Fast Code on Super Sampling Anti Aliasing

Wenlin Mao

Electrical and Computer Engineering

Carnegie Mellon University

Pittsburgh, US

wenlinm@andrew.cmu.edu

Mitchell Yang

Electrical and Computer Engineering

Carnegie Mellon University

Pittsburgh, US

mfy@andrew.cmu.edu

Chang Liu

Electrical and Computer Engineering

Carnegie Mellon University

Pittsburgh, US

changl7@andrew.cmu.edu

I. Abstract

The project that we want to optimize is the Super Sampling Anti Aliasing (SSAA[1]) algorithm which is in the graphics domain. This is normally used for removing aliasing in real time renders, such as game, animation, or SVG displayers. Aliasing is a very common problem in real time rendering since unlike the real world, lines on a computer screen are shown by a collection of equally sized small squares. The jagged effect created by these small squares is what we call aliasing. To remove aliasing, the traditional methods like Fourier Transforms are computationally demanding and require a lot of repeated work. The Super Sampling Anti Aliasing algorithm offers a cheaper and better solution; however, SSAA is more expensive memory-wise than other anti-aliasing algorithms and can significantly increase rendering times. This is a crucial problem for real time renderers since in real time renderers, we always want to render each frame within 16ms so that we can maintain at least 60 fps frame rate. A slower anti-aliasing algorithm will cause disasters to a real time render which causes the animation or game to break up entirely. By optimizing the SSAA algorithm, we could reduce both the run time and space used for this algorithm so that we would produce a better frame rate in real time rendering.

The SSAA algorithm mainly takes the idea that given a high resolution version of the original image, we can blend all pixels nearby to create smooth curves or lines. The idea is simple; however, it can be very expensive in a real time renderer from both the perspective of time and space. Since the SSAA algorithm requires the program to create a high resolution version of the original image then store it in memory for future sampling, the space that the high resolution image used would be multiple times larger than the original image. In addition to the memory problem, it is also very challenging from a computational perspective, since for each point in the original image, we want to sample the high resolution image multiple times to compute the interpolation. There are many places in the original algorithm we can optimize. For memory problems, we should be able to find a better way to encode the higher resolution image instead of storing them completely in buffers. For computational problems, we could also change the way the renderer samples from the high resolution image.

The codebase that we will be working on is Project 1 of Computer Graphics(15-662). This is a real time renderer that can take a SVG file and display it on the screen. Although there are many

parts in the SVG Renderer, the algorithm we will be focusing on is still the Super Sampling Anti Aliasing, with a subset of the rasterization method to make the SVG Renderer display. The baseline code we will use as comparison is the existing solution code for Project 1. We extracted the line drawing and circle drawing functions and timed them for our benchmarks.

The baseline code is mainly focusing on the SVG rendering. However, encoding and decoding SVG is not our main focus of the project. Instead, we decided to extract the data from the SVG file and hard code them as our input to the improved version of the circle and line drawing algorithms. The data from the original SVG dataset that comes with the baseline project would be enough for our testing purpose. Moreover, we have written code in the driver that is able to randomly generate data in the same format that a SVG file would provide to us. This kind of testing data fits our purpose since we would expect our line and circle drawing algorithm to draw objects visually similar to baseline code with random line and circle input.

II. Performance peak

The platform that we want to work on is the ECE machine 010 which is a Dell PowerEdge R430 with 2 Intel Xeon CPU E5-2640 v4 @ 2.40GHz which is part of the Intel® Xeon® Processor E5 Family and uses X86 architecture. It has L1 data cache of 8 way 10 x 32 KB, L2 cache of 8 way 10 x 256 KB, and L3 cache of 20 way 25 MB. The microarchitecture for E5-2640 is Broadwell and it supports AVX. It supports all the SIMD instructions we want.

The latency and throughput for each instruction we use is shown in Table 1.

Instruction	Latency	IPC
(SIMD_LOAD) vmovapd	7	2
(SIMD_STORE) vmovapd	9	1
(SIMD_ADD) vaddpd	3	1
(SIMD_MUL) vmulpd	3	2
(SIMD_FMA) vfmaddxxxpd	5	2
(SIMD_FLOOR) vroundpd	6	.5
(SIMD_CMP) vcmppd	3	1
(SIMD_AND) vandpd	1	1
(SIMD_BROADCAST) vbroadcastsd	8	2

Table 1. Latency and Throughput[2]

The theoretical peak of the line kernel was computed with bottleneck at floor IPC * SIMD length * number of flops computed = 0.5 * 4 * 1 = 2 flops / cycle. The theoretical peak of circle is bottlenecked at compare so 1 * 4 * 1 = 4 flops / cycle.

III. Performance baseline

We used image size as the major target to differentiate. By increasing the size of the image from 1x1 to 256x256, we can see that the cycle the algorithm takes increased dramatically. The result have shown in Figure 1 and Figure 2 below. At larger sizes the baseline code either takes an hour to compute or directly segfaults.

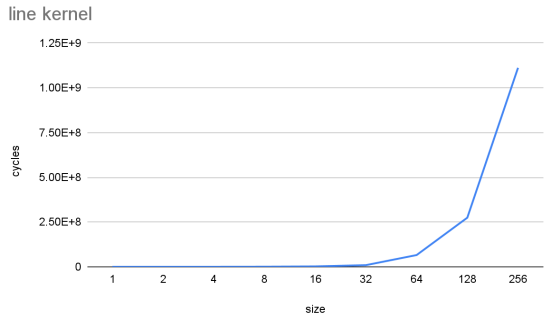

Figure 1. Line Kernel Baseline

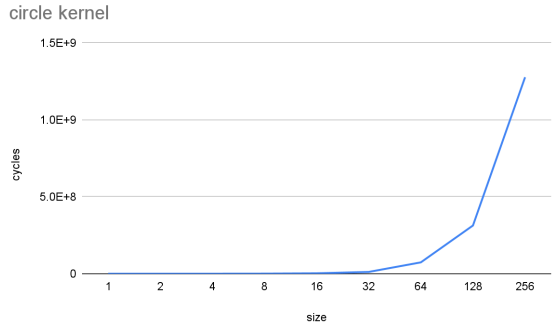

Figure 2. Circle Kernel Baseline

For the line kernel, based on our analysis, pipeline 1 would be the bottleneck. So the peak would be 4 FLOPS/cycle. Thus, the number of operation we use for the line algorithm is:

width * height *256

which is the FLOPS we need to use for each. For 256 x 256 example:

256 x 256, 256 * 256 * 4 = 16777216 FLOPS
16777216/1111831935.449 = 0.015089 FLOPS/cycle.
0.000235/2 = 0.000117 = .377%

of theoretical peak.

For the circle drawing algorithm, CMP operation would become the bottleneck since we CMP operation has IPC of 1 which is slower than FMA which has IPC of 2. Thus, the circle kernel is bounded by the pipeline 1. The number of operations we have for the circle algorithm is:

width * height * sample rate^2 * 1.

For 256 x 256 example, we have:

256 * 256 * 16 * 16 = 16777216 FLOPS.
16777216/1277023888 = 0.01313 FLOPS/cycle.
0.01313/4 = 0.00328443 = 0.328%

of theoretical peak.

IV. Design

The algorithm that we want to run is the Super Sampling Anti Aliasing algorithm. We want to run the portions for the lines and for circles. Our kernels will be circle kernel and line kernel. The parameters for deciding the size of the kernels is the number of independent operations needed to place between floors. The upper bound is the number of registers that we have available which is 16. We chose to use a 5 16 x16 pixel large kernel to perform Xiaolin Wu's line algorithm[3] for the line kernel. The circle kernel uses 16 registers when performing SSAA on a 16 x 16 supersampled target for an actual pixel.

For the line kernel, since we identified the bottleneck to be pipeline 1, we tried to make sure it was as full as possible. We use SIMD LOAD and SIMD BROADCAST to set the initial values. SIMD CMP and SIMD AND to compare the values against the boundaries to make sure not beyond limit and to set them, and SIMD FMADD, SIMD FLOOR, SIMD ADD, SIMD SUB to calculate the y coordinates relative to each x value as well as the color

for the pixel. Since there are not enough registers to have enough independent instructions, we have to use SIMD LOAD and SIMD STORE to store intermediate values for future use. See Figure 3 below for the independent and dependent chains for the calculations to get the pixel colors. I tried working with 4 pixels at a time, but encountered bubbles in the pipeline as well as more importantly I could only get a steady state on the pipeline 1 for a few instructions only. The prologue is fairly expensive as it has bubbles when waiting for simd_floor, but once in steady state can use the adds and subs to fill in the rest of the pipeline. Using 5 pixels makes it possible to fill up the registers fully, but will still run out registers and have bubbles, but smaller than the 4 pixel one. We also tried a 8 pixel implementation with spilling to see if it would work faster as it removed the bubbles seeing if that hid the implicit loads and stores which was worse. It is probably because in this model, it was not possible to keep 2 sets of information without evicting data from registers. Therefore, we ended up using the implementation with 5 pixels which had to perform extra loads and stores

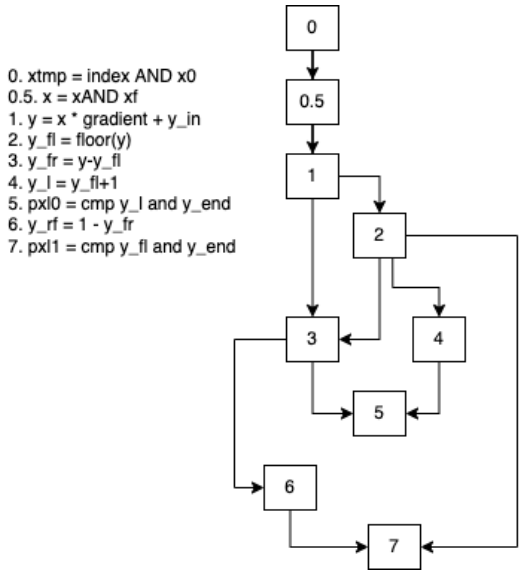

Figure 3. Line Instruction Chain

for the calculation of 6 and 7 in Line Instruction Use. The register use is in Line Register Use.

Register							1. y = x * gradient + y_in
0		gradient	y_l	pxl0	y_fl		2. y_fl = floor(y)
1	xf	y	y_fr		y_rf		3. y_fr = y-y_fl
2	xf	y	y_fr		y_rf		4. y_l = y_fl+1
3	xf	y	y_fr		y_rf		5. pxl0 = cmp y_l and y_end
4	xf	y	y_fr		y_rf		6. y_rf = 1 - y_fr
5	xf	y	y_fr		y_rf		7. pxl1 = cmp y_fl and y_end
y_rf		x0	y_fl		y_end	y_end	
y_rf		x0	y_fl		y_end	y_end	
8		x0	y_fl		y_end	y_end	
9		x0	y_fl		y_end	y_end	
10		x0	y_fl		y_end	y_end	
11		x		y_l	pxl0	y_fl	pxl1
12		x		y_l	pxl0	y_fl	pxl1
13		x		y_l	pxl0	y_fl	pxl1
14		x		y_l	pxl0	y_fl	pxl1
15		x	one				pxl1

Figure 4. Line Register Use

The final reduction uses SIMD ADD, SIMD MULT, and sequential additions for the pixel color average. We moved the multiplication of color to the scaling factor from the original implementation to here as it would have been redundant calculations as we would have to do it 64 times per pixel instead of once per pixel. SIMD MUL and SIMD FMA will use functional units 0 or 1. SIMD ADD, SIMD SUB, SIMD, SIMD CMP, and SIMD FLOOR will use functional unit 1. SIMD BROADCAST and LOAD will use functional units 2 or 3. SIMD STORE will use one of the functional units 2 or 3 or 7 and functional unit 4.

As for the circle, we came up with two versions of design. The first design uses SIMD MUL to calculate $y_j^2 = y_j \times y_j$ and SIMD ADD to calculate $R_{ij}^2 = y_j^2 + x_i^2$. Then it uses SIMD CMP to test $mask_{ij} = R_{ij}^2 < R_0^2$ ($mask_{ij}$ would have either all 1s if $R_{ij}^2 < R_0^2$ or all 0s

otherwise, which serves as a bitwise mask) and SIMD AND to get $color_{ij} = mask_{ij} \,\&\, color$. Finally, it calculates $sum_i = color_{ij} + sum_i$. The final result adds sum_i together and divide by 256.

Since SIMD CMP is a latency 3 and throughput 1 instruction, we initially identified it as our bottleneck. However, as the experiment went on, we realized that our bottleneck is actually not on a single instruction, but on functional unit 1 (p1). For this design, we have to perform 2 SIMD ADDs and a SIMD CMP on p1 in order to calculate the final result, which means we will reach our theoretical peak as long as we manage to fill the pipeline of p1.

For each actual pixel, a total of 256 supersampled pixels will be calculated. Each supersampled pixel will have to perform 3 float operations on functional unit 1 (p1). So our total number of float operations performed on our bottleneck is:

$$FLOPS = \frac{256 \times 3}{(Number\ of\ cycles)}$$

In order to hide the latency of SIMD CMP and SIMD ADD, it needs at least 3 independent chains. Also, to hide the latency of SIMD AND (whose latency 1 and throughput 1), we need one more independent chain to fill the gap between SIMD CMP and SIMD ADD. So, the implementation has 4 independent chains, which could fill both the pipeline of p1 and all the registers.

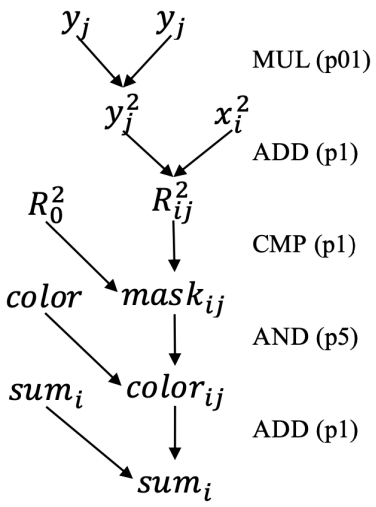

Figure 5. Circle Instruction Chain (V1)

	p0	p1	p5
	MUL1		
	MUL2		
	MUL3		
Steady State		ADD1	
		ADD2	
		ADD3	
		ADD4	
		CMP1	
		CMP2	
		CMP3	
		CMP4	AND1
		ADD1	AND2
		ADD2	AND3
		ADD3	AND4
		ADD4	
		ADD1	
		ADD2	
		ADD3	
		ADD4	
		CMP1	
		CMP2	

Figure 6. Circle Schedule (V1)

	SIMD
ymm0	$x_0^2 \sim x_{15}^2$
ymm1	
ymm2	
ymm3	
ymm4	y_i
ymm5	offset
ymm6	$(work)$
ymm7	
ymm8	
ymm9	
ymm10	sum0
ymm11	sum1
ymm12	sum2
ymm13	sum3
ymm14	color
ymm15	R_0^2

Figure 7. Circle Register Use (V1)

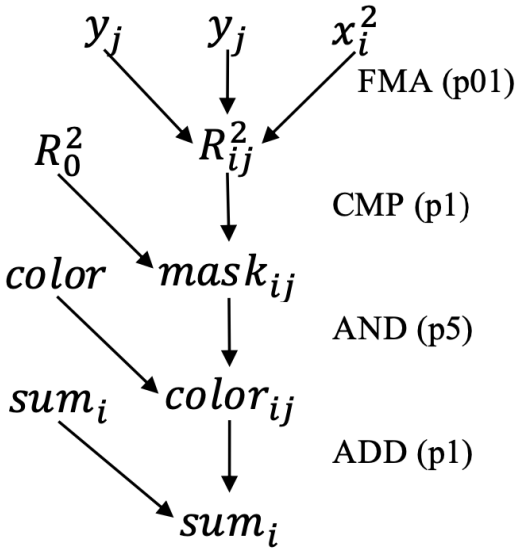

Figure 8. Circle Register Use (V2)

The second design uses SIMD FMA to calculate ${R_{ij}}^2 = y_j \times y_j + {x_i}^2$. Then it uses SIMD CMP to test $mask_{ij} = {R_{ij}}^2 < {R_0}^2$ ($mask_{ij}$ would result in all 1s if ${R_{ij}}^2 < {R_0}^2$ and all 0s otherwise, which serves as a bitwise mask) and SIMD AND to get $color_{ij} = mask_{ij} \,\&\, color$. Finally, it calculates $sum_i = color_{ij} + sum_i$. The final result adds sum_i together and divide by 256.

This design is different from the first version. It uses SIMD FMA to calculate ${R_{ij}}^2$. Also, for this design, we only have to perform a SIMD ADD and a SIMD CMP on p1 in order to calculate the final result, which means we will reach our theoretical peak as long as we manage to fill the pipeline of p1. Similarly, we will have to perform 2 float operations on p1. So, the total number of float operations performed on our bottleneck is:

$$FLOPS = \frac{256 \times 2}{(Number\ of\ cycles)}$$

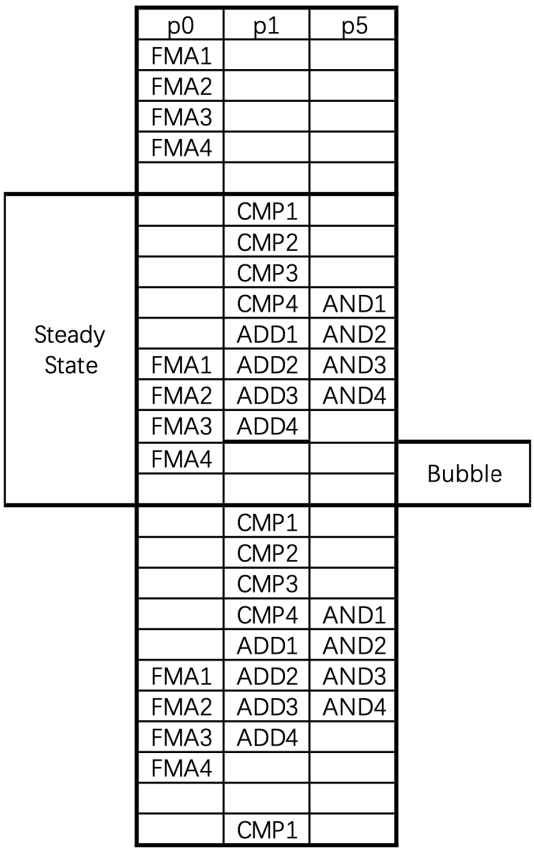

Figure 9. Circle Schedule (V2)

	SIMD
ymm0	$x_0^2 \sim x_{15}^2$
ymm1	
ymm2	
ymm3	
ymm4	y_i
ymm5	offset
ymm6	$(work)$
ymm7	
ymm8	
ymm9	
ymm10	sum0
ymm11	sum1
ymm12	sum2
ymm13	sum3
ymm14	color
ymm15	R_0^2

Figure 10. Circle Register Use (V2)

In this design, we need to hide the latency of SIMD FMA. The latency of SIMD FMA is 5 and its throughput is 2. Since it can be performed on functional units 0 or 1 (p0 and p1) and our p1 is busy enough, we can assume it only uses p0 and its actual throughput becomes 1. So we need only 5 independent chains to hide the latency of it.

V. Parallelism

The parallelization is applied to both the draw line and the draw circle algorithms to improve the performance of preparation code that prepares data needed for both kernels. The parallelization is mainly implemented by the OpenMP library.

On line algorithm, we identified that each write from the line kernel to the render target is independent. Thus, in the first version, we tried to add parallelization for each pixel that was written in the rendering target. However, since the number of iterations inside the draw line only iterates the pixels around the line, it is not big enough to beat the overhead of using the parallelization. We need to add more work for each thread so that we won't bound our performance by the overhead of parallelization. Therefore, we finally decided to parallelize the draw object loop so that each object would be drawn by a single thread. This will give us sufficient work for each thread to work on in parallel and improve our performance.

On our circle algorithm, we identified that read, SSAA, and write for each of the pixels are independent. We are able to apply the parallelization on each pixel of the region where we draw the circle. Unlike the line kernel, we need to iterate through all pixels of the region that contains the circle. There is sufficient work for us to use the parallelization on the loop that iterates through each pixel and executes the circle kernel in parallel. Thus, we rewrote the loop and applied another parallelization to have threads work on each pixel of the rendering target. In addition, we also parallelized the object loop so that it would bring us another level of improvement.

Overall, the performance is improved about 60%+ compared to the algorithm with a single thread for line algorithm. The performance is improved about 70%+ compared to the algorithm with a single thread for the draw circle algorithm.

VI. Results

For the line kernel, the baseline segfaults when run at larger sizes, so could only compare to smaller sizes of baseline. The line kernel gets around 55% of peak. The plots would not be possible to see anything on the same plot if we did not do a log scale on y axis as for example, at 256x256, the baseline is 427 times slower than the line kernel. When scaling the dimension of the problem, the baseline is 17139 times slower at 1024 x 1024.

We think there are a couple of reasons why we do not have a very high

performance. With 5 pixels, it occupies as pictured in Line Register Use, almost all of the registers are used in the 5 pixel implementation. There are bubbles especially during the prologue. For example, every other instruction is dependent on loading the first index. Since the register use is very tight, the broadcasts, loads, and stores have the very tail end get bubbles. We schedule these as soon as the register is free, but there are not enough registers to place more independent instructions, nor enough registers to be able to store data elsewhere.

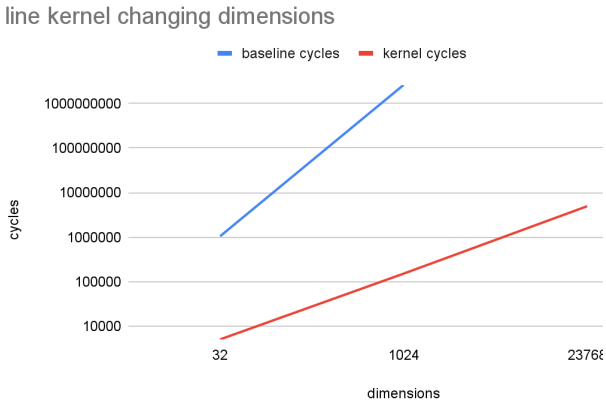

Figure 11. Line - Dimension vs Cycle

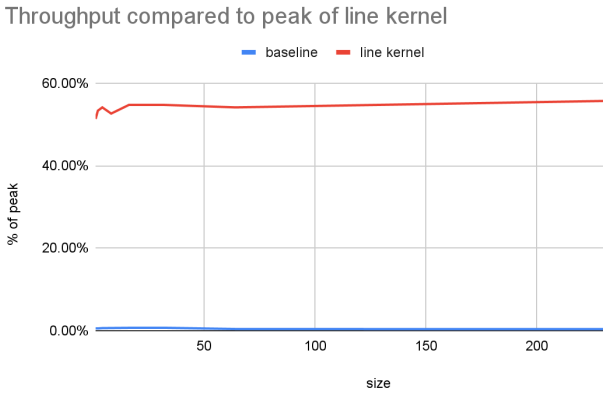

Figure 12. Line - Size vs Peak

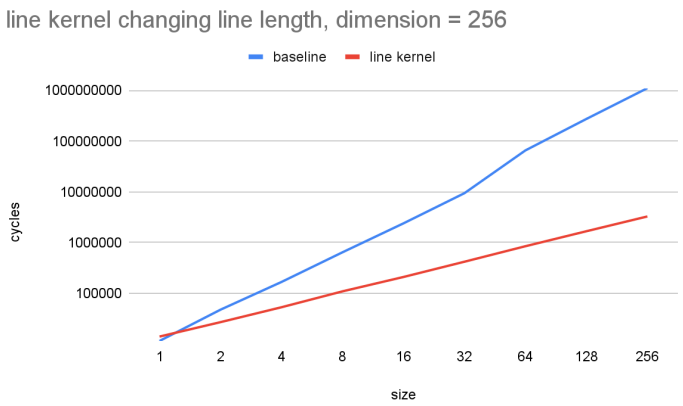

Figure 13. Line - Size vs Cycle

For the circle kernels, we conducted several tests, trying to find its ideal and actual performance.

First, we tested their performance by drawing a single pixel. This experiment aims at finding its capability of reaching its theoretical peak in an ideal situation (i.e. no memory load overhead, etc.). In this experiment, Design 1 managed to obtain 86.07% of the theoretical peak performance while Design 2 managed to obtain 75.75% of the theoretical peak.

Peak	Cycle	FLOPS	Percentage
4.0	223.07	3.4429	86.07%

Table 2. Circle Kernel (V1) Performance

Peak	Cycle	FLOPS	Percentage
4.0	168.98	3.0298	75.75%

Table 3. Circle Kernel (V2) Performance

For Design 2, we did not have enough registers, so we only managed to have 4 independent chains. Considering the fact that we do redundant calculations ($y_j * y_j$) and we have bubbles in our pipeline, this result is reasonable. From the Fast Code perspective, this design might be inefficient.

However, it actually outputs the same correct result with less cycle. That is because we identified the bottleneck and bypassed it with other instructions. Meanwhile, we utilized more functional units to calculate our result.

Then, we tested their performance by drawing practical pictures. This experiment aims at evaluating their performance in actual drawing tasks. We applied our kernel to different sizes of circles, measured their time and calculated their performance. It turns out that the performance is relatively stable through all sizes of pictures.

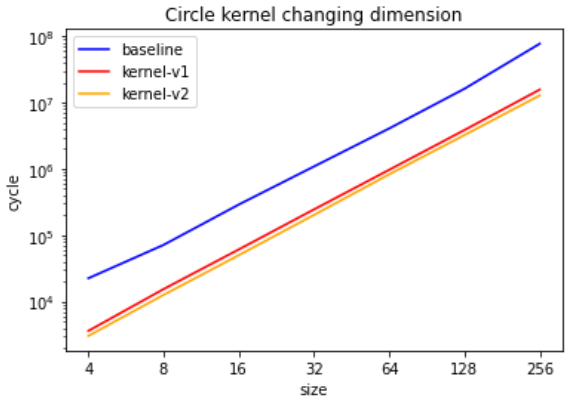

Figure 14. Circle Kernel Changing Dimension

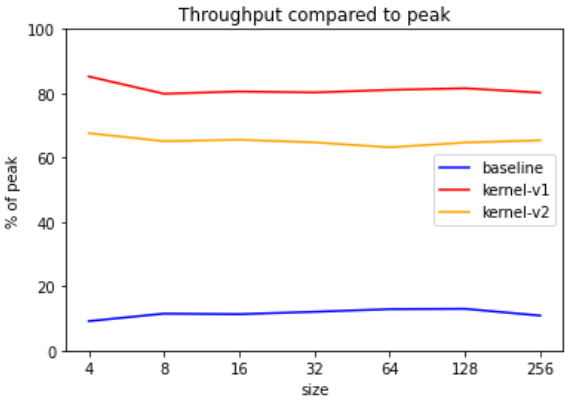

Figure 15. Circle Kernel Throughput Compared to Peak

Based on our testing data, for the line algorithm, using 4 threads would provide the best performance. As the size of the image increases, the performance of the multithreading version would be even better compared to the single thread version. For 2048x2048 images, we are able to achieve 66.37% of improvement compared to the single thread version.

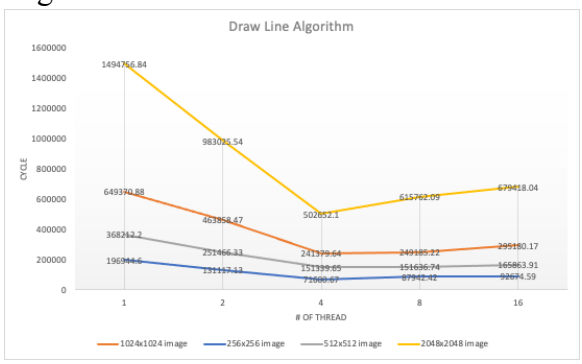

Figure 16. Line Parallel Performance

For the draw circle algorithm, using 4 or 8 threads provides the best performance. Same as the line algorithm, as the size of the image increases, the performance of the multithreading version would be even better compared to the single thread version. For 2048x2048 images, we are able to achieve 74.77% of improvement compared to the single thread version.

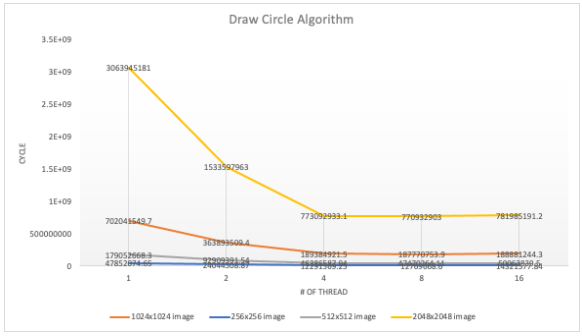

Figure 17. Circle Parallel Performance

VII. Future Work

If we had more time, we could implement the steep version of the line. We think we can flip the x and y to get the algorithm to work that way. This would allow the lines that are able to be drawn to

not be as limited as it currently is. Something else that would be nice to do in future works is to improve the packing code that is used when preprocessing data for the line kernel. It is currently a sequential implementation that does not consider independent chains, nor use SIMD. I could also investigate machines with 32 registers and see if it would be possible to have enough independent instructions in there.

For the circle kernel, an obvious improvement is to fill pipeline 1 with an extra chain, hence we could eliminate the bubbles in pipeline 1 and make it more efficient. Although we have tried to use some techniques to solve this problem(i.e. Register renaming), we end up having a performance drop. If we have more time to try it, we might overcome this problem.

One thing we didn't get time to try is to do the parallelization for the algorithm that finds and sorts the intersection point of line and each pixel in it's supersample tile. By adding parallelization on this portion of the code, we should be able to further improve the line drawing algorithm for large size images. The idea here would be to split the intersection vector into different pieces and sort them separately in their own container. Then, we do a parallel merge back to get the correct output. We are able to do this since intersection points are also being calculated independently and only the final order matters. However, in the end, we decided to not implement this parallelization since it is relatively complex and it won't bring much improvement if the image size is small as there isn't sufficient work to surpass the overhead of parallelization.

In addition, our implementation still sometimes produces incorrect results against randomly generated data. That happens since we didn't control the random data to avoid the steepness of the line. The random dataset might produce some data that is not supported by our implementation. If we had more time, we would definitely test with more data and produce a more reasonable dataset.

VIII. Reference

[1] Kristof Beets, Dave Barron. Super-sampling Anti-aliasing Analyzed: http://www.x86-secret.com/articles/divers/v5-6000/datasheets/FSAA.pdf

[2] Uflops.info - Table: https://uops.info/table.html.

[3] Xiaolin Wu. 1991. An efficient antialiasing technique. SIGGRAPH Comput. Graph. 25, 4 (July 1991), 143–152. DOI:https://doi.org/10.1145/127719.122734