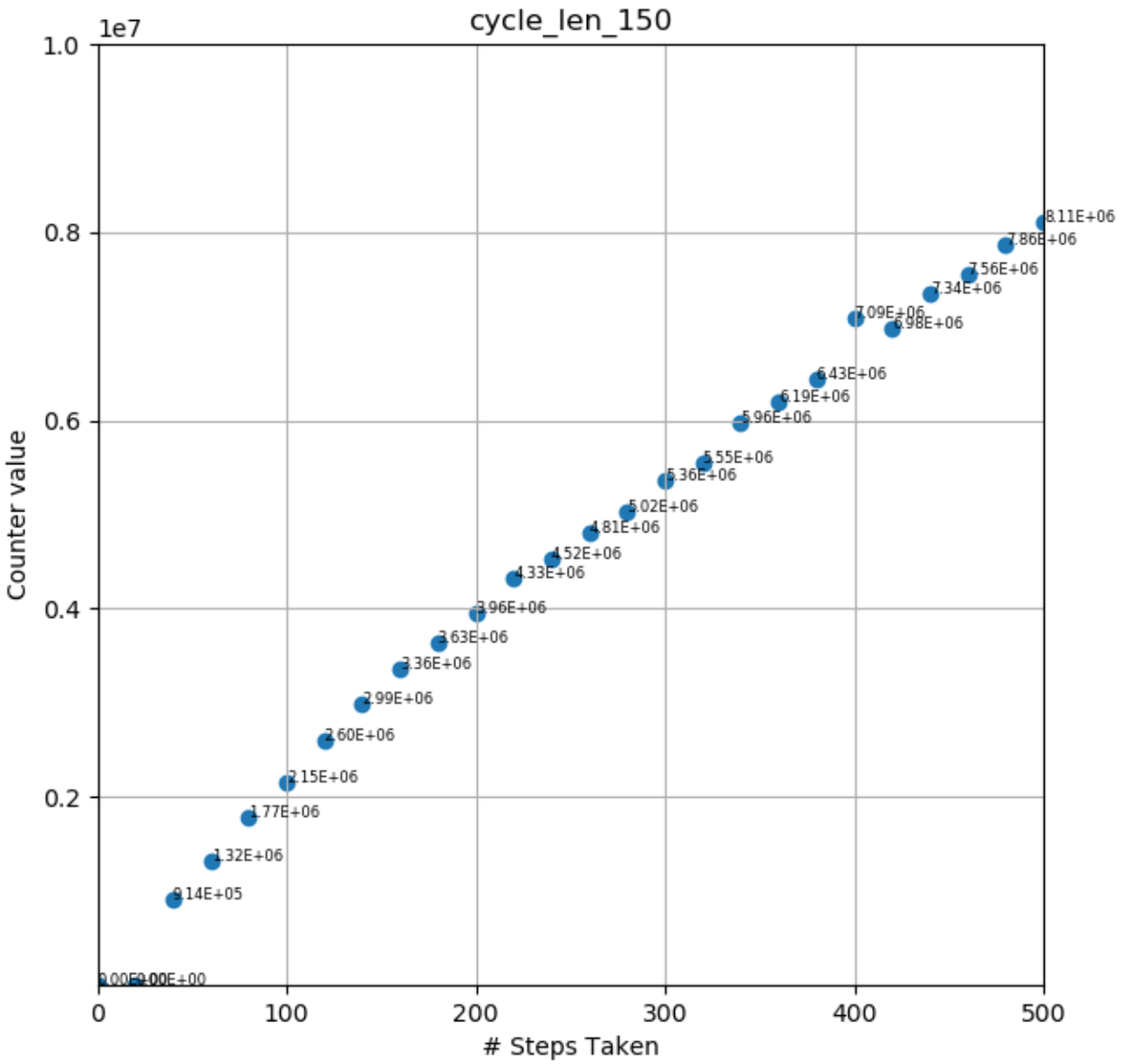


Results from the cycle-walking test, where the cycle has length 100.

SLOPE: ~22,000 Counter Increments per Step at 100 steps and below
~14,000 Counter Increments per Step at 100 steps and above

Cycle-Walking Test Results

NVIDIA GeForce GTX 1060 6GB PCIe SSE2 OpenGL 45 core

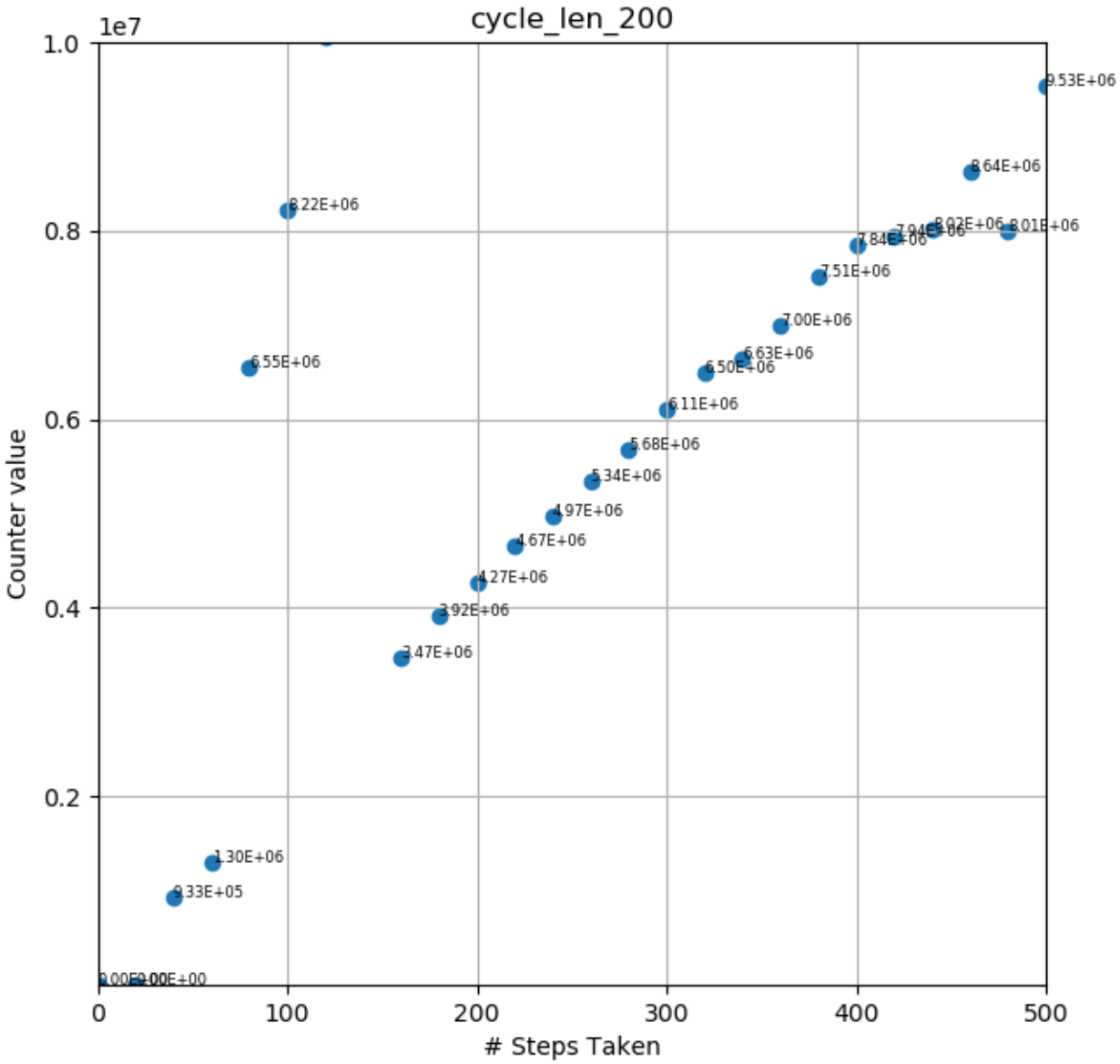


Results from the cycle-walking test, where the cycle has length 150.

SLOPE: ~22,000 Counter Increments per Step at 150 steps and below
~14,000 Counter Increments per Step at 150 steps and above

Cycle-Walking Test Results

NVIDIA GeForce GTX 1060 6GB PCIe SSE2 OpenGL 45 core

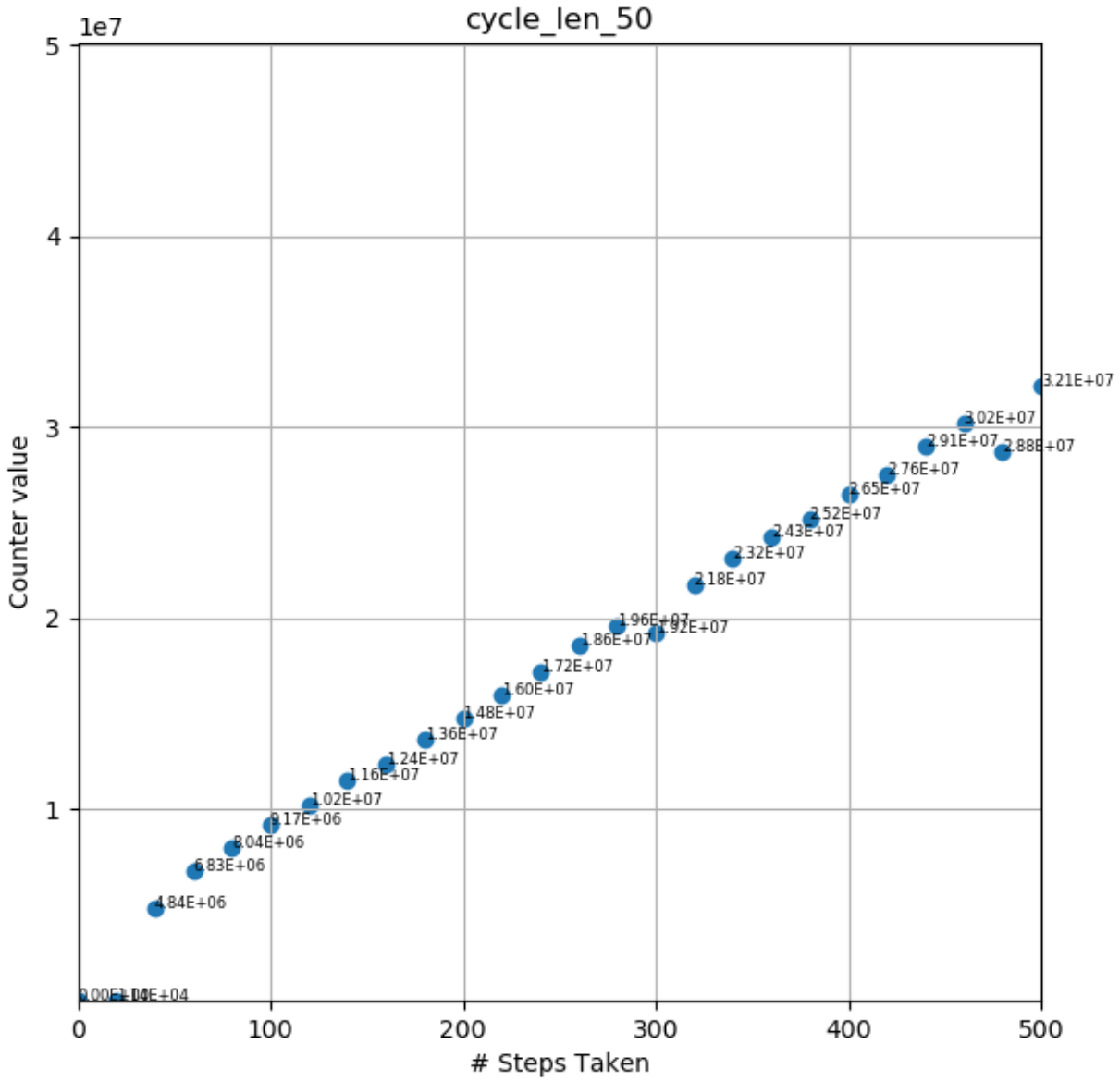


Results from the cycle-walking test, where the cycle has length 200.

SLOPE: ~21,000 Counter Increments per Step at 200 steps and below
~18,000 Counter Increments per Step at 200 steps and above

Cycle-Walking Test Results

NVIDIA GeForce GTX 1070 PCIe SSE2 OpenGL 45 core

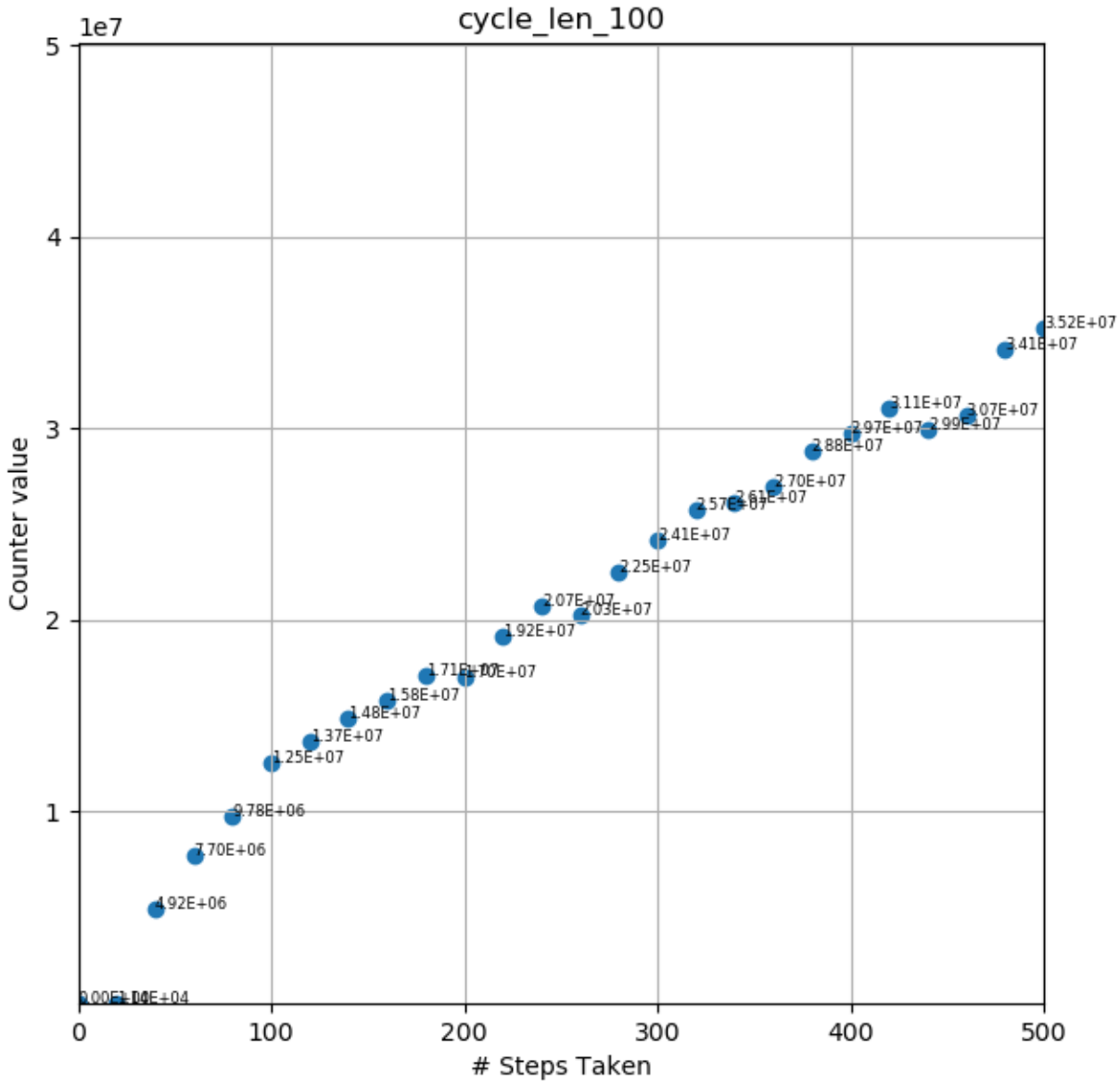


Results from the cycle-walking test, where the cycle has length 50.

SLOPE: ~57,000 Counter Increments per Step

Cycle-Walking Test Results

NVIDIA GeForce GTX 1070 PCIe SSE2 OpenGL 45 core

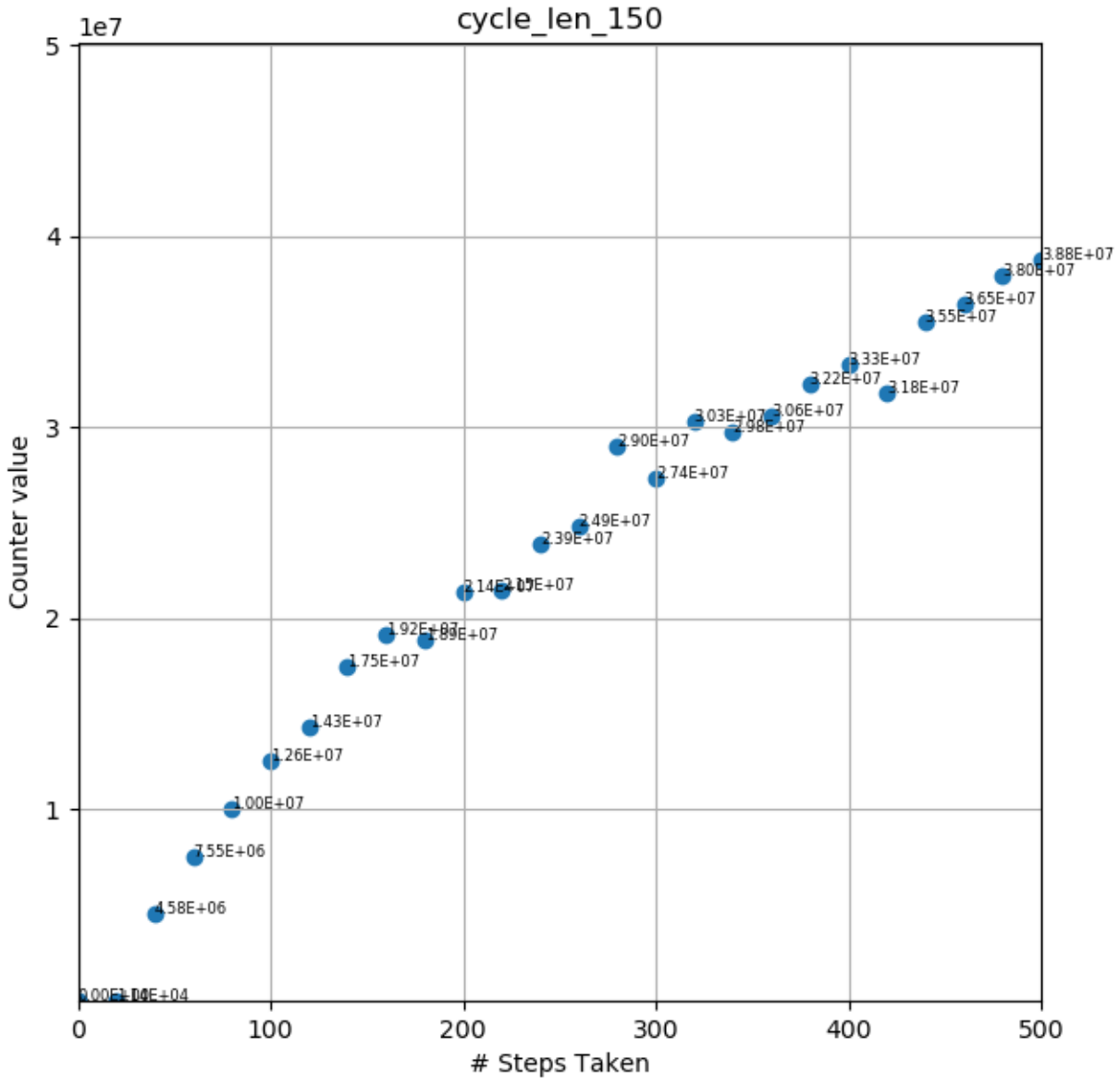


Results from the cycle-walking test, where the cycle has length 100.

SLOPE: ~125,000 Counter Increments per Step at 100 steps and below
~56,000 Counter Increments per Step at 100 steps and above

Cycle-Walking Test Results

NVIDIA GeForce GTX 1070 PCIe SSE2 OpenGL 45 core

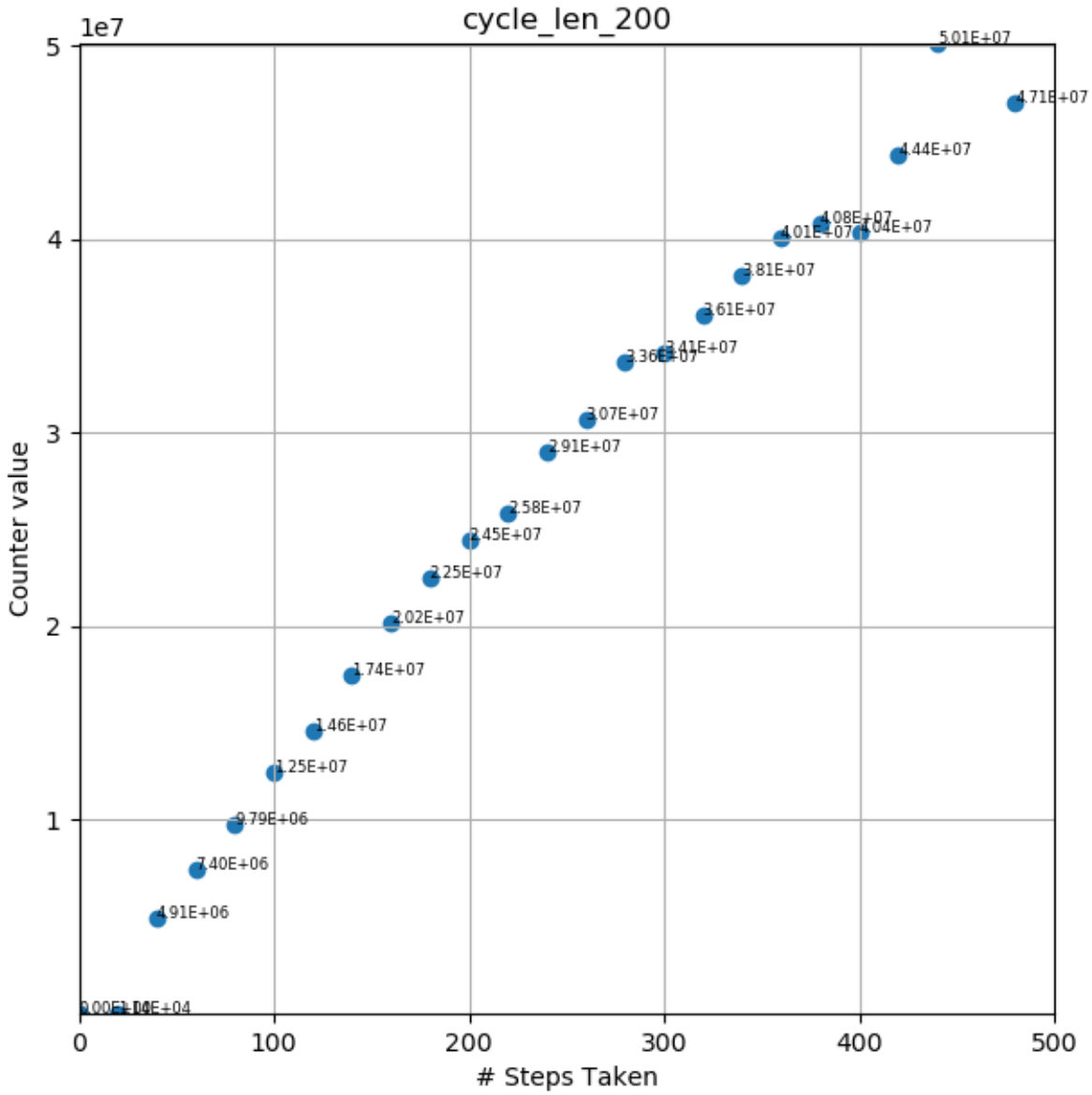


Results from the cycle-walking test, where the cycle has length 150.

SLOPE: ~125,000 Counter Increments per Step at 150 steps and below
~58,000 Counter Increments per Step at 150 steps and above

Cycle-Walking Test Results

NVIDIA GeForce GTX 1070 PCIe SSE2 OpenGL 45 core

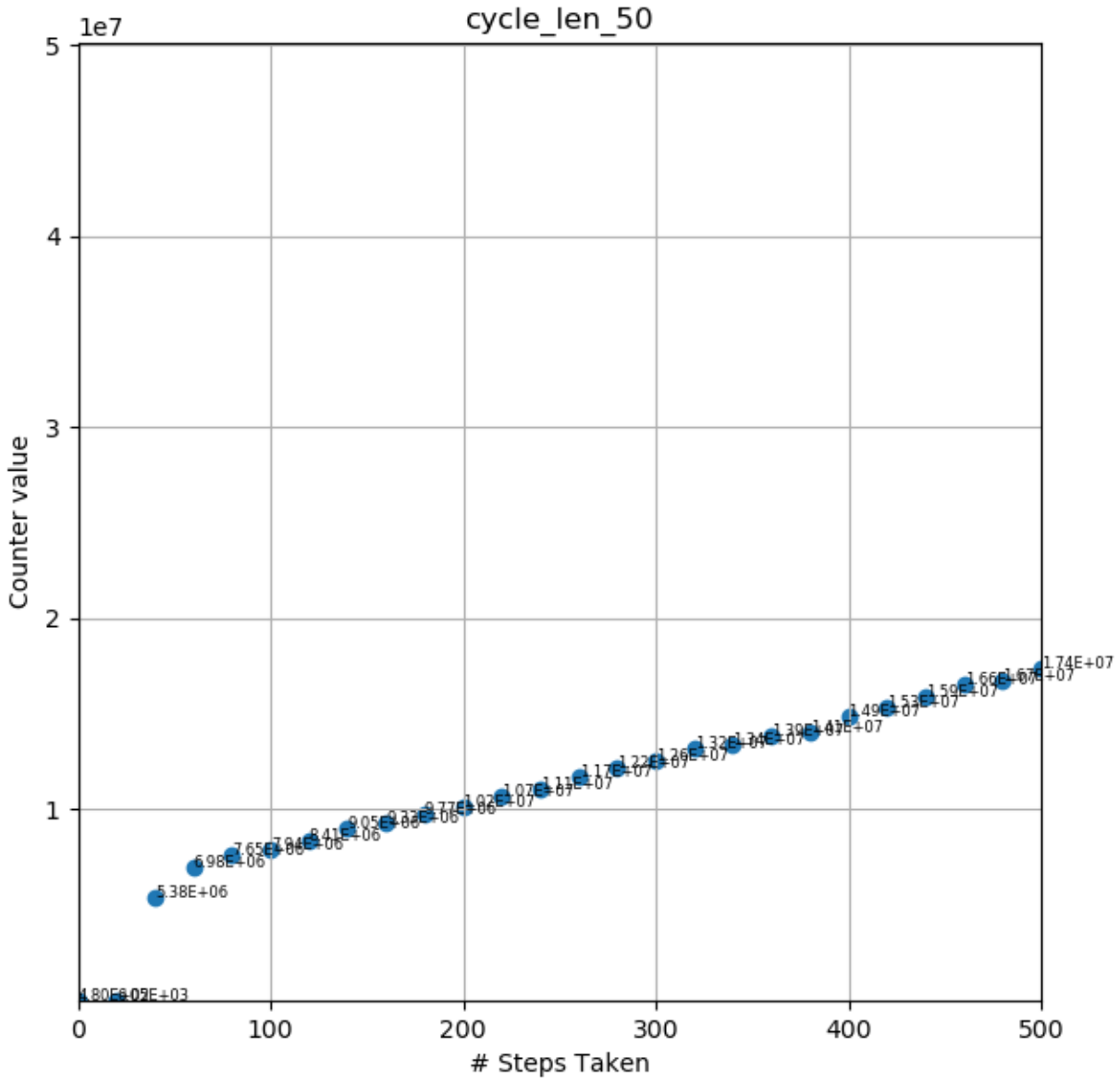


Results from the cycle-walking test, where the cycle has length 200.

SLOPE: ~123,000 Counter Increments per Step at 200 steps and below
~90,000 Counter Increments per Step at 200 steps and above

Cycle-Walking Test Results

NVIDIA GeForce GTX 1080 PCIe SSE2 OpenGL 45 core

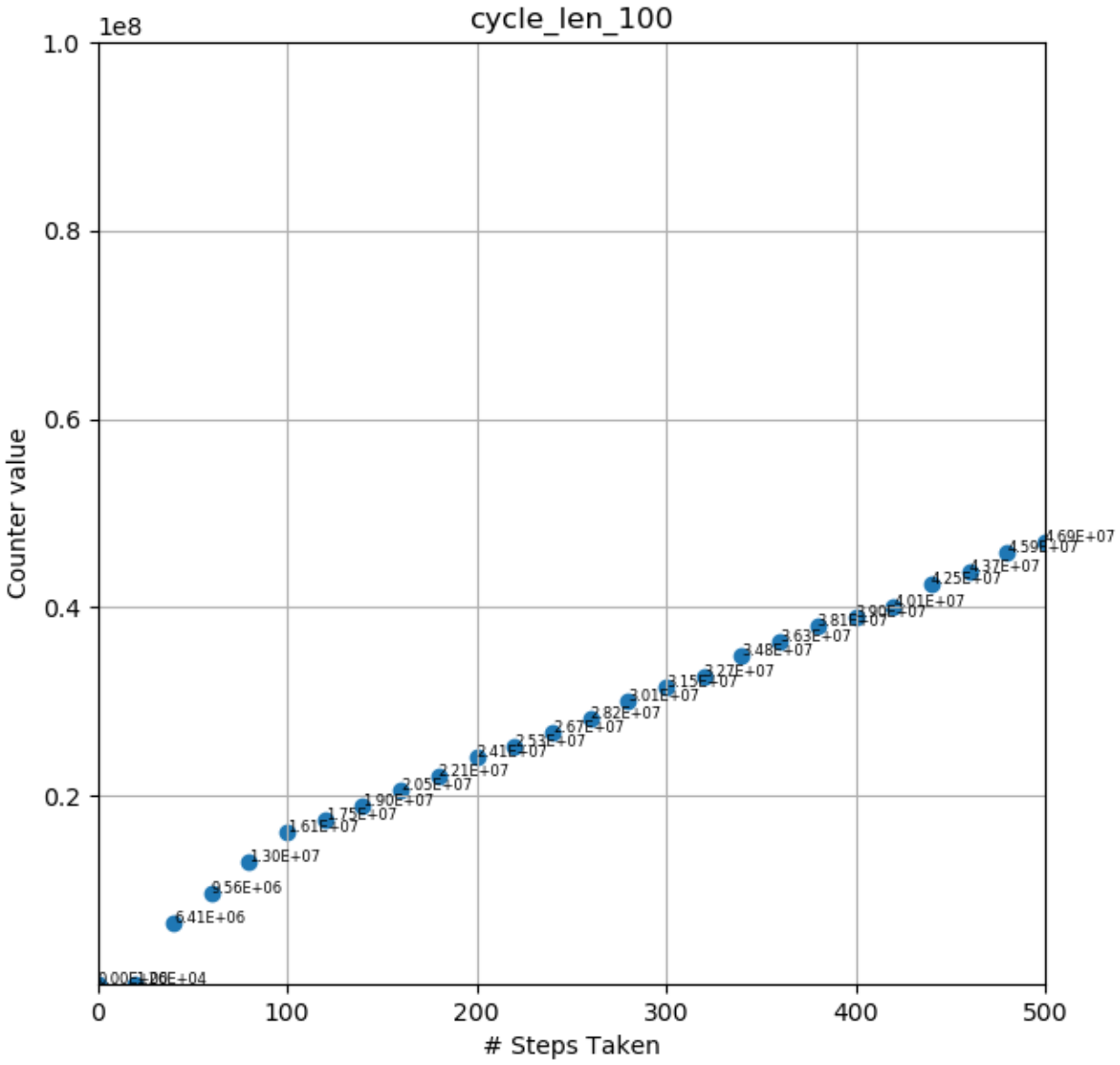


Results from the cycle-walking test, where the cycle has length 50.

SLOPE: ~49,000 Counter Increments per Step,
only takes data points at 60 through 500 steps into account.

Cycle-Walking Test Results

NVIDIA GeForce GTX 1080 PCIe SSE2 OpenGL 45 core

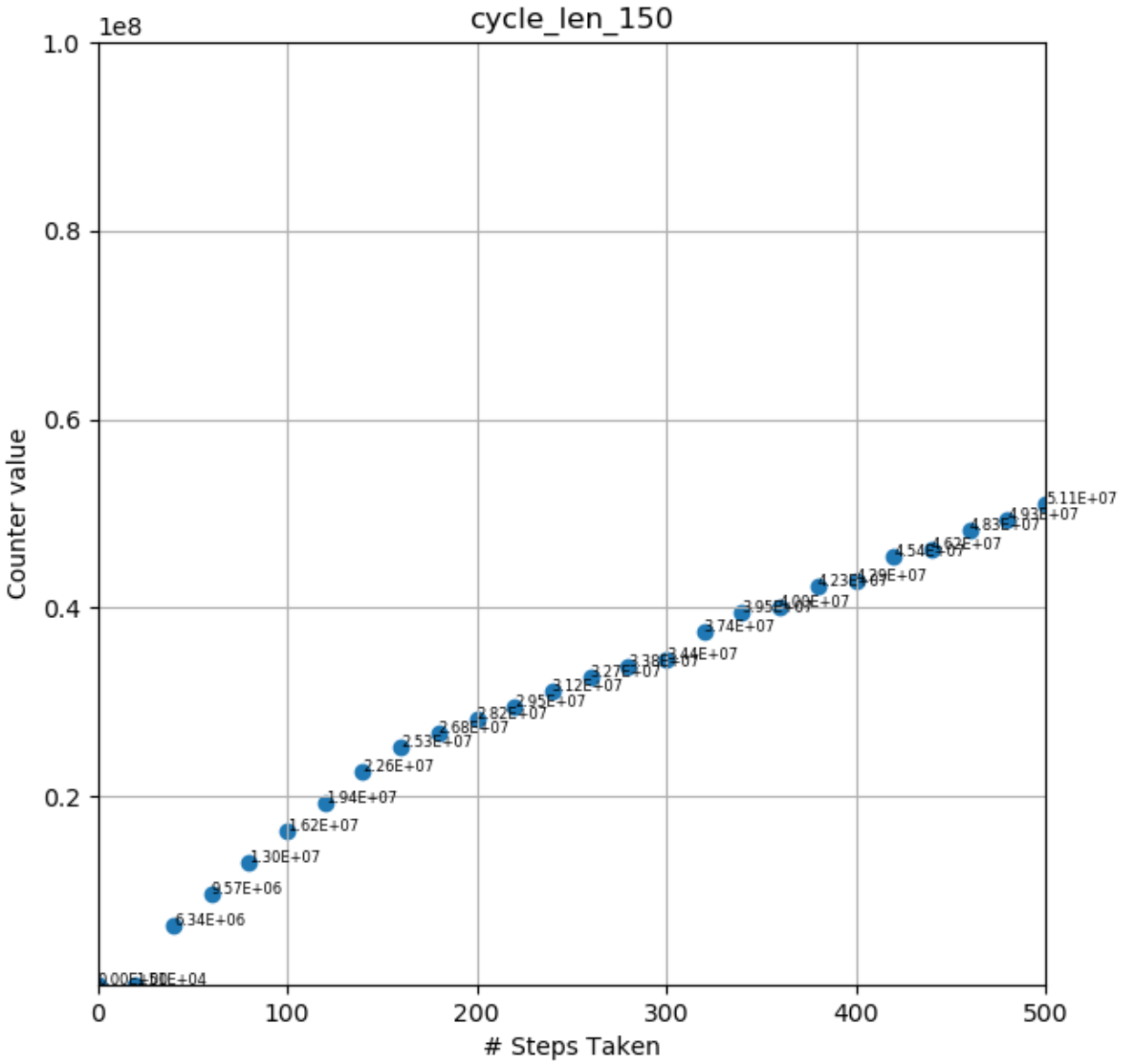


Results from the cycle-walking test, where the cycle has length 100.

SLOPE: ~161,000 Counter Increments per Step at 100 steps and below
~77,000 Counter Increments per Step at 100 steps and above

Cycle-Walking Test Results

NVIDIA GeForce GTX 1080 PCIe SSE2 OpenGL 45 core

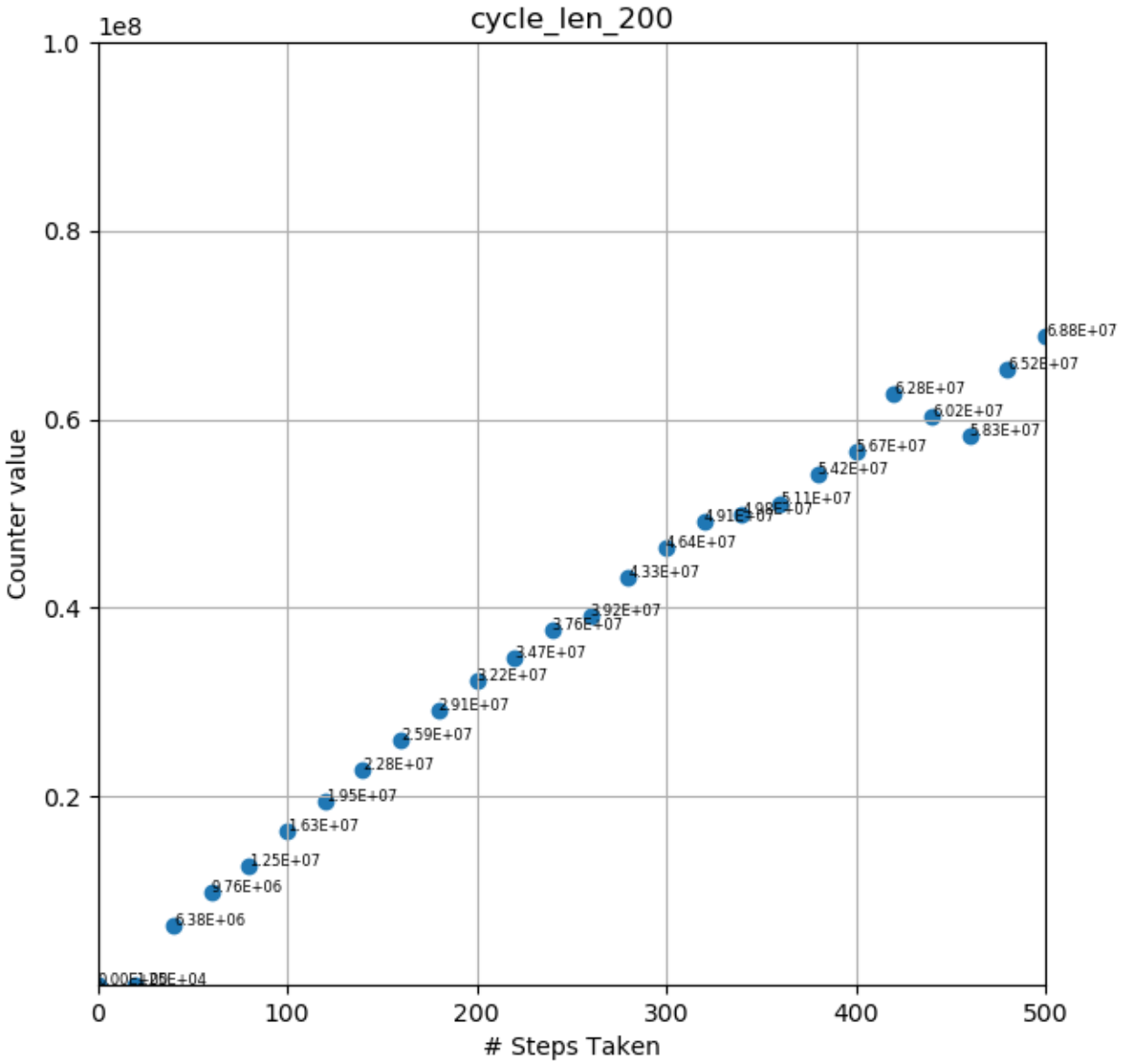


Results from the cycle-walking test, where the cycle has length 150.

SLOPE: ~161,000 Counter Increments per Step at 150 steps and below
~76,000 Counter Increments per Step at 150 steps and above

Cycle-Walking Test Results

NVIDIA GeForce GTX 1080 PCIe SSE2 OpenGL 45 core

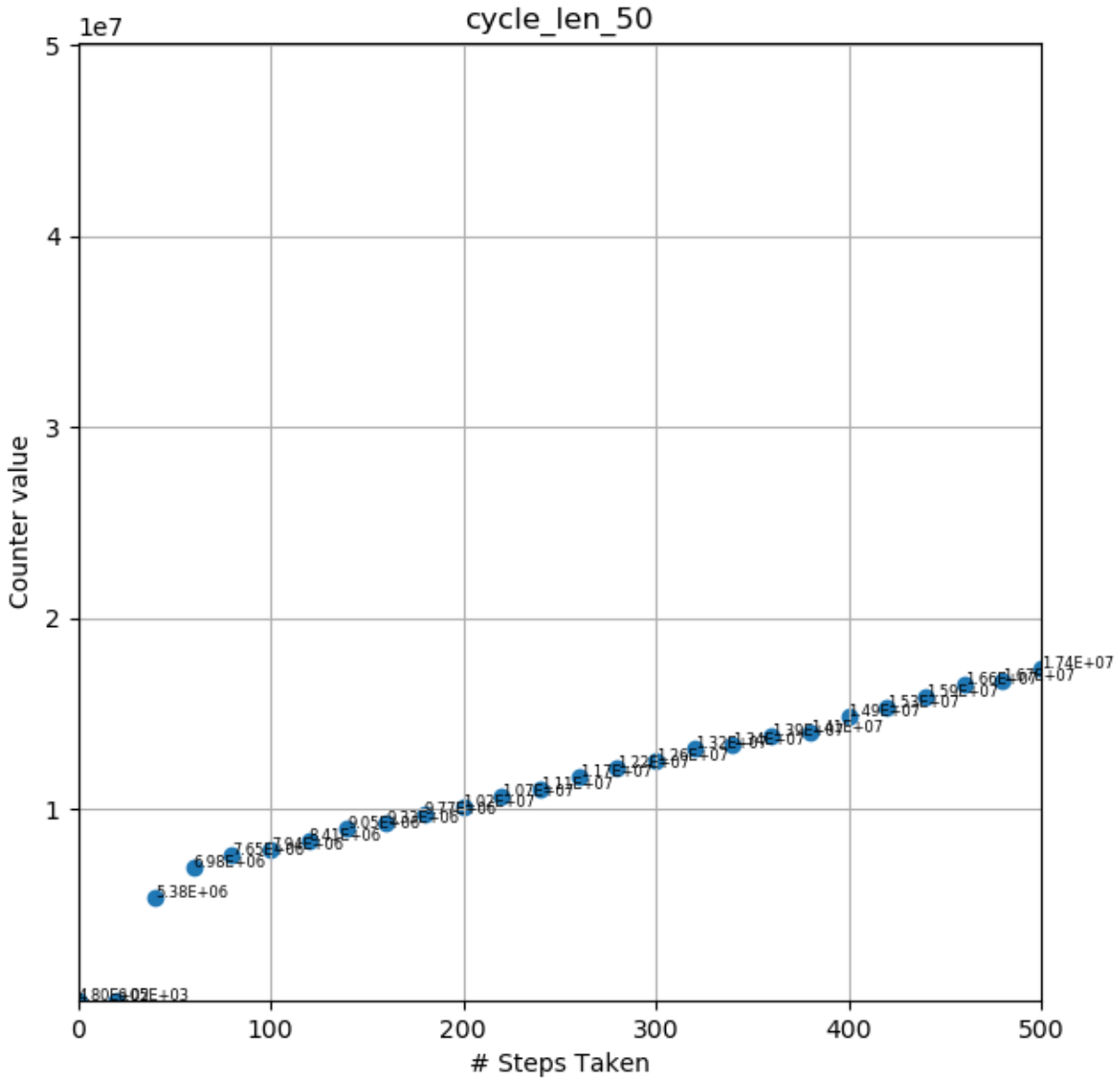


Results from the cycle-walking test, where the cycle has length 200.

SLOPE: ~161,000 Counter Increments per Step at 200 steps and below
~122,000 Counter Increments per Step at 200 steps and above

Cycle-Walking Test Results

NVIDIA GeForce RTX 2080 SUPER PCIe SSE2 OpenGL 45 core

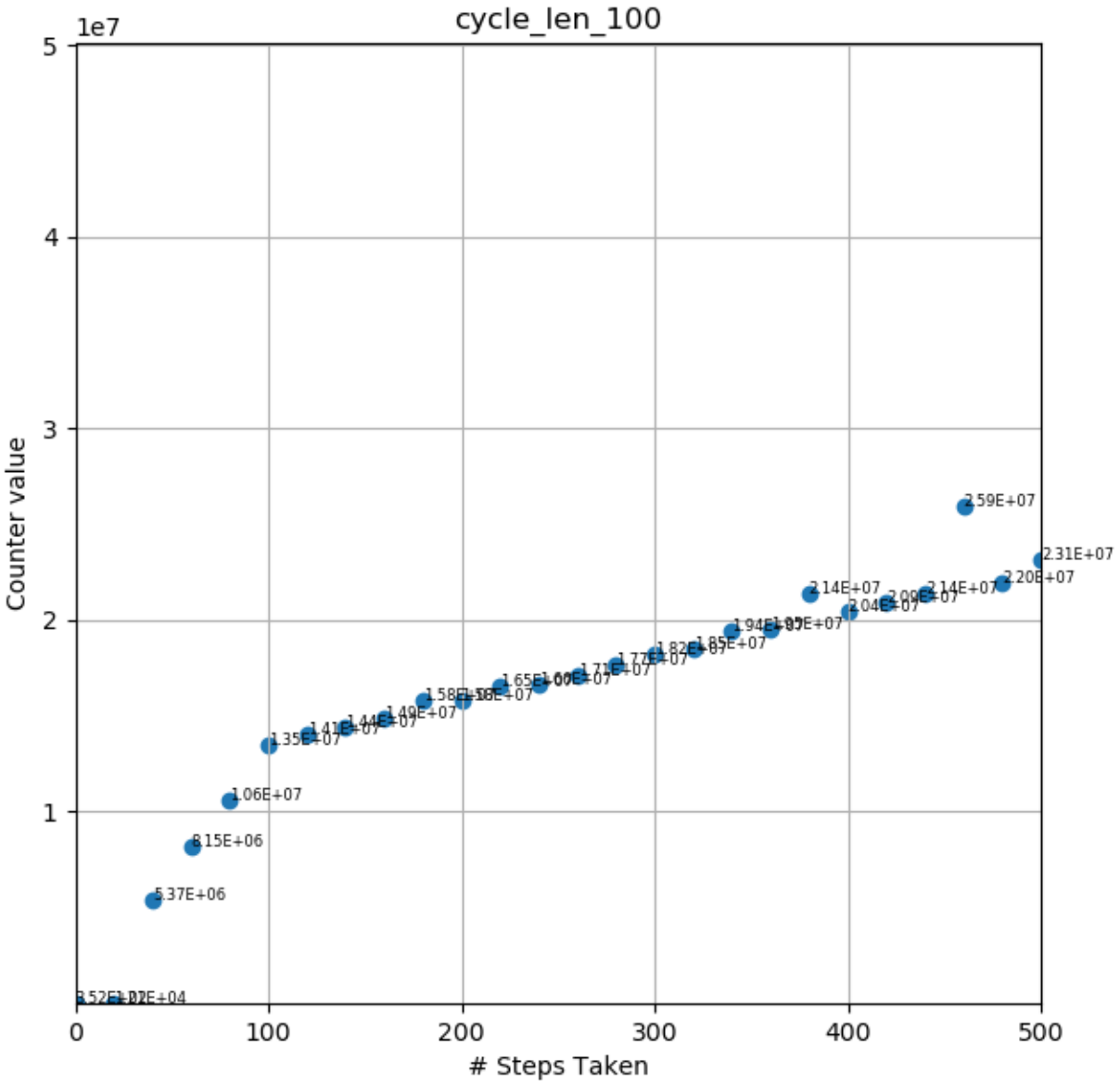


Results from the cycle-walking test, where the cycle has length 50.

SLOPE: ~24000 Counter Increments per Step,
only takes data points at 60 through 500 steps into account.

Cycle-Walking Test Results

NVIDIA GeForce RTX 2080 SUPER PCIe SSE2 OpenGL 45 core

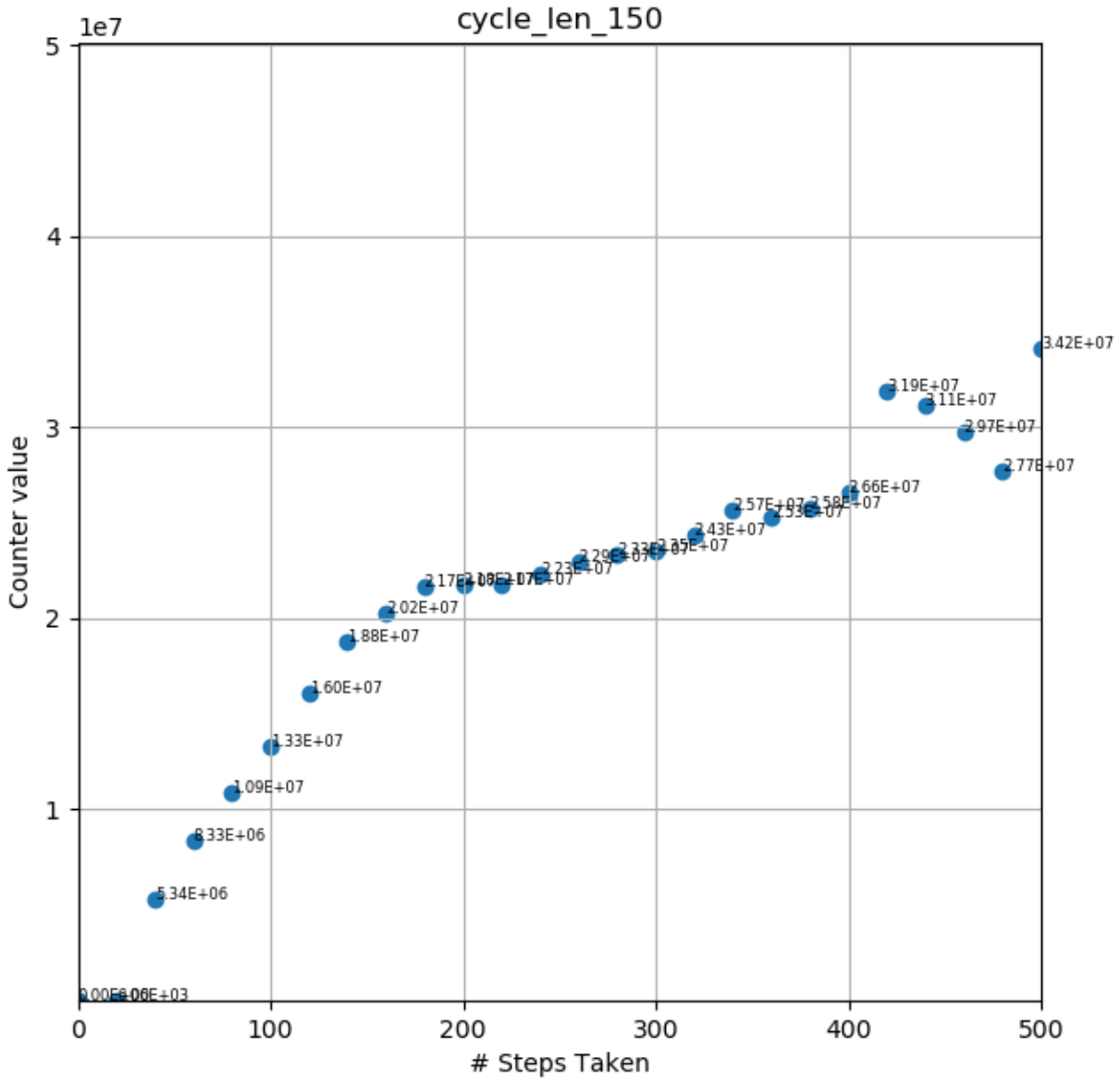


Results from the cycle-walking test, where the cycle has length 100.

SLOPE: ~135,000 Counter Increments per Step at 100 steps and below
~22,000 Counter Increments per Step at 100 steps and above

Cycle-Walking Test Results

NVIDIA GeForce RTX 2080 SUPER PCIe SSE2 OpenGL 45 core

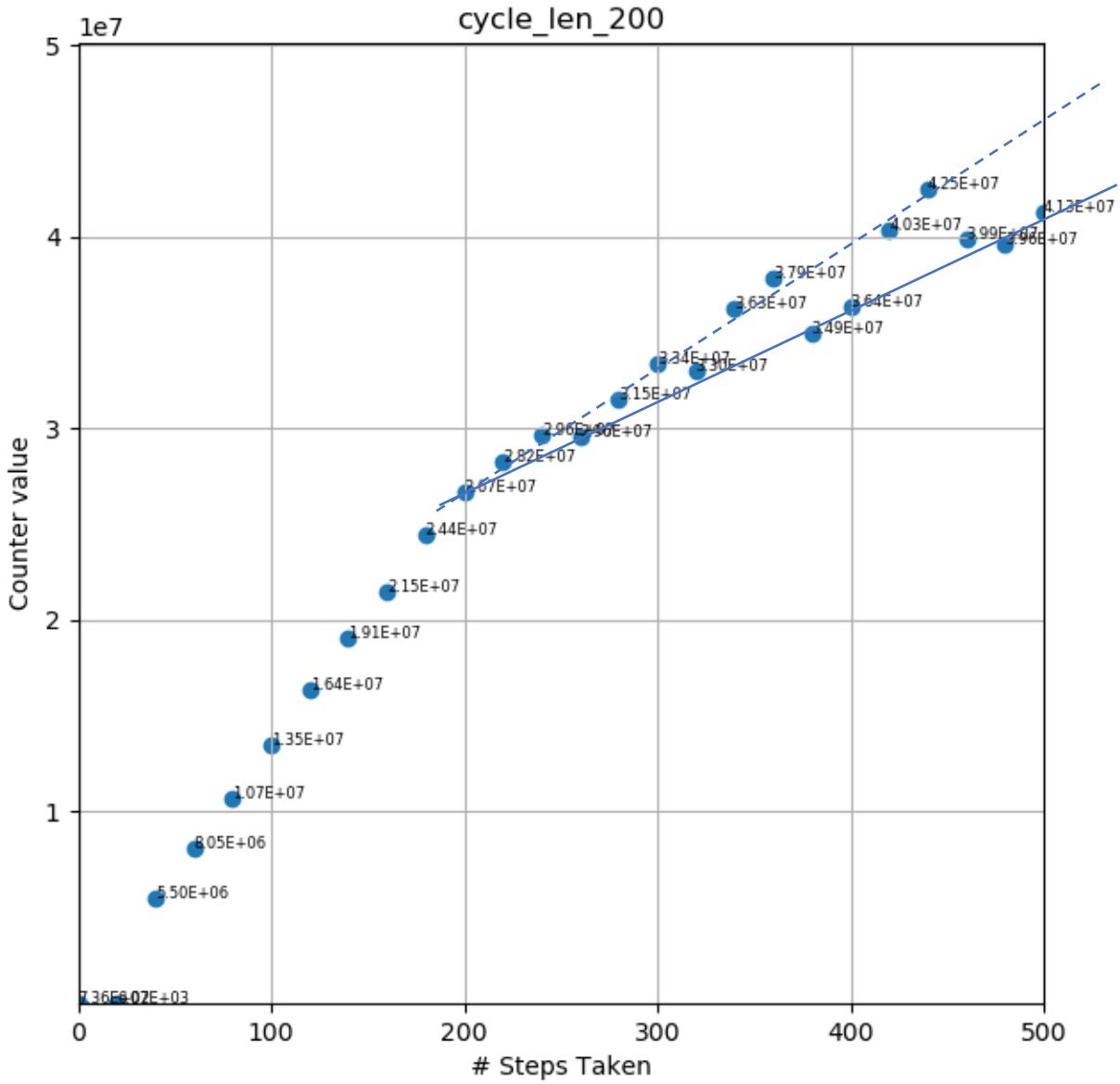


Results from the cycle-walking test, where the cycle has length 150.

SLOPE: ~134,000 Counter Increments per Step at 150 steps and below
~27,000 Counter Increments per Step at 150 steps and above

Cycle-Walking Test Results

NVIDIA GeForce RTX 2080 SUPER PCIe SSE2 OpenGL 45 core

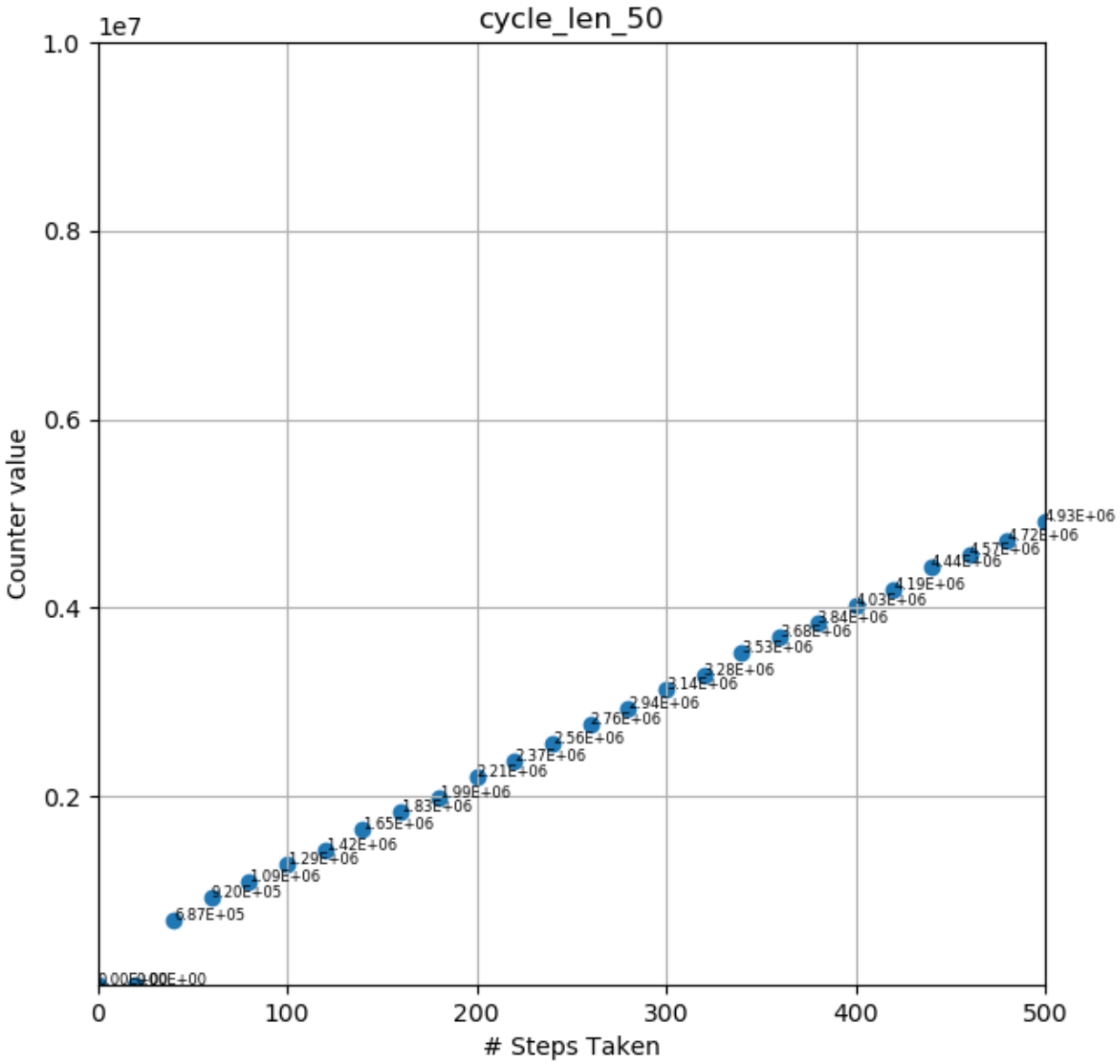


Results from the cycle-walking test, where the cycle has length 200.

SLOPE: ~134,000 Counter Increments per Step at 200 steps and below
~66,000 Counter Increments per Step at 200 steps and above – dashed line
~48,000 Counter Increments per Step at 200 steps and above – solid line

Cycle-Walking Test Results

NVIDIA Quadro P620 PCIe SSE2 OpenGL 45 core

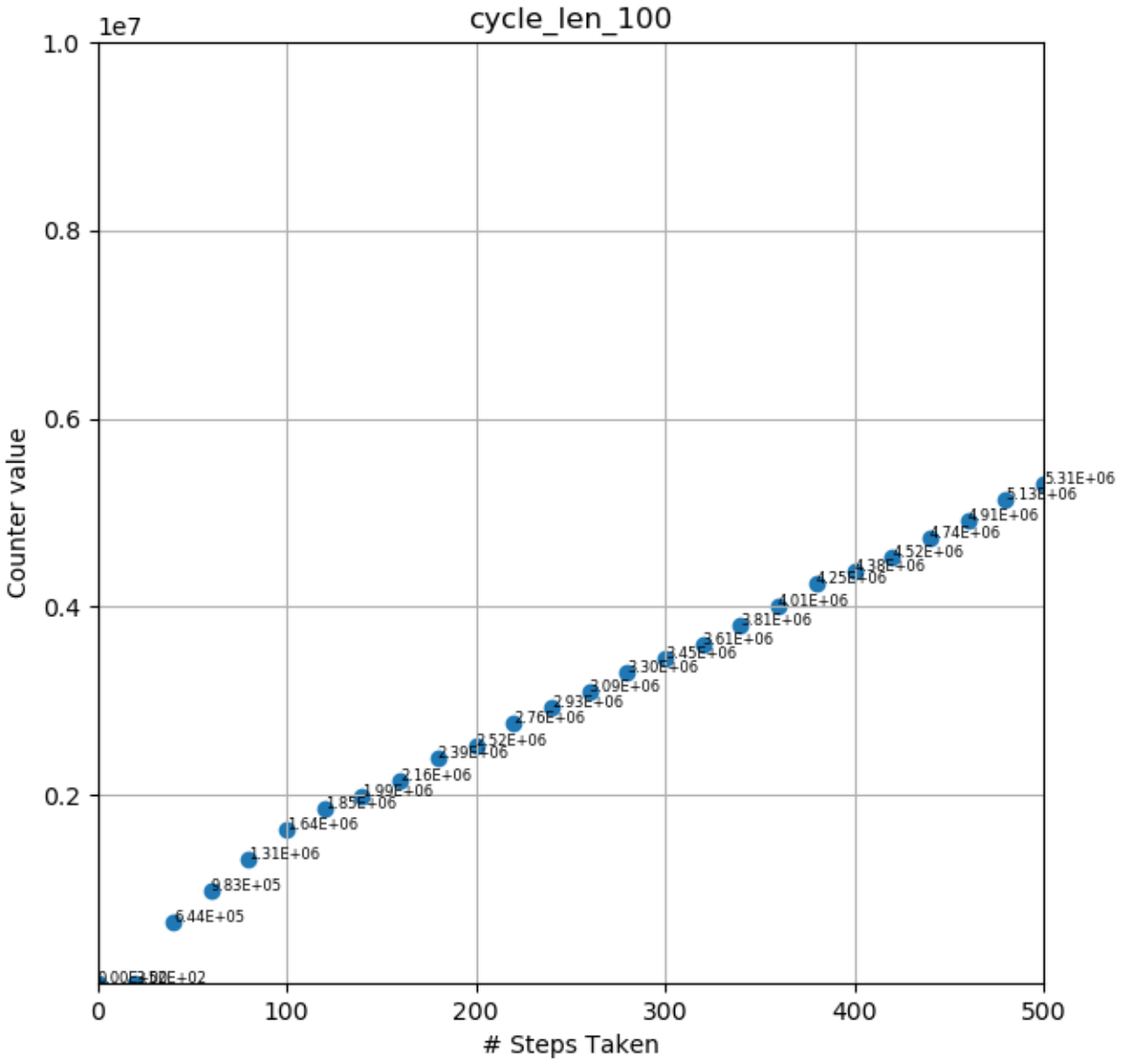


Results from the cycle-walking test, where the cycle has length 50.

SLOPE: ~9,000 Counter Increments per Step

Cycle-Walking Test Results

NVIDIA Quadro P620 PCIe SSE2 OpenGL 45 core

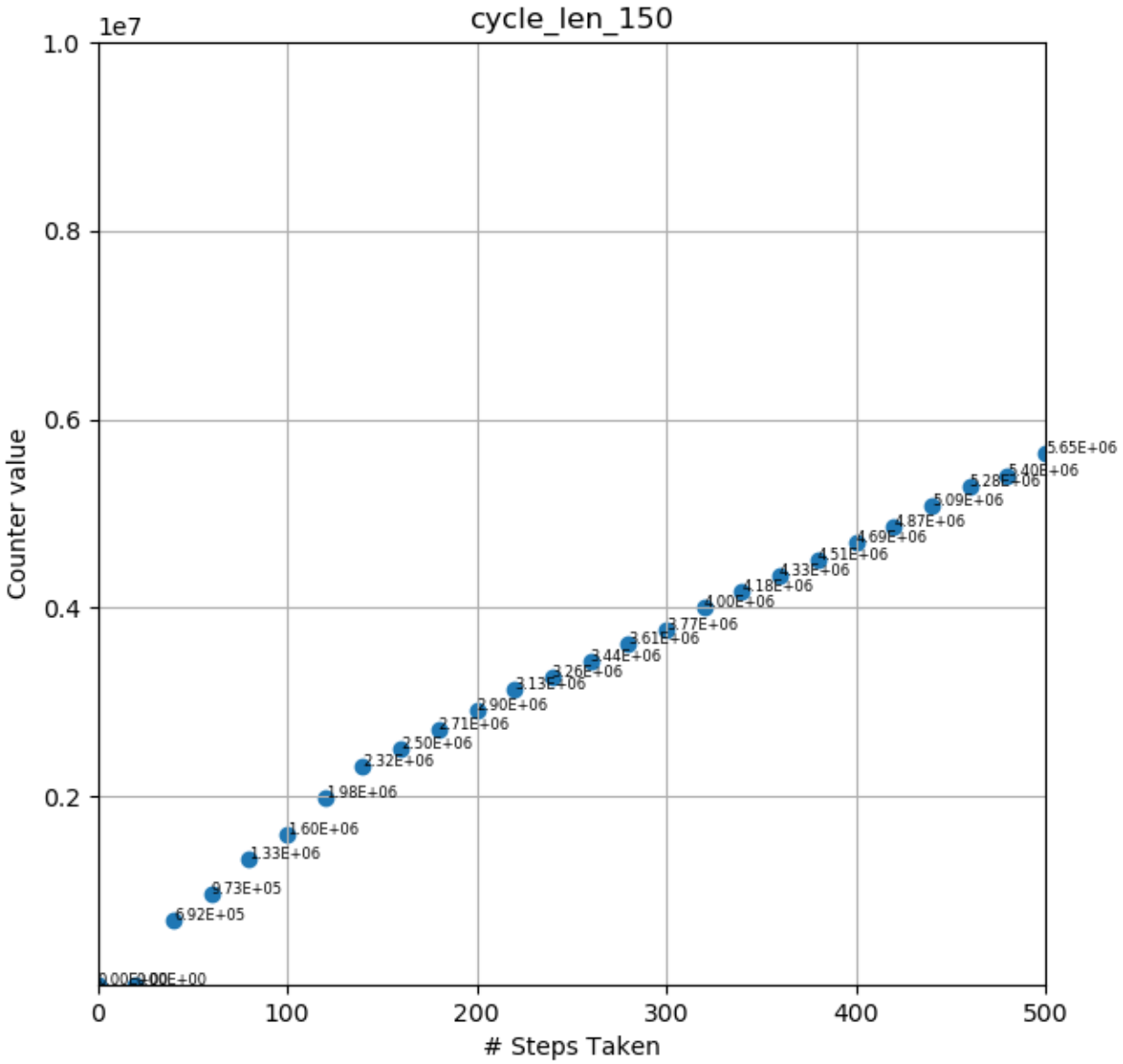


Results from the cycle-walking test, where the cycle has length 100.

SLOPE: ~16,000 Counter Increments per Step at 100 steps and below
~9,000 Counter Increments per Step at 100 steps and above

Cycle-Walking Test Results

NVIDIA Quadro P620 PCIe SSE2 OpenGL 45 core

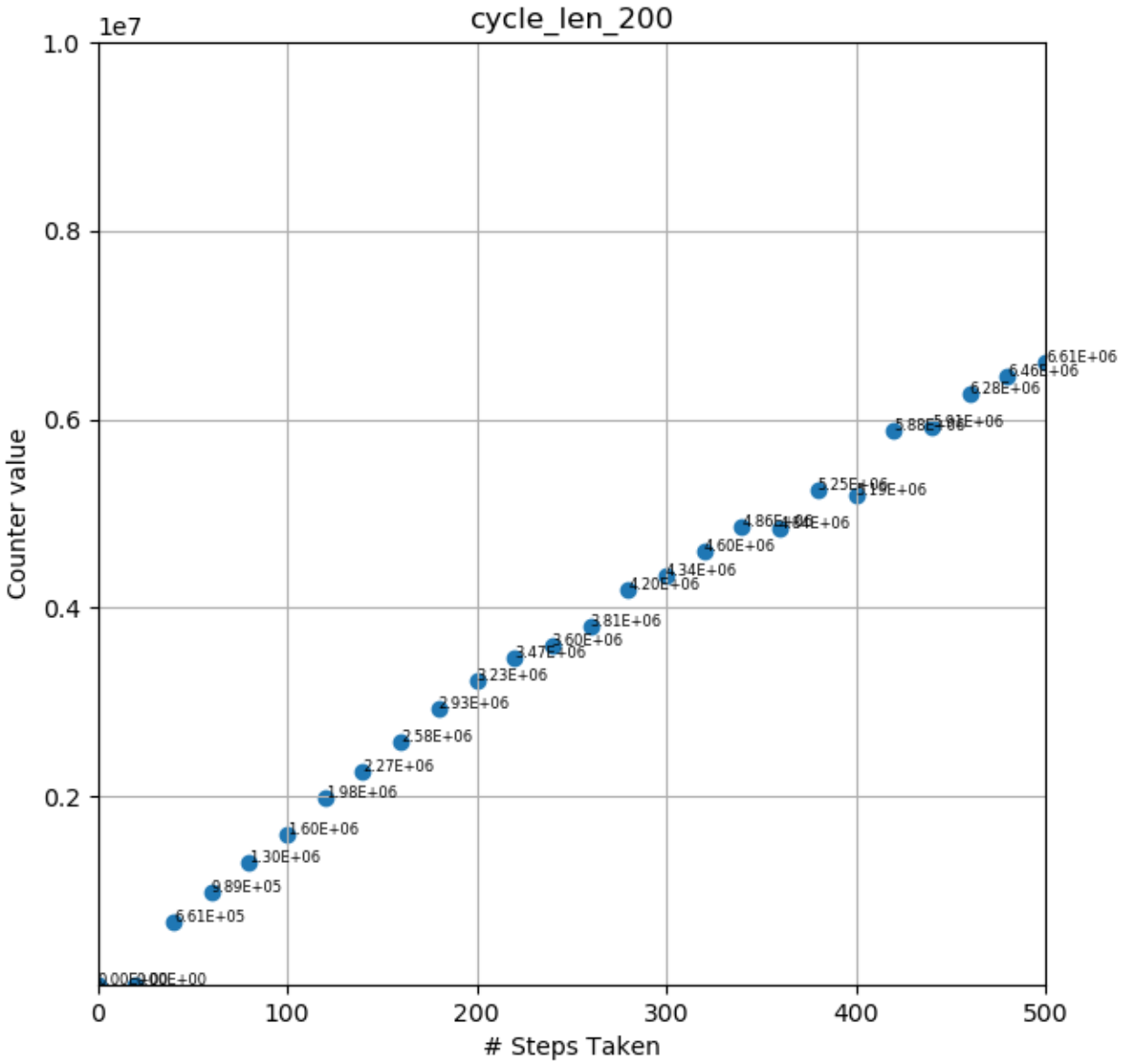


Results from the cycle-walking test, where the cycle has length 150.

SLOPE: ~17,000 Counter Increments per Step at 150 steps and below
~9,000 Counter Increments per Step at 150 steps and above

Cycle-Walking Test Results

NVIDIA Quadro P620 PCIe SSE2 OpenGL 45 core

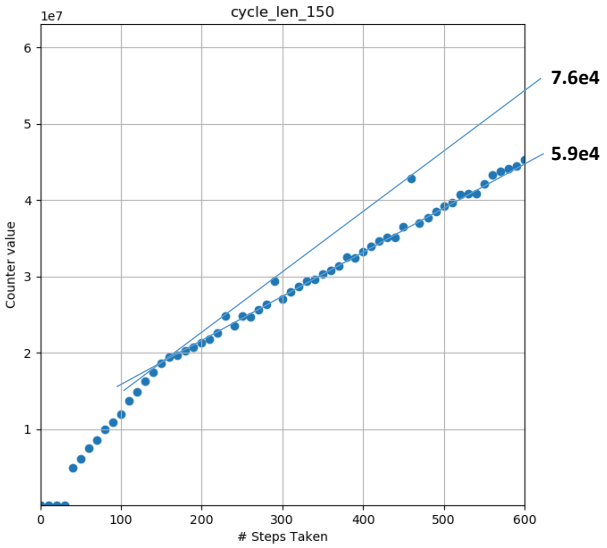


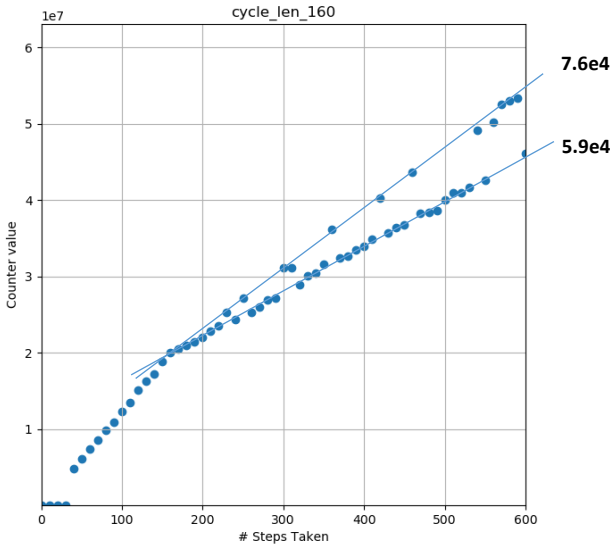
Results from the cycle-walking test, where the cycle has length 200.

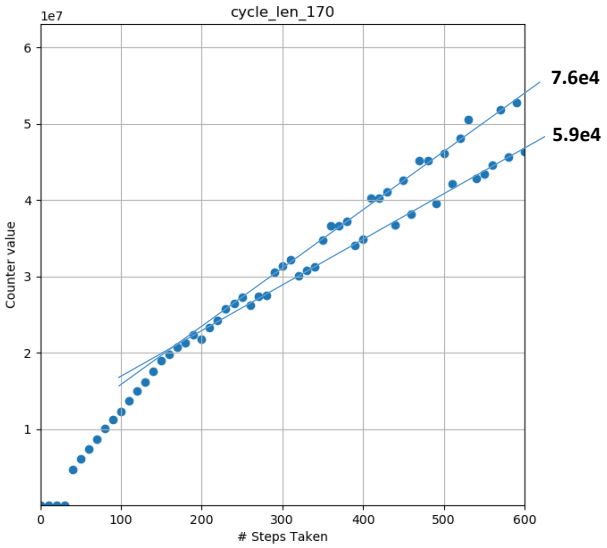
SLOPE: ~16,000 Counter Increments per Step at 200 steps and below
~11,000 Counter Increments per Step at 200 steps and above

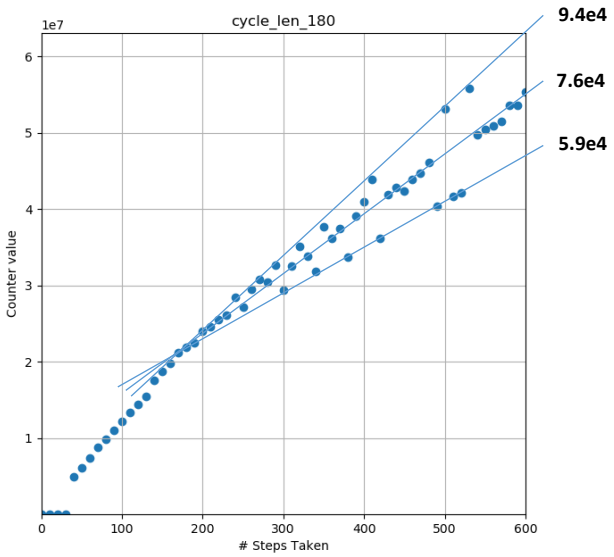
Extended Cycle-Walking Test Results: NVIDIA GeForce 1070 PCI3 SSE2 OpenGL 45 core

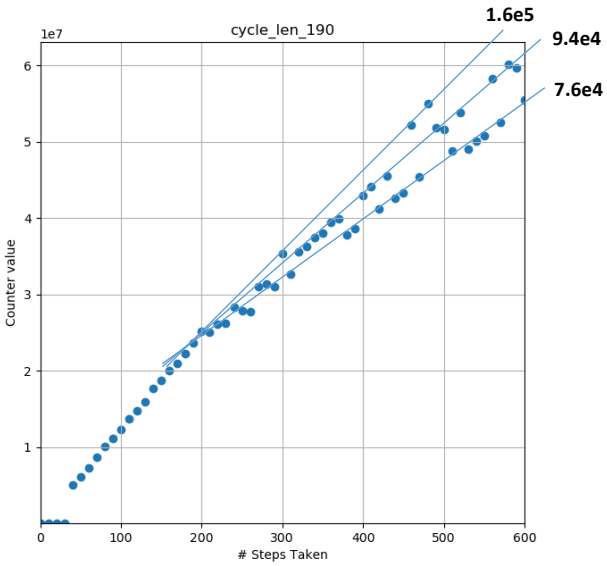
Increments per step are indicated by the numbers to the right of each line.

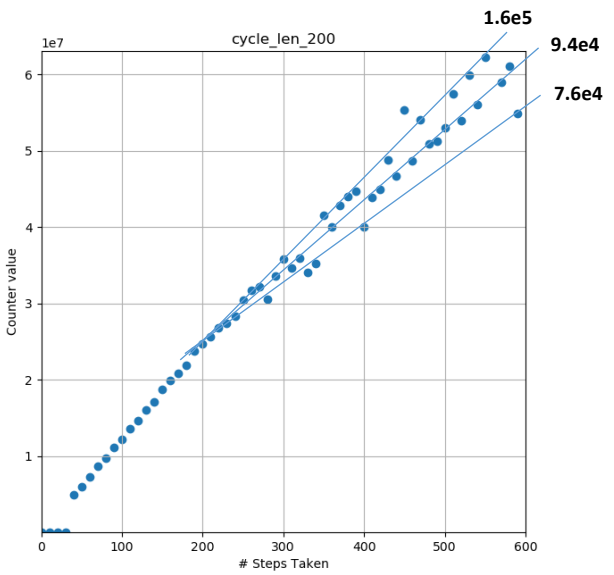




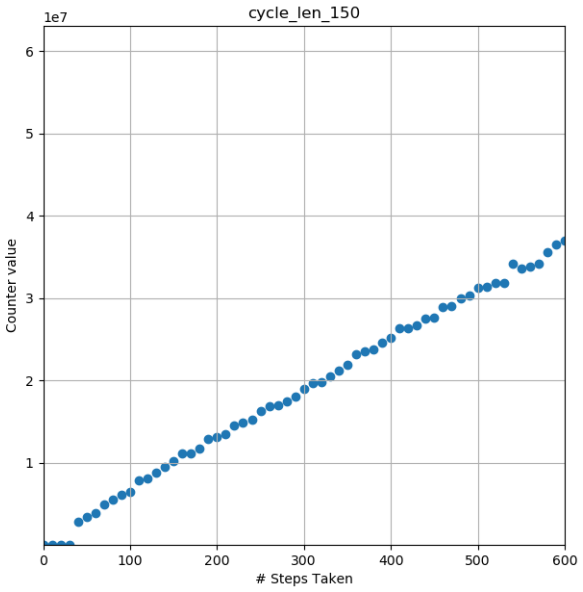






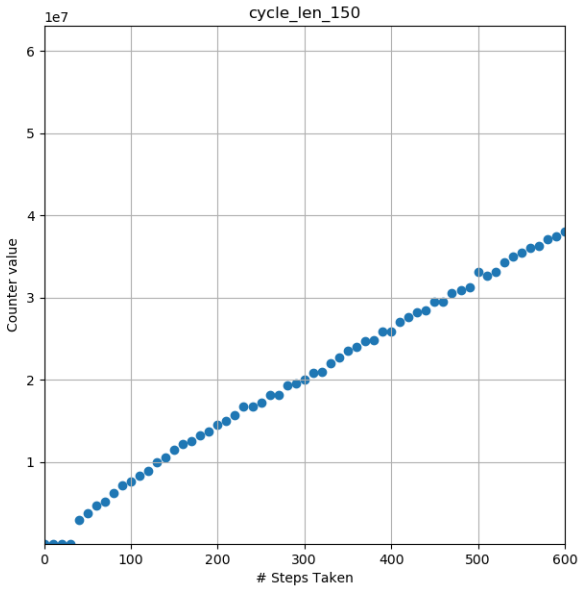


Linear Cycle-Walking Test Results
NVIDIA GeForce GTX 1070 PCIe OpenGL 45 core



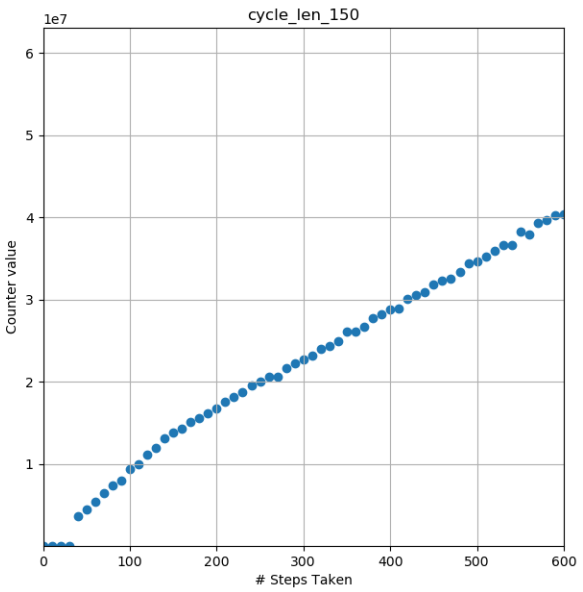
Results from the linear cycle-walking test,
where the stride length is 1, and the cycle length is 150.

SLOPE: ~68,000 Counter Increments per Step at 150 steps and below.
~60,000 Counter Increments per Step at 150 steps and above.



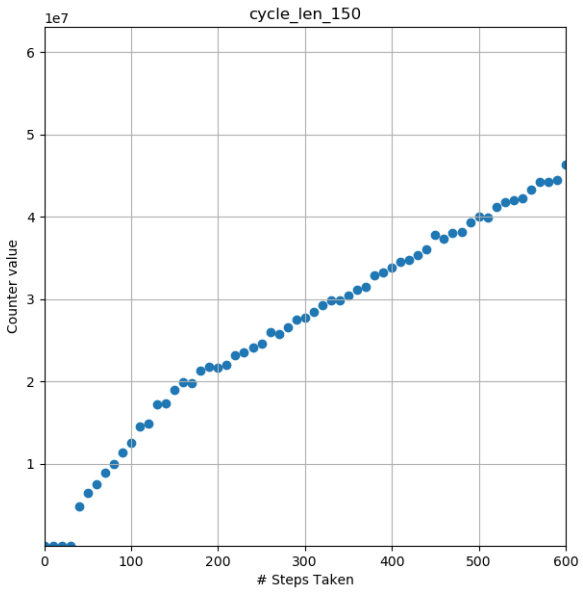
Results from the linear cycle-walking test,
where the stride length is 2, and the cycle length is 150.

SLOPE: ~77,000 Counter Increments per Step at 150 steps and below.
~60,000 Counter Increments per Step at 150 steps and above.



Results from the linear cycle-walking test,
where the stride length is 4, and the cycle length is 150.

SLOPE: ~93,000 Counter Increments per Step at 150 steps and below.
~60,000 Counter Increments per Step at 150 steps and above.



Results from the linear cycle-walking test,
where the stride length is 8, and the cycle length is 150.

SLOPE: ~126,000 Counter Increments per Step at 150 steps and below.
~60,000 Counter Increments per Step at 150 steps and above.