



**SAVONIA**

THESIS - MASTER'S DEGREE PROGRAMME  
SOCIAL SERVICES, HEALTH AND SPORTS

# GENERAL PURPOSE COMPUTATIONS ON GRAPHICS PROCESSING UNIT (GPGPU) COMPUTATION ROU- TINES FOR HIGH-PERFORMANCE DATA VISUALIZA- TION IN MEDICAL AND OTHER APPLICATIONS

Author/s: Athanasios Margaritis

Field of Study Social Services, Health and Sports			
Degree Programme Master's Degree Programme in Digital Health			
Author(s) Athanasios Margaritis			
Title of Thesis GPGPU computation routines for high-performance data visualization, in medical and other applications			
Date	17.12.2019	Pages/Appendices	45
Supervisor(s) Mr. Arto Toppinen, Principal Lecturer Mr. Mikko Pääkkönen, Senior Lecturer			
Client Organization /Partners Arction Ltd. Mr. Pasi Tuomainen, CEO			
<p>Abstract:</p> <p>Data visualization is an important part of computing that grows both in complexity and computing resources demand. Numerous medical fields require the necessary resources to take advantage of the modern visualization techniques. JavaScript is the most commonly used programming language for web applications that can run in a variety of devices like IoT devices, embedded systems, mobile phones and computers so it is a perfect candidate for use in modern patient-oriented medical solutions for example in visualizing information and patient statistics on Telemedicine applications, making charts for understanding Big Data in epidemiological studies, or in real-time device assisted surgery</p> <p>Arction Oy's LightningChart JS visualization library is designed to offer an easy but robust visualization solution for JavaScript based applications. But due to the nature of JavaScript, which is inherently single core, an effort to gain multithreaded performance is made by using General Purpose Computations on Graphics Processing Unit (GPGPU) Technologies.</p> <p>Even though the graphics pipeline has been exposed to JavaScript via the WebGL API since 2011 there has yet to be an official GPGPU solution adopted by the industry. Therefore, several independent solutions have appeared. The first part of this paper focuses on choosing the best solution available right now that offers performance, ease of use, and broad platform support. After reviewing 5 candidates, GPU.js was chosen.</p> <p>Then gpu.js was used to create a GPGPU acceleration module for LightningChart JS. The Iterative Waterfall model was used throughout the development process. All the routines that were requested to be accelerated by gpu.js were written both for the Central Processing Unit (CPU) and Graphics Processing Unit (GPU) and then thoroughly benchmarked. Numerous optimization techniques were used for writing the GPU accelerated code.</p> <p>According to the results, on most routines GPU showed a significant overhead over the CPU and the GPU usually overcame the CPU only for computations larger than 1 Million points and higher, but around the 10 Million point mark most browsers run into memory allocation issues thus negating the acceleration. The platform and the code complexity highly affected the result with some browsers performing better than others. Best results were seen when drawing the results directly to the Canvas element, negating a second memory copy back to the CPU.</p> <p>While this implementation shows great potential, the fact that it is so highly platform dependent passes the burden of GPGPU acceleration choice to the end user who might not be equipped to make such a decision. Nevertheless gpu.js offers great results on server-side data preparation and manipulation.</p> <p>Future development aspects can include further optimization solutions to make the code more stable and the results more constant between platforms and making sure that the code keeps in pace with the technological advancements like taking advantage of new emerging Graphical Application Programming Interfaces (API's) and embedded solutions.</p>			
Keywords GPGPU , Accelerated Computing, Javascript , Visualization, LightningChart JS			

## CONTENTS

1	INTRODUCTION .....	5
1.1	Data Visualization in Medicine .....	6
1.2	LightningChart and LightningChart JS.....	7
1.2.1	JavaScript Environment .....	8
1.3	Performance Requirements Increase .....	9
1.3.1	The end of Moore's Law .....	9
1.3.2	Different ways to combat that. Better Faster computers and/or parallelization. ....	10
1.3.3	GPGPU Basics .....	10
1.4	GPGPU and JS .....	11
2	PARALLEL COMPUTING.....	12
2.1	Basics of Parallel Computing vs Traditional (Serial) Computing.....	12
2.2	Massively Parallel Computing .....	14
2.3	GPGPU .....	14
2.3.1	Understanding GPGPU .....	15
2.3.2	CUDA .....	15
2.3.3	OpenCL .....	16
2.3.4	DirectCompute.....	16
2.3.5	Before CUDA.....	16
2.3.6	GPGPU Benefits .....	16
2.3.7	GPGPU Issues and Downfalls .....	17
3	FINDING A GPGPU LIBRARY FOR JAVASCRIPT.....	19
3.1	WebGL and GLSL .....	19
3.2	JavaScript based GPGPU.....	20
3.3	Requirements of the Library .....	20
3.3.1	The Requirement gathering process .....	20
3.3.2	The requirements.....	21
3.4	Choosing the right candidate .....	22
4	OPTIMIZATION TECHNICUES FOR GPU.JS.....	24
4.1	Memory Optimization .....	24
4.2	Optimized Data Formats.....	24
4.3	Optimize Algorithm .....	25

4.4	Minimizing Host to Device Copies .....	25
4.5	Direct Rendering .....	27
5	MAKING THE LIBRARY .....	29
5.1	Identifying What parts of the Product will benefit from GPU Acceleration .....	29
5.1.1	Theoretical .....	31
5.1.2	Practical .....	33
5.2	Writing the code. ....	33
6	RESULTS .....	35
6.1	Benchmarks CPU vs GPU .....	35
6.1.1	Interpolators.....	35
6.1.2	2D Extrapolators .....	37
6.1.3	Heatmap .....	37
6.1.4	Financial and other Routines .....	38
6.2	Coding with gpu.js .....	39
6.3	How did our Results match our Expectations? .....	39
6.3.1	The Browser Effect.....	40
6.4	Where to from now? .....	40
7	REFERENCES .....	42
7.1	References .....	42
7.2	Figures and Tables.....	44

## INTRODUCTION

Data Visualization has always been integral part of computing. From the early punched tapes, to the first video output computers, the method for a computer to output the results has almost always been either via printed paper or screen depictions. A notable example of an early computer with video display is Whirlwind all the way back in 1951 that is depicted in Figure 1.

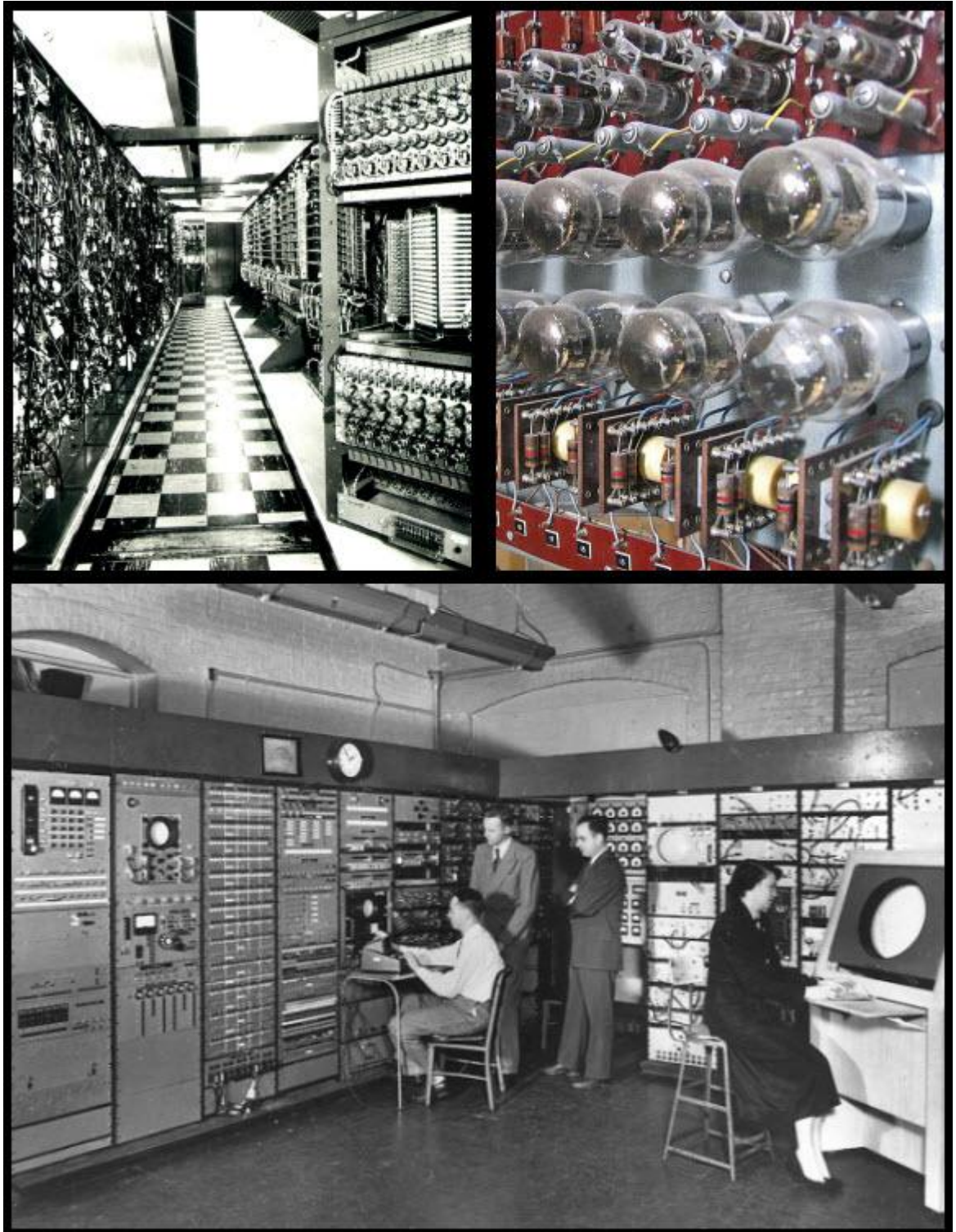


Figure 1: Whirlwind. The first computer to use video display for output 1951 (Top left and bottom images curtesy of the Computer History Museum, MountainView CA. Top right Image Dpbsmith at English Wikipedia [CC BY-SA 3.0])

As time passed, the visual capabilities of the modern PC grew in parallel with its performance. Computer output evolved from difficult to read cathode tube screens with text only graphics, to more visually oriented environments like Norton Commander, Windows and finally to the 3D era of today. As our computers become faster, and our needs bigger, the visualization techniques often follow. And this has never been truer than today, where new concepts such as Big Data Analysis, Machine Learning and Quantum Computing, require new and innovative ways to visualize data.

## 1.1 Data Visualization in Medicine

Medicine is a field that is steadily requiring more and newer ways of data visualization. Up until recently, with most Health Records being paper, and data acquisition being a hard and difficult process, data visualization was in the realm of the researcher and the scholar. And those devices that were there to help the physician had, as a rule, rudimentary and simplified graphics. Nowadays the modern hospital or doctor's office has a myriad of interconnected computing devices, from Health Record processing computers to real time visual enhancing robotic units in operating rooms. And this brand new and evolving ecosystem comes with its own needs (Steele, 2010). To quote Harold Thimbleby on his 2013 Journal publication about Technology and the future of healthcare "Infusion pumps used to be clockwork (and before that, gravity fed) and now almost everything contains a computer and has a colourful screen and lots of buttons." (Thimbleby, 2013.)

A good example of the ever growing and ever expanding need for new visualization techniques is Big Data Analysis. Big Data Analysis is essentially the process of obtaining useful information and drawing helpful conclusions from a very large dataset. It has been around for decades but has seen an unprecedented boom in the last 10 years. This is partly the result of the introduction to our lives of a host of small devices with computing and networking capabilities. This swarm of devices brought into foreplay the Internet of Things (IoT) and gave us a huge amount of data ready to be analysed. The other major factor that brought the Big Data Analysis into the medical fore is Medical Record Digitalization. Across the globe, billions of pen and paper medical records that were stored in drawers, became digitalized and easy to share. (Thimbleby, 2013.)

As a result, or in parallel with the Big Data Analysis growth, the growth of Artificial Intelligence and Machine Learning which are sciences that consume huge amount of data on their own, created paradigms that needed new visualization techniques to be implemented, giving great results in fields like epidemiologic studies, disease monitoring, early disease detection, genome studies etc. (Ristevski & Chen, 2018 ; He, Ge, & He, 2017.)

Another field where Medicine meets visualization is Real time depiction of data. From sensors in Telemedicine and IoT devices, to control of Robotic Arms and Volume rendering of 3D depictions of the body during operations, the modern physician needs to see accurate, multi-layered and latency-free information. And apart from better diagnostics and better control in the Surgery room, or the practitioner's office, the possibilities are there for remote diagnostics and even remote operations

where the patient and the health practitioner are in different parts of the world. From vital signs depiction, or Virtual reality assisted surgery, the medical technology advances offer an immense amount of possibilities, and the Data Visualization Technologies follow closely. (Kubler, Bauer, Heinze, & Raczowsky, 2002.)

## 1.2 LightningChart and LightningChart JS

LightningChart is a visualization library that offers developers the ability to add fast and easy to use visualizations in their applications. It is developed by Arction Oy, a Kuopio Based company. As a leader in performance-oriented visualizations they offer a myriad of features such as simple and 3D highly interactive charts and diagrams, mapping solutions, heatmap depictions and in the latest version of the library, Volume Rendering which can provide ultra-fast real time Volume rendering depiction (Figure 2) that can be useful to applications such as CAT and MRI scanners .

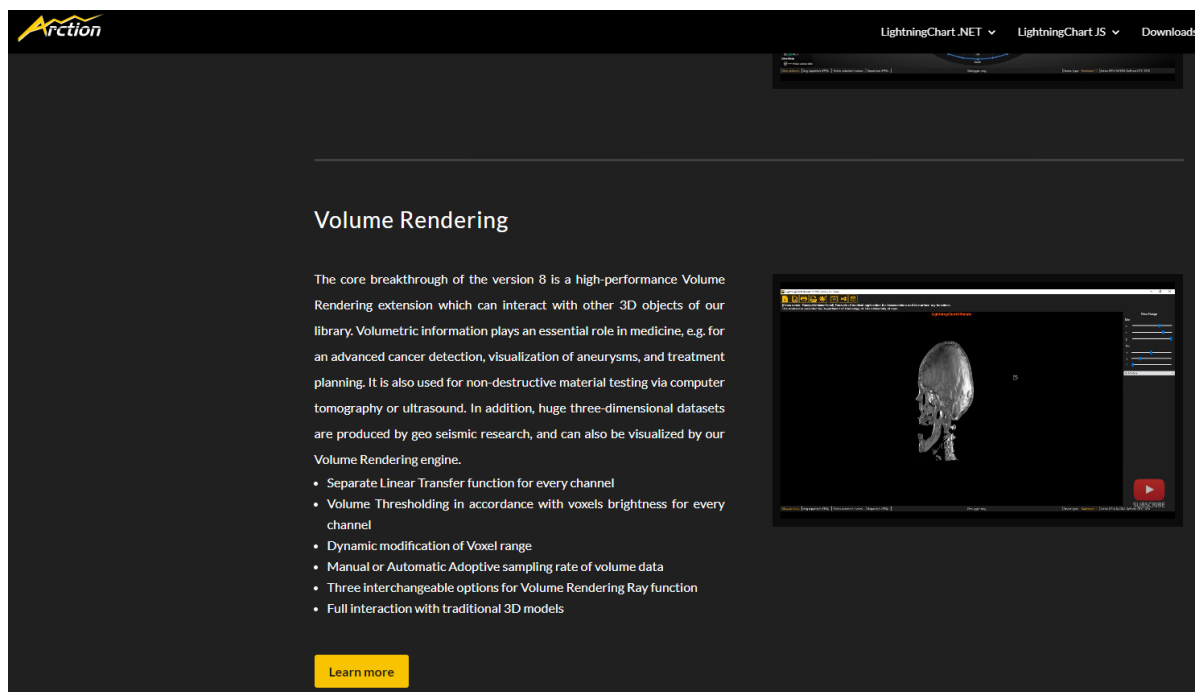



Figure 2: Arction LightningChart Volume Rendering feature (Image Courtesy of Arction Oy)

Arction's latest product is called LightningChart JS (Figure 3) and it is an implementation of LightningChart that runs in JavaScript. Running on JavaScript can offer the library the ability to run on a much larger variety of devices and platforms such as IoT and embedded devices, phones, and even computers that do not have the ability to install software and need a client-server web application. As with all other Arction products, the library is available for free for students.



**LightningChart JS Trading showcase application**

Trading chart with live data fetched from worldtradingdata.com. Features:

- OHLC
- Simple Moving Average
- Exponential Moving Average
- Bollinger Bands
- Volume
- Relative Strength Index

[Github example](#)

---

**About Lightningchart JS**

LightningChart JS is a WebGL based, cross-platform charting library that has been developed for delivering an outstanding performance of your charts which ensures high refresh rates and smooth animations of your application. It features accessible API documentation, extensive functionality and an innovative dashboard control. Optimal for engineering, medicine, industrial process control and trading.

Figure 3: LightningChart JS Promotional Webpage (Image Courtesy of Arction Oy)

Since it is based written for JavaScript, LightningChart JS can be implemented in any device that has network app functionality. This includes IoT devices like activity trackers (or their control software if they lack a screen), mobile phones and tablets, embedded devices like heartrate monitors or signal analysers, and even light computer terminals with low processing power, as can be found in ambulances, mobile hospitals or anywhere there is a need for a web-based decentralized application.

### 1.2.1 JavaScript Environment

JavaScript is a high-level scripting language designed to offer dynamic content on web-browser environments. It was first released with Netscape Navigator, a pre-cursor of today's Mozilla Firefox, in 1995 (Brown, 2016, pp. 2-5) and has since become one of the core technologies in World Wide Web. (Flanagan, 2011.)

Because of its appeal to the public, and the explosion of Internet capable Smartphones, Web Applications and hence JavaScript reached a far larger audience. Adding to this boom was the advent of Internet of Things devices, that added another big amount of internet capable devices on the hand of users around the globe.

And since the user pool grew larger, so did the number of developers using JavaScript. So much so that technologies like Node.JS emerged, that enabled the use of JavaScript scripting for server-side applications, when by then JavaScript was purely client-side (node.js foundation, 2019).



One inherent issue about JavaScript is that it is bound to the Browser as an execution platform and therefore has no direct access to the computer resources. This characteristic of JavaScript has two main effects. It means that execution times, compatibility, robustness and overall usability is highly dependent on each Browser or independent JS platform implementation, and that since it runs in a browser sandbox (somewhat similar to a Java VM), it has limited resources.

Up until recently this sandbox approach meant that JavaScript applications had no direct access to the devices Graphics Processor. All graphical computations had to be executed on the Central Processing Unit (CPU), so any Graphic Intensive JavaScript applications were chunky and slow. This changed with the introduction of WebGL in 2011. WebGL allowed JavaScript applications direct access to the Graphics Processing Unit (GPU) resources and take advantage of hardware acceleration (Ghayour & Cantor, 2018.)

### 1.3 Performance Requirements Increase

With the advent of IoT devices and the surge of Big Data, the computational needs of data visualization grows exponentially. Combined with visual complexity of modern visualization techniques the need for better performance is more apparent than ever (Statista Research Department, 2016).

Especially on complicated visualizations that require real-time accurate data depiction, the requirement for more performance is more evident than ever. This is also exaggerated by the fact that JavaScript runs in a variety of devices, including low power ARM-Based IoT devices, remote devices, embedded devices, phones and tablets (Flanagan, 2011).

#### 1.3.1 The end of Moore's Law

"The number of transistors in an Integrated Chip doubles approximately every 28 months"  
Gordon Moore

As we slowly reach the physical hard limit of transistor size, we reach the limit of the available frequency performance we can get out of the silicone. In order to keep up with Moore's Law, the industry has compensated to the lack of higher frequencies with more cores per CPU (Figure 4).

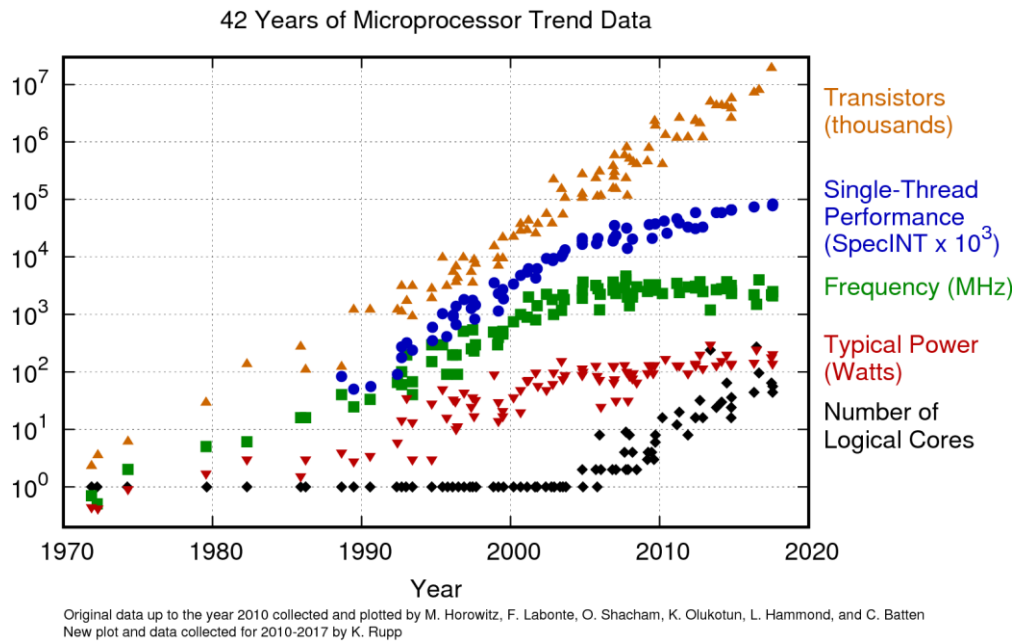


Figure 4: 42 Years of Microprocessor Trend Data (Rupp, 2017)

Since the JavaScript Engine is single-threaded, this means that the latest increase in performance does not in fact affect the performance of JavaScript execution.

The only way to get JavaScript to take advantage of the current generations CPU performance increase is to make it multithreaded.

### 1.3.2 Different ways to combat that. Better Faster computers and/or parallelization.

There have been a lot of attempts to add multithreading computation on JavaScript. HTML5 introduced "workers" which allowed the main JavaScript application to call on smaller executables called "workers" which have the ability to run on a different thread. But they're only suited for background work since they cannot directly access the Document Object Model (DOM) of a webpage and are limited to the number of cores or threads on the CPU (Mozilla Developer Network, 2019).

Another Solution is to take advantage of the WebGL API and the exposed resources and use it for a GPGPU implementation for JavaScript.

### 1.3.3 GPGPU Basics

The idea of GPGPU, or General-Purpose Computing on Graphics Processing Unit is simple in its inception. Utilize a GPU, which is essentially a multi-core processing unit designed to do a lot of parallel floating-point calculations (The nature of Graphical Computations), to do computations other than graphics. This essentially gives the user the ability to use more than 1000-core computing devices, if the computational needs are similar to graphic computations. This has the potential to give Mainframe-grade computational abilities to consumer-grade computers. It has been so successful that it has been implemented in Datacentres creating cost effective supercomputer arrays around the world (Figure 5).

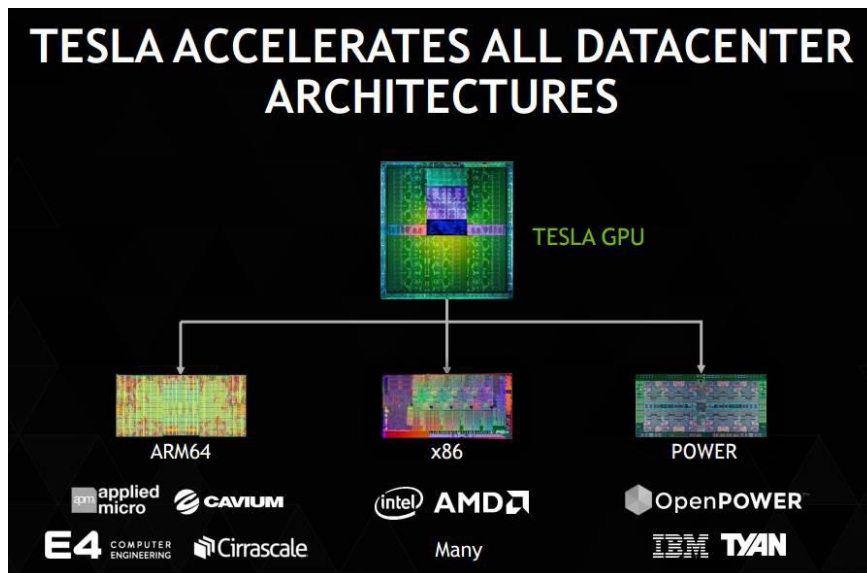


Figure 5: GPGPU Architecture in Data Centres (source: Nvidia used under Fair usage)

#### 1.4 GPGPU and JS

While JavaScript has implemented workers, it is still at its core a single threaded process. Implementing GPGPU solutions on JavaScript can offer the ability to have multithreaded computations on web applications. WebGL on its own opens up the JavaScript engine to have access to GPU accelerations, but it does not provide an inherent way to use the GPU as a computing device. It only allows graphical computations. On that end, we need a JavaScript library to translate the JavaScript code in the page, into graphical computations so they can be run on the GPU and take advantage of the Massively Parallel abilities of a modern graphics subsystem. The next chapters delve more into detail on Parallel Computing and its possible implementations on JavaScript.

## PARALLEL COMPUTING

### 2.1 Basics of Parallel Computing vs Traditional (Serial) Computing

In traditional (serial) computing, every command is executed one after the other in the computational pipeline (GeeksforGeeks, n.d.). All computational problems are divided in instructions, and each one of those instructions is executed once the previous is executed.

On Parallel Computing on the other hand, the problems are divided into smaller problems, and those problems are computed in parallel instead of serially. This can provide a big advantage in performance since the more resources are used, the less is the execution time.

One of the earliest examples of parallel execution in consumer hardware was Intel's Hyperthreading (Figure 6) which was implemented on the Intel Pentium 4 CPUs (Xbit Labs, 2002) and onwards and allowed a compatible OS to see one physical core as 2 logical cores, both of which share a common set of resources for the execution stage but have a duplicated architectural state and employing superscalar architecture (Johnson, 1991). This paved the way for Consumer Devices Multithreaded and then Multicore CPUs and started the trend of multithreaded aware OS's for consumers. The first OS to support it was Windows XP that were based on the NT kernel of Windows 2000.

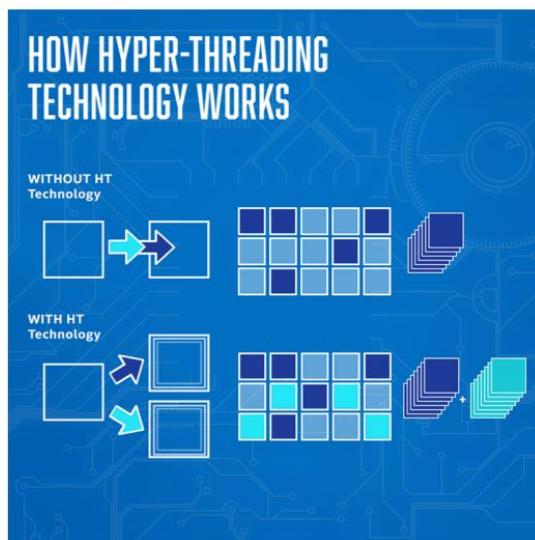


Figure 6: Intel Hyper-Threading Technology (Source: Intel, used under fair usage)

A big concern in Parallel computer is the fact that the problem that needs to be computed ought to be compatible for parallel computation. That means that the data that has to be computed should be able to be accessed at the same time, and that the computations for it do not require the results of the previous computations to finish. The more the computations need previous computations to finish, the less the performance benefit. (Grama, Gupta, Karypis, & Kumar, 2003.)

To put things into perspective, let's consider this simple example. We have a CPU that does 1 addition per clock, and 3 similar CPU's in parallel. An example like this  $(a+b) + (b+c) + (c+d)$ .

A single threaded CPU would have to do these calculations

1<sup>st</sup> tick:  $a+b$

2<sup>nd</sup> tick:  $b+c$

3<sup>rd</sup> tick:  $c+d$

4<sup>th</sup> tick: (result of 1<sup>st</sup> tick) + (result of 2<sup>nd</sup> tick)

5<sup>th</sup> tick: (result of 4<sup>th</sup> tick) + (result of 3<sup>rd</sup> tick)

So, the whole calculation will need 5 cycles to be completed.

On a Parallelized CPU the calculation would be as follows

1<sup>st</sup> tick:  $a+b$  on the first CPU,  $b+c$  on the second CPU,  $c+d$  on the third CPU

2<sup>nd</sup> tick:  $(a+b)+(b+c)$

3<sup>rd</sup> tick: (Result of 2<sup>nd</sup> tick)+ $(c+d)$

As we can see, there is an obvious performance increase in Parallel computing, that in turn increases as the number of parallel processes becomes higher, and the data calculations are not dependent with each other results. And even in this example, there is still a big part that requires serial calculations. An even more prominent example, and the one that is being used more often to test parallel execution performance is the example of Matrix Multiplication.

Imagine two equally sized matrices, with a height of  $H$  and a width of  $W$ . A serial CPU would take  $H*W$  clock cycles to go through the whole data set, but a Parallel Processing unit with  $X$  computational cores would be able to do in  $H*W/X$  clock cycles (Figure 7).

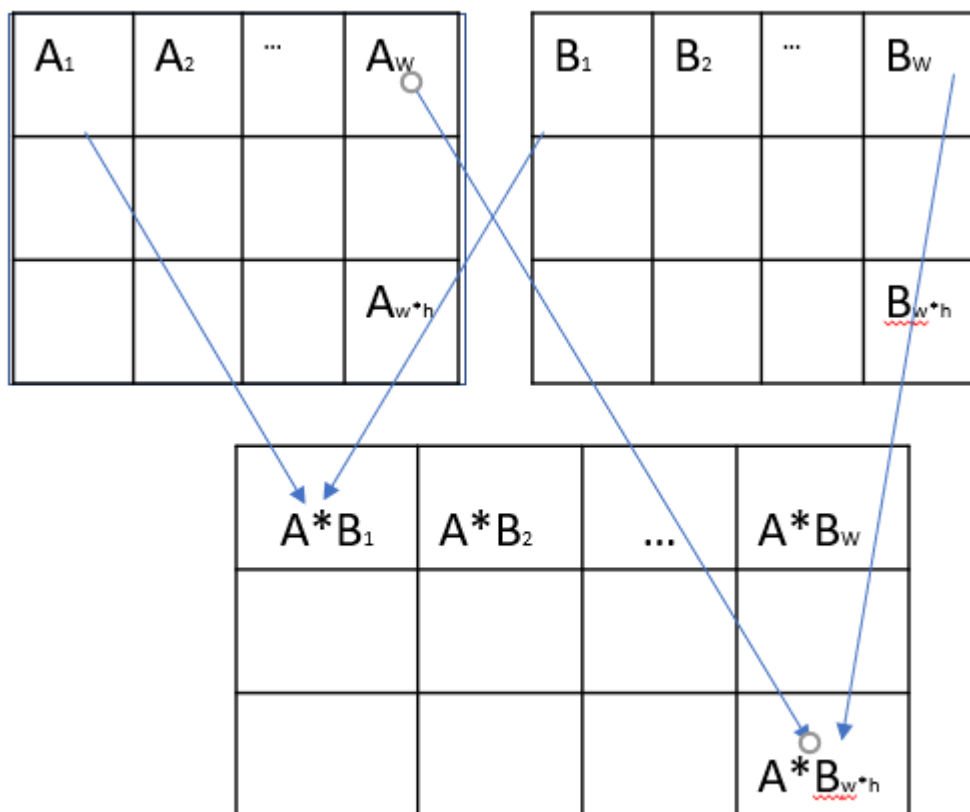


Figure 7: Matrix Multiplication Example

This is in fact a best-case scenario, where all our data is independent of each other and the parallelization is absolute. Thankfully though, this scenario is actually very common in a lot of image manipulation algorithms where every pixel is recalculated independently, similar to the rendering process of the graphical pipeline.

## 2.2 Massively Parallel Computing

Massively Parallel Computing is just Parallel computing in a much larger scale, usually used in research. It can be achieved in a variety of different ways and technologies, depending on the nature of the datasets, the equipment used etc.

One approach to Massively Parallel Computing is Grid Computing, where the processing power comes from a big number of heterogenous computer systems, that opportunistically join the group whenever they are available. Those systems usually get a fraction of the dataset, run the required calculation and return the result to a controller. (E-sciencecity, 2013.)

Clustering Computing is another approach where usually homogenous computer systems are closely connected together and share resources to achieve better performance.

This in time got scaled down from a cluster of numerous computers, to contained integrated circuits that utilize a large number of processing units and shared memory, called Massively Parallel Processor Arrays or MPPAs. A modern Graphics Card can be considered in a way a specialized MPPA focused on graphical calculations, but they differ on the fact that GPUs are Single Instruction Multiple Data (SIMD) devices. (Butts, 2008.)

One difference between Massively Parallel Computing and Symmetric Multiprocessing (SMP) which is used in normal consumer CPUs, is that MPP systems do not necessarily share resources such as main memory and computing architecture, including Operating System. (Butts, 2008)

Applying this method to a large database, gave birth to the term Massively Parallel Processing Databases. (Riso, 2018), and it can also be applied to Big Data Analysis, and Deep Learning/Machine Learning (Pivotal Data Science Labs, 2014).

## 2.3 GPGPU

General Purpose Computations over GPU (GPGPU) is a technology that allows the use of Massively Parallelism on a consumer grade computer. As the name implies, this technology utilizes the computers Graphics Processing Unit (GPU) for computations that would have otherwise been handled by the CPU.

### 2.3.1 Understanding GPGPU

As was already mentioned a GPU is essentially a Computational Unit specialized massively parallel floating-point calculations that is required by the graphics pipeline. That means that a GPU is optimized to handle operations on Matrices and Vectors. Early GPGPU solutions required that all programs had to be transposed into graphical primitives supported by the 2 main graphical APIs at the time, DirectX and OpenGL. Soon after that, Nvidia's CUDA implementation gave programmers the freedom to directly access the GPU resources and use them as they see fit, instead of having to translate everything, and gave the GPU core computational hardware a slightly less rigid structure to allow for a larger set of instructions to be run to it. (Tarditi, Puri, & Oglesby, 2006; Che, et al., 2008.) The program that runs in parallel for every data element is called a Kernel (Houston, 2007).

Microsoft DirectCompute API that was launched together with DirectX 11, offered a vendor-independent solution for programmers to utilize the potential of GPGPU, and even allowed applications to use and scale between different vendors and GPU technologies with Direct X 12. This allowed for example game developers to utilize the Primary GPU for graphical calculations, and a secondary integrated GPU that would otherwise have been unused, to provide Physics calculations for more realistic gaming.

In a very simplified Graphics pipeline, to form a picture we would need textures, which are images that comprise our scene. Textures are essentially 2 dimensional arrays with pixel values. To render the scene, for every pixel we would have to run a small program called a shader, which calculates the final colour values of every pixel according to what goes on the screen. For example, if a pixel is covered by a shadow it will be darker, or brighter if it's under direct light. Essentially a shader is a calculation that is run on every single pixel.

If we take a texture and replace the image data on it, with any set of 2-dimensional array data, and we replace the shader program with our own calculations, we eventually have our very own massively parallel calculation, one simple calculation running for every element in our data grid.

That is exactly what early solutions of GPGPU did before CUDA and OpenCL arrived.

### 2.3.2 CUDA

CUDA, which stands for Compute Unified Device Architecture, is a proprietary programming API from Nvidia that added GPGPU functionality in their cards. Released in 2007, it was the first Major API from a GPU Manufacturer that enabled programmers without advanced programming skills to be able to write code that runs in the GPGPU. While it only runs on Nvidia GPU's it's considered a pioneer in the sense that it brought GPGPU computations in the mainstream market and sparked the GPGPU explosion. After CUDA's success, a lot of alternatives started to appear in the Market. (Houston, 2007; Margaritis & Kaporis, 2014)

### 2.3.3 OpenCL

Open Computing Language (OpenCL) is a programming framework, released in 2009, 2 years after CUDA that allowed programmers to write code that can run across a multitude of heterogeneous platforms like CPUs, GPUs, DSPs, FPGAs etc. It is maintained by the Khronos Group who also maintains OpenGL. It has a broader device scope than Cuda, and its programming APIs are based on C and C++. It has support from a lot of manufacturers including AMD, Apple (even though they have now moved to Metal), ARM, Intel, Nvidia and others. (Khronos Group, 2011.)

### 2.3.4 DirectCompute

DirectCompute is Microsoft's attempt at GPGPU APIs. It was announced with DirectX 11 in 2008 and released in 2009 but offered some compatibility with DirectX 10. It is offering a lot of similar features with its competitors, and it is mainly used for Physics applications in gaming to provide realistic environment interactions. (Nvidia, 2019.)

### 2.3.5 Before CUDA

Before CUDA exposed the GPU pipeline to a broader spectrum of calculations, the only way to achieve GPGPU acceleration was with directly translating the data into graphical primitives and trick the GPU into thinking it is doing graphical computations. There were some languages and APIs that made this very difficult task a little easier and heavily influenced the future GPGPU implementations. Some of those were Brook, with its BrookGPU implementation (Stanford University, n.d.), and sh/Rapindmind who was then acquired by Intel (Intel, 2010).

### 2.3.6 GPGPU Benefits

GPGPU Solutions offer a huge performance increase in situations where Parallel Programming is applicable.

- **Low cost compared to MPP solutions**

A GPGPU solution is highly cost effective. Most modern computers and portable devices already have some sort of GPU. And even in the case where a separate GPU needs to be purchased, the necessary investment is much lower than a dedicated Massively Parallel mainframe CPU. In fact, it's so cost effective that, as was mentioned earlier, a lot of Datacentres start to implement GPGPU solutions.

- **Easily available**

In the past Massively Parallel programming was in the realm of expensive mainframes and datacentres, with difficult to maintain and buy equipment. GPUs on the other are massively produced and easily available. This means that GPGPU solutions are available "off the shelf" on almost all computer components stores, and at very affordable prices.



- **GPUs are present in a multitude of devices**

Apart from being in almost every computer, GPU's also exist in a huge variety of other devices. Mobile phones, certain IoT and Embedded Systems, set-top boxes, car computers etc. They all have some kind of mobile CPU and mobile GPU.

- **Possibility of very large performance increases**

Depending on the dataset and on the complexity of the calculations, the performance increases could be very substantial, sometimes tens or even thousands of orders of magnitude.

### 2.3.7 GPGPU Issues and Downfalls

Apart from benefits though, GPGPU has some downfalls, especially since it shares the downfalls of every Massively Parallel implementation.

- **Only shows performance on certain datasets and computations**

Just like all MPP solutions, GPGPU is highly dependent on the amount of Parallelization of the data that needs to be calculated, and of the complexity of the computations. Especially in the case of JavaScript GPGPU solutions, Kernels with a high amount of branched statements like IF or CASE take a big hit in performance. Since it is a SIMD architecture (Single Instruction Multiple Data) it means that the calculations run on the dataset are uniform throughout the set. In simple terms, the same exact program, the kernel, runs for each and every one piece of data on our dataset independently, without having access to the results of the calculations than run in parallel. This requirement reduces the cases where GPGPU is effective. More information on determining what datasets are good candidates for GPGPU acceleration are discussed in a further chapter.

- **Large computational overhead**

GPGPU is an Heterogenous System Architecture. In an Architecture like that, CPUs and GPUs reside on the same bus to reduce the latency between the devices. Still though, GPU and CPU have direct access only to their respective memories, and memory copy operations between CPU and GPU are very costly in time and all GPU computations need to be synced. Future implementations and GPGPU solutions aim into delegating these issues, and either allow the GPU direct access to the system memory and vice versa or creating a hybrid environment where System and GPU share the same memory (Ridley, 2019). Faster memory solutions and more importantly faster interconnection paths between the CPU and GPU also help reduce this overhead, though it still remains a big latency factor.

- **Data needs to be optimized for GPU computations**

GPU's are highly optimized for Floating point operations. Integer operations have to be translated into Floating Point to better make use of this optimization. This means that there is an added overhead where the compiler must wrap the data into float values. Some modern solutions have started to allow the use of integer values directly which greatly increases performance. Another issue with floating point operations is that there is the so-called floating-point precision error. Essentially when a GPGPU does high precision floating point operations, there might be some minor least significant

digit rounding errors that might end up showing up in the final calculation. These errors have very little significance in Graphical calculations, where for example a pixel colour value might have a slight shade difference small enough not be visible to the human eye, but when Integers are wrapped as floats weird behaviour might occur. For example in one of the routines written for this project, the algorithm was expecting an integer from a division between numbers, but the actual number returned was a float with a infinitesimal difference(2.00000000001 instead of 2) This was enough to cause issues to the whole program and steps where needed to be taken to make sure that all returned values where flattened to integer values again. Once more though modern compilers for the most common GPGPU solutions have built-in procedures to catch these rare occasions and repair the result on the fly.

- **GPGPU Solutions need to be explicitly provided by some sort of API, or otherwise it is very difficult to program**

GPGPU programming are based on proprietary hardware and the solutions must either be Proprietary API's (like CUDA), Industry wide open API's (like OpenCL), or they need to be transpiled to be used as actual graphical data.

In the first case, the whole solution comes from the company that usually provides the hardware itself and there is a high level of continuity and compatibility. The whole solution is optimized to work for the hardware, and the hardware is designed to offer the best results with the proprietary solution. This well built and robust ecosystem though has the disadvantage of being tied up to one company and only to the supported hardware.

On the second case, the solutions are accepted Industry wide and are based on working groups with members throughout the industry working together to bring unified standards (Khronos Group, 2019). While this approach offers wide support and compatibility between different hardware manufacturers, it still lacks the optimization and fine-tuning that proprietary solutions have.

The final case, which is the case with JavaScript GPGPU, means that there is no actual effort by the hardware manufacturers to provide a GPGPU solution. This was the case in early GPGPU efforts before it became an industry practice and meant that the whole operation had to wrapped to seem like a Graphical Operation. No optimizations for code that does not fall under the graphical computation's norm are implemented and these solutions face higher memory overhead and incompatibilities.

## FINDING A GPGPU LIBRARY FOR JAVASCRIPT

In order to get more performance out of the product (LS JS) we need the ability to get multicore performance. This can be achieved with giving the JavaScript Library GPGPU abilities. For that there is a need for a Library to allow GPGPU computations via the Browser Sandbox where JavaScript Resides. That means that we essentially need a Library to transpose JavaScript code, into WebGL code, and wrap the data into WebGL compatible data, so the computation can be run in WebGL

### 3.1 WebGL and GLSL

Before 2011, all web applications had no access to the PC's Graphics Card. But in 2011 Khronos Group, published the WebGL specifications, bringing the much-needed hardware acceleration to Web Browsers. In 2014 with the advent of WebGL 2.0, Apple also offered support in its Browsers which meant that WebGL was supported on almost all available web enabled devices.

Khronos Group is an open industry Consortium focused on creating open standard and royalty free API's that provide hardware acceleration standards for 3D graphics and other media. They are behind some Industry leading standards such as OpenGL and OpenGL ES, OpenCL, Vulkan and many others. (Khronos Group, 2019.)

With the introduction of WebGL, and the adoption of the standard by most modern browsers including Internet Explorer, Chrome and Firefox, web developers had now access to 3D acceleration for their JavaScript based Web Application, since WebGL exposed the GPU Layer to a JavaScript application. By working on the HTML 5 element <canvas> the web developer could now create more vibrant, realistic and interactive applications.

Since WebGL is derived from OpenGL (specifically OpenGL ES 2.0) it uses the same core architecture characteristics and the same graphics pipeline elements. This means that it also follows the base concept of having a texture element and running a vertex shader program on it. For WebGL as with OpenGL, those programs use the OpenGL Shading Language (GLSL). (Ghayour & Cantor, 2018, pp. 97-107.)

OpenGL Shading Language (GLSL or also mentioned as ELSL) is the language used by OpenGL and subsequently WebGL for programming the shaders on the graphics pipeline. WebGL is using the OpenGL ES variable like OpenGL ES. Its syntax is similar to C so it can offer Graphic Developers a high-level programming environment instead of the far more difficult Graphics Assembly Languages like ARB Assembly used in versions predating OpenGL 1.4

## 3.2 JavaScript based GPGPU

Since WebGL exposed the GPU resources to the Web Browser and to the JavaScript API, the way to create a GPGPU solution for JavaScript became a possibility. Even though WebGL came with the promise of WebCL, an OpenCL equivalent for Web Browsers, that project has not yet come into fruition, and users must look for independent libraries that offer GPGPU functionality to JavaScript Applications, the same way that early GPGPU solutions had to translate datasets and programs to textures and shaders.

The idea of having GPGPU acceleration in JavaScript is not new. In fact, since the early stages of JavaScript a solution based on the OpenCL standard was proposed. WebCL was a proposed standard by the Khronos Group. Its first and only draft appeared in November 2014 but has never been implemented by any of the major Browsers, so it is still in obscurity.

Since there is no native way to get GPGPU acceleration on JavaScript, a number of independent efforts started making their appearance, that took inspiration from the early days of GPGPU where the only solution was to transpile and encapsulate the calculations and the data as graphical calculations and then parse the results from the results of the shading process. A large portion of this Thesis is to find the best library which does that and use it to accelerate the LightningChart JS product.

## 3.3 Requirements of the Library

### 3.3.1 The Requirement gathering process

As with any software project, our first step is to gather the requirement for the project and have a clear understanding of the needs and timeline. For the whole development process, an Iterative waterfall development method (Figure 8) was used (Geeks for Geeks, n.d.).

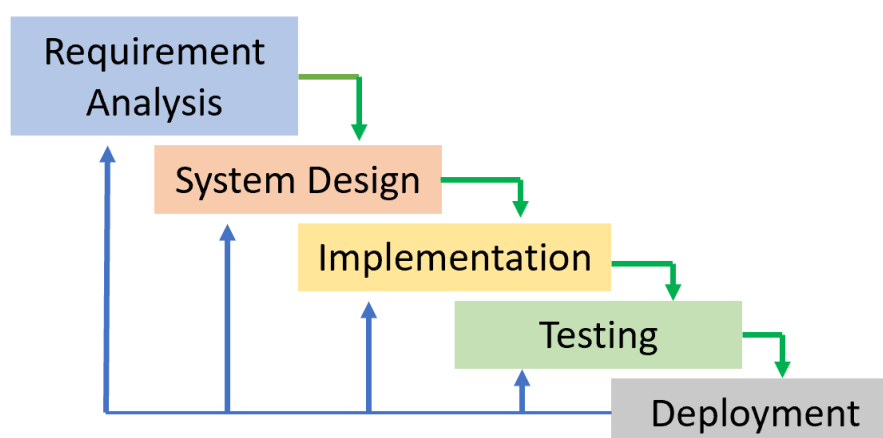


Figure 8: The Iterative Waterfall Process

### 3.3.2 The requirements

During the requirement analysis process, it was decided that the library should have certain specifications. First came performance. It is essential that the Library should be able to offer a performance increase over the CPU on code that is easily parallelized. Then comes platform independence. That means that the library should be able to run in as many platforms as possible while using the same code. Platform independence translates to a much broader market pool. The final prerequisite is Ease of coding. The Library should be easy to code for, by people with JavaScript experience. Even though it is difficult to specifically quantify and compartmentalize this, some characteristics are as follows.

- **Data Wrapping**  
Are the data wrapped into graphical data automatically by the library, or does the developer have to do it themselves. A library that does the data wrapping automatically offers great ease of use for the developer, but might have some negative effect on performance
- **Memory Operations Control**  
Whether the library offers direct control of when the memory operations occur or if it entirely transparent to the developer has an effect on the control the developer has on the overall program. Better control means better ways to troubleshoot and optimize but it might come with the caveat of development complexity. The perfect combination would be a library that either allows proper control without complexity or handles memory operations already on the most optimized manner.
- **Kernel Language**  
The language in which the kernel is written is also an important factor of the library. It is requested that the library should only use JavaScript or one of the derivative technologies like Microsoft's Typescript. Libraries that need the kernel to be written in GLSL are preferred to be avoided.
- **Kernel Mapping**  
Since a large number of the operations are done on 2 dimensional datasets like images, it is imperative that the kernel has a way to access the datasets in a 2 or 3-dimensional format. That means that there must be operators that show both the location of the output dataset and the location of the input dataset.
- **Updates/Community Support**  
Preferably the library should be updated often with new features and have good community support. Projects that are either Proof of Concept or are not developed anymore should in best case be avoided as they will not be able to follow the technological changes in the field and provide only time limited benefits
- **Open Source/Free**  
Finally, the library should be free of charge and Open Source.

### 3.4 Choosing the right candidate

While looking for available candidates, 5 solutions that offered some GPGPU solution for JavaScript were discovered. Those 5 are, Gpu.js, Turbo.js, WebCLGL, MaiaVictor/Webmonkeys and Gpgpu.js

**GPU.js** is the library that offers the easiest platform to code. Kernel creation is done in JavaScript in an easy manner, it offers adequate position identifiers for 1D, 2D and 3D data matrices, allowing better data manipulation. It also offers the best platform compatibility since it was able to run on all common browsers on Windows, and on android phone without issues. As mentioned before, the kernel is written in JavaScript and is automatically translated into GLSL by the library. This allows for much easier Kernel development. While there is no direct memory copy control, the memory copy processes were among the fastest of the group. Lastly but very important, it offers a seamless fall-back to CPU when there is no valid GPU for the acceleration, making the programming of CPU fall-back routines obsolete. (gpu.js, 2019.)

**Turbo.JS** needs the GPU kernel to be coded in GLSL, but the data structures are automatically transformed for WebGL compatibility. GLSL offers better control over the code of the Kernel since the kernel is not translated from JavaScript but is substantially more difficult to code. This becomes even more apparent when you consider the lack of a position identifier. So, the data can only be manipulated in regard to their self, and not in association with their position on a data matrix. Another important drawback of Turbo.js is the inability to run on a big number of android devices, since it requires the OES\_texture\_float WebGL v1 extension that is lacking in older generation Adreno GPUs, and almost all Mali GPU's. Together with the fact that it offers no automatic CPU fall-back, this makes Turbo.js a very bad candidate when it comes to platform compatibility and interoperability. (turbo.js, 2019.)

#### **WebCLGL**

The WebCLGL library requires as well to be coded in strict GLSL which makes coding for it difficult, while at the same time not offering as good data wrapping as gpu.js or turbo.js. It only offers basic numerical calculations on the GPU and offers only 1 dimensional position identifier which makes it really bad for 2 dimensional operations like heatmaps or 2D extrapolators. (webclgl, 2017.)

#### **Webmonkeys**

The Webmonkeys library has overall the best control on memory writes-reads but does so at a speed cost. Kernel programming is done in GLSL which affects the ease of use negatively, and it does not offer any embedded CPU fallback options. Preliminary benchmarking showed very slow performance between Host to Device and Device to Host memory copy operations. (Webmonkeys, 2018.)

#### **GPGPU.js**

Gpgpu.js is just a barebones library with very little usability, very complex kernel creation and almost no data wrapping for use with WebGL. It was mostly written as a proof of concept library and

even though it is considered one of the pioneers, it does not have any of the important features that are needed from a library. (gpgpu.js, 2016.)

During our testing session we did 2 culling sessions to narrow down the applicants. First, we tested the ease of use for each library. Since it was important that the library should be used by JavaScript developers, we had to discard the libraries that had kernel programmed in GLSL and not JavaScript. After that we tried to code a simple extrapolator routine to test both ease of coding and performance. Next, we discarded the ones that we found too difficult or impossible to code for. On the remaining candidates we benchmarked, tested for platform compatibility and chose the best candidate.

In conclusion, gpu.js is best overall solution between all the candidates. Gpu.js is the easiest and more hassle-free library to code in, while at the same time giving very good performance results, Native CPU fallback and the best platform compatibility. It is also a library that is still actively being worked on and upgraded and had the release of its version 2 during the duration of this Thesis that solved a lot of issues and gave important performance increase.

## OPTIMIZATION TECHNICUES FOR GPU.JS

### 4.1 Memory Optimization

Memory Optimization is a key component in any Massive Multiprocessing and GPGPU implementation because of the huge memory overhead penalty. This is also true in Gpu.JS. For any gpu.js kernel to be effective, all memory copy operations between the host and the device, should be kept to a minimum, and preferably there must be only 2, once at the beginning and once at the end. Even that can also be reduced to only 1 operation if we use a Direct Rendering method, as we see later in this chapter.

### 4.2 Optimized Data Formats

GPU.js uses data formats that are strongly typed, since GLSL uses a subset of C++. Even when the user uses weak type formatted date, GPU.js will translate them into strongly typed. Since it offers a big variety of Argument types, like Array to Array(4) up to Array3D(4), its best to make sure your data are packed into 1 type, so only 1 memory operation is needed.

Also, in earlier versions, GPU.js has significant performance gains when using flattened arrays, especially when returning. To make calculations in flattened arrays easier by still using the x, y ,z lookup positioners, gpu.js offers the Input type for the kernel(Figure 9). (gpu.js, 2019.)

```
const { GPU, input, Input } = require('gpu.js');
const gpu = new GPU();
const kernel = gpu.createKernel(function(a, b) {
  return a[this.thread.y][this.thread.x] + b[this.thread.y][this.thread.x];
}).setOutput([3,3]);

kernel(
  input(
    new Float32Array([1,2,3,4,5,6,7,8,9]),
    [3, 3]
  ),
  input(
    new Float32Array([1,2,3,4,5,6,7,8,9]),
    [3, 3]
  )
);
```

Note: `input(value, size)` is a simple pointer for `new Input(value, size)`

Figure 9: Example of the Input Type for kernels at the GPU.js website (gpu.js, 2019)



Gpu.js also uses different data types that emulate textures when communicating between kernels, such as when using the pipelining options. Choosing the proper type can enhance performance.

### 4.3 Optimize Algorithm

Apart from optimizing the data that are being calculated, optimizing the algorithm that does the calculations is also very important, and can have a big effect on performance.

First of all, as with all parallel computation solutions, the code must take as much as advantage of parallelization as possible. That means the more of our code runs in parallel, the better the performance, even if to achieve that parallelization, the algorithm becomes more complex. If there are enough iterations of the serial data, then the complexity increase is overcome by the gain in performance by running them in parallel.

Another way to optimize the kernel algorithm is to avoid conditionals. Even though newer shader processors have received optimizations that offer decent branch prediction, the GLSL ES is based on a GLSL version that does not have proper branch prediction (Apple, 2018), and during the calculations, all conditions must be calculated and then discarded. So, conditionals inside a kernel, especially in a loop, have a severe effect in performance and should be avoided. (The Orange Duck, n.d.)

Finally, a best practice to optimize the kernel algorithm is to avoid large loops. Loops more often than not are good candidates for parallelization, so it makes more sense to spawn a second kernel to do the operation than have the loop run in the kernel. Even in cases where the loop is not capable of being run in parallel, the number of iterations should be passed beforehand to the kernel as an attribute. Conditionals inside loops offer the worst performance hit. (gpu.js, 2019.)

### 4.4 Minimizing Host to Device Copies

As has been already mentioned before, one of the most important hits of performance for the GPGPU computations is the memory copy from host to the device. In GPGPU terminology, Host is considered the CPU and various memory levels it has, from Level 1 cache all the way to the System Memory. Device is the GPU device, with the GPU cores, the cache and the Graphics Memory. So, since the GPU and the CPU do not actually share a memory space, they can both directly access, all communication between them happens by memory copy operations which have to go through the much slower PCI Express bus (or platform dependent alternative). This memory copy is what makes the Host to Device and the Device to Host memory operations so expensive in computational time. Thankfully for this reason GPU.js offers 3 different ways of minimizing those memory operations, when there is a need for more than 1 kernel to be run on a set of data.

First is the Combining kernels option (Figure 10), which allows one kernel to receive as input the output of a previous kernel, while keeping the result of the operations in the GPU memory. (gpu.js, 2019.)

```

const add = gpu.createKernel(function(a, b) {
  return a[this.thread.x] + b[this.thread.x];
}).setOutput([20]);

const multiply = gpu.createKernel(function(a, b) {
  return a[this.thread.x] * b[this.thread.x];
}).setOutput([20]);

const superKernel = gpu.combineKernels(add, multiply, function(a, b, c) {
  return multiply(add(a, b), c);
});

superKernel(a, b, c);

```

Figure 10: Example of the Combining Kernels method from the gpu.js website (gpu.js, 2019)

The second way to avoid memory operations is the Create Kernel Map (Figure 11). This method allows multiple operations in one kernel and saving the output of each operation. This is for example useful in machine learning operations. (gpu.js, 2019.)

### object outputs

```

const megaKernel = gpu.createKernelMap({
  addResult: function add(a, b) {
    return a + b;
  },
  multiplyResult: function multiply(a, b) {
    return a * b;
  },
}, function(a, b, c) {
  return multiply(add(a[this.thread.x], b[this.thread.x]), c[this.thread.x]);
}, { output: [10] });

megaKernel(a, b, c);
// Result: { addResult: Float32Array, multiplyResult: Float32Array, result: Float32Array }

```

### array outputs

```

const megaKernel = gpu.createKernelMap([
  function add(a, b) {
    return a + b;
  },
  function multiply(a, b) {
    return a * b;
  }
], function(a, b, c) {
  return multiply(add(a[this.thread.x], b[this.thread.x]), c[this.thread.x]);
}, { output: [10] });

megaKernel(a, b, c);
// Result: { 0: Float32Array, 1: Float32Array, result: Float32Array }

```

Figure 11: Example of the Kernel Map method from the gpu.js website (gpu.js, 2019)

The third method that gpu.js offers to alleviate the need for memory copy operations is called pipelining. Pipelining (Figure 12) allows value to be sent directly from kernel to kernel via a texture. In recent versions, the ability to clone the kernel outputs was added as well. (gpu.js, 2019.)

```
const kernel1 = gpu.createKernel(function(v) {
  return v[this.thread.x];
})
.setPipeline(true)
.setOutput([100]);

const kernel2 = gpu.createKernel(function(v) {
  return v[this.thread.x];
})
.setOutput([100]);

const result1 = kernel1(array);
// Result: Texture
console.log(result1.toArray());
// Result: Float32Array[0, 1, 2, 3, ... 99]

const result2 = kernel2(result1);
// Result: Float32Array[0, 1, 2, 3, ... 99]
```

Figure 12: Example of the pipelining method from the gpu.js website (gpu.js, 2019)

One final method of reducing memory copy operations is by rendering the output of a kernel directly to the Canvas.

#### 4.5 Direct Rendering

Direct Rendering is the process of displaying the result of the GPGPU computations, directly on an HTML5 canvas, without having to do a memory copy back to the CPU memory and then parse the data from a texture file. This offers a very big performance boost since it essentially halves the memory copy overhead and gives the ability to have real time depiction of the GPGPU computations. To render directly to a canvas with gpu.js you have to use the Graphical Output mode. This is done by setting the setGraphical option to true and have a 2-dimensional array as output (Figure 13). Also, the kernel must have as return value a `this.color(r,g,b)` or `this.color(r,g,b,a)` to correspond to a pixel colour value. For this we also have to make sure we are using a visible HTML5 canvas, and not a hidden one, as is customary for non-graphical computations. (gpu.js, 2019.)

```
const render = gpu.createKernel(function() {
  this.color(0, 0, 0, 1);
})
.setOutput([20, 20])
.setGraphical(true);

render();

const canvas = render.canvas;
document.getElementsByTagName('body')[0].appendChild(canvas);
```

Figure 13: GPU.js graphical output code example from gpu.js GitHub page (gpu.js, 2019)

## MAKING THE LIBRARY

### 5.1 Identifying What parts of the Product will benefit from GPU Acceleration

By Arction it was requested that the following items be accelerated via GPGPU. They are considered some of the most power intensive parts of the LightningChart JS product, and they all have strong indications that they can be accelerated by Parallel Computing. To this end, interviews with the Project Manager and CEO took place, and then a quick run-down of the code with the current developers to identify possible candidates. What was sought after was long and complicated serial loops that could in theory be later calculated in parallel. The result of this work was the following calculation routines.

#### Data Upsamplers (Interpolators)

Interpolators (Figure 14) are routines that are used to increase the resolution of a digital signal. They use different mathematical techniques to estimate the values between two known samples.

The most common techniques are:

Linear, where the unknown values are calculated from a straight line connecting the two values, Nearest Neighbour, where the unknown values are calculated by taking the value of the nearest sample, and finally Spline which is the most intricate. In Spline the Values are calculated by a polynomial called spline, named after the elastic rulers used in drawings for shipbuilding (Hazenwinkel, 1994).

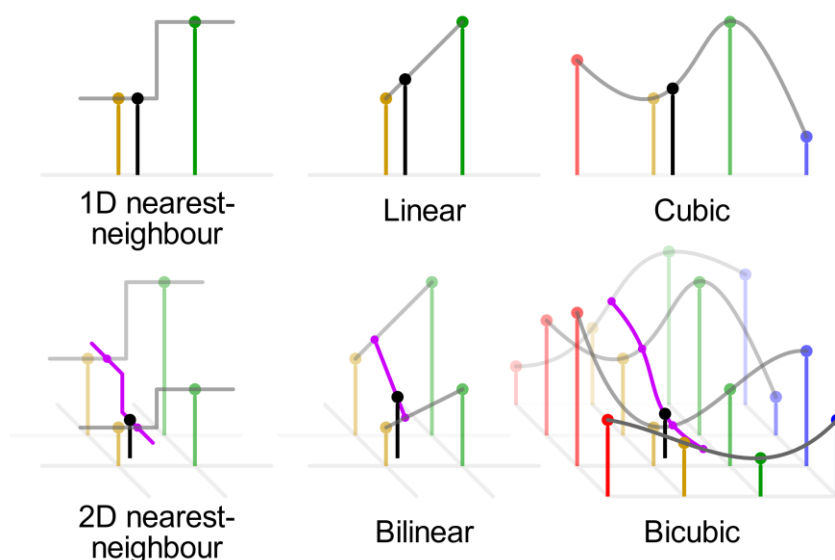


Figure 14: One- and Two-Dimension Interpolator examples (By Cmglee in English Wikipedia, CC BY-SA 4.0,)

Interpolators are important in signal and image manipulation. They can be used for example to recreate a very low-resolution heart-rate signal, or to recreate missing frames from a live-streaming video such as can be found in Telemedicine applications.

## Down Samplers (Extrapolators)

Down Samplers or extrapolators are mathematical calculations that reduce the resolution of a sample so it can fit into a screen or take less file size. In Extrapolators, a group of samples is represented by one value. This value is usually the maximum value of the sample, the Average value of all calculated samples, or the minimum value, depending on the case. Downsamplers are essentially the exact opposite of Interpolators and are often used on the opposite side of a broadcast. A device with low bandwidth, such as a portable electrocardiograph or an MRI machine, can lower its resolution using a Down sampler, and at the receiving end, on the Physicians terminal perhaps, that resolution can be rebuilt in close approximation to the original using an Interpolator.

Extrapolators and Interpolators are also often used in image resizing, either to make an image larger (Interpolator) or an image smaller (Extrapolator).

## Heatmap Data (2D)

Heatmaps are visual representation of a two-dimensional array of values, where each value has its own colour, usually but not exclusively, from deep blue to red. As a two-dimensional array, Heatmaps are often Extrapolated or Interpolated to fit the screen canvas. They are often used to portray temperature in areas (hence the name heatmap) but are can be used for any kind of value. We can see an example of a heatmap portraying the rain intensity on the United States of America on the figure bellow (Figure 15).

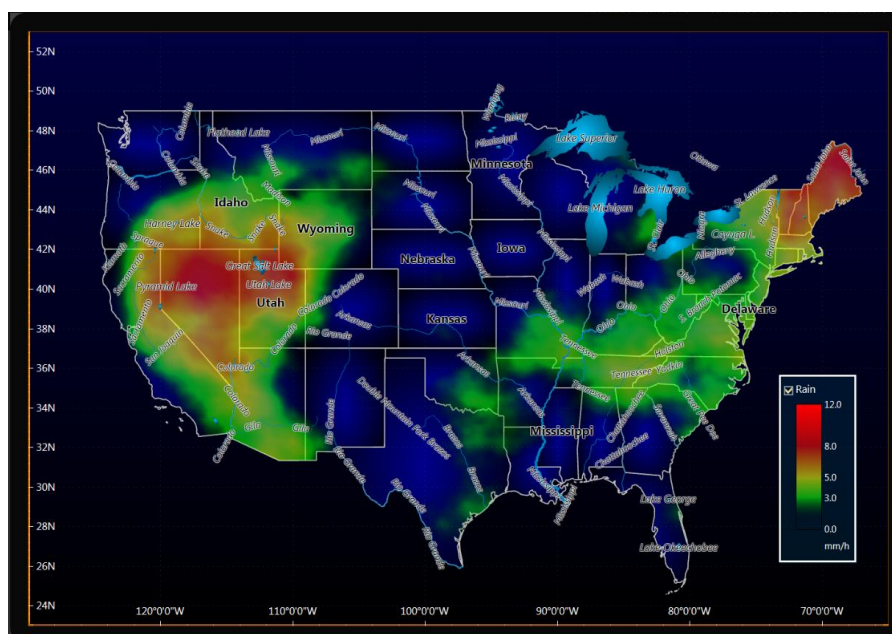


Figure 15: Heatmap example portraying rain intensity in the USA made by LightningChart (Image courtesy of Arction Oy)

Heatmaps have various implementations in the field of medicine and healthcare. Since they are used to portray the intensity of a value in a certain area, they can be used in epidemiological studies, to show for example the number of disease carriers in an area, or what percentage of the population in

a location has a certain genome. It can help detect diseases or genetic tendencies for population groups in conjunction with their geographical location.

## Financial Routines

### 1. Relative Strength Index (RSI Calculations)

RSI is a financial tool that helps visualize the momentum of a stock. It calculates at least the last 14 values of a stock and shows a trend of the stock being oversold or overbought. It is a simple calculation in complexity but requires a large amount of calculations especially if the sample is big, for example 2 or 3 years long. This makes it a really good candidate for parallelization. (Chen, 2019.)

### 2. Fibonacci retracement

Fibonacci retracement is essentially a tool that applies the Golden Ratio of Mathematics into financial sciences. In Mathematics, two quantities have a golden ratio when their ratio is the same as the ratio of their sum to the larger of the two quantities. Fibonacci retracement essentially creates border lines based on Golden ratio percentages as limits to try and calculate stock behaviour. For example, stocks that have a drop in price over 23.6% are treated differently by traders that stocks that have a drop of 38.2%, 50% and 61.8%, all those numbers being implementations of the Golden Ratio. (Mitchell, 2019.)

### 3. Moving Average Convergence Divergence (MACD)

MACD is an indicator that allows traders to see the relationship between two moving averages. To calculate it, one must calculate a 26-day Exponential Moving Average and a 12-day Exponential Moving Average and the first from the later.

#### 5.1.1 Theoretical

Since GPGPU is essentially a SIMD architecture, it essentially means that every computing unit, has only access to its own set of data in the memory, and cannot alter the output of others. So only the data calculations of an algorithm that allow these kinds of operations are getting any actual performance increase. How much of a performance increase we would get, could be roughly estimated by Amdahl's Law.

In 1967, computer scientist Gene Amdahl presented a formula that which theoretically calculates the performance benefit of an application from being run on multiple processors (Jenkov, 2015).

According to Amdahl's law  $S_{\text{latency}} = \frac{1}{(1-p) + \frac{p}{s}}$  where  $S_{\text{latency}}$  is the execution speedup,  $s$  is the part

of the program that can be parallelized, and  $p$  is the proportion of execution time that the part benefiting from parallelization originally occupied (Figure 15).

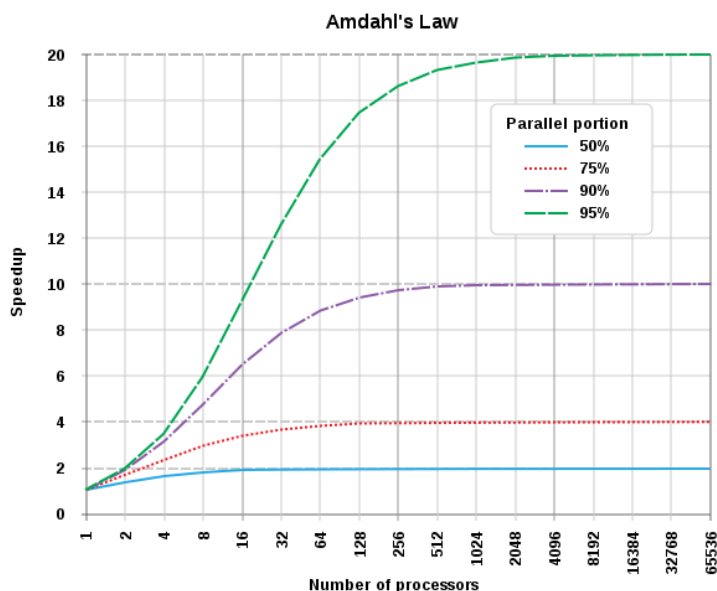


Figure 16: Amdahl's Law speed gain per resources. (By Daniels220 , CC BY-SA 3.0)

As we can see the gain from the extra processors slowly deteriorates the more processors we add. This happens because as much as we reduce the execution time of the part of the code that can be parallelized, the serial part of the code is still taking the same time to execute. This is even more visible when the cost of parallelization, i.e. the overhead from the parallelization process it also being taken into consideration.

In 1988 on the other hand, John L Gustafson and Edwin H Barsis, in an article called "Reevaluating Amdahl's Law", calculated the speedup in latency of task with fixed execution time (compared to a fixed problem size that Amdahl's law uses). The thinking behind this was that when developers have more resources, they will match their code to take advantage of all the available resources and produce more results. For example, if the problem would be to render a video file with a certain quality, when given enough resources, the developers will then choose to opt for a higher quality. Gustafson's law states the speedup  $S$  gained by using  $P$  processors for a task with a serial fraction  $a$  is  $S=P-a(P-1)$  (Figure 16)



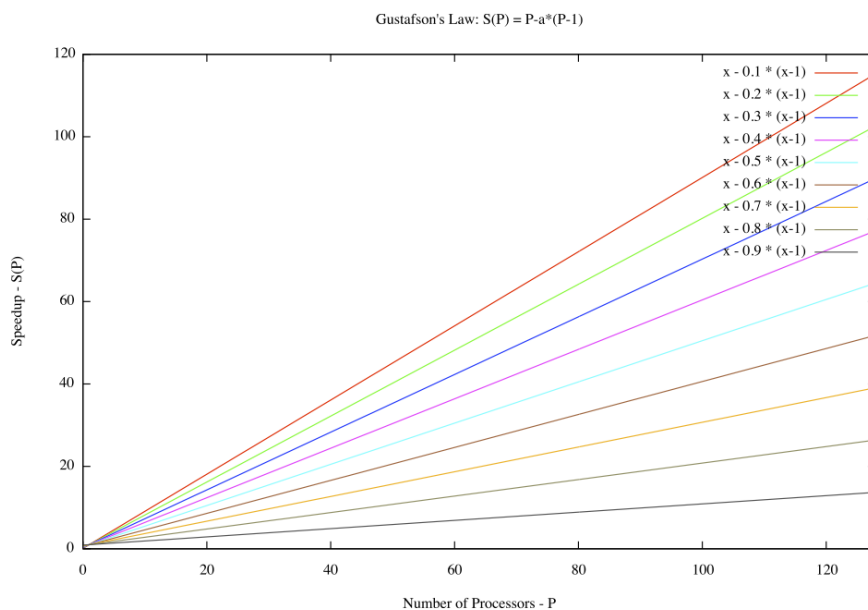


Figure 17: Gustafson's Law Speedup per Number of Processors (image By Peahihawaii - Own work, CC BY-SA 3.0)

### 5.1.2 Practical

Apart from the theoretical aspect, the practical aspects of the coding also come into play. For example, a Fast Fourier Transformation (FFT) algorithm could be written for GPU.js, but in reality, the end result is quite below expectations. This is mostly because of the complexities of the code, the larger overhead and GLSL language limitations. Considering the fact that most Browsers already have highly optimized FFT functions build-in, we can see that there seems to be no sense in developing a GPGPU solution for FFT.

Another aspect that affects performance is the actual transpilation of the JavaScript code itself into GLSL by the gpu.js library. Even though the library does its best to optimize the code, the whole process is not visible to the end user, and there is little control of the end result, which might often be less optimized than native GLSL written code. This is also exaggerated by the fact that the data wrapping is also invisible to the user, and the library decides for the most optimal way to do it. Essentially the best and safest way to test the practical real-life gains of the code is to actually write it and see the end result.

## 5.2 Writing the code.

Once the identification of the parts of LightningChartJS that are good candidates for the GPGPU implementation was complete, the task of applying the theoretical aspects of parallelization to get a theoretical representation of what could be expected. From the beginning it was obvious that 2 of the financial routines were not capable of proper parallelization. While the both had a lot of code being repeated, every run of the code depended on the result of the previous. That means that the necessary independence between the datasets was not there. Those 2 calculations were discarded.

On the other hand, applying Amdahl's and Gustafson's law to the rest of the algorithms showed the theoretical potential for significant performance gain.

With the theoretical part of the way the actual coding started. All of the parallelized parts of the code were translated into GPU.JS kernel code. For proper benchmarking, all those routines were also re-written as they were into pure JavaScript code so there can be a proper comparison in performance. During this time, the `gpu.js` community pages, especially on GitHub were visited to find solutions in different problems. Most of the problems were already been referenced in one way or another, and in the cases where there was no information available, the developers answered questions fast and in a helpful manner. Once every routine was completed, there was a constant check for ways to optimize it even further, and to make sure that there were no updates to the `gpu.js` library that offered new tools or solutions. More often than not, it was necessary to update the library and the code to take advantage of new features.

## RESULTS

The next step after completing the process of writing the code for the requested GPGPU module, is to assess how the module performs, especially in comparison with the CPU equivalents. To put it in simpler terms, how much faster is the GPGPU accelerated code than a normal serial code.

### 6.1 Benchmarks CPU vs GPU

All computational routines were done both in CPU and GPU. At first the CPU results were based on the CPU fail-over function of the gpu.js, but this offered a substantial overhead that gave erroneous results, so the CPU routines were written in pure JavaScript. All the benchmarks were run on all possible available solutions, mainly the most common web browsers available. The benchmarks used a variety of samples with different sizes to more clearly show how well the performance benefits matched the bigger computation sizes.

#### 6.1.1 Interpolators

All interpolator routines were run on both the CPU and GPU in various Browsers. Different benchmarks were run for each interpolator and getting 1000, 1 Million and 10 Million points out of 300 and 1000 samples for a total of 6 results per Interpolator. The next 3 tables show us the result of the benchmarks.

#### 1. Linear

Linear Interpolator		Chrome		Firefox		Opera		Edge		Edge Beta (chromium)	
		CPU	GPU	CPU	GPU	CPU	GPU	CPU	GPU	CPU	GPU
1000 Points	300 samples	3	26	2	70	4	76	2	70	4	89
1 Million	300 samples	175	320	255	321	188	294	101	2061	184	250
10 Million	300 samples	1549	7465	2933	2887	1467	5879	3723	Crash	1829	2518
1000 Points	1000 samples	2	30	2	71	3	72	2	71	4	85
1 Million	1000 samples	295	269	657	356	353	265	268	2078	394	284
10 Million	1000 samples	2940	8225	6507	3496	3407	6210	3244	132675	3149	3265

Table 1: Linear Interpolator on different browsers (execution time in milliseconds)

## 2. Nearest Neighbour

Nearest Neighbour		Chrome		Firefox		Opera		Edge		Edge Beta (chromium)	
		CPU	GPU	CPU	GPU	CPU	GPU	CPU	GPU	CPU	GPU
1000Points	300 samples	<b>3</b>	26	<b>2</b>	55	<b>2</b>	20	<b>1</b>	55	<b>3</b>	22
1M	300 samples	<b>126</b>	229	<b>215</b>	324	<b>187</b>	285	<b>140</b>	1681	<b>193</b>	207
10M	300 samples	<b>1449</b>	5622	<b>1974</b>	2597	<b>1454</b>	5745	<b>1664</b>	130593	<b>1447</b>	2472
1000Points	1000 samples	<b>6</b>	28	<b>3</b>	56	<b>6</b>	18	<b>1</b>	58	<b>5</b>	24
1M	1000 samples	378	<b>322</b>	544	<b>324</b>	372	<b>261</b>	<b>307</b>	1651	370	<b>241</b>
10M	1000 samples	<b>3504</b>	5698	5351	<b>3181</b>	<b>3302</b>	6018	<b>3290</b>	131921	<b>3255</b>	2493

Table 2: Neighbour Neighbour Interpolator on different browsers (execution time in milliseconds)

## 3. Spline

Spline		Chrome		Firefox		Opera		Edge		Edge Beta (chromium)	
		CPU	GPU	CPU	GPU	CPU	GPU	CPU	GPU	CPU	GPU
1000 Points	300 samples	<b>8</b>	30	<b>2</b>	66	<b>7</b>	21	<b>2</b>	73	<b>7</b>	25
1 Million	300 samples	<b>188</b>	243	<b>267</b>	302	<b>182</b>	226	<b>146</b>	1677	<b>181</b>	221
10 Million	300 samples	<b>1426</b>	5906	<b>2328</b>	3104	<b>1946</b>	5756	<b>1742</b>	135400	<b>1859</b>	2131
1000 Points	1000 samples	<b>7</b>	32	<b>10</b>	78	<b>8</b>	69	<b>4</b>	70	<b>8</b>	90
1 Million	1000 samples	351	<b>272</b>	650	<b>309</b>	348	<b>258</b>	<b>311</b>	1724	354	<b>279</b>
10 Million	1000 samples	<b>2981</b>	5665	5556	<b>3313</b>	<b>3564</b>	5955	<b>3353</b>	135421	<b>3382</b>	2757

Table 3: Spline Interpolator on different browsers (execution time in milliseconds)

All 3 cases show the same behaviour. The CPU is faster than the GPU when interpolating 300 samples. Things change when interpolating 1000 samples, where the GPU overcomes the CPU on 1 Million points. On Firefox this trend continues on 10 million points as well, but Chrome, Opera and Chromium based browsers reach graphic memory allocation issues and the GPU performance deteriorates again. This is even more exaggerated on Edge, where the GPU results for 10 Million points are either very high or the browser crashes with memory errors.

### 6.1.2 2D Extrapolators

The 2D Extrapolators benchmarks were run on the most common use cases. 2 different samples were used, one 4000x4000, one 8000x8000. They were both down sampled to 100x100 and 1000x1000. Lastly a very common industry use case was tested, down sampling from a 4K resolution of 3840x2160 to a Full HD video of 1920x1080 resolution. We can see the exact results of the Extrapolators benchmark on Table 4.

			Chrome		Firefox		Opera		Edge		Edge Beta (chromium)	
Extrapolators												
	Original Sample	Resized Sample	CPU	GPU	CPU	GPU	CPU	GPU	CPU	GPU	CPU	GPU
Maximum	4K	100	46	143	24	162	80	211	25	340	70	194
	4K	1K	111	344	62	193	106	284	68	381	108	351
	8K	100	109	528	76	406	96	596	101	1103	96	557
	8K	1K	228	1004	176	484	226	1241	175	1203	234	1056
	4K	FullHD	113	296	75	159	109	315	61	270	111	330
Minimum	4K	100	46	188	22	155	44	194	34	360	49	193
	4K	1K	115	350	66	187	113	324	70	373	111	456
	8K	100	111	552	75	408	112	560	87	1170	112	553
	8K	1K	253	906	174	488	248	1103	172	1331	255	1058
	4K	FullHD	121	306	65	171	116	295	59	269	114	322
Average	4K	100	41	184	27	140	41	182	34	391	36	182
	4K	1K	129	337	64	176	127	268	97	379	99	457
	8K	100	90	546	98	381	89	547	266	1174	90	955
	8K	1K	207	910	205	471	209	886	248	1574	204	1732
	4K	FullHD	107	296	68	144	100	291	99	257	101	334

Table 4: Extrapolation routines on different browsers (execution time in milliseconds)

### 6.1.3 Heatmap

Heatmap is a routine where the gpu.js library can show its biggest potential. Since a part of the routine is to render the calculated image directly to the screen, we have the ability to save on one big memory operation from the Device to Host. This lowers the overhead substantially and lowers the threshold of calculations where the GPU overtakes the CPU. The benchmarks are done on the most common resolutions that are found in desktop and mobile devices right now, so they are as close to the actual possible use scenarios (Table 5).

Heatmap	Chrome		Firefox		Opera		Edge		Edge Beta (chromium)	
	CPU	GPU	CPU	GPU	CPU	GPU	CPU	GPU	CPU	GPU
Resolution: VGA (640x480)	<b>12</b>	36	11	31	<b>8</b>	23	275	<b>38</b>	<b>9</b>	36
HD (1280x720)	<b>13</b>	55	45	<b>33</b>	<b>13</b>	37	808	<b>51</b>	<b>13</b>	52
FullHD (1920x1080)	73	<b>69</b>	44	<b>35</b>	<b>24</b>	68	1828	<b>62</b>	<b>23</b>	82
4K (3480x2160)	<b>75</b>	230	123	<b>49</b>	<b>74</b>	269	7458	<b>146</b>	<b>118</b>	275
8K (7680x4320)	<b>587</b>	1549	445	<b>98</b>	<b>415</b>	889	586	<b>453</b>	<b>564</b>	902

Table 5: Heatmap Routine on different browsers (execution time in milliseconds)

As we can see, this test is the most ambiguous when it comes to cross platform results. For chrome itself, the GPU overcomes the CPU for Full HD resolutions which means a little over 2 million samples but again we run across memory performance issues and the CPU retakes the lead for 4K and onwards. On the other hand, this does not replicate on the chromium-based Edge beta where the CPU has a constant advantage on the GPU. Firefox as usual shows the best results, where the GPU is faster than the CPU on HD resolutions and higher. Opera clearly favours the CPU which same as the Edge Beta, has no results that favour GPGPU acceleration. Finally, Edge has the most inconsistent results, with the GPU outperforming the CPU on all cases. This is most likely a bug on the Edge JavaScript implementation. The final results are the average of 5 runs of the same test. In Edge's case the first run always had results similar to the other browsers and much lower than the GPU, but the other four iterations of the benchmark shows substantial degradation in CPU performance.

#### 6.1.4 Financial and other Routines

For the financial calculations, the same methodology was used as before. RSI calculations were run on stock samples of 1 thousand, 1 million and 10 million points. Even though RSI have a strong serial component, the parallel component of the code was easily optimized for the GPU and have really good results even despite the initial overhead. In most cases the GPU was faster from 1Million points onward on all browsers with the exception of Edge, where the CPU results were always faster as seen on Table 6. As stated before, Fibonacci Retracement and MACD were not suitable candidates for GPGPU acceleration.

tutions		(chromium)									
RSI	CPU	GPU	CPU	GPU	CPU	GPU	CPU	GPU	CPU	GPU	
1K points	1	3	1	1	1	2	1	1	1	2	
1M points	37	25	24	21	33	26	44	59	34	28	
10M	320	242	291	184	321	233	392	572	310	219	
Fibonacci Retracement	Does not support Parallelization										
MACD	Does not support Parallelization										

Table 6: Financial Calculations on different browsers (execution time in milliseconds)

## 6.2 Coding with gpu.js

Writing code with the `gpu.js` library was fairly simple and straightforward to anyone with JavaScript knowledge. There is of course the learning curve where the developer needs to get used for writing parallelized SIMD code, but for anyone with some experience this is not an issue. Writing the kernels on `gpu.js` is also very intuitive especially to anyone with CUDA experience since they share the same reference methodology. To easily reference your position in the dataset, CUDA uses a variable called `threadIdx.x` and `threadIdx.y`. `Gpu.js` emulates this by using the `this.thread.x` and `this.thread.y` constants in the kernel. What can prove difficult to master for someone who switches from CUDA or any other native GPGPU solution to `gpu.js` is the lack of memory control. Unfortunately, `gpu.js` does all the memory operations transparently and the developer has very little control over it. Finding the most optimized ways to present your data to library takes a little time to get used to and still the control is limited. Ways to keep the data into the GPU and avoid memory copy operations back and forth are provided but up until recently the documentation was somewhat hazy. Lately this has been improved with better documentation.

Finding ways to optimize the kernel code also takes some time. The developer needs to work out how the JavaScript code transpiles into GLSL and then work around that to make as the code as optimized as possible. A very positive aspect of the whole experience was the very active and helpful community. The library developers answer most questions on GitHub (The Open source version control website that hosts the development of the code) and there are constantly new updates and bug fixes.

## 6.3 How did our Results match our Expectations?

While comparing our results to what we hoped to gain from our Theoretical overview of the code, the results were below our expectations. The memory overhead of the JavaScript engine in each browser is higher than expected. That is most likely due to the extra overhead from the `GPU.js` library to wrap the data into textures to be injected in the graphical pipeline.

Algorithm complexity also contributed a lot to the performance degradation. The shader language used in WebGL is based on a much older version of OpenGL language, and does not have the

proper optimizations like branch prediction, which affects the performance. Still the GPU manages to overcome the CPU in most of the cases, but this gain was later negated by memory limits on most browsers.

The turning point and at the same time the sweet spot for GPGPU seems to be around the 1 million calculation mark, depending heavily on the kernel complexity, especially when it comes to having loops inside the kernel. On this Firefox proved to have the most optimized implementation of the JavaScript engine and offered by far the best results. Most results that favoured the GPU were over the 1 Million calculation limit, and Firefox continued to improve performance after that.

### 6.3.1 The Browser Effect.

As was proven by the results, one of the most important factors in the performance gain, or lack thereof, of the GPGPU accelerations over JavaScript, is the JavaScript implementation itself. Different browsers give varying results on the exact same hardware running the exact same code.

In all the benchmarks, there were varying results not only on the GPGPU implementation itself but on the CPU execution times as well. That means that not only is each browser treating the WebGL code differently, but also the pure, non-graphical JavaScript code as well.

This is a severe limiting factor when trying to establish a baseline performance increase, especially when trying to choose if an implementation is worth it in terms of performance or not. The main result of this inconsistency is that the burden of choice for the implementation falls on the end user who chances are will not actually have the expertise to make a decision like this. This makes the end product much less user-friendly and can create a bad reputation for the product as an unstable solution because of the possible bad implementations.

## 6.4 Where to from now?

GPGPU on JavaScript is still in early stages. The results unfortunately have not been conclusive in favour or against using GPGPU acceleration. While there are indeed techniques where GPGPU acceleration in real-time visualization can offer some performance increase, especially when drawing directly to the canvas and avoiding part of the memory overhead penalty, even those increases are highly platform and implementation dependent.

GPGPU accelerated JavaScript shows strong benefits in data preparation, especially in server-side applications when using technologies like Node.JS to implement the JavaScript Engine instead of a browser. Matching all these shortfalls with the disadvantage of WebGL's shader language that is based on older versions of OpenGL and is far less optimized for non-graphical application, we can clearly see a need for the underlying technologies be updated. To this end GPU.js claims to be able to take advantage of the Vulkan graphics API (gpu.js, 2019) soon instead of just WebGL, with future API implementations soon to follow. It is safe to assume that Apple's Metal is soon to follow.



This might be enough to allow GPGPU JavaScript acceleration to overcome its teething issues and become much more competitive with hardware native GPGPU technologies. An actual implementation of WebGL from the browsers will be a sure step towards this direction. This can exponentially grow the infiltration of GPGPU acceleration since every mobile phone, IoT capable device, and computer will have the option to be a platform-independent mini supercomputer. There can be no doubt that an industry wide solution for JavaScript GPGPU implementation will be most beneficial to developers and in the end of the day the final user. This happened in the Operating System level when the industry switched from 3<sup>rd</sup> party solutions to CUDA and then to OpenCL and it is safe to expect that the same benefits will apply to the JavaScript sandbox as well.

In the field of Medicine and Healthcare the applications of GPGPU are numerous as was discussed earlier. IoT devices with small processing power can do faster data preparation using JavaScript GPGPU acceleration. This would save bandwidth if the device is in an area with bad internet connectivity, especially in cases like signal processing. Simple signals like Electrocardiograms (ECG's) or Encephalograms (EEG's) can be processed and have their resolution reduced with extrapolation to save broadcasting time, and then be interpolated back into higher resolution for the physician to diagnose. Right now, in mobile devices like that which are mostly web-application based, GPGPU acceleration varies from non-existent to highly undependable. The time when every mobile device will have the ability to run fast, accurate, low-latency and processing power intensive mobile health applications is almost here.

## REFERENCES

## 7.1 References

- Apple. (2018). *Apple Developer. OpenGL ES Programming Guide*. Retrieved November 25, 2019, from Best Practices for Shaders:  
[https://developer.apple.com/library/archive/documentation/3DDrawing/Conceptual/OpenGLES\\_ProgrammingGuide/BestPracticesforShaders/BestPracticesforShaders.html#//apple\\_ref/doc/uid/TP40008793-CH7-SW12](https://developer.apple.com/library/archive/documentation/3DDrawing/Conceptual/OpenGLES_ProgrammingGuide/BestPracticesforShaders/BestPracticesforShaders.html#//apple_ref/doc/uid/TP40008793-CH7-SW12)
- Brown, E. (2016). *Learning JavaScript*. O'Reilly.
- Butts, M. (2008). Multicore and Massively Parallel Platforms and Moore's Law Scalability. Silicon Valley.
- Che, S., Boyer, M., Meng, J., Tarjan, D., Sheaffer, J. W., & Skadron, K. (2008). A performance study of general-purpose applications on graphics processors using CUDA. *68*(10).
- Chen, J. (2019, May 16). *Relative Strength Index - RSI*. Retrieved December 11, 2019, from Investopedia:  
<https://www.investopedia.com/terms/r/rsi.asp>
- E-sciencecity. (2013). *What is the Grid*. Retrieved November 14, 2019, from  
<https://www.loc.gov/item/lcwa00096054/>
- Flanagan, D. (2011). *Javascript - The definitive guide*. O'Reilly & Associates.
- Geeks for Geeks. (n.d.). *Software Engineering , Iterative Waterfall Model*. Retrieved September 12, 2019, from Geeks for Geeks: <https://www.geeksforgeeks.org/software-engineering-iterative-waterfall-model/>
- GeeksforGeeks. (n.d.). *Introduction to Parallel Computing*. Retrieved September 23, 2019, from Computer Science Portal for Geeks: <https://www.geeksforgeeks.org/introduction-to-parallel-computing/>
- Ghayour, F., & Cantor, D. (2018). *Real-Time 3D Graphics with WebGL 2 (Second Edition)*. Packt Publishing.
- gpgpu.js. (2016). *gpgpu.js on github*. Retrieved May 25, 2019, from github:  
<https://github.com/teatime77/gpgpu.js>
- gpu.js. (2019). *GPU.js GPU accelerated Javascript*. Retrieved from <https://gpu.rocks/>
- gpu.js. (2019). *Gpu.js on Github , Create kernel map*. Retrieved November 25, 2019, from  
<https://github.com/gpujs/gpu.js#create-kernel-map>
- gpu.js. (2019). *GPU.JS on Github , Graphical Output*. Retrieved November 25, 2019, from Github:  
<https://github.com/gpujs/gpu.js/#graphical-output>
- gpu.js. (2019). *Gpu.js on Github, Loops*. Retrieved November 25, 2019, from Gpu.js on Github:  
<https://github.com/gpujs/gpu.js#loops>
- gpu.js. (2019). *Gpu.js on Github, Pipelining*. Retrieved November 25, 2019, from Gpu.js on Github:  
<https://github.com/gpujs/gpu.js#pipelining>
- gpu.js. (2019). *Gpu.js on Github, Types*. Retrieved November 25, 2019, from Gpu.js on Github:  
<https://github.com/gpujs/gpu.js#flattened-typed-array-support>
- gpu.js. (2019). *Gpu.js on Github. Combining Kernels*. Retrieved November 25, 2019, from Gpu.js on Github:  
<https://github.com/gpujs/gpu.js#combining-kernels>
- gpu.js. (2019). *GPU.js on Github. Issue #414*. Retrieved November 27, 2019, from GPU.js on Github :  
<https://github.com/gpujs/gpu.js/issues/414>
- Grama, A., Gupta, A., Karypis, G., & Kumar, V. (2003). *Introduction to Parallel Computing (2nd Edition)*. Pearson Education Limited.
- Hazenwinkel, M. (1994). Spline Interpolation. In *Encyclopedia of Mathematics*.

- He, K., Ge, D., & He, M. (2017). Big Data Analytics for Genomic Medicine. *International Journal of Molecular Sciences*, 18(2).
- Houston, M. (2007). *Stanford University Graphics Department*. Retrieved September 11, 2019, from Advanced Programming (GPGPU): [https://graphics.stanford.edu/~mhouston/public\\_talks/cs448-gpgpu.pdf](https://graphics.stanford.edu/~mhouston/public_talks/cs448-gpgpu.pdf)
- Intel . (2010). *Sh: A high level metaprogramming language for GPU's*. Retrieved November 13, 2019, from <http://libsh.org/>
- Jenkov, J. (2015, June 26). *Amdahls Law*. Retrieved December 11, 2019, from Jenkov Media Labs: <http://tutorials.jenkov.com/java-concurrency/amdahls-law.html>
- Johnson, M. (1991). *Superscalar Microprocessor Design*. Prentice-Hall.
- Khronos Group. (2011). *OpenCL Overview*. Retrieved October 14, 2019, from Khronos Group: <https://www.khronos.org/opencl/>
- Khronos Group. (2019). *About the Khronos Group*. Retrieved November 24 2019, from The Khronos Group Inc: <https://www.khronos.org/about/>
- Kubler, C., Bauer, L., Heinze, P., & Raczkowski, J. (2002). Realtime Textured 3D-Models for Medical Applications. *Studies in health technology and informatics*.
- Margaritis, A., & Kapiris, I. (2014). Bachelor's Thesis: Massively Parallel Computing on GPUs. Heraclion: Technical Education Institute of Crete. Retrieved from [http://nefeli.lib.teicrete.gr/browse/stef/epp/2014/KapirisIoannis,MargaritisAthanasios/attached-document-1409080386-732520-32081/MargaritisAthanasios\\_KapirisIoannis2014.pdf](http://nefeli.lib.teicrete.gr/browse/stef/epp/2014/KapirisIoannis,MargaritisAthanasios/attached-document-1409080386-732520-32081/MargaritisAthanasios_KapirisIoannis2014.pdf)
- Mitchell, C. (2019, April 9). *fibonacci Retracement Definition and Levels*. Retrieved December 11, 2019, from Investopedia: <https://www.investopedia.com/terms/f/fibonacciretracement.asp>
- Mozilla Developer Network. (2019). *Using Web Workers*. Retrieved September 12, 2019, from Mozilla Developer Network: [https://developer.mozilla.org/en-US/docs/Web/API/Web\\_Workers\\_API/Using\\_web\\_workers](https://developer.mozilla.org/en-US/docs/Web/API/Web_Workers_API/Using_web_workers)
- node.js foundation. (2019). *About Node.js*. Retrieved November 22, 2019, from Node.js: <https://nodejs.org/en/about/>
- Nvidia. (2019). *Nvidia Developer Network. High Performance Computing*. Retrieved September 17, 2019, from Direct Compute: <https://developer.nvidia.com/directcompute>
- Pivotal Data Science Labs. (2014, December 17). *Pivotal Io*. Retrieved November 13, 2019, from Distributed Deep Learning on MPP and Hadoop: <https://content.pivotal.io/blog/distributed-deep-learning-on-mpp-and-hadoop>
- Retro Delight*. (n.d.). Retrieved from Gallery of Early Computers: <https://royal.pingdom.com/retro-delight-gallery-of-early-computers-1940s-1960s/>
- Ridley, J. (2019, July 20). *AMD joins low-latency CPU to GPU interconnect group founded by Intel*. Retrieved December 11, 2019, from PCGamesN: <https://www.pcgamesn.com/amd/compute-express-link-cpu-gpu-interconnect>
- Riso, J. (2018, 07 13). *What is an MPP Database?Intro to Massively Parallel Processing*. Retrieved November 13, 2019, from Flydata: <https://www.flydata.com/blog/introduction-to-massively-parallel-processing/>
- Ristevski, B., & Chen, M. (2018). Big Data Analytics in Medicine and Healthcare. *Journal of integrative Bioinformatics*, 15(3).
- Rupp, K. (2017). *42 years of Microprocessor Trend Data*. Retrieved November 15, 2019, from <https://github.com/karlrupp/microprocessor-trend-data/tree/master/42yrs>  
<https://github.com/karlrupp/microprocessor-trend-data/tree/master/42yrs>

- Stanford University. (n.d.). *Brook GPU*. Retrieved November 14, 2019, from BrookGPU:  
<http://graphics.stanford.edu/projects/brookgpu/>
- Statista Research Department. (2016, November 27). *Statista Portal*. Retrieved from Internet of Things - number of connected devices worldwide 2015-2025: <https://www.statista.com/statistics/471264/iot-number-of-connected-devices-worldwide/>
- Steele, J. (2010, 08 09). *Data Visualization in Medicine*. Retrieved November 3, 2019, from Forbes:  
<https://www.forbes.com/2010/09/08/data-visualization-medicine-technology-autopsy.html#4aca190526cc>
- Tarditi, D., Puri, S., & Oglesby, J. (2006). Accelerator: Using Data Parallelism to Program GPUs for General-Purpose Uses.
- The Orange Duck. (n.d.). *The Orange Duck*. Retrieved from Avoiding Shader Conditionals:  
<http://theorangeduck.com/page/avoiding-shader-conditionals>
- Thimbleby, H. (2013). Technology and the Future of Healthcare. *Journal of Public Health Research*.
- turbo.js. (2019). *Turbo.js Easy GPGPU*. Retrieved from <https://turbo.js.org/>
- webcgl. (2017). *WebCLGL GPGPU Javascript Library*. Retrieved from Github:  
<https://github.com/stormcolor/webcgl>
- Webmonkeys. (2018). *Webmonkeys on github*. Retrieved May 25, 2019, from Github:  
<https://github.com/MaiaVictor/WebMonkeys>
- Xbit Labs. (2002, November 14). *Intel Pentium 4 3.06GHz CPU with Hyper-Threading Technology: Killing Two Birds with a Stone...* Retrieved October 23, 2019, from Xbit Laboratories:  
<https://web.archive.org/web/20140531105602/http://www.xbitlabs.com/articles/cpu/display/pentium4-3066.html>

## 7.2 Figures and Tables

Figure 1: Whirlwind. The first computer to use video display for output 1951 (Top left and bottom images curtesy of the Computer History Museum, MountainView CA. Top right Image Dpbsmith at English Wikipedia [CC BY-SA 3.0]).....	5
Figure 2: Arction LightningChart Volume Rendering feature (Image Curtesy of Arction Oy) .....	7
Figure 3: LightningChart JS Promotional Webpage (Image Curtesy of Arction Oy) .....	8
Figure 4: 42 Years of Microprocessor Trend Data (Rupp, 2017) .....	10
Figure 5: GPGPU Architecture in Data Centres (source: Nvidia used under Fair usage).....	11
Figure 6: Intel Hyper-Threading Technology (Source: Intel, used under fair usage).....	12
Figure 7: Matrix Multiplication Example .....	13
Figure 8: The Iterative Waterfall Process .....	20
Figure 9: Example of the Input Type for kernels at the GPU.js website (gpu.js, 2019) .....	24
Figure 10: Example of the Combining Kernels method from the gpu.js website (gpu.js, 2019) .....	26
Figure 11: Example of the Kernel Map method from the gpu.js website (gpu.js, 2019) .....	26
Figure 12: Example of the pipelining method from the gpu.js website (gpu.js, 2019).....	27
Figure 13: GPU.js graphical output code example from gpu.js GitHub page (gpu.js, 2019).....	28
Figure 14: One- and Two-Dimension Interpolator examples (By Cmglee in English Wikipedia, CC BY-SA 4.0,) .....	29

Figure 15: Heatmap example portraying rain intensity in the USA made by LightningChart (Image courtesy of Arction Oy) .....	30
Figure 16: Amdahl's Law speed gain per resources. (By Daniels220 , CC BY-SA 3.0).....	32
Figure 17: Gustafson's Law Speedup per Number of Processors (image By Peahihawaii - Own work, CC BY-SA 3.0) .....	33
Table 1: Linear Interpolator on different browsers (execution time in milliseconds) .....	35
Table 2: Neighbour Neighbour Interpolator on different browsers (execution time in milliseconds) ..	36
Table 3: Spline Interpolator on different browsers (execution time in milliseconds) .....	36
Table 4: Extrapolation routines on different browsers (execution time in milliseconds).....	37
Table 5: Heatmap Routine on different browsers (execution time in milliseconds) .....	38
Table 6: Financial Calculations on different browsers (execution time in milliseconds).....	39