CS3211 Project 1

Irvin Lim Wei Quan
A0139812A

Part 1

Hardware

In this project, two machines were used for executing and benchmarking the various execution time and speedup of various programs. The details of each machine are tabulated below, which is collated from both the Intel® specification sheets[1,2] as well as by running `lscpu` on the relevant machines:

	LAB MACHINE	TEMBUSU NODE
CPU	Intel® i5-2400S	2 x Intel® Xeon® E5-2620 v2
BASE FREQ	2.50 GHz	2.10 GHz
CPU CORES	4	6
CPU THREADS	4	12
L1 CACHE SIZE	32 KB	32 KB
L2 CACHE SIZE	256 KB	256 KB
L3 CACHE SIZE	6 MB	15 MB
TOTAL THREADS	**4**	**24**

Figure 1: CPU specifications and differences between the two types of machines used

We can see that the core differences between the two machines would be the total number of physical threads that are available, the size of the L3 cache, and the presence of hyper-threading in the Intel® Xeon® E5-2620.

The most significant difference in the number of cores would simply mean that the Tembusu node is able to run more concurrent processes (i.e. not context-switched). Simplistically speaking, this should allow programs to split up the same amount of work to be completed among a greater number of cores in a shorter time.

The size of the L3 cache would affect the rate of cache misses, which would in turn affect the number of read operations that are required from memory. More data that has to be read from the main memory into the higher-level CPU caches would mean that program execution would have to incur more time for data transfer and I/O, affecting overall execution time.

Hyper-threading makes the CPU appear to have double the number of logical processors to the OS, but can actually only run at most 12 of those 24 processors concurrently. The added advantage in hyper-threading is that each virtual processor can be better pipelined since there is double the number of CPU registers, allowing instructions and data to be loaded into the registers while waiting for its turn to be executed. This reduces the number of clock cycles that are required for pipeline stalling in order to resolve hazards, for example.

[1] https://ark.intel.com/products/52208/Intel-Core-i5-2400S-Processor-6M-Cache-up-to-3_30-GHz
[2] https://ark.intel.com/products/75789/Intel-Xeon-Processor-E5-2620-v2-15M-Cache-2_10-GHz

Speedup

By running the matrix multiplication program (`mm-shmem`) on the lab machines as well as the Tembusu node, we can measure the time taken for program execution when varying the number of threads used, as well as the size of the input (i.e. matrix dimension). For each pair of (`threads, matrix_dim`), the program was executed 5 times, and the minimum of those timings was recorded. For the lab machine, execution time up to 8 threads is tabulated in Figure 2 below, while for the Tembusu node, execution time up to 40 threads is tabulated in Figure 34 in the Appendix.

The minimum timing was used instead of the average or maximum, because it is not possible for a program to run faster than the theoretical maximum of 100% parallelization with zero overhead. It is more likely that recorded results are slower than the theoretical speeds, possibly due to extra overhead such as data I/O or context switching by the host OS. Other processes being run on the CPUs would also cause the execution time to be longer, since the OS has to schedule for those processes to be run as well.

Threads / Matrix Dim.	1	2	3	4	5	6	7	8
128	0.0192	0.0079	0.0058	0.0046	0.0073	0.0059	0.0053	0.0065
256	0.1217	0.0675	0.0494	0.0378	0.0434	0.0415	0.0421	0.0428
512	1.0616	0.5488	0.4163	0.3378	0.3784	0.3625	0.3387	0.3573
1024	10.0943	5.2432	3.9648	3.2174	3.3142	3.2441	3.2681	3.2141
2048	115.2394	59.1137	43.7514	34.4941	34.8483	34.0204	33.5351	28.8867

Figure 2: Matrix multiplication execution time (min. of 5 runs) on the lab machine

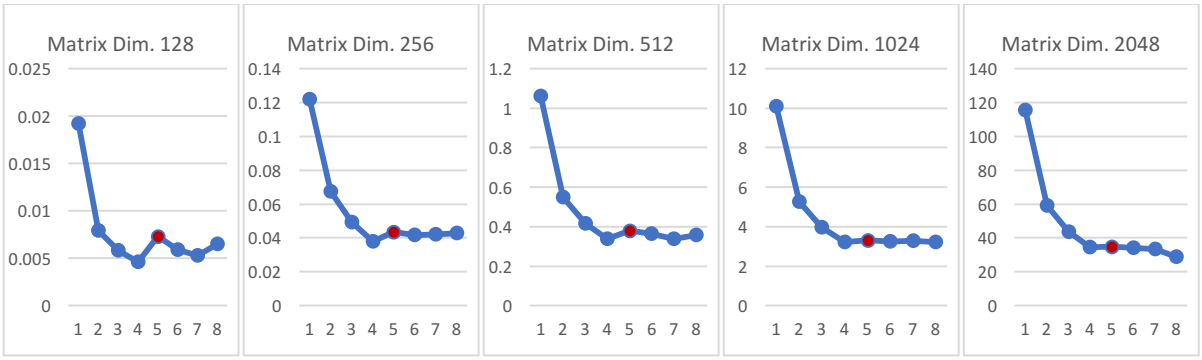

Figure 3: Execution time against number of threads on the lab machine, split by matrix dimensions

There is a significant observed discontinuity in timing between 4 and 5 threads across all matrix dimensions in Figure 3 above, which are also highlighted in Figure 2. Since the lab machine has only 4 cores, an additional thread would cause unnecessary overhead, such as through more context switching of the additional process. Since context switching requires dumping and reloading the contents of the CPU registers onto the stack, frequent context switching can cause overall runtime to be increased.

As for the Tembusu node, we observe similar trends occurring. From the charts in Figure 4 as well as the tabulated results in Figure 34 in the Appendix, we can see that there are two points of discontinuity, between 12 to 13 threads, as well as between 24 to 25 threads, across all input matrix sizes. Since the Tembusu node has 24 threads, the reason for the 24-25 thread discontinuity is similar as to what was observed on the lab machine.

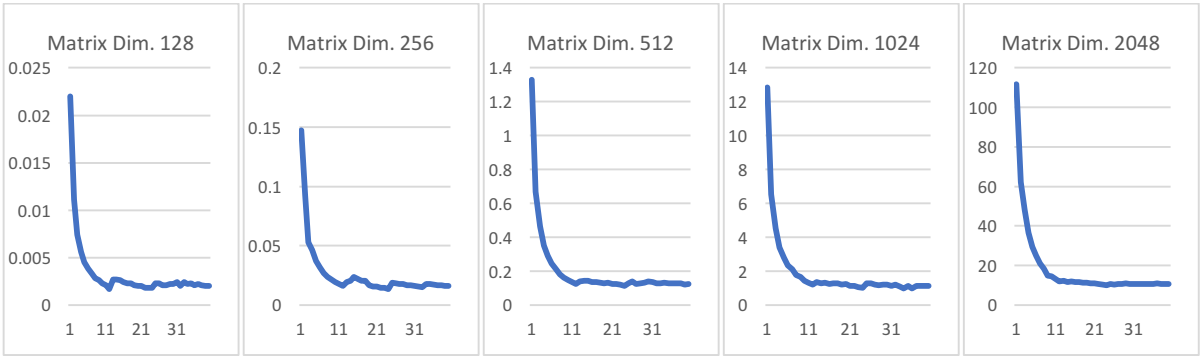

Figure 4: Execution time against number of threads on the Tembusu node, split by matrix dimensions

However, since the CPUs on the Tembusu node have hyper-threading, only 12 threads are actually being run concurrently, with the other 12 being pipelined for faster execution. The discontinuity from 12 to 13 threads is likely because of additional overhead required to invoke the 13th virtual processor, or because it is unlikely for the additional virtual processor to be scheduled efficiently such that all 13 threads are able to complete the task at the same time. However, the time decreases steadily from 13 threads to 24 threads, because each thread handles less work and thus the overhead becomes less significant.

We can also analyse the speedup of the program execution with respect to the number of processors in two different ways. For practical purposes, only the results on the Tembusu node was analysed. Using Amdahl's model, where we keep the problem size constant, we can calculate the speedup factor on the different matrix dimensions running from 1 to 40 threads, which is shown in Figure 5 below.

By Amdahl's law, we expect that if problem size is fixed, that speedup would be limited by the serial portion of the program, and would hence "taper off" as number of processors increase. Regardless of the problem size, we observe that the performance increase does exhibit similar trends, with a sharp increase up to around 12 threads. However, since the chart uses non-theoretical data, discontinuities in speedup is also affected by the same reasons for the discontinuities for execution time as explained above.

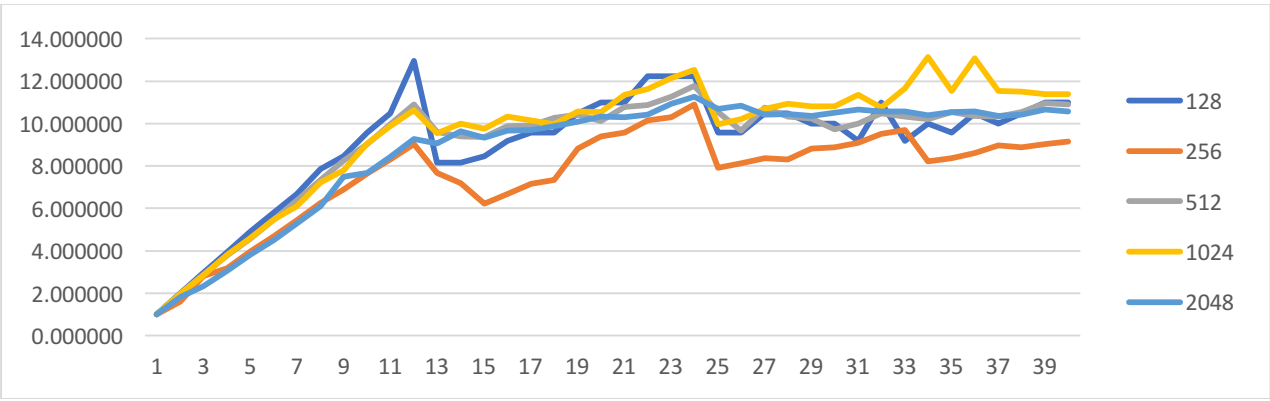

Figure 5: Amdahl speedup against number of threads on the Tembusu node for different matrix dimensions

However, by Gustafson's law, we expect that, if the execution time is fixed, the speedup would not be limited in the same way as expected using Amdahl's model. Instead, the argument is that more work can be done in the same fixed time T by using more processors, and hence speedup should be calculated based on the time taken for a single processor to do the same

amount of work that can be done by multiple processors, within the fixed time T. Hence, we would expect that the Gustafson speedup would increase linearly with respect to the number of processors used, assuming that the amount of work done is adjusted according to a fixed time T.

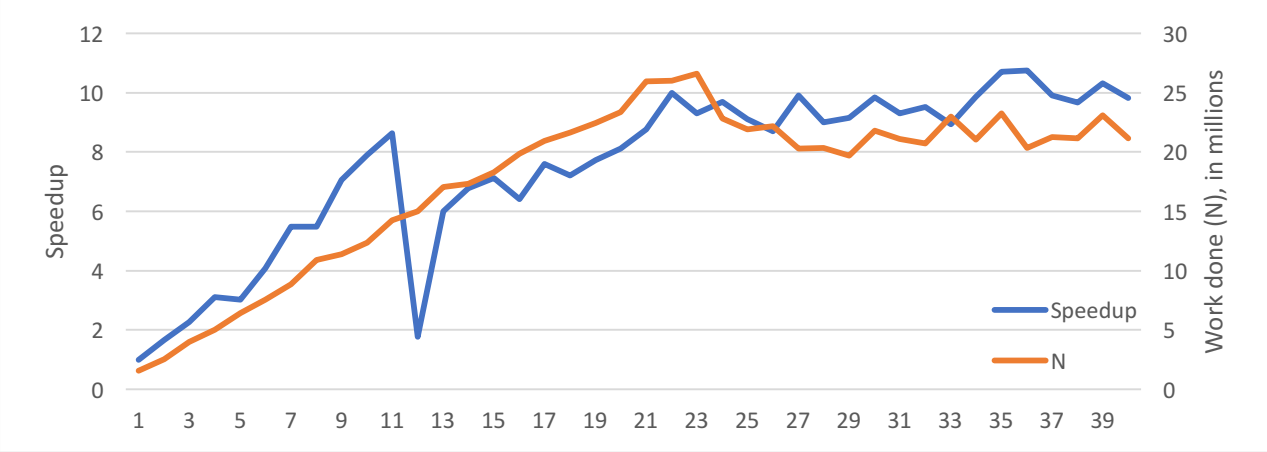

Figure 6: Gustafson speedup against number of threads on the Tembusu node, fixed time T = 100 ms

From Figure 6 above, the speedup was calculated for each P by estimating the amount of work that can be done by P processors within a fixed time $T = 100\text{ms}$, and measuring the time that a single processor would take to do the same amount of work. The results are tabulated in Figure 36 in the Appendix as well.

We can see that the Gustafson speedup as shown in the chart is slightly more optimistic with respect to an increase in number of processors. However, the results are largely limited by real-world phenomena, such as memory effects, which would reduce the extent that a linear improvement of speedup is seen in the experimental results.

Memory Effects

In this section, we will be investigating memory effects on the execution time of a program. A quick glance at the source code of `testmem.c` shows that the program executes a constant number of operations (64M operations), irrespective of the independent variable (i.e. size of array). However, lookups and writes are performed on every 64 bytes in the array for each operation, looping back to the start of the array once the array size limit is reached.

SIZE (KB)	**4**	**8**	**16**	**32**	**64**	**128**	**256**	**512**
TIME (S)	0.1517	0.1458	0.1469	0.1455	0.1540	0.1552	0.1596	0.1913
SIZE (KB)	**1,024**	**2,048**	**4,096**	**8,192**	**16,384**	**32,768**	**65,536**	**131,072**
TIME (S)	0.1938	0.1944	0.2506	0.7598	0.9549	0.9519	0.9627	0.9713

Figure 7: Program execution time of constant number of operations against different size arrays on lab machines

The above table shows the execution time of the program, with the minimum time recorded across 5 successive runs for each of the array sizes. We can see that, apart from the very first run (4 kB array), there is a general increase in execution time with respect to the size of the array.

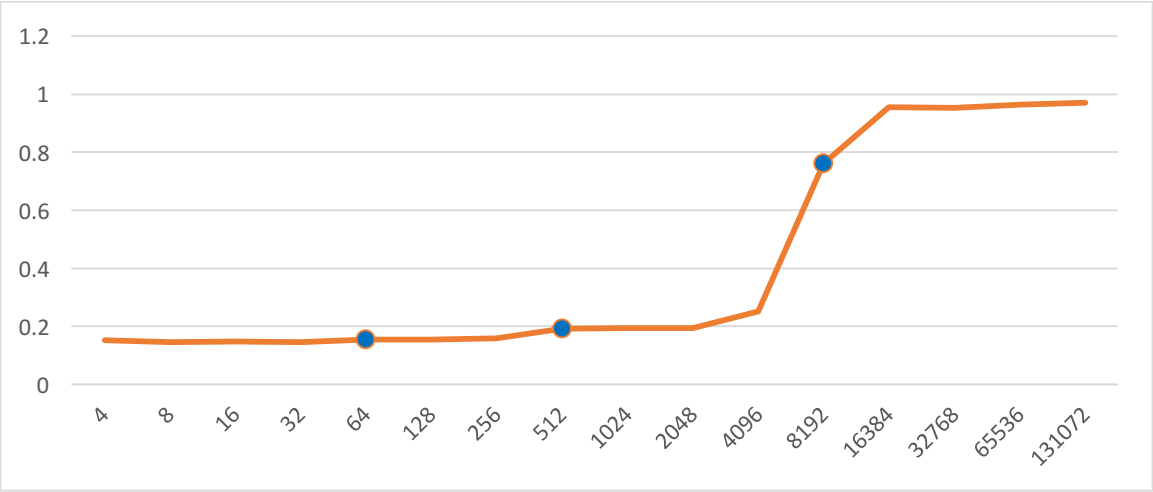

Figure 8: Program execution time of constant number of operations against different size arrays on lab machines

From Figure 8 above, we can observe small discontinuities between 32 kB to 64 kB and 256 kB to 512 kB. Since the lab machine has a L1 and L2 cache size of 32 kB and 256 kB respectively, there is a significantly increased number of cache misses due to the linear access pattern of the array, such that an increase of array size causes the entire array to be too large to fit in the L1 and L2 data caches respectively. This cache-busting access pattern would cause program execution to be much slower since data needs to be transferred from higher-level caches to lower-level caches, incurring significant waiting time.

The largest discontinuity observed is between 4096 kB to 8192 kB. Since the last-level L3 shared CPU cache size is 6 MB, the missed cache data for an 8 MB array would have to be continuously fetched from the memory into the last-level cache (LLC), which is several orders of magnitude slower than accessing from any part of the CPU caches.

Accuracy

When running `fpadd1`, we observe a peculiarity where every number summed after 16,500,000 produces 16,777,216, as follows:

```
$ ./fpadd1
Adding 1 to 0, lots of times...
...
Adding 16000000 1s to 0 gives this result: 16000000.0
Adding 16500000 1s to 0 gives this result: 16500000.0
Adding 17000000 1s to 0 gives this result: 16777216.0
Adding 17500000 1s to 0 gives this result: 16777216.0
...
```

This is because the underlying data type is a single-precision floating point number (IEEE 754) declared in C using `float`, which uses 32 bits in memory to store 3 different values as such:

Sign bit (1 bit)	Exponent (8 bits)	Significand precision / Mantissa (23 bits)

Figure 9: Diagram illustration the structure of a single-precision IEEE 754 floating point number

We can get the value represented by an IEEE 754 number using the formula below:

$$\text{value} = (-1)^{\text{S}} \times 1.\text{M} \times 2^{\text{E}-127}$$

$16{,}777{,}216$ is equal to 2^{23+1}, which has an unbiased exponent value of 24 (or $\text{E} = 151$). Thus, by incrementing the mantissa by 1, we would get $16{,}777{,}218$ since there are only 23 bits in the mantissa. Because the program increments the float by 1 each iteration, the number $16{,}777{,}217$ cannot be precisely represented, which is still represented using the same value for $16{,}777{,}216$. This results in every subsequent iteration to always store the same value in the float each time without actually incrementing the value.

On the other hand, when running `fpadd2`, we observe that peculiarities exist for the numbers beyond $1{,}000{,}000.3$, which actually show a loss of floating point accuracy:

```
$ ./fpadd2
Adding 1 to 0.3, lots of times...
Adding 500000 1s to 0.3 gives this result:   500000.3
Adding 1000000 1s to 0.3 gives this result:  1000000.3
Adding 1500000 1s to 0.3 gives this result:  1500000.2
Adding 2000000 1s to 0.3 gives this result:  2000000.2
...
Adding 4000000 1s to 0.3 gives this result:  4000000.2
Adding 4500000 1s to 0.3 gives this result:  4500000.0
...
```

In the same vein, since the underlying data type is a single-precision IEEE 754 floating point number which is 32-bits long, the amount of discrete numbers that can be represented gets more and more sparse as the number diverges from 0. Because there are a limited number of bits in the mantissa, the number $1{,}500{,}000.3$ is actually stored as $1{,}500{,}000.25$, which is then displayed as $1{,}500{,}000.2$ due to rounding-off errors.

When running `fporder`, some sample program output is as follows, which show a difference when the order in which the numbers are summed up is reversed:

```
$ ./fporder
Adding 20 (pseudo)randomly generated floating point numbers for 1 to 20 and 20
down to 1

Sum from 1 to 20 is 759.563782
Sum from 20 to 1 is 759.563721
```

Because of limitations of real numbers which can be precisely represented using floating point numbers, floating point addition is commutative but not necessarily associative (i.e. the order of addition operations matters), due to rounding off errors on intermediate values.

To demonstrate, let us assume that there are only 3 terms in the sum, which follows the following explicit formulae as denoted by the program:

$$\text{Sum}_{1\to3} = (a_1 + a_2) + a_3$$
$$\text{Sum}_{3\to1} = (a_3 + a_2) + a_1$$

Since floating point addition is commutative but not associative, it means that the following are not equal:

$$\text{Sum}_{1\to3} = (a_1 + a_2) + a_3 \neq a_1 + (a_2 + a_3) = \text{Sum}_{3\to1}$$

The same concepts of non-associativity of floating point numbers can also be seen when running the `fpomp` program as follows, with added print statements within each thread:

```
$ ./fpomp
Adding 100000 randomly generated floating point numbers using 4 threads

Local sum thread 0 is 769448.937500
Local sum thread 2 is 765209.125000
Local sum thread 1 is 769626.687500
Local sum thread 3 is 766283.062500
Final sum is 3070567.750000

Local sum thread 2 is 765209.125000
Local sum thread 3 is 766283.062500
Local sum thread 1 is 769626.687500
Local sum thread 0 is 769448.937500
Final sum is 3070568.000000
...
```

Note that the individual sums within each thread is the same across different repetitions, but the final sum differs. By looking at the source code, we note that the final summation occurs in a critical section of the OpenMP parallel block (denoted by `#pragma omp critical`).

Since each thread may start and finish in any order, the order of which the threads enter the critical section will also be different each time. Since floating point addition is commutative but not associative, the successive addition of floating point numbers using the += operator would yield different results for a different order in which the numbers are summed.

Running the same program on the Tembusu node yields the same peculiarities, except that the number of threads that are involved is 24 instead of 4. However, the maximum error margin observed on the Tembusu nodes is 1.00, which is much larger as that observed on the lab machines, which is 0.25. Because there are 24 addends with 24 threads, every additional operand increases the possibility for the loss of precision, widening the error margin with every addition.

Communication and Speedup

The effect of using MPI (Message Passing Interface) on the execution time of running a matrix multiplication program, `mm-mpi`, was also investigated. MPI allows us to utilise and communicate with other machines (through SSH), in order to parallelize the workload across a greater number of cores. This investigation was performed on the Tembusu node.

Running MPI on several other machines on the Tembusu cluster, allows MPI to better parallelize each process to machines based on the number of cores available, so that oversubscription is minimised to improve execution time. In this investigation, the program

utilised between 8 to 64 threads, as well as between 1 to 16 remote machines within the same cluster.

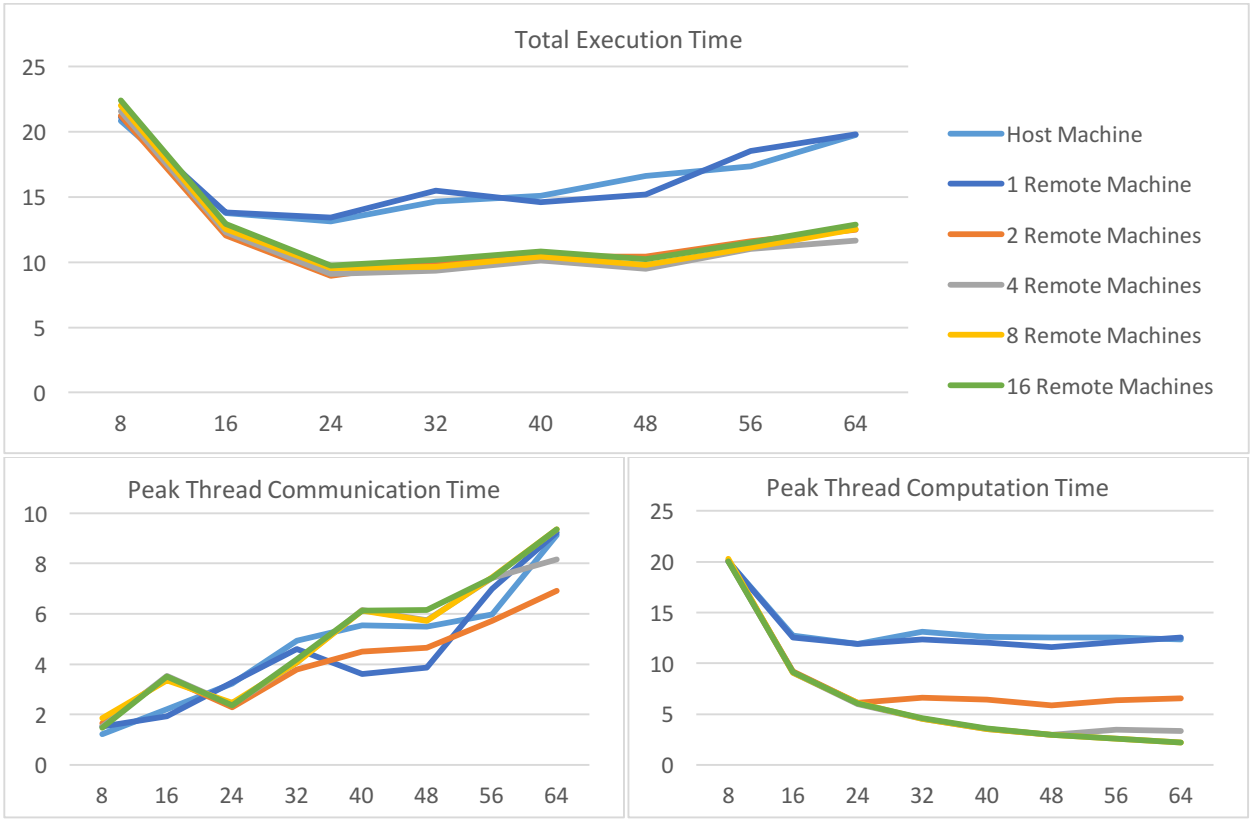

Figure 10: Total execution time and peak thread communication/computation time against number of processes, run on several nodes on the Tembusu cluster

Measuring the overall execution time, as well as the peak thread communication and computation time allows us to investigate the communication-to-performance trade-off that might arise from a significant overhead in transferring data for communication between more processes. The results can be seen in Figure 10 above, as well as in Figure 37 in the Appendix.

From the chart, we can see that there is a general trend regardless of number of machines used such that execution time decreases sharply when more threads are used up to 24, but starts to increase steadily up to 40 threads. This is because there is a greater amount of data that needs to be sent between more slave processes and the master process, as well as other transmission factors that may come into play such as bandwidth limits over the network. This is supported by the peak thread communication time chart, which shows a steady increase in communication time with more processes, which might have countered the performance increase from greater parallelization of work.

Additionally, we note that using 2 or more machines to parallelize the workload shows an absolute decrease in execution time when using 16 or more processes. Furthermore, the peak thread computation time decreases with more processes, but plateaus differently depending on the number of machines. This is because since each machine has only 12 physical cores, any additional processes would be context-switched. Allowing more cores to handle the spread of work helps to decrease overall execution time and thread computation time.

Part 2

Introduction

We will be investigating the reasons for a low framerate in the demo application that is using GPU.js. The hardware used for this investigation is my personal computer, a Macbook Pro 2014, with a discrete NVIDIA GTX 750M GPU.

The demo application uses the HTML5 `getUserMedia` API, which provides access to the computer's camera, as well as GPU.js for reading in the image data, optionally filtering it with an embossing convolution kernel, followed by drawing a graphical output on a HTML5 Canvas. This is done in a loop, using the HTML5 `requestAnimationFrame` API which helps to run a function in succession repeatedly, throttling the frames to maintain a reasonable framerate (60 fps) if needed.

We will be investigating the low framerate issue solely on the "GPU mode" of the demo application, when filtering is enabled. On my machine, the framerate is about 17 fps on average, which is shy of the peak 60fps framerate.

The following diagram is a high-level overview of the pipeline of tasks that are done in each animation frame, assuming that GPU mode is enabled:

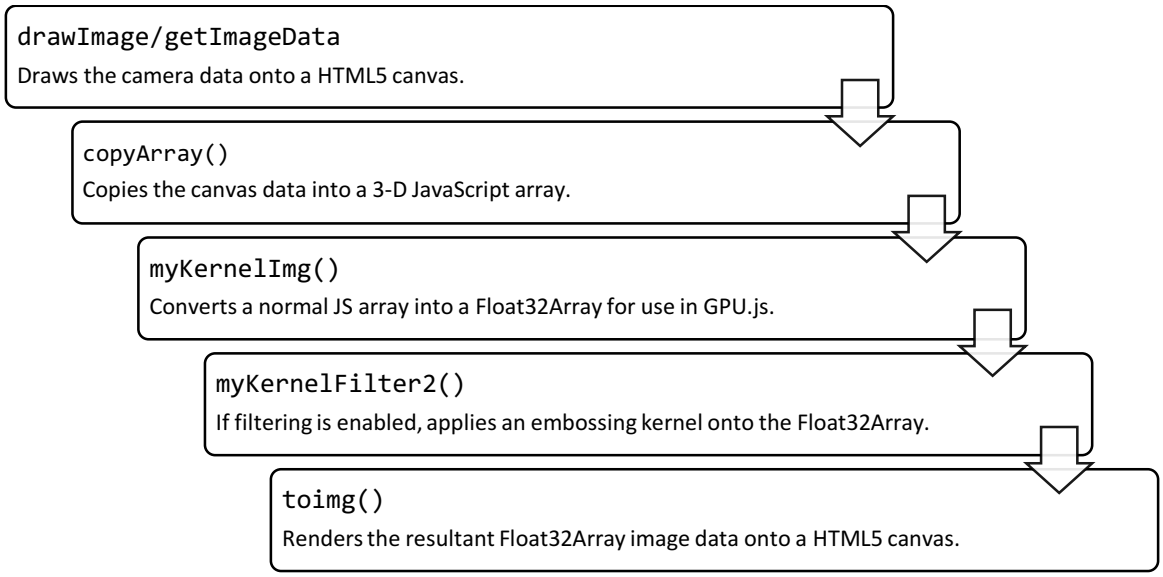

Figure 11: High-level overview of the pipeline of tasks during each animation frame

Initial Investigation

In order to investigate the reason for a low framerate, we will need to find out the average time taken for each of the tasks in the pipeline above, followed by optimising certain routines in those tasks. We will be tweaking the JavaScript for easier profiling, which can be found in the attached ZIP file, under the `part2` directory.

Firstly, we note that there are two animation frame sequences that are being called concurrently. Since this makes it difficult to measure runtime in the form of a pipeline, we can

combine both of them into a single pipeline within the `renderLoop` function (see `test01.html`).

Next, we can do some initial profiling of the application using Chrome's built-in profiler. The following screenshot shows the breakdown of JavaScript tasks, as well as other things such as GPU, over the timeframe of a single animation frame, which took 59.6ms. In order to reach 60fps, the animation frame should complete within 17ms or less.

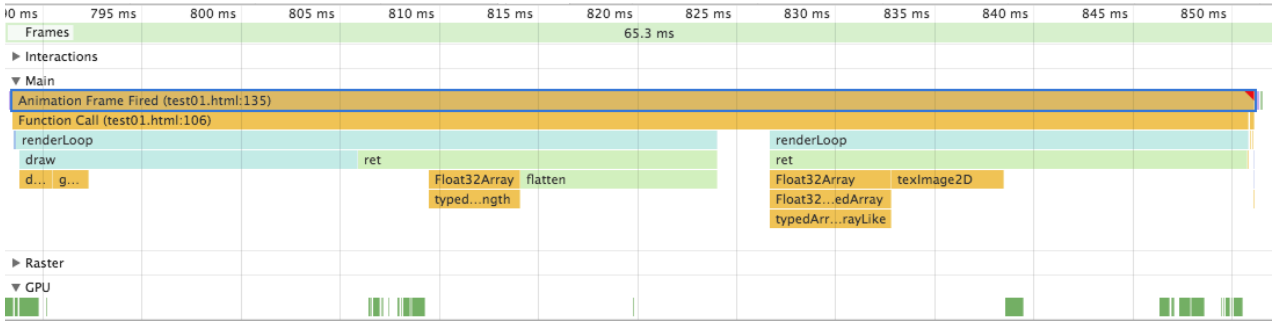

Figure 12: Screenshot of Chrome profiler on a single animation frame of the demo application, total 59.6ms

As expected, the bulk of the time is spent within `renderLoop`, which performs the pipeline of tasks as outlined above. Looking at the first occurrence of `renderLoop`, we see that the `draw` function takes up around a third of the pipeline's time, corresponding to both task 1 and 2 of our pipeline.

Due to the limitations of the profiler, we cannot observe the time taken by JavaScript statements or internal API calls, which appears as `ret` in the screenshot above. Instead, we will need to refactor the code according to the pipeline of tasks by encapsulating the actual code within their own functions (see `test02.html`).

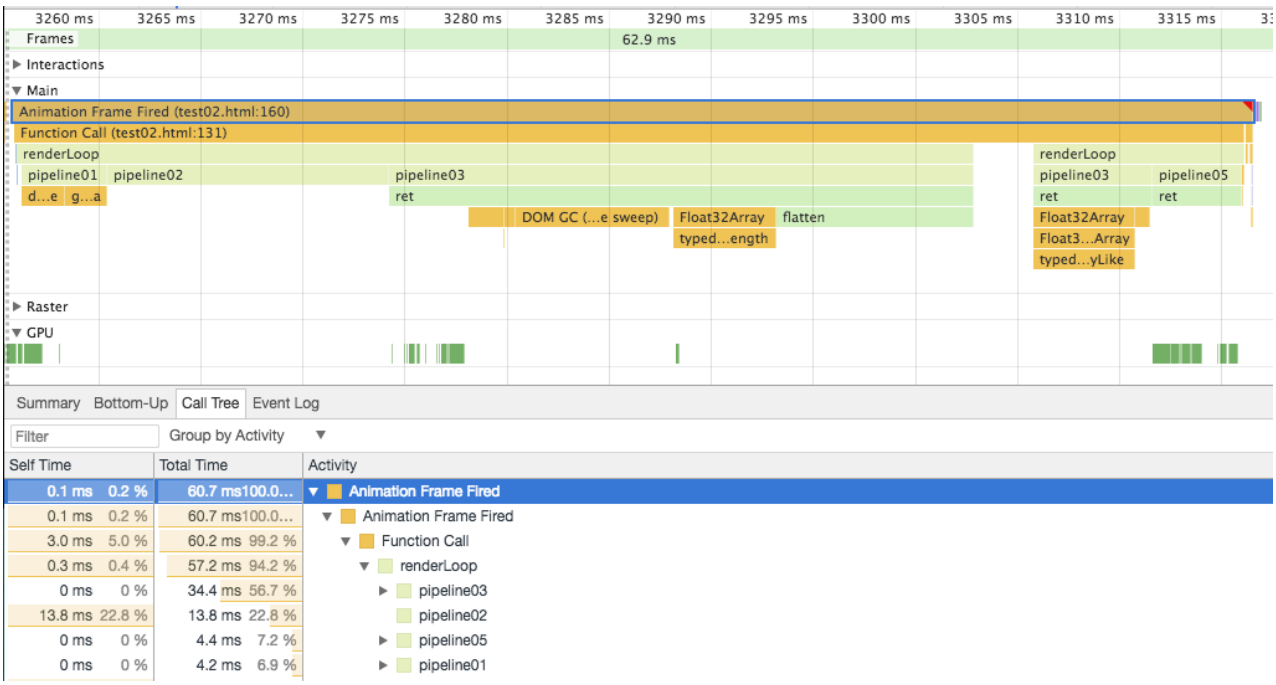

Figure 13: Screenshot of Chrome profiler on a single animation frame after refactoring into pipeline tasks

We see that pipeline task 3 takes up the most time here, with 34.4ms out of 57.2ms for `renderLoop`, which is even more than what we had discovered previously.

In order to be more scientific and accurate, instead of depending on Chrome's profiler to estimate the proportion of time spent within each task, we can instead use simple JavaScript APIs to run a timer for each of the tasks (see `test03.html`), which gives us a breakdown as follows:

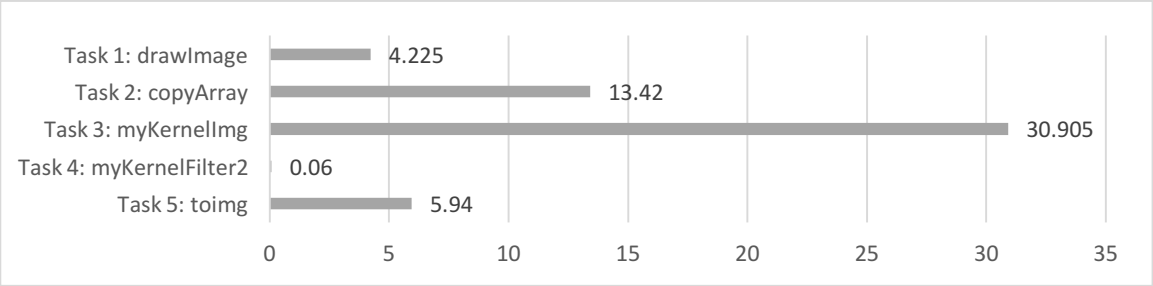

Figure 14: Breakdown of time taken by pipeline tasks, averaged over 200 samples

Through this, we can confirm our suspicions that `myKernelImg` takes up the longest time within the pipeline, taking up more than 30.905ms out of 54.54ms on average.

Digging Deeper

Inspecting deeper into the codebase, we realise that the latency incurred by `myKernelImg` is mostly unavoidable, since the data needs to be copied from the CPU to the GPU. While we can try to reduce the size of the array that is being transferred (we will do so later), we can also try and optimise other tasks that are CPU-bound.

For example, `copyArray` looks like it could be optimised since all it is doing is to copy an array into another array. Since `myKernelImg` is also copying an array, except in parallel from CPU to GPU, we could combine these two tasks. By modifying the serial array access code in `copyArray` to a parallel version in `myKernelImg` (see `test04.html`), we would effectively be removing one task from the pipeline. Once done, we can re-profile the timing, as follows:

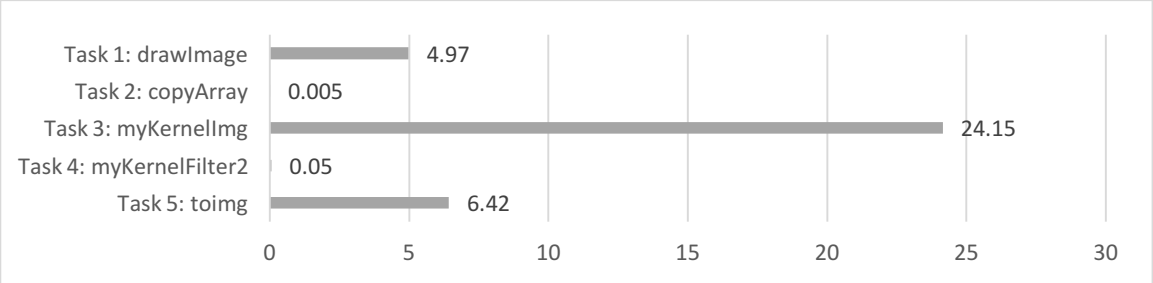

Figure 15: Breakdown of time taken by tasks after combining task 2 & 3, averaged over 200 samples

We see that the overall time for one frame is now 35.595ms, which is reduced by almost 19ms since task 2 is now combined into task 3. Additionally, task 3 saw an improvement in runtime as well, which could possibly be due to memory effects. The average FPS now is also about 30fps.

Further Optimizations

Another optimization that is possible is to remove the alpha channel from the array that is being worked on. Since the camera input does not produce an alpha channel, it does not make sense to have an extra 800x600 array being passed around which is all 1s.

We can remove this extra alpha channel, reducing the size of the working array from 800x600x4 to 800x600x3, which is a 25% decrease. This requires changing of the dimensions for each of the GPU.js kernel functions (see `test05.html`). The improvement can be seen as follows:

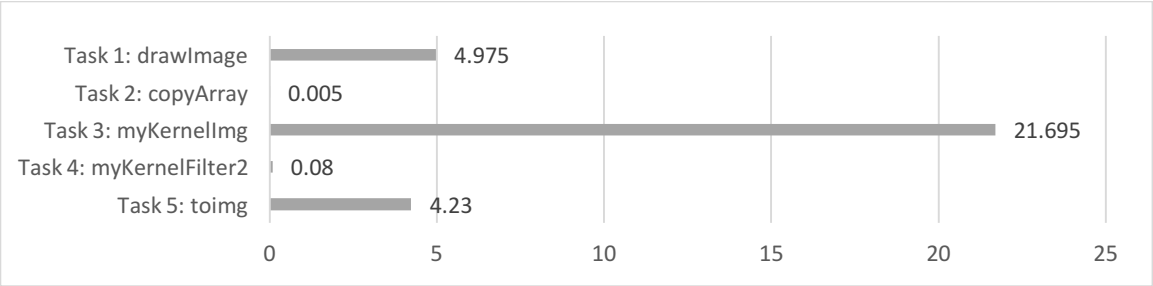

Figure 16: Breakdown of time taken by tasks after removing the alpha channel, averaged over 200 samples

We see that there is an improvement in both tasks 3 and 5 as expected, since they are the GPU.js kernel functions that work on the array directly. A smaller array would mean that the GPU shaders are able to complete all of the work in a shorter time, as can be seen with a 2-3ms improvement for each of `myKernelImg` and `toimg` functions. The overall time for one frame is now 26.01ms, with an average framerate of 38fps.

Analysis

Through the various optimisations, we can plot a bar chart to summarise the reduction in execution time as follows on GPU when filtering is enabled:

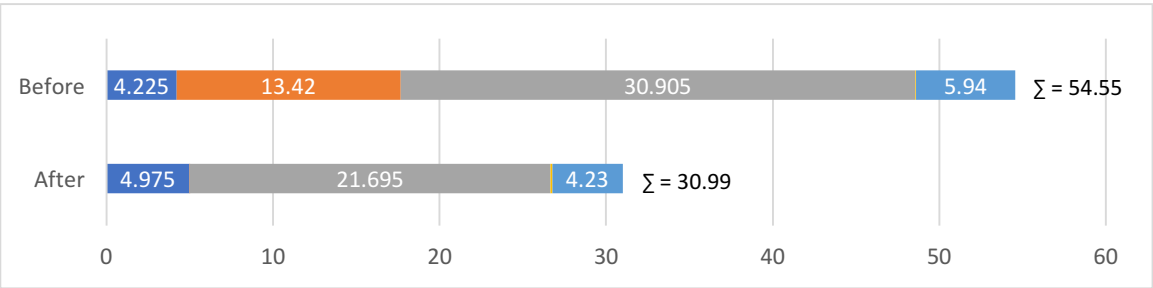

Figure 17: Comparison of runtime of tasks within one animation frame, before and after optimization

We can see that there is a rather significant improvement simply by removing redundant CPU-bound tasks and parallelizing it, as well as to reduce the size of data, allowing better usage of CPU caches and memory.

Additionally, we can look at the speedup of using the GPU as compared to CPU, which is tabulated below:

	BEFORE				*AFTER*			
	CPU		**GPU**		**CPU**		**GPU**	
	Filtering	*No Filter*	*Filtering*	*No Filter*	*Filtering*	*No Filter*	*Filtering*	*No Filter*
Task 1	4.150	3.650	4.225	4.210	4.200	3.545	3.855	3.860
Task 2	13.250	13.650	13.420	13.700	0.000	0.000	0.000	0.000
Task 3	83.200	85.450	30.905	30.360	53.550	53.480	17.720	17.505
Task 4	129.400	0.000	0.060	0.000	108.400	1.380	0.060	0.000
Task 5	24.250	25.800	5.940	6.270	18.600	18.740	3.405	3.605
TOTAL TIME	*254.250*	*128.550*	*54.550*	*54.540*	*184.750*	*77.145*	*25.040*	*24.970*

Figure 18: Comparison of runtime of tasks within one animation frame, across CPU/GPU, before/after optimisation

From Figure 18 above, we can see that the recorded timings are in line with our optimisations and explanations. We see that after our optimisations, runtime for Task 2 is completely eliminated. Additionally, the runtime for Tasks 3 to 5 have been reduced considerably, for both CPU and GPU, due to a smaller array being used, leading to better cache hit rates.

We can measure the speedup in terms of before and after the optimisation, as well as in terms of parallelizing the work onto the GPU, as follows:

	Before/After Optimisation Speedup				*CPU/GPU Speedup*			
	CPU		**GPU**		**Before**		**After**	
	Filtering	*No Filter*	*Filtering*	*No Filter*	*Filtering*	*No Filter*	*Filtering*	*No Filter*
Task 1	0.988	1.030	1.096	1.091	0.982	0.867	1.089	0.918
Task 2	-	-	-	-	0.987	0.996	-	-
Task 3	1.554	1.598	1.744	1.734	2.692	2.815	3.022	3.055
Task 4	1.194	-	1.000	-	2,156.67	-	1,806.67	-
Task 5	1.304	1.377	1.744	1.739	4.082	4.115	5.463	5.198
TOTAL TIME	*1.376*	*1.666*	*2.179*	*2.184*	*4.661*	*2.357*	*7.378*	*3.090*

Figure 19: Amdahl speedup with respect to before/after optimisation, as well as CPU/GPU speedup

We can see that optimisation speedup is more significant on the GPU than the CPU after optimisation. Additionally, the speedup is most significant with respect to CPU vs GPU timings, especially Task 4, which is the filtering kernel function. Task 4 takes less than a millisecond to compute on the GPU, which is mainly because the entire array is already in memory and does not need to be copied to and fro the CPU, unlike Tasks 3 and 5. On the other hand, running the same filtering function on the array serially on the CPU would be orders of magnitude slower.

Part 3: Image Processing on GPU.js

In my implementation of the image processing filters, I chose to implement a few distinct filters that could complement well together, to demonstrate the usefulness as well as the various techniques required to implement them. The following are some screenshots of the application in action:

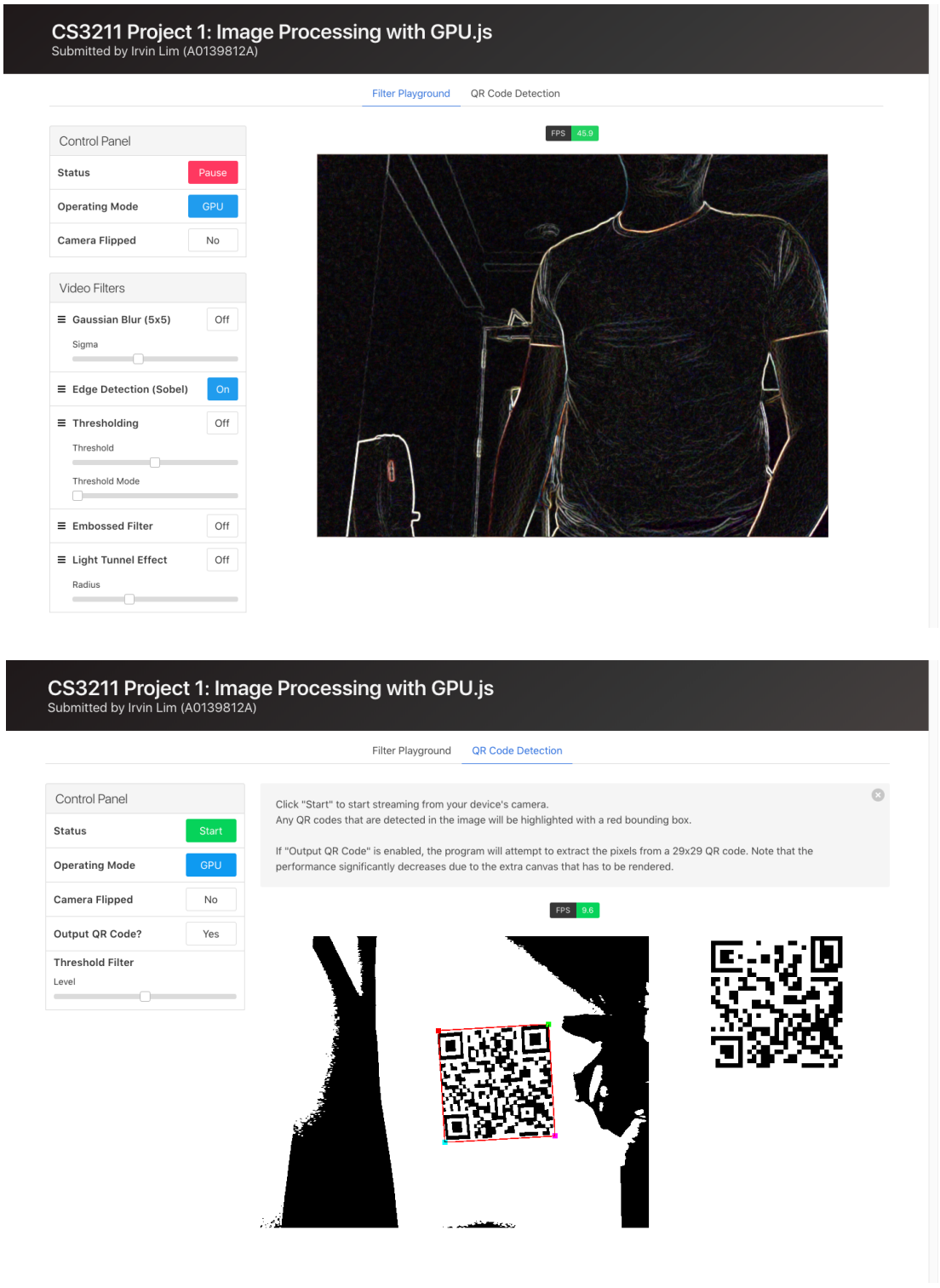

Figure 20: Screenshots of the video filtering and QR code recognition applications in action

Features Implemented

The following is a list of features that are implemented on the application, which will be further elaborated in later sections below:

- **Video filters**
  - *Gaussian Blur*: Uses the Gaussian kernel to convolve the image array. Configurable.
  - *Edge Detection*: Uses the Sobel kernel to convolve the image array.
  - *Thresholding*: Thresholds image pixels depending on their lightness. Configurable.
  - *Light Tunnel Effect*: Fun video filter, inspired by Apple's Photo Booth. Configurable.
- **Image recognition**
  - Detects 29x29 QR codes, as long as all finder markers are visible.
  - Optionally transforms and extracts the bits to recreate the QR code.
- **Features**
  - Video processing can be paused and resumed in real-time.
  - Video filters can be configured by parameters using draggable sliders.
  - Video filters can be pipelined and toggled in real-time.
  - Video filters can be reordered in real-time using a drag-and-drop interface.
  - Camera can be flipped so that it is similar to front cameras on smartphones.
- **Special techniques**
  - *Computational time reduction*: Use faster algorithms when appropriate.
  - *Data transfer reduction*: Using smaller arrays when appropriate, and also prevent unnecessary CPU-GPU copying.
  - *Accuracy improvements*: Reduction of false positives for QR code detection.
  - *Loop parallelization:* Parallelization loops within the GPU when appropriate in order to speed up runtime.

Libraries

This project uses the v1 release of GPU.js, which provides a new API and boasts improved performance. Most notably, the speed had improved considerably, allowing each animation frame to run at 60fps comfortably without any filters enabled, as compared to the maximum framerate of 38fps achieved in Part 2.

Other JavaScript and CSS libraries include the Bulma CSS framework[3] and html5sortable[4], which are used for the user interface of the application.

Setup Instructions

The application runs in the browser, preferably on Google Chrome, and is packaged under two separate HTML filters, namely `filters.html` and `qr.html`, holding the video filter functionality and QR code image recognition, respectively.

[3] https://bulma.io/

[4] https://github.com/lukasoppermann/html5sortable

Since the application makes use of the device's camera, the HTML files must be run on the `http` protocol, rather than the `file` protocol. This can be done by starting a static HTTP server, either with `http.server` (Python 3), `SimpleHTTPServer` (Python 2) or `http-server` (Node.js). More instructions can be found in the README file in the attached ZIP file.

Explanation of Video Filters

Like what was shown in Part 2, the architecture of the video filtering application is similar in nature, with a pipeline that looks like as follows:

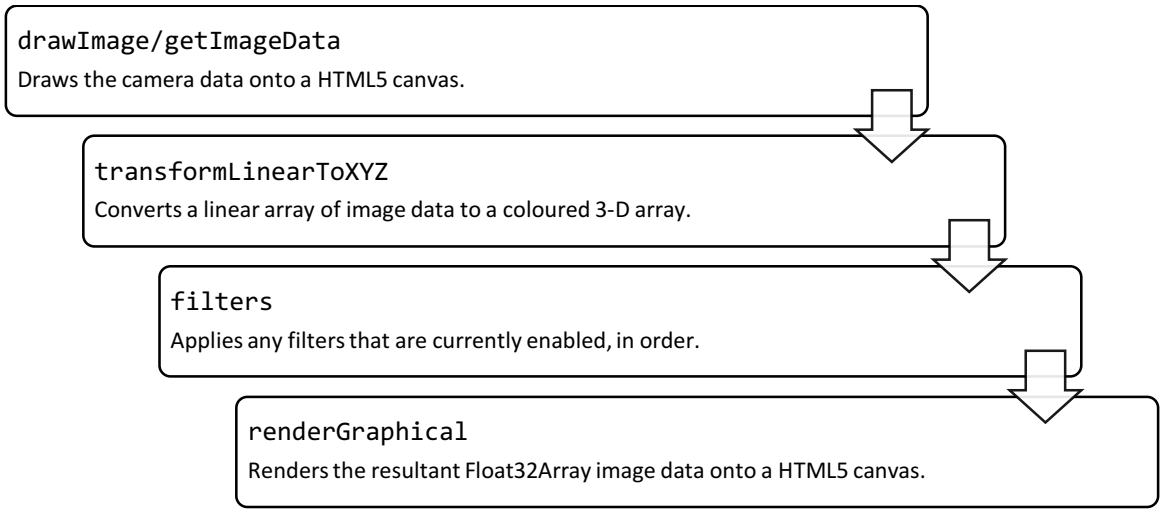

Figure 21: High-level overview of the pipeline of tasks during each animation frame

Gaussian Blur

Though there are different types of blur, I chose to implement the Gaussian blur filter because it is easy enough to implement in parallel. The basis of a Gaussian blur is the Gaussian function $G(x)$, which can when multiplied, gives us the function in two dimensions $G(x, y)$:

$$G(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{x^2}{2\sigma^2}}$$
$$G(x, y) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}}$$

The 2-D Gaussian function is then used to construct a kernel which would then be used to convolve the image with. In my implementation, I chose to construct a 5x5 kernel, which would be of the following form:

$$\begin{bmatrix} G(2,2) & G(1,2) & G(0,2) & G(1,2) & G(2,2) \\ G(1,2) & G(1,1) & G(0,1) & G(1,1) & G(1,2) \\ G(0,2) & G(0,1) & G(0,0) & G(0,1) & G(0,2) \\ G(1,2) & G(1,1) & G(0,1) & G(1,1) & G(1,2) \\ G(2,2) & G(1,2) & G(0,2) & G(1,2) & G(2,2) \end{bmatrix}$$

Note that the kernel is symmetrical in 2 dimensions around the central element, which decreases as the distance from the centre increases. Also, since σ represents the standard deviation of the function, a larger value of σ would mean a wider and more even spread of values in the matrix.

This can be easily converted into a kernel function to be run in parallel on each pixel per thread, since we can simply convolve the input matrix with the Gaussian matrix with respect to a given input σ, which would produce the resultant matrix in a single pass. The resultant matrix would need to be normalized such that the sum of all terms in the matrix is 1, in order to maintain the lightness of the image pixels.

Optimisations and Features

Since the Gaussian function is rather computationally intensive in its raw form, one optimisation could have been to precompute the kernel values. However, the value of σ is parameterized in my implementation, which means that the user can modify the extent of the blur effect by adjusting σ using a slider, which means that the kernel cannot be precomputed.

Hence, one alternative is to compute the kernel in the CPU together with memoization. This means that the Gaussian coefficients would only be recomputed when the value of σ changes, and would be passed directly to the kernel. The speedup for implementing this optimisation on both the CPU and GPU is obtained as follows by timing the time taken by the kernel function alone:

	CPU		GPU	
	Original	**Improved**	**Original**	**Improved**
AVG. FRAMERATE	1.4 fps	6 fps	53 fps	53 fps
RUNTIME (MS)	641.1350	137.9760	0.0575	0.0650
SPEEDUP	**4.6501**		**NEGLIGIBLE**	

Figure 22: Speedup from optimisations in computing Gaussian coefficients

We note that there is no improvement to the kernel function on the GPU with such an optimisation, while the Amdahl speedup on the CPU is 4.65x. Since the computation was already fully parallelized on the GPU, it could be that the time for the additional data transfer of the coefficients from CPU to GPU outweighed the time savings from not computing 6 Gaussian coefficients in each shader. However, since there is no parallelization on CPU mode, shaving off 6 computations per pixel would mean a reduction of 8,640,000 Gaussian function computations, which is much more significant.

Other possible optimisations to this filter include implementing a quicker blur algorithm, such as the box blur.

Edge Detection

Different types of edge detection algorithms exist as well, though I chose to implement an easier one by using the Sobel operator. Similar to the Gaussian blur kernel, Sobel edge detection involves the convolution of an input matrix with the Sobel matrices $\mathbf{G}_x$ and $\mathbf{G}_y$ as follows:

$$\mathbf{G}_x = \begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix} * \mathbf{A};\ \mathbf{G}_y = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} * \mathbf{A}$$

The two Sobel operators approximate the gradients of neighbouring values, which correlates to edges in an image when the brightness of a pixel changes sharply from its adjacent pixels. The $\mathbf{G}_x$ operator approximates the gradients in the x-direction, while $\mathbf{G}_y$ approximates the gradients in the y-direction. To coalesce the result of these two convolutions, we can take the magnitude of the two gradients, as follows:

$$\mathbf{G} = \sqrt{{\mathbf{G}_x}^2 + {\mathbf{G}_y}^2}$$

Similar to the implementation of the Gaussian blur kernel, we can simply compute the convolution for each pixel in its own thread to produce the resultant image matrix in a single pass.

Thresholding Filter

Another filter that was implemented was the thresholding filter, which can be used to binarize a greyscale image by taking all pixels greater/lesser than a threshold value and substituting it with black or white. Such an effect is often used in applications such as optical character recognition (OCR), augmented reality (AR) marker detection, QR code detection, etc.

I opted to implement the simplest thresholding algorithm, where a pixel's colour is dependent solely on its lightness value. While the lightness value of a greyscale pixel is simply its value (out of 1), computing the lightness for a colour image is different. There various ways to compute lightness of a RGB pixel, with the first and simplest being the average of the R, G and B values, as follows:

$$L = \frac{1}{3}(R + G + B)$$

Another method, which is used in the HSL (hue-saturation-lightness) colour model, is to take the average of the maximum of the RGB values with the minimum of the values, as follows:

$$L = \frac{1}{2}\max(R, G, B) + \frac{1}{2}\min(R, G, B)$$

Lastly, another colour model, HSP (hue-saturation-perceived lightness) provides yet another way of computing it[5]:

$$L = \sqrt{0.299R^2 + 0.587G^2 + 0.114B^2}$$

I chose to adopt the last method, as it seems to have better results on gradients, though there is no correct choice of method.

[5] http://alienryderflex.com/hsp.html

Additionally, the thresholding filter allows the user to modify two different parameters: the threshold value, as well as the thresholding mode. The thresholding value is straightforward, which determines the lightness level at which pixels should be replaced or not. There are 5 available thresholding modes, which are based on the available modes in OpenCV[6]:

1. `THRESH_BINARY`: White if beyond threshold; black otherwise
2. `THRESH_BINARY_INV`: Black if beyond threshold; white otherwise
3. `THRESH_TRUNC`: Original pixel if beyond threshold; white otherwise
4. `THRESH_TOZERO`: Original if beyond threshold; black otherwise
5. `THRESH_TOZERO_INV`: Black if beyond threshold; original pixel otherwise

The implementation for the thresholding filter is also straightforward. Since the resultant pixel can be determined from only the input pixel itself, we can compute the resultant image matrix in a single pass with each pixel in its own thread.

Light Tunnel Effect

This filter is different from the rest, as it is not a generic filter that is often used for other purposes. Instead, this was inspired by Apple's Photo Booth filters, similar to the "fun" filters available on Snapchat and Instagram:

Figure 23: Screenshot of video effects available on Photo Booth, with the Light Tunnel effect highlighted.

This filter is straightforward to implement, as it only requires simple trigonometry to compute the value of each pixel in the resultant image. Since we know the origin and radius of the circle in the centre of the image, we can use Pythagoras' theorem to determine if a given pixel is within the circle or not, given a circle $C(x_0, y_0, r)$ and point $P(x, y)$:

[6] https://www.docs.opencv.org/trunk/d7/d4d/tutorial_py_thresholding.html

$$withinCircle(C, P) = \sqrt{\left|P_x - C_{x_0}\right|^2 + \left|P_y - C_{y_0}\right|^2} < C_r$$

From this, if an input pixel is within the circle, it will not change, but if it is outside of the circle, we will replace it with the pixel that is at the circumference of the circle, at the same angle to the origin:

$$\theta = \text{atan2}(C_{x_0} - P_y, C_{y_0} - P_x$$
$$P_{out}(x, y) = (C_{x_0} - \lfloor r \cos\theta \rfloor, C_{y_0} - \lfloor r \sin\theta \rfloor)$$

The implementation is equally straightforward since we can access the pixel values for all other pixels in the same thread, so that we can compute the resultant image matrix in a single pass.

Optimisation and Features

One obvious optimisation would be to eliminate the use of the square root for distance comparison, since it is computationally expensive. We can optimise the $withinCircle$ function so that it does not use a square root, and still remain mathematically sound by squaring both side of the inequality:

$$withinCircle(C, P) = \left|P_x - C_{x_0}\right|^2 + \left|P_y - C_{y_0}\right|^2 < {C_r}^2$$

We can measure if there is any CPU or GPU speedup when doing this optimisation, by time the time taken by the kernel function alone:

	CPU		GPU	
	Original	**Improved**	**Original**	**Improved**
AVG. FRAMERATE	5.5 fps	6 fps	53 fps	53 fps
RUNTIME (MS)	141.0545	131.8650	0.0290	0.0265
SPEEDUP	**1.1858**		**NEGLIGIBLE**	

Figure 24: Speedup from optimisations in comparing Pythagorean distances

It seems that there is a minute speedup on the CPU, which arises from eliminating 1,440,000 unnecessary square root operations. Meanwhile, since the only computation that is saved is a single square root operation on each GPU thread, we would not expect any large speedup on the GPU.

Analysis

We will look at the time taken for various filters using both CPU and GPU. Since I have implemented the filters in a pipeline, the time taken for the rest of the pipeline should remain constant while the filter in the middle changes, and hence is not plotted below in Figure 25.

	CPU	GPU	SPEEDUP
GAUSSIAN BLUR	131.474	0.178	738.618
EDGE DETECTION	67.115	0.114	591.322
THRESHOLDING (MODE 1)	39.175	0.040	979.375
THRESHOLDING (MODE 3)	54.190	0.045	1204.222
EMBOSSED FILTER	69.360	0.119	582.857
LIGHT TUNNEL EFFECT	115.525	0.026	4359.434

Figure 25: Breakdown of time taken for different filters to run on both CPU and GPU, as well as the Amdahl speedup

As shown in the table, the speedup for each filter is consistently very high, where all processing in the GPU consistently taking less than 0.2ms, while the CPU takes anywhere from 40ms to 130ms, depending on the filter used. Since there is no data transfer that is occurring in the GPU kernel function, all time spent should be completely spent working on each pixel in parallel.

Meanwhile, on the CPU, since processing is done in serial, many things may affect the runtime of the filters. Firstly, the number of floating-point operations would directly affect the runtime, which is heavy within the embossed filter.

Secondly, the more significant reason for the disparity between the runtimes of the filters on CPU would be memory effects. For the Gaussian blur, edge detection, embossed filter and light tunnel effect, these kernel functions depend on other values in the array to determine the value for a specific pixel. Should there be a cache miss within the CPU cache, more time would be lost. Since the Gaussian blur kernel is larger than the edge detection kernel (5x5 vs 3x3), we can expect that there might be more cache misses in the Gaussian blur filter, as exemplified above. The runtime for the light tunnel effect is equally as high as the Gaussian blur filter, also because rendering each pixel requires arbitrary access within the array, and exhibits much lower spatial and temporal locality as compared to convolution operations, which operate on neighbouring elements.

Explanation of QR Code Recognition

The recognition of QR codes used in this application is a simplified and fairly accurate implementation, which was chosen so that it could be applied on a parallel computing environment fairly well.

The basis for the recognition of QR codes is on two parts. Firstly, the QR code is a square matrix made up of binary blocks, usually black and white, which helps with the detection of image features. Secondly, the QR codes have 3 **Finder Patterns (FPs)**, located at the top left, top right and bottom left of the code, which is made up of a series of alternating black and white blocks, conforming to the ratio of $1:1:3:1:1$ as follows:

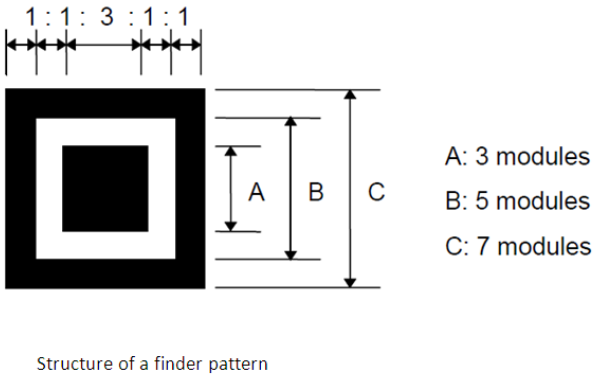

Figure 26: Structure of a QR code finder pattern (FP). Source: KeepAutomation[7]

This ratio holds true regardless if the FP is rotated (as a result of rotated image captures), which allows the FP to be a good gauge to determine if an image contains a QR code.

The following things needed to be done for this implementation. Firstly, we need to draw a bounding box around the detected QR code, in the same orientation as the QR code in the image capture. Secondly, we need to identify which FPs belong to which corner of the QR code. As shown in Figure 27 below, the top-left FP is marked as the red square in the corner, the top-right being green, bottom-left being cyan and bottom-right being magenta. This should (preferably) hold true regardless of the rotation of the QR code in the image capture. Lastly, we should also be able to extract the blocks in the QR code into a separate canvas, as shown below as well.

Figure 27: Left: Bounding box drawn around QR code, FPs identified and orientation detected. Right: Extraction of QR code blocks from the detected QR code.

Various assumptions are held in order to simplify the implementation. Firstly, we assume that QR codes are always in black on a white background, which is the most widely displayed form of QR codes. Secondly, we assume that the QR code is not too far away from the camera, which means that the QR code should not be too small in the image capture. Thirdly, we assume that the QR code is not too warped in the image capture (due to the angle at which the code was captured), because we will be using an affine transformation (rather than a perspective warp) for simplicity. Lastly, for the QR code extraction feature, we assume that the QR code is 29x29, in order to simplify the implementation.

[7] http://www.keepautomation.com/tips/qr_code/functions_of_qr_code_function_patterns.html

With these assumptions in place, we can produce a simplified implementation of a QR code recognition application, as outlined in the following high-level pipeline diagram:

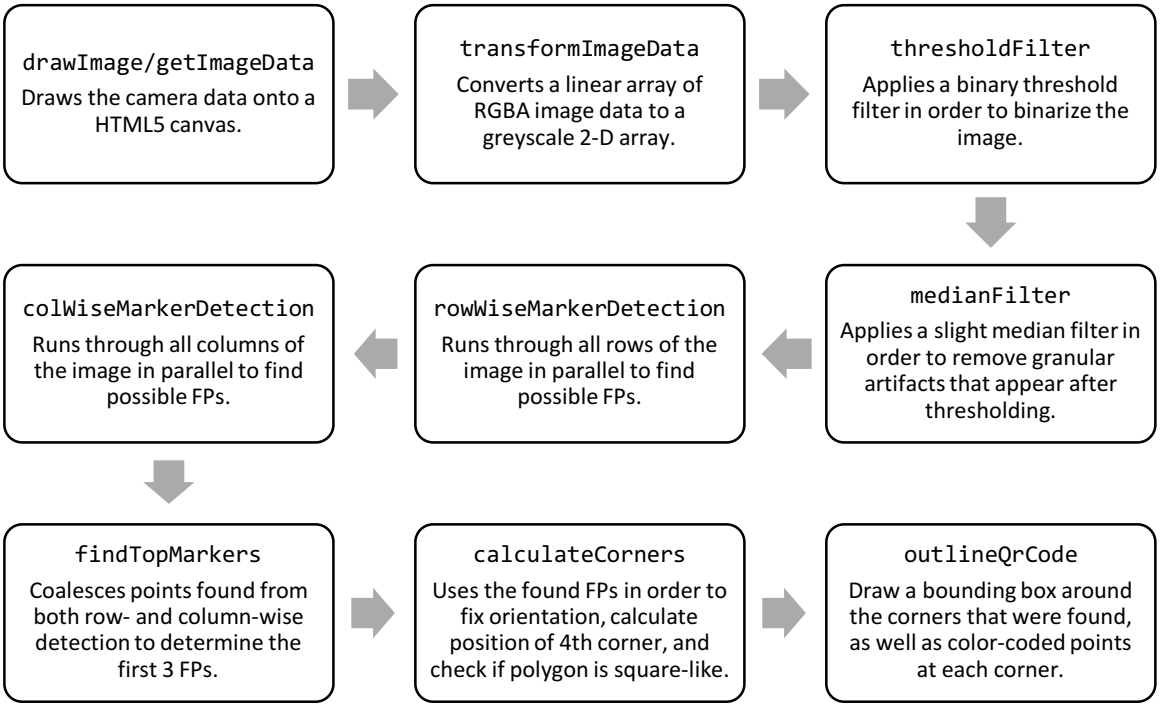

Figure 28: High-level overview of the pipeline of tasks for QR code detection

Since many of the above tasks have either been elaborated on elsewhere in this paper, or are trivial enough, only some selected pipeline tasks will be explained in the sections below.

Row/Column-wise FP Detection

As mentioned previously, all FPs have alternating block sequences conforming to a $1:1:3:1:1$ ratio, regardless of orientation. With this in mind, we can loop through all rows as well as columns to find if there are pixel sequences which follow such a ratio, subject to a variance of at most 50%.

The algorithm searches a row or column in a linear fashion, counting the number of black and white pixels that it detects, as well as when a pixel changes from white to black (or vice versa). Once it finds a chain of 5 alternating pixel sequences, it checks if this sequence follows the given ratio; if it does not, it continues searching down the row or column.

Since the search requires to search the entire row or column, the implementation of this algorithm cannot be done efficiently using a one-pixel-per-thread approach, since each row would have at most 2 possible FPs. On the other hand, running the algorithm completely within a single thread, or on the CPU, is also impractical due to the size of the array.

Instead, I opted to parallelize the algorithm by rows/columns, since each search can run independently of other rows/columns. This way, for a $w \times h$ image matrix, we would only need to run $w + h$ threads for a single frame, instead of $w \times h$ which is considerably larger. A third function, `findTopMarkers` would have to be used in order to coalesce the results from the row-wise and column-wise searches into a single list of points in 2D space.

Coalescing of Possible FPs

Once the possible points of FPs are found by searching the rows and columns, we will have to tally the two resultant arrays together. Since a point that was found by searching row-wise could be a coincidence (perhaps within the error margin introduced with the 50% variance), we need to make sure that a x-value reported as a point in the row-wise result at column y, has a corresponding point in the column-wise result at row x, and that point should be equal to y.

This algorithm would then be able to give us the origin of the FP, since only the middle point would satisfy both conditions as outlined above. This is illustrated in Figure 29 below, where the green circle represents one of the row-wise results, the red circle represents one of the column-wise results, while only the orange circle satisfies both results simultaneously:

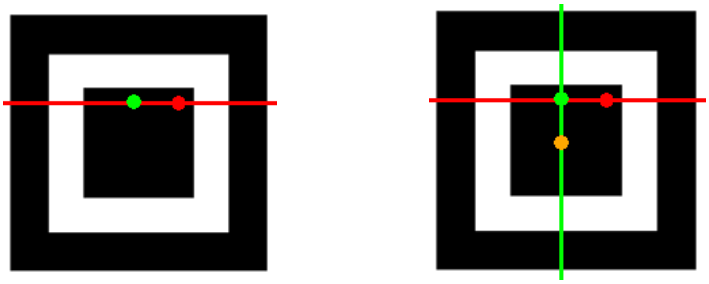

Figure 29: Illustration of coalescing of row- and column-wise points to give us the orange circle. Source: AI Shack [8]

The naïve implementation of the algorithm would run through a $w \times h$ matrix, checking both arrays if the point is satisfied. Instead, I opted to only run through the rows of the matrix (i.e. size h) and retrieving the corresponding column as needed, since the arrays are indexed by row/column order.

Additionally, since we only want to find 3 FPs, I opted to run only 3 GPU threads for this computation, rather than a $w \times h$ matrix of Booleans that correspond to whether it was a FP. This requires us to iterate through all rows within each thread, and only returning once it had found enough FPs. Since there is no way for threads in GPU.js to communicate with one another, I had to settle for this option.

Calculating Corners

Once we have found the FPs from the previous step, we can start to compute the corners of the QR code. This part involves 3 steps: fixing orientation of FPs, extrapolating the position of the 4th FP, followed by running sanity checks on the resultant quadrilateral to check if it satisfies certain conditions, in order to reduce false positives.

In order to re-orient the markers, we have to position them at the correct corners. In order to reference the corners by name, we label the top-left as 1, top-right as 2, bottom-left as 3 and bottom-right as 4. These labels will correspond to the FPs, as well as the colours of the corners when the bounding box is drawn.

[8] http://aishack.in/tutorials/scanning-qr-codes-verify-finder/

We then observe that since the FPs should ideally form somewhat of a right-angled triangle, we know that the sides of the triangle are both shorter than the hypotenuse. Since the previous steps tells us nothing about the orientation of the FP, we have to calculate the Pythagorean distance between each of the 3 FPs, and swapping the FPs until the distance between FP2 and FP3 is larger than other pairs of FPs. As shown in Figure 30 below, the green line should be the longest as compared to each of the red lines.

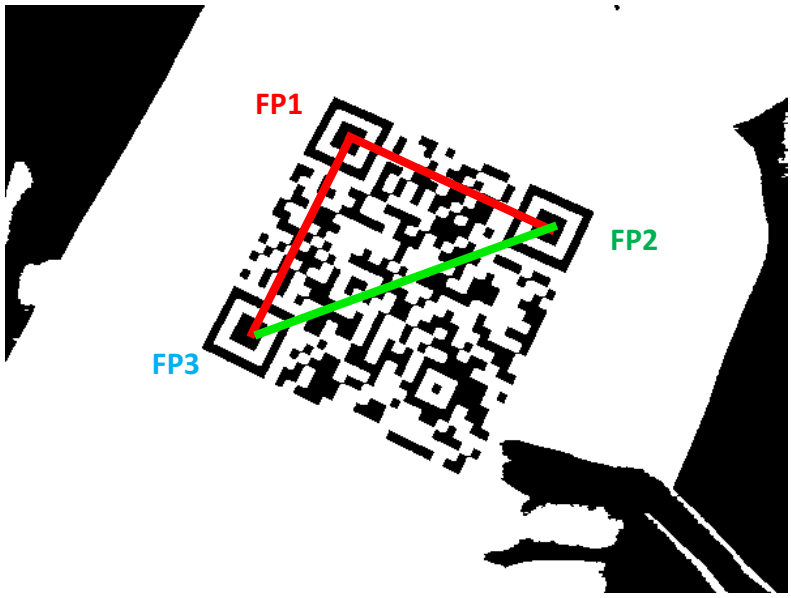

Figure 30: Illustration of re-orientation of FPs using Pythagorean distances

Next, we can estimate the location of the 4th corner, by assuming that both sides of the quadrilateral are parallel. We can estimate the position of the 4th corner as follows, which is also illustrated in below:

$$x_4 = x_3 + (x_2 - x_1)$$
$$y_4 = y_2 + (y_3 - y_1)$$

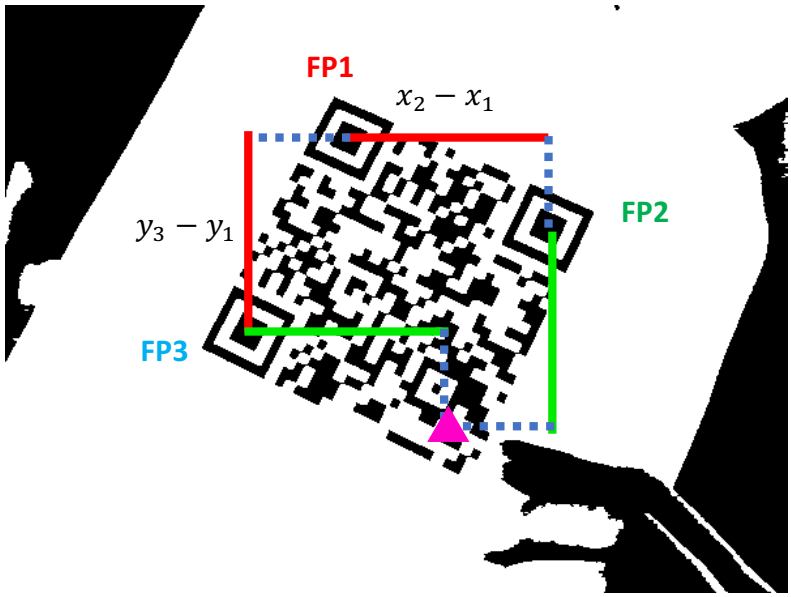

Figure 31: Illustration of estimation of the 4th corner, which is drawn in magenta

Lastly, we will do some checks to reduce the chances of false positives. These checks include ensuring that all FPs are within the canvas, no two points are too close together, that the area of the quadrilateral is at least a fixed size, and that both parallel sides of the quadrilateral have similar lengths (prevent rectangles).

The implementation of this entire procedure is completed within 4x2 threads, since there are only 4 corners to find, and 1 thread each for the x and y-coordinates each.

Extraction of QR Code Blocks

In this implementation, we hold the assumption that the QR code is always a 29x29 one, although it should be trivial to extend it to other sizes with enough effort. Hence, the basis of the extraction would be to take the area of the image matrix bounded by the 4 corners found earlier, partition it into 29x29 partitions, and perform an affine transformation of the parallelogram back to a square canvas.

The technical details of the implementation of the transformation will not be discussed. Instead, because we need to render the extracted QR code on a separate canvas from the main canvas, we have to sacrifice some of the runtime to transfer the image texture to CPU, before passing it back to the GPU for rendering. Otherwise, only 1 image would be rendered due to limitations with GPU.js.

Hence, this feature is not switched on by default so that runtime would not be impacted as much when starting the application for the first time. Additionally, the reliability of this feature is not as good, since the corners which are calculated in the previous step are only approximations, subject to the various assumptions that we held earlier. As such, though it has been implemented, this particular feature is far from complete due to the various reasons mentioned above.

Analysis

The time taken for running the various in the pipeline on both CPU and GPU, with QR code block extraction disabled and enabled respectively, are as follows:

	CPU	GPU	Speedup
transformImageData	3.960	35.446	0.112
thresholdFilter	4.292	0.353	12.160
medianFilter	3.870	0.217	17.836
markerDetectionRowWise	6.880	0.333	20.659
markerDetectionColWise	15.603	0.410	38.056
markerDetectionTop	0.166	0.333	0.496
calculateCorners	0.072	0.439	0.163
outlineQrCode	91.215	1.396	65.317
renderLeftImageColor	5.430	0.904	6.010
TOTAL TIME	**131.487**	**39.832**	**3.301**

Figure 32: Breakdown of time taken for QR code detection to run on both CPU and GPU

	CPU	GPU	Speedup
transformImageData	4.090	5.235	0.781
thresholdFilter	3.181	0.299	10.639
medianFilter	3.594	0.207	17.365
markerDetectionRowWise	6.895	0.394	17.501
markerDetectionColWise	14.475	0.422	34.342
markerDetectionTop	0.199	0.379	0.526
calculateCornersAsArray	0.077	60.051	0.001
outlineQrCode	88.758	1.043	85.057
renderLeftImageColor	5.250	0.873	6.010
convertToArray	2.392	48.276	0.050
affineTransform	8.351	1.764	4.734
renderQrCode	0.619	0.850	0.729
TOTAL TIME	**137.881**	**119.793**	**1.151**

Figure 33: Breakdown of time taken for QR code detection to run on both CPU and GPU, with QR extraction enabled

As shown in the above tables, we can observe various phenomena. Firstly, we observe that `transformImageData` is significantly slower on GPU when QR code extraction is disabled, which is rather puzzling. The time taken which is significantly higher than CPU is expected, since data does need to be transferred between the CPU and GPU, which involves loading from the main memory. However, the decrease from 35.446ms to 5.235ms when QR code extraction is enabled is unexplained, which might have to do with CPU caches due to how I used GPU.js for rendering on two separate canvases.

Secondly, we can see large discontinuities between GPU and CPU in some of the methods, such as `markerDetectionColWise`. Because `markerDetectionRowWise` and `markerDetectionColWise` effectively operate on the same number of loop iterations – the former has h threads doing w loop iterations, while the latter has w threads doing h loop iterations – we would expect that the time taken would be the same. However, on the CPU the time taken to iterate through each column is more than double than to iterate through each row. This is because memory is being accessed in column-major order, which might not be as optimised for caching as it is to do so in row-major order.

Lastly, we can see that the two tasks in which the GPU takes significantly longer than others would be `calculateCornersAsArray` and `convertToArray`, which incurs heavy time penalties from transferring the data from the GPU back to main memory, and to the CPU again. This is largely unavoidable due to technical limitations with GPU.js as explained in the previous sections. Hence, the reason to use two separate kernels, `calculateCorners` vs `calculateCornersAsArray` is justified here, since the former can make good use of pipelining within the GPU.

Machine-specific Caveats

Since this application depends on many different hardware, it is imperative for each of them to be sufficiently performant in order to get good performance from the application. For example, although the application is meant to be run on the GPU, the GPU.js library falls back to

CPU mode if the target browser does not support GPU hardware acceleration. This would mean that the runtime and user experience of the user running the application is largely dependent on the device that he or she is using. Additionally, we do not expect that the application will run on devices such as mobiles and tablets, which do not support the `getUserMedia` API, nor do they have discrete GPUs as well.

First of all, it should be obvious that the hardware specifications of the system would directly impact the performance of the application. Having discussed various topics such as memory effects in the CPU cache, these factors would directly impact the runtime of the application and hence the results obtained above.

The performance of the application is also dependent on the available system resources, such as the number of processes, CPU utilization of the system and memory available, as we can expect that there would be more context switching when there are more processes, more aggressive scheduling policies when CPU utilization is higher, and even virtual paging that may occur if system memory is low. These factors may affect the results and performance of the user who is using the application.

Appendix

Figure 34: Program execution time (minimum of 5 runs) against number of threads on the Tembusu node

Processors	128	256	512	1024	2048
1	0.0220	0.1472	1.3305	12.8430	111.6758
2	0.0111	0.0925	0.6686	6.5394	61.9725
3	0.0074	0.0527	0.4630	4.5330	48.1927
4	0.0056	0.0463	0.3519	3.3889	36.6455
5	0.0045	0.0372	0.2909	2.7905	29.3754
6	0.0038	0.0315	0.2435	2.3510	24.8730
7	0.0033	0.0271	0.2088	2.1103	21.0838
8	0.0028	0.0236	0.1815	1.7829	18.2977
9	0.0026	0.0214	0.1617	1.6476	14.9092
10	0.0023	0.0193	0.1475	1.4174	14.5814
11	0.0021	0.0177	0.1335	1.2997	13.2491
12	0.0017	0.0163	0.1222	1.2092	12.0453
13	0.0027	0.0192	0.1384	1.3471	12.3412
14	0.0027	0.0205	0.1418	1.2851	11.6051
15	0.0026	0.0237	0.1422	1.3186	11.9665
16	0.0024	0.0221	0.1348	1.2437	11.5445
17	0.0023	0.0206	0.1344	1.2670	11.5091
18	0.0023	0.0201	0.1296	1.2879	11.3148
19	0.0021	0.0167	0.1276	1.2156	11.0845
20	0.0020	0.0157	0.1317	1.2188	10.8161
21	0.0020	0.0154	0.1236	1.1300	10.8462
22	0.0018	0.0145	0.1223	1.1053	10.7283
23	0.0018	0.0143	0.1181	1.0590	10.2195
24	0.0018	0.0135	0.1130	1.0252	9.9119
25	0.0023	0.0186	0.1264	1.2897	10.4482
26	0.0023	0.0181	0.1378	1.2589	10.3044
27	0.0021	0.0176	0.1237	1.2020	10.7197
28	0.0021	0.0177	0.1287	1.1739	10.6941
29	0.0022	0.0167	0.1300	1.1891	10.7803
30	0.0022	0.0166	0.1369	1.1867	10.6150
31	0.0024	0.0162	0.1332	1.1315	10.4925
32	0.0020	0.0155	0.1270	1.1935	10.5777
33	0.0024	0.0152	0.1288	1.1017	10.5713
34	0.0022	0.0179	0.1302	0.9788	10.7514
35	0.0023	0.0176	0.1262	1.1143	10.5870
36	0.0021	0.0171	0.1284	0.9828	10.5651
37	0.0022	0.0164	0.1289	1.1144	10.7777
38	0.0021	0.0166	0.1261	1.1177	10.7208
39	0.0020	0.0163	0.1214	1.1290	10.4808
40	0.0020	0.0161	0.1222	1.1291	10.5818

Figure 35: Amdahl speedup against number of processors, for each matrix dimension on the Tembusu node

Processors	128	256	512	1024	2048
1	1.000000	1.000000	1.000000	1.000000	1.000000
2	1.981982	1.591351	1.989979	1.963942	1.802022
3	2.972973	2.793169	2.873650	2.833223	2.317276
4	3.928571	3.179266	3.780904	3.789725	3.047463
5	4.888889	3.956989	4.573737	4.602401	3.801678
6	5.789474	4.673016	5.464066	5.462782	4.489840
7	6.666667	5.431734	6.372126	6.085865	5.296759
8	7.857143	6.237288	7.330579	7.203433	6.103270
9	8.461538	6.878505	8.228200	7.794975	7.490395
10	9.565217	7.626943	9.020339	9.060957	7.658784
11	10.476190	8.316384	9.966292	9.881511	8.428935
12	12.941176	9.030675	10.887889	10.621072	9.271317
13	8.148148	7.666667	9.613439	9.533813	9.049023
14	8.148148	7.180488	9.382934	9.993775	9.622993
15	8.461538	6.210970	9.356540	9.739876	9.332370
16	9.166667	6.660633	9.870178	10.326445	9.673507
17	9.565217	7.145631	9.899554	10.136543	9.703261
18	9.565217	7.323383	10.266204	9.972048	9.869887
19	10.476190	8.814371	10.427116	10.565153	10.074952
20	11.000000	9.375796	10.102506	10.537414	10.324960
21	11.000000	9.558442	10.764563	11.365487	10.296307
22	12.222222	10.151724	10.878986	11.619470	10.409459
23	12.222222	10.293706	11.265876	12.127479	10.927717
24	12.222222	10.903704	11.774336	12.527312	11.266841
25	9.565217	7.913978	10.526108	9.958130	10.688521
26	9.565217	8.132597	9.655298	10.201763	10.837681
27	10.476190	8.363636	10.755861	10.684692	10.417810
28	10.476190	8.316384	10.337995	10.940455	10.442749
29	10.000000	8.814371	10.234615	10.800605	10.359248
30	10.000000	8.867470	9.718773	10.822449	10.520565
31	9.166667	9.086420	9.988739	11.350420	10.643393
32	11.000000	9.496774	10.476378	10.760788	10.557664
33	9.166667	9.684211	10.329969	11.657439	10.564056
34	10.000000	8.223464	10.218894	13.121169	10.387094
35	9.565217	8.363636	10.542789	11.525621	10.548390
36	10.476190	8.608187	10.362150	13.067766	10.570255
37	10.000000	8.975610	10.321955	11.524587	10.361747
38	10.476190	8.867470	10.551150	11.490561	10.416741
39	11.000000	9.030675	10.959638	11.375554	10.655274
40	11.000000	9.142857	10.887889	11.374546	10.553573

Figure 36: Gustafson speedup against number of processors on the Tembusu node, fixed time T = 100 ms

P	N	Dimension	$\mathbf{time}(N, 1)$	$\mathbf{time}(N, P)$	Speedup
1	1577237	116	0.015000	0.015000	1.000000
2	2563362	137	0.024400	0.014700	1.659864
3	4021409	159	0.037900	0.016700	2.269461
4	5047724	172	0.047600	0.015300	3.111111
5	6449870	186	0.060000	0.019800	3.030303
6	7581404	196	0.070200	0.017200	4.081395
7	8879559	207	0.082900	0.015100	5.490066
8	10888353	222	0.103200	0.018800	5.489362
9	11392673	225	0.107200	0.015200	7.052632
10	12380849	231	0.116000	0.014700	7.891156
11	14265328	243	0.135500	0.015700	8.630573
12	15029987	247	0.142500	0.080500	1.770186
13	17042145	257	0.162400	0.027100	5.992620
14	17304568	259	0.166700	0.024600	6.776423
15	18270266	263	0.178000	0.025000	7.120000
16	19858903	271	0.195900	0.030600	6.401961
17	20952778	276	0.205700	0.027100	7.590406
18	21663266	279	0.215000	0.029800	7.214765
19	22420344	282	0.222300	0.028800	7.718750
20	23349926	286	0.241800	0.029800	8.114094
21	25937297	296	0.259600	0.029600	8.770270
22	26015568	296	0.260000	0.026000	10.000000
23	26618688	299	0.262500	0.028200	9.308511
24	22832163	284	0.236500	0.024400	9.692623
25	21915368	280	0.269300	0.029600	9.097973
26	22174556	281	0.219400	0.025200	8.706349
27	20272513	273	0.246500	0.024900	9.899598
28	20356401	273	0.246600	0.027400	9.000000
29	19701529	270	0.233600	0.025500	9.160784
30	21781247	279	0.267400	0.027200	9.830882
31	21094323	276	0.253100	0.027200	9.305147
32	20730099	275	0.251500	0.026400	9.526515
33	22991004	284	0.292400	0.032700	8.941896
34	21026177	276	0.254200	0.025800	9.852713
35	23272949	286	0.298500	0.027900	10.698925
36	20337057	273	0.247200	0.023000	10.747826
37	21268879	277	0.256400	0.025900	9.899614
38	21175836	277	0.257200	0.026600	9.669173
39	23070524	285	0.292100	0.028300	10.321555
40	21150040	277	0.258000	0.026300	9.809886

For the above table, even though time was fixed at $T = 0.100\ s$ to calculate N, due to memory effects the actual running time of re-running the program with the same number of processors P at the problem size N yielded very different runtimes. Hence, the speedup was calculated based on the resultant runtimes for using a single processor as well as P processors at the new value of N.

Figure 37: Communication, computation and total execution time for MPI program (best of 3) on the Tembusu cluster

# PROCESSORS		**8**	**16**	**24**	**32**	**40**	**48**	**56**	**64**
Host Machine	Comm.	1.230	2.220	3.240	4.940	5.540	5.490	5.970	9.120
	Comp.	20.060	12.740	11.880	13.110	12.610	12.540	12.560	12.340
	Total	20.839	13.769	13.133	14.643	15.092	16.631	17.353	19.745
1x Remote	Comm.	1.520	1.920	3.280	4.600	3.620	3.860	6.990	9.240
	Comp.	20.030	12.530	11.880	12.360	12.050	11.600	12.070	12.520
	Total	21.133	13.841	13.452	15.475	14.596	15.182	18.524	19.818
2x Remote	Comm.	1.650	3.450	2.280	3.780	4.490	4.660	5.710	6.910
	Comp.	20.080	9.210	6.130	6.640	6.400	5.830	6.340	6.520
	Total	21.234	12.076	8.941	10.022	10.426	10.429	11.627	12.501
4x Remote	Comm.	1.520	3.530	2.390	4.060	6.160	5.750	7.430	8.160
	Comp.	20.120	9.100	6.010	4.550	3.540	2.930	3.470	3.350
	Total	21.564	12.304	9.117	9.369	10.127	9.499	11.021	11.668
8x Remote	Comm.	1.850	3.350	2.460	4.070	6.130	5.720	7.450	9.350
	Comp.	20.270	9.080	6.110	4.530	3.550	2.960	2.550	2.170
	Total	22.041	12.566	9.532	9.666	10.445	9.807	11.142	12.540
16x Remote	Comm.	1.480	3.520	2.370	4.200	6.120	6.160	7.430	9.350
	Comp	20.070	9.110	6.030	4.610	3.590	2.960	2.560	2.220
	Total	22.397	12.918	9.769	10.198	10.815	10.233	11.532	12.903

Figure 38: Sample QR code for use in the QR code detection application