Analysis and Evaluation of Embedded Graphics Solutions for Critical Systems

Bachelor's Thesis

Author: Marc Benito Bermúdez
Director: Dr. Leonidas Kosmidis
Tutor: Dr. Miquel Moretó Planas

2019-10-16

Abstract

In the safety-critical systems domain, which includes automotive, avionics and space systems, more compute power is needed to provide additional functional value and safety. In order to achieve this, new hardware architectures are considered from industry for future critical systems. One of this approaches is the use of mobile GPUs, which have excellent performance capabilities for intensive computational tasks and low-power consumption. However, current programming models for general purpose programming of GPUs like CUDA and OpenCL do not comply with the safety standards of safety critical systems. On the other hand, there are alternative programming solutions based on graphics, namely OpenGL SC 2 and Brook Auto, which are certification-friendly.

In this thesis, we perform an analysis of these safety-critical programming models for GPUs and we explore the different aspects of the development of general purpose software in them. We present our experience with porting two applications from two distinct safety-critical domains, aerospace and avionics, in several graphics-based APIs (OpenGL 2, OpenGL ES 2, OpenGL SC 2 and Brook Auto) and the evaluation of our produced versions. In terms of functionality and performance, no difference has been observed, whereas we noticed a big gap in the development complexity and productivity between pure graphics solutions and Brook Auto.

Resum

En el camp dels sistemes crítics, que inclou l'automotriu, l'aviònica i els sistemes espacials, es necessita més capacitat de computació per aportar tant valor funcional com seguretat addicional. Per aconseguir-ho, la indústria està considerant noves arquitectures per futurs sistemes crítics. Una de les possibles opcions és l'ús de targetes gràfiques mòbils, que tenen un rendiment excel·lent per tasques computacionals complexes i un baix nivell de consum. Per desgràcia, les eines actuals de desenvolupament per programació de propòsit general de targetes gràfiques com CUDA o OpenCL no compleixen amb les regulacions dels estàndards de seguretat dels sistemes crítics segurs. Per altra banda, hi ha altres solucions per programar per gràfics, com ara OpenGL SC 2 i Brook Auto, que són fàcils de certificar.

En aquest projecte, analitzem aquestes solucions per programar per targetes gràfiques i explorem els diferents aspectes del desenvolupament de programari de propòsit general amb elles. Us presentem la nostra experiència adaptant codi de dues aplicacions de dos sectors diferents de sistemes crítics, l'aviònica i els sistemes espacials, a diferents *APIs* (OpenGL 2, OpenGL ES 2, OpenGL SC 2 i Brook Auto) i l'avaluació de les versions que nosaltres hem generat. En funcionalitat i rendiment, no s'ha observat cap diferència, tot i que sí que hem notat un gran salt comparatiu en la complexitat del desenvolupament i la productivitat entre eines orientades només a sistemes gràfics i Brook Auto.

Resumen

En el campo de los sistemas críticos, que incluye el automotriz, la aviónica y los sistemas espaciales, se necesita más capacidad de computación para aportar tanto valor funcional como seguridad adicional. Para conseguirlo, la industria está considerando nuevas arquitecturas para futuros sistemas críticos. Una de las posibles opciones es la utilización de tarjetas gráficas móviles, que tienen un excelente rendimiento para tareas computacionales complejas y un bajo nivel de consumo. Por desgracia, las actuales herramientas de desarrollo para programación de propósito general de tarjetas gráficas como CUDA o OpenCL no cumplen con las regulaciones de los estándares de seguridad de los sistemas críticos seguros. Además, hay otras soluciones para programar para gráficos, como OpenGL SC 2 y Brook Auto, que son fáciles de certificar.

En este proyecto, analizamos estas soluciones para programar para tarjetas gráficas i exploramos los diferentes aspectos del desarrollo del software de propósito general para ellas. Os presentamos nuestra experiencia adaptando código de dos aplicaciones de dos sectores diferentes de sistemas críticos, la aviónica y los sistemas espaciales, a diferentes *APIs* (OpenGL 2, OpenGL ES 2, OpenGL SC 2 y Brook Auto) i la evaluación de las versiones que nosotros hemos generado. En funcionalidad y rendimiento, no se ha observado ninguna diferencia, aunque sí que hemos notado un gran salto comparativo en la complejidad del desarrollo y la productividad entre herramientas orientadas solo a sistemas gráficos y Brook Auto.

Acknowledgements

This final year project has been performed at the Computer Architecture/Operating System Interface (CAOS) group in the Barcelona Supercomputing Center (BSC) under an Internship agreement between UPC and BSC (Practicas de Empresa). It has received financial support, equipment and software from the GPU4S (GPU for Space) project funded by the European Space Agency (ESA) and the collaboration between Airbus Defence and Space at Getafe, CoreAVI and BSC. We would like to thank Airbus Defence and Space for kindly providing the demo Vulkan application used in this project.

Furthermore, I want to thank my family and specially my parents, which have been giving me support from the beginning of this degree until the very end, helping me in whatever they could.

To Mónica, who has been with me through all this project's journey and had to bear my moments of frustration.

And finally, to my friends from different associations in the UPC at the Omega Building, mainly from l'Oasi and Distorsió at floor 1 office 103, who have helped me in many ways and are one of the biggest reasons I persevered until the end.

Contents

1 Context

Nowadays, computing systems are introduced in more and more different systems with distinct requirements, even in domains that in the past relied only on mechanical/hydraulic systems. The common characteristic in all of them is the addition of more functionality which increases their features and make everyday life easier and in some cases safer.

In this diploma thesis we focus on a particular subset of computing systems which are known as *critical* systems. Failures in those systems have to be contained since they can potentially result in catastrophic consequences. In order to serve this purpose, critical systems have to be designed in accordance with safety regulations, whose compliance has to be reviewed before the systems are deployed for operation. This essential process is called *certification*. In this work, we analyse and evaluate already certified and/or soon to be certified software and hardware solutions based on graphics, which have as a purpose to increase the performance of existing critical systems.

2 Introduction

Critical systems include computer, electronic or electromechanical systems which require a high degree of reliability because their failure can have severe human or economic consequences. Such systems vary from a protection system in a chemical plant to a customer accounting system in a bank. The former category is known as *safety* critical, since a failure can affect the safety of humans, while the latter as *mission* critical, since a malfunction can affect business operations. In this thesis we will focus on safety critical systems like the ones used in automotive, avionics and aerospace, which have similar requirements and safety standards. We will analyse the state of the art and will identify possible solutions to actual shortcomings.

Nowadays, all companies developing products in the safety critical systems domain are incorporating more functionalities in order to increase their competitive advantage, as well as to make transportation more effective and safer. For example, in the automotive domain there is a lot of research going on by all major automotive companies, towards the development of autonomous vehicles. In other domains like in space, new scientific missions are processing larger amount of data due to higher resolutions of their observation instruments. Moreover, in avionics more and more functionalities are now computer assisted, facilitating pilot's efforts to navigate even in rough climate conditions.

As it is obvious, all these functionalities can only be provided by powerful hardware components. However, traditional hardware for critical systems are simpler compared to desktop and high performance computing, due to the need to comply with safety regulations in these domains, as we explain in deeper detail

later in the document. For this reason, nowadays critical domains are looking to higher performance solutions, such as Graphics Processing Units (GPUs).

This project is part of a larger collaboration between the CAOS (Computer Architecture/Operating Systems Interface) group at Barcelona Supercomputing Center (BSC) and large companies in the above domains.

In particular, our work has contributed in the following research projects:

- GPU4S (GPU for Space): This project which is funded by the European Space Agency (ESA) and coordinated by the thesis director, Dr. Leonidas Kosmidis, explores the possibility of using GPUs in the space domain. Airbus Defense and Space (ADS) Toulouse, which is the main European satellite provider is also participating in the project, providing insights about the aerospace on-board software for current and future missions. So far the project has only focused on benchmarking of high-end GPUs, which only support OpenCL or CUDA. For this reason, a set of benchmarks, called GPU4S benchmark suite, written in these languages has been developed, resembling essential algorithmic basic blocks found in representative space software. However, the aforementioned programming models cannot be certified with the existing safety standards, due to the fact that are based on pointers and dynamic memory allocation, two features that are strictly forbidden in safety critical systems. Nonetheless, the existing safety certified graphics based APIs, can be used on lower-end devices, since they are subsets of their defacto programming model, OpenGL ES 2. In this thesis we have performed a preliminary analysis of how these GPGPU benchmarks written in CUDA or OpenCL can be ported in this universal graphics programming model for embedded graphics OpenGL ES 2 and we have performed the porting of an important benchmark of the suite, the Fast Fourier Transform (FFT). In this way, this thesis contributed to extend the benchmarking of embedded GPUs towards low-end platforms.

- ADS Getafe: this project is a collaboration between the CAOS group at BSC and the Airbus Defence and Space (ADS) Avionics division based in Getafe, in the Madrid metropolitan area. In this project, which is the first project to explore the use of GPUs and most importantly their certification in avionics, BSC provides its expertise in both GPU hardware understanding and programming as well as in development of tooling for certification. As a part of this collaboration, we got access to a GPU used in avionics and a safety-certified graphics driver and API (OpenGL SC 2 [14] from the project partner CoreAVI which is a supplier of safety critical graphics drivers and they are leading the definition on the safety critical APIs within the Khronos standardisation committee. Moreover, we got access to a demo application developed by Airbus, which resembles a safety critical prototype application and it is written in a non-certified graphics API such as Vulkan [15]. In this thesis, we studied the differences between non-certified graphics APIs and certified ones and ported the application

to two safety certified/certifiable APIs, OpenGL SC 2 and Brook Auto. Then we have also performed a comparison between the different versions in terms of performance and programmability.

3 Stakeholders

This project had different people which were directly or indirectly involved:

- Analyser: I performed the analysis of the potential solutions for safety certified graphics APIs and their differences with general purpose ones. Next, I prepared the software needed to perform the different evaluations as required. Although the initial plan was to work on only one of the two software options and projects presented in the previous Section, I have worked with both, and I have used multiple graphics APIs. Finally, I performed the evaluation and provided an analysis of the differences between safety certified and non-safety certified solutions.
- Director and Tutor: The director played an important role on the project. His role was to guide and supervise this thesis to reach this project's objective, including my necessary training. The Tutor was the necessary relation with the university, because the director at the time was not associated with the UPC. Moreover, the director is an expert in the embedded GPU domain and certification for safety critical systems, so he was aware of all the objectives we needed to achieve. Additionally, the outcome of this thesis was essential for the projects he coordinates, which guaranteed close interaction and engagement from his side.
- Barcelona Supercomputing Center: The BSC has provided me the necessary hardware and software as well as funding and a place to work so that I could properly develop my project. In exchange, my work has contributed and complemented the tasks in the above projects in which BSC is involved.
- Beneficiaries: The end-user companies which are interested in upgrading their critical systems as well as companies in the supply chain. For example in the above projects they are the ESA and ADS from both the space and avionics divisions, as well as CoreAVI which provide certified graphics drivers. In addition, other companies in safety critical domains like in automotive, can also benefit from the findings of our work once it is published in a scientific publication which is scheduled to be submitted after the thesis defence. Moreover, the companies might use our solutions or methodology to perform similar benchmarking on other GPUs or safety certified APIs.

4 State of the art

Avionics is the application of electronics into the aviation. These systems can range from communications to navigation, meaning there are different systems taking place in a plane at the same time and each one can have different criticality. There are 4 levels of criticality ranging from DAL-A (highest) to DAL-D (lower criticality). In this thesis we focus on DAL-A solutions. The overall code size in these systems is constantly increasing as we can see in Figure 1 extracted from [1], requiring an equivalent increase in processing power. A similar trend is also observed in the space domain [11]. This has forced these domains to start exploring new hardware platforms moving away from single core CPUs to more powerful hardware. In our particular project this new hardware is the GPU and our work provides necessary feedback on whether they are sufficient for compute-heavy tasks, as well as their trade-offs in programmability and certification.

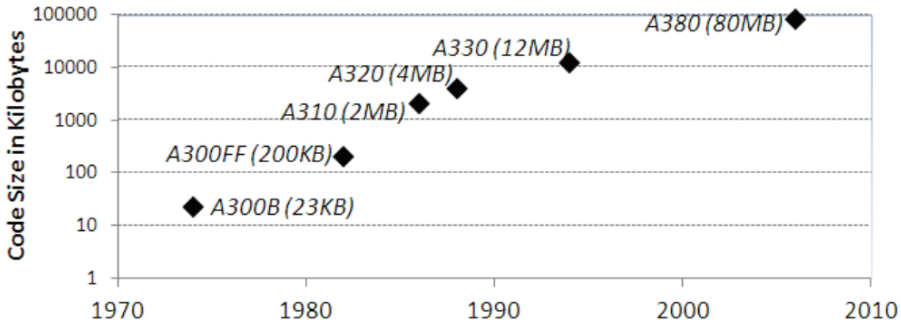


Figure 1: Code size evolution for safety critical systems in Airbus aircraft [1].

4.1 Graphics Processing Units (GPUs)

Graphics accelerators have been originally designed to handle computation intensive graphics workloads, for use in displays. However, in the last decade they have obtained general purpose programming capabilities, achieving very high performance processing rates for massively parallel tasks. For this reason, graphics accelerators as high-performance, high-consumption hardware, are mainly found in desktop or high performance computing environments.

However, safety critical domains are known to be very conservative in the adoption of new technologies like this. For example, current avionics systems use GPUs only for displaying information in the cockpit's screens, but not for general purpose computations. On the other hand, in the space domain, which has very strict consumption limitations, GPUs were discarded so far. In the automotive sector, GPUs are also used for display tasks and are currently explored

for compute tasks to enable autonomous driving in prototype cars, however the compute programming models such as CUDA and OpenCL are violating programming guidelines for safety critical systems, hindering their certification [4]. Since smartphones, mobile/embedded Graphics Processing Units with high-performance but low-consumption have appeared. These mobile GPUs are behind the ones usually found in desktop environments in terms of performance, but the low-consumption characteristic makes them especially interesting in avionics and other critical domains, where heat dissipation cannot be guaranteed through fans, because they might break due to excessive vibrations.

In [2], a study is made about different types of GPUs and their possible use in space. Though avionics are more strict in certification but less constrained in size, weight and Power (SWaP) as space critical systems, they share similar needs, so we have used this detailed information to give an idea of why and what type of embedded systems are chosen for this case.

Unlike in desktop GPUs, which are only from NVIDIA and AMD (and to a lesser extent, Intel), there are several big embedded GPU IP providers like ARM, Imagination Technologies or Adreno. These GPUs are ASICs (Application-Specific Integrated Circuits), typically as part of a larger SoC (System-on-a-Chip), which usually contains CPUs and other devices, for example DSPs, embedded memory, etc. We can divide them further in two types, depending on their computing power.

Low-end GPUs: Some of the advantages of this class of embedded GPUs are simpler architectural design and lower power consumption. Also, we can found them in single-board computers, set-top-boxes, tv-sets, automotive systems and FPGAs. Their widespread nature and older designs means that potential bugs in their design are already identified and solved or work arounds have been found, which makes them more reliable in next iterations and therefore more appropriate for safety critical systems. These GPUs support only graphics APIs, such as OpenGL ES 2.

High-end GPUs: The main advantage about this type of embedded GPUs is their support of GPGPU APIs like OpenCL and the higher computational power, although they are less power efficient. Because their architecture is newer, they are more prone to have some design errors, which can impact the reliability in a system. Another issue is the decreasing interest from manufacturers to support OpenCL because of its little usage. For example, Google has officially dropped support for OpenCL in their Android operating system, which is only used in certain flagship mobile phones models from companies such as Samsung or Sony, to accelerate custom versions of their applications.

Low-end and high-end GPUs offer diverse advantages and trade-offs between them, so it's in the scope of the particular application to decide one or the other. In our case, we decided to use low-end ones mainly for the low power

consumption that high-end GPUs cannot deliver, as well as because current safety certified programming models better suit low-end GPUs.

4.2 Safety Critical Application Programming Interfaces

As we mentioned before, mobile GPUs have one major shortcoming if we want to use them as GPGPU (General Purpose GPU), and that is the incompatibility with majority of low level programming models such as OpenCL and CUDA. Every mobile GPU supports graphics operations, compatible with at least the OpenGL ES 2 API, which cannot be used to implement OpenCL for a number of reasons explained in previous work, as in [3].

Another problem for avionics and other critical systems are the certification requirements and safety standards. For example, the safety standard used in automotive, the ISO26262 [12], which is mandatory in automotive systems. CUDA and OpenCL violates this specifically because their utilization of pointers and dynamic memory allocation. As we can see in [4], there are five rules violated by every program written in those APIs: restricted use of pointers, no dynamic memory allocation, static verification of program properties, resilience to faults and fault propagation.

4.2.1 Brook Auto

The solution proposed in [4] for this problems is using an existing API and adapting it to the majority of the mobile GPU market and make it certification friendly. In this same article [4], this API is Brook, an open source language developed at Stanford University that is the base for NVIDIA's CUDA and commercially exploited in the past by their rival, too, ATI (currenly AMD).

As the paper says "[...]Brook abstracts the programmer from the graphics complexities so that she/he has only to identify the algorithmic part to be offloaded (kernel) and its input/output.[...]". With this in mind and using a subset that meets the certification requirements and safety standards mentioned before, Brook Auto is born.

Note that other domains have their own standards. For example in avionics we find DO178 [13], which however has the same restrictions identified above as ISO26262 [12]. For this reason, it is relatively easy to adopt a certified solution from one critical domain to another, such as Brook Auto, or the OpenGL SC we examine next.

4.2.2 OpenGL SC

Another solution is OpenGL SC 2 [14], a subset of OpenGL ES 2 that is designed primarily for safety critical graphics hardware running on embedded de-

vices. This API has been designed to share the same interests with avionics and automotive requirements for safety critical software. As part of this thesis, we perform an analysis review the standard, as well as it earlier version (OpenGL SC 1.0) which is still used in critical systems and identify which are the differences with OpenGL ES 2 and which are the limitations in programmability and in performance.

4.2.3 Vulkan

Vulkan [15] is a new redesign from the ground-up of a graphics API for graphic accelerators which gives total control to the developers of the performance, efficiency, and capabilities of GPUs. Even though there are few embedded GPUs that support it, some work has been made in the domain of critical systems.

As with the change from OpenGL to Vulkan with recent GPUs, Vulkan SC [16] is the focus to use this modern API to execute high computational tasks in more recent embedded systems although it is still in development.

VkCore SC [39] is a safety critical API based of a subset of Vulkan 1.0 to address safety critical concerns for avionics and automotive platforms, as their web-page describes. It is worth to note that the definition of this standard is currently under discussion. Although we initially envisioned to use it in this project by getting access to an early version of that API through BSC's collaboration with CoreAVI, delays in the release didn't allow for this to happen.

5 Scope

5.1 Objectives

The main objective of this project was to demonstrate how a safety-critical prototype GPGPU application written in a non-certifiable programming model such as OpenCL or CUDA can be converted to use a safety-critical graphics API. We identified what changes are required and what is the impact in performance and programmer productivity.

Because there are several API possibilities in using low-end embedded GPUs, we used several cases with the different tools mentioned before. Moreover, we list the equipment provided by BSC and Airbus which were used in this study.

5.2 Resources

There were several resources used in the entirety of the project, as well as ones used only temporary.

- Dell Latitude 7490[17].
- NVIDIA Corporation GP107 [GeForce GTX 1050 Ti] (rev a1) [18].
- NVIDIA Corporation GP102 [GeForce GTX 1080 Ti] (rev a1) [19].
- AMD Radeon E8860 [21].
- HiKey 970, featuring an ARM Mali G72 embedded GPU [24].
- NVIDIA Xavier [23]

The first one was the computer that I worked on, which is provided by BSC.

There were two NVIDIA desktop graphics cards entirely at our disposal to do testing, and to get accustomed to some methodologies of GPGPU programming. The AMD card is the main test unit we have for avionics, on which the avionics case study from Airbus was designed.

We also got access to two NVIDIA embedded GPUs, the Jetson Xavier [23] and the NVIDIA Jetson TX2 [22], which are platforms developed for automotive use, and in particular to enable autonomous driving. The former has also been evaluated in the GPU4S project, as well as the HiKey 970 [24], and as such they are good candidates to compare our evaluation on both graphics and general purpose programming modes on the same platform.

The main platforms used for the actual software development of this thesis were based on the availability of the selected safety certified APIs (Brook Auto and OpenGL SC 2) and the baseline software we ported (avionics application written in Vulkan). In particular we used the laptop for the development, the AMD GPU for the avionics application and the Xavier for the GPU4S benchmark porting.

5.3 Methodology

We divided our methodology in two important, complementary parts. The first one was completely theoretical but absolutely necessary, since it involved studying the state of the art in the use of GPUs in critical domains. In particular, we will performed an analysis of the existing graphics solutions for safety critical systems out of the ones mentioned before, like OpenGL SC or Brook Auto. In fact, part of this analysis had already been started during the GEP course, in order to produce a first draft of the document, by studying all the related scientific literature in the form of publications in top tier conferences, which are included in the bibliography. This also included works not related to GPUs, but about software and certification for critical systems from different domains such as automotive, avionics and space, including subsets of their respective standards such as ISO26262 [12] and DO178 [13]. A very important contribution for this was the training I obtained at the CAOS group during my internship. In fact I have attended the PUMPS summer school organised by UPC and BSC on the programmability of NVIDIA GPUs using CUDA, a CAOS group training course on ISO26262 and several weekly presentation from CAOS members on safety-critical systems.

Our initial study presented in the State-of-the-art Section was extended to cover in more detail the aforementioned solutions, like for example the standard of the OpenGL SC 2.

As a second part in our methodology, we studied how to port the required software in the safety critical API we analysed, OpenGL SC 2 and Brook Auto. We also compared our implementations with the same software in non-safety critical APIs and we assessed the potential cost in performance or programmability which comes at expense of certifiability.

The methodology we followed for the software development of the applications which we ported in the project was Scrum. We had daily meetings with the thesis director discussing the project progress and the obstacles which appeared. This allowed us to have a clear understanding of the short term goals and the steps required in order to be achieved. Moreover it provided the flexibility to adapt our development in an agile manner.

Finally, at the end of the project, as we initially planned we will pursue the publication of our results in a scientific publication in the domain of critical systems.

6 Activities

In this section we provide a list of the activities which we performed during the project. Their duration and their distribution in the project timeline as well as their dependencies are shown in the Gantt chart provided in the Appendix.

6.1 Project bootstrapping

Whenever you start a new project, the first thing you need to do is looking at the state of the art and familiarize with its environment. This was especially true in our case, since computing is a very wide area and the particular area of a new project might not fall within one's expertise, especially when the new area is a niche market, as safety critical systems.

For this reason, this project has started by understanding the safety critical domain and the awareness of the problem, which was presented in the previous sections. In the References section we have identified a list of different scientific papers ([1], [2], [3], [4], [5], [6], [7], [8], [9], [10]) that introduced us to the characteristics of critical systems and gave us some insight on the issues which should be addressed, from 3 different safety critical domains, namely the avionics, aerospace and automotive.

6.2 Studying functional safety standards

From the beginning of the project until the very end, we were studying the documentation about the most important critical safety standards: ISO26262 [12] and DO178 [13]. Note that the size of these standards exceeds thousands of pages and they are written in such a way that guidelines are provided as opposed to particular solutions. For this reason, they are open to interpretation of the particular avionics or automotive computing system in which they are applied.

In order to understand their specifications and characteristics and familiarize with these standards, this task was almost as long as the project itself. Each week we used some time to understand their singularities and to take them to heart. Moreover, the training course I attended in the CAOS group on ISO 26262 eased significantly my work in this task.

6.3 Studying and Experimenting with the state-of-the-art SC APIs

In this task, we had to study state of the art so we could understand what are the advancements in this field and what the known problems or shortcomings are. We read documents about embedded graphics cards with GPGPU

programming based on safety critical graphics APIs, critical systems and how they are applied to the automotive domain and avionics to ensure the code in execution is compliant with the ISO26262 [12] and the DO178 [13] standards respectively. Once we knew what has been done in these areas of research, we began to approach all the necessary tools to start working on the project.

Because in the analysis of our study we considered several different safety critical graphics APIs, most of the work here was be concurrent. In fact we ended up iterating between many more APIs which were initially envisioned. In particular, we have experimented with desktop OpenGL 1.3 (the first programmable version for fragment shaders in terms of GPU assembly language), the desktop OpenGL 2 (the first version programmable in a high level GPU language, GLSL), OpenGL ES 2, OpenGL SC 2 [14], Brook Auto [4] and Vulkan [15]. The work in this task involved reading documentation about each of the tools, get access to existing different tests and benchmarks and familiarise ourselves by executing them. If for a given API no benchmark was available, the familiarisation required building small tests to demonstrate their capabilities and limitations.

Based on this analysis, we documented the differences between the examined safety critical APIs and their differences from their non-critical counterparts ie. non SC versions, as well as the evolution of the APIs. The outcome of this task is provided in Section 10.

6.4 Application/Benchmark Porting

Once we selected the APIs for our target system, we ported the required software into that safety critical API. Since we worked in two projects, depending on the selected API, the starting point was written in a non-safety critical API variant, such as OpenGL 1.3 in the FFT benchmark of the GPU4S benchmark suite and Vulkan in the prototype safety critical application from Airbus.

6.5 Evaluation and Comparison

When the porting was over, we evaluated the implementation of the software in the selected API. This allowed us to compare our results with implementations of the same software in non-safety critical APIs, showing the trade-off in programmability in these specific conditions.

6.6 Possible extra work

The previous tasks were sufficient for the successful completion of this project. However, we initially planned that if we reached this point ahead of time to perform more work if possible. In our case, the additional, optional work which

we performed was to implement both projects instead of only one, and study provide implementation in more than one safety critical API, namely OpenGL SC 2 and Brook Auto. This required an excessive amount of optional work which was successfuly implemented and added great value in both projects.

6.7 Obstabcles, Risks and their Mitigation

Because in this project we needed the collaboration with big business companies like Airbus and CoreAVI, we identified in our planning that there was a potential risk that some work involving the AMD Radeon E9171 MCM [20], the E8860 [21] and the safety critical application prototype could be delayed. Moreover, CoreAVI which provides the safety critical drivers and is driving the Khronos SC standardisation process, including the upcoming Vulkan SC, would provide the first alpha release of the safety critical driver for this graphics API. However, it wasn't possible to get access to this alpha release on time for inclusion in the project. As we initially planned we continued with a backup plan, due to the inherent redundancy built in this project. In particular, since our work contributed in two research projects, any delay or potential obstacle which prevented us to complete the work in one project, could be compensated with more work in the other one.

Based on our preparation for the worst case, we achieved to contribute in both research projects within a single bachelor's thesis.

In particular, since the baseline software of the avionics case study was not available at the beginning of this thesis, we started with the platforms (Xavier) and a benchmark (FFT) used in the GPU4S. However, when we faced significant difficulties towards the end of this task, the avionics application became available and we were able to switch in that, while we could benefit from the obtained knowledge and software infrastructure we built in the first hald of this thesis.

Moreover, since the breadth of the project was flexible, it allowed for such project adaptations without compromising the goals and the objectives of the project. In particular, not only we contributed in both projects, but also we were able to port the avionics application in more than one safety critical APIs, a task which was initially planned as optional.

7 Economic management

7.1 Gantt chart

As already mentioned in the the previous section, in the Appendix we present a Gantt chart to visualize how much time each section lasted, which was directly related to the total budget of the project and how it is decomposed.

7.2 Budget

In this subsection we will examine the budget which was required to finish the project under the previously explained conditions.

Human resource	Dedication (hours)	Price/hour (euros/hour)	Total cost (euros)
Project Director	50	50	2500
Project Tutor	10	50	500
Support team	100	20	2000
Internship Developer	735	8	5880
TOTAL	845	-	10880

Table 1: Human Resources and person month allocations

This budget consists of a Human resource table (table 1), which specifies the time and wages for each of the members implicated in the project.

The Total budget table (Table 2), mentions the needed hardware and software to develop the project, the cost of the developer and other human resources, the expected hours required to develop each section of the project and the possible contingency we might have.

As we mentioned before, we accounted for some potential delay coming from software and hardware from our industrial partners, which we utilised for the duration of the project. Our initial provisioning included the possibility to add an extra month of work until the presentation date in case of a delay, which added further cost in the budget allocation. Our provisioning for an additional 17% contingency value to make sure the possible problems mentioned beforehand and other that may arise are within our expectations was proven enough, since despite the expected delays we were able to work on both projects.

Resource	Cost/unit (euros)	Lifespan (years)	Amortization (euros/hour)	Total cost (euros)
Project bootstrapping				**Time: 35 Hours**
Dell Latitude 7490 [17]	1699	5	0.038	27.93
Internship Developer	-	-	8	280
Studying and experimenting with the state of the art SC APIs				**Time: 240 Hours**
Dell Latitude 7490 [17]	1699	5	0.038	8.74
AMD Radeon E9171 MCM [20]	212	5	0.004	0.92
AMD Radeon E8860 [21]	260	5	0.006	1.38
NVIDIA GTX 1080 Ti [19]	789	5	0.018	4.14
NVIDIA GTX 1050 Ti [18]	182	5	0.004	0.92
NVIDIA Jetson TX2 Developer Kit [22]	517	5	0.011	0.06
NVIDIA Jetson AGX Xavier Developer Kit [23]	1199	5	0.027	0.14
Hikey 970 [24]	265	5	0.006	1.38
Internship Developer	-	-	8	1920
Studying functional safety standards				**Time: 110 Hours**
Dell Latitude 7490 [17]	1699	5	0.038	3.80
Internship Developer	-	-	8	880
Application/Benchmark Porting				**Time: 210 Hours**
Dell Latitude 7490 [17]	1699	5	0.038	7.60
AMD Radeon E9171 MCM [20]	212	5	0.004	0.80
AMD Radeon E8860 [21]	260	5	0.006	1.20
NVIDIA GTX 1080 Ti [19]	789	5	0.018	3.60
NVIDIA GTX 1050 Ti [18]	182	5	0.004	0.80
NVIDIA Jetson TX2 Developer Kit [22]	517	5	0.011	0.06
NVIDIA Jetson AGX Xavier Developer Kit [23]	1199	5	0.027	0.14
Hikey 970 [24]	265	5	0.006	1.38
Internship Developer	-	-	8	1680
Evaluation and Comparison				**Time: 140 Hours**
Dell Latitude 7490 [17]	1699	5	0.038	5.32
AMD Radeon E9171 MCM [20]	212	5	0.004	0.56
AMD Radeon E8860 [21]	260	5	0.006	0.84
NVIDIA GTX 1080 Ti [19]	789	5	0.018	2.52
NVIDIA GTX 1050 Ti [18]	182	5	0.004	0.56
NVIDIA Jetson TX2 Developer Kit [22]	517	5	0.011	0.06
NVIDIA Jetson AGX Xavier Developer Kit [23]	1199	5	0.027	0.14
Hikey 970 [24]	265	5	0.006	1.38
Internship Developer	-	-	8	1120
Project Director	-	-	50	2500
Project Tutor	-	-	50	500
Support team	-	-	20	2000
Aggregated total	Time: **735 Hours**			**10956.37**
Contingency	Percentage: **17%**			**1862.58**
TOTAL	Time: **735 Hours**			**12818.95**

Table 2: Hardware and software budget

8 Sustainability analysis

In this section we perform an analysis of the sustainability of the project which implies different points to take into account. We use these points to explain the characteristics of our project from the environmental, economic and social perspectives.

8.1 Environmental analysis

From an environmental standpoint, our project targets low consumption because the limitations imposed by their end-products. As our project deals with automotive and aerospace systems, we have to consider the restrictions of using certain hardware because they have a low finite energy source i.e. batteries, oil or solar panels, and in addition to the energy consumption of the computing element they have to take also other energy related aspects such as their weight and their cooling.

For this reason, the ecological footprint is limited in terms of the energy consumption of each computing element. However, in order to manufacture these low consumption boards and their energy sources we need to gather the required resources, some of which are extracted in countries with limited wealth and poor health conditions. Even though we cannot do anything about those work conditions, we keep in mind the hardware's origin and try to maximize its usage. We also keep in mind that this hardware will be installed in vehicles with oil consumption, which have a strong environmental impact. Making the correct choice in terms of performance then is very important to reduce the amount of fossil fuel required to give electricity to the various electronic components.

For prototyping our solutions, we had to execute code and test different approaches to the solution in desktop GPUs, such as the Nvidia GTX 1080Ti [19] and 1050Ti [18]. These GPUs are high performance but they also have a high energy consumption in the range of hundreds of Watts, beyond the power limit of tenths of Watts of electronic components used in critical transportation systems. However, despite their high power consumption of these desktop graphic cards, these are usually quite energy efficient (measuring it in performance per Watt) as shown by the top super computers in the Green 500 list [25]. Nonetheless, we expect that the total consumption generated in this tests does not surpass the future executed software using this work as a guideline for choosing the tools to implement it. In particular, the deployed solutions will be executed in embedded GPU boards, such as AMD Radeon E9171 MCM [20] and E8860 [21] provided by an avionics company or the GPUs used in the GPU4S project, as they were listed in detail in Section 5.2.

Regarding these embedded GPUs we used in this project, it is worth to note that they are more efficient than CPUs with same power consumption because

current software cannot usually fully exploit the multiple CPU cores. In addition to power consumption this is also reflected in the pure computational power that a GPU can provide performance compared to multiple CPUs and which is needed in order to provide new functionalities as we described in earlier sections. Furthermore, a single GPU may weight less than multiple independent, interconnected CPUs as it is the case of the state-of-the-art in critical systems, which implies less fuel consumption and, in extension produce less CO2 emissions.

Moreover, in these critical systems we cannot use active cooling (a fan exhausting the heat generated from the CPU and/or GPU) because they can break from vibrations generated by different inside or outside elements during their operation. For this reason, the systems need to be low power by design, and embedded GPUs are the perfect candidates, since they are designed with similar goals eg. no active cooling can be used a mobile phone, neither it is needed to be using a heavy battery.

Resource	Consumption (Watt)	CO2 Emission (CO2 kg/hour)
Dell Latitude 7490 [17]	60	69.3
AMD Radeon E9171 MCM [20]	35	40.4
AMD Radeon E8860 [21]	35	40.4
NVIDIA GTX 1080 Ti [19]	150	173.3
NVIDIA GTX 1050 Ti [18]	70	80.9
NVIDIA Jetson AGX Xavier Developer Kit [23]	10(*)/30(*)	11.6(*)/34.7(*)
Hikey 970 [24]	10	11.6

(*) Depending the consumption mode

Table 3: Power consumption table

Table 3 represents the consumption and the CO2 emissions for each of the hardware used in the project. The CO2 emission is converted with the data extracted form [26], which explains the emissions of CO2 generated from the Spanish electrical mix, which is 321g CO2/kWh.

8.2 Economic analysis

In the previous section we presented a table with the project's estimated budget, which mainly involved a computer to work on and the graphic cards used to test the graphics APIs. These resources are not cheap and need an significant investment so we (and the other people in our department and BSC) can use them. In our case, there were mainly three people involved in the project, but we also needed to communicate with other persons working in other projects (within or outside BSC).

However, apart from the analysis of the cost of the system, our project has the potential to generate additional economic value, if it is succeeded to increase the use of embedded GPUs in critical systems. In that case, as we describe in greater detail in the social analysis section, our project has the potential to create new jobs in the critical markets. In addition, our stakeholders, the companies which were described in Section 3 will be able to increase their product sales in the critical markets. Since avionics, automotive and aerospace are multi-billion markets, the economic benefits from the use of embedded GPUs can be substantial both for the companies involved, as well as for the economy in general, including the workers. In particular, although the electronic systems ie. CPUs or microcontrollers used in these systems only represent a tiny fraction of their cost, the inclusion of embedded GPUs in them can lead to new opportunities that were not possible until now eg. increasing their autonomy. Self-driving cars or unmanned aerial vehicles can create a whole new type of products in these markets, with immense impact in economy.

8.3 Social analysis

As a student and author, this project has changed my point of view of automotive and avionics. Before I started this project, I knew they were an important part of our every day life but I was not aware that they were based so much on computing. Moreover, I was familiar with GPUs and graphics but I didn't know that big companies like major players in avionics were exploring their use, in addition to automotive companies that are publicly advertising such efforts. Also, thanks to the study of the state of the art and the training I received during my internship, I now appreciate all the requirements and limitations of these systems and the difficulties of putting everything together respecting all the critical safety standards like ISO26262 [12] and DO178 [13].

Our work contributes towards GPUs to be used in critical systems to provide the missing performance needed for autonomous operations for projects such as self-driving cars or Unmanned Aerial Vehicles (UVA). This way, this project can open the door to achieve more efficient transportation with an impact resulting in less traffic, less accidents or less time looking for parking which can result to a upgrade in the quality of life of the society. Also, because these graphic cards have to be programmed, new jobs are arising, creating new workplaces for GPU programmers, who add new uses and functionalities or software upgrades.

In the case of the code used in critical domains, most software is proprietary because the tools qualified to do the certification and control of the execution are expensive to develop. Also, in the instance of closed source, in case a problem with the system arises the company who developed the code can be held responsible, whereas open source software licenses exclude liability. For example, our collaborator and stakeholder CoreAVI's implementation of the driver for OpenGL SC 2 [14] is proprietary for these reasons. However, the specification of this API is open, which enables competitors to work on compatible solutions.

On the other hand, some of the APIs we explored in this project, like Brook Auto [4] are open source.

We think the results of the project will help to increase the analysed APIs usage. As a consequence, if OpenGL SC 2 is selected as a solution, it can help to boost both CoreAVI and their competitor sales in the critical markets and therefore get more contracts (and clients) with avionics or car manufacturers. On the other hand, if Brook Auto is the API of choice, it can allow smaller players to join the critical domain, since they will be able to develop software without relying on expensive licenses of proprietary software.

9 The project

In this Section we explain in detail each stage of the development of the software we ported in this project.

As we explained earlier in the description of the tasks, we first started with reading and experimenting with the different APIs in order to become familiar with them. Next we started working on the benchmark from the GPU4S project which we got very close to the completion, before moving to the ADS project in the middle of the thesis. In the latter one, not only we completed the porting to OpenGL SC 2, but we also completed the optional part which was to port to another API, Brook Auto.

Below, we provide more details about each of these steps.

9.1 First steps

Before starting working on this thesis, my background was on graphics but only with the desktop OpenGL 3.3, which is taught during the undergraduate courses at the FIB/UPC like IDI (Interaction and Interface Design). However, I didn't have any experience with OpenGL ES 2, which is the graphics API which is the predecessor of OpenGL SC 2 and is also used within Brook Auto. As a consequence, my first programming experimentation in the project started with familiarisation with OpenGL ES 2. In particular, the goal we set with the director of the thesis was to be able to reproduce the content of his publication [3] based only on the paper description, which was the first one to apply general purpose computations on low-end mobile GPUs with OpenGL ES 2.

Thanks to my previous knowledge in desktop OpenGL, my job was easier towards the embedded programming model. The first step was start reading about OpenGL ES 2 and starting working with tutorials in order to learn the basics which are required to perform general-purpose computations with OpenGL ES 2. After that, the idea was to understand the specifics of OpenGL SC 2 and make the necessary changes to the code.

The tutorial we used as a boilerplate code in our development was from the public repository [28] from the OpenGL ES 2 Programming Guide [27]. This programming guide helped to familiarize with the specifics of this version so the project could start as quickly as possible.

The first step was to compile everything and test that a triangle appeared on screen, which was one of the examples given by the programming guide. This was important to test the OpenGL ES 2 compatibility in the laptop and in the Nvidia Xavier, where all the work was done.

Then, we started modifying the code so we could execute some general purpose computation on the GPU. With this objective in mind, another example from

this book was used, which was using a simple texture. After that, the addition of the two textures was implemented in the fragment shader.

This practice helped us understand that, to use GPU as a general purpose programming hardware, the textures are used as storage for both input and output and we read the information from this output texture directly from the memory in the GPU. This allowed us to get the output without requiring a display and therefore read the exact pixel values, which contain the result of the general purpose computation we are implemeting in the shader.

Once two textures and their addition were implemented, a case study which is essentially a vector addition of two matrices, I faced a problem which appeared when running experiments with large textures, like thousand pixels in width and height. It turned out that in embedded platforms like the Xavier, it is not possible to use large statically allocated buffers at the CPU side for initialising textures. The problem did not appear at compilation but at runtime, when the driver was trying to access this data. Instead dynamic memory allocation using mallocs solved this issue.

After solving this problem, I completely understood the reason why general purpose computations on low-end GPUs were not possible until the appearance of [3]. Because CPUs and GPUs do not encode equally the values of the data, a function to convert to one another was needed. This functions translates a CPU encoding of a type of variable to the GPU one so the accelerator can use it in its operations and convert it back again so we can manipulate them properly.

That is the reason why we implemented texture additions with different data types in them. These operations could only be done with the same type for each of the operands and the output. These types are unsigned and signed character, unsigned and signed integer and double floating point. Since the complexity of implementing these encoding and decoding operations from scratch would require extensive effort from our side, while not contributing in the skills required to complete this thesis, the director provided the GLSL and corresponding CPU code implementing these transformations. The reason is that the necessary milestone for my training was to be able to run general purpose computations and being able to see which limitations exist and how they can be overcome with the proposed solutions in [3].

For the sake of knowing the limits of our hardware, we started increasing the size of the OpenGL window and therefore increasing the number of elements used in our general purpose computation. Although the textures could be created with large sizes, the addition operation between them resulted in zero when we stepped out of 430 pixels per axis. This deemed strange and took us more time than expected to figure out.

Apparently, the problem was that we performed the computations in the on screen window and copy the data from the framebuffer. However, the Nvidia Xavier we used for experimentation was used as a headless remote server, however the X server was running with a minimum resolution, and therefore ev-

erything exceeding that resolution was not computed. To solve this issue we relied on texture rendering using texture attached to the framebuffer to save the output of the addition operation in the fragment shader.

After that, we studied the differences between OpenGL SC 2 and OpenGL ES 2, as well Brook Auto, which performs the steps we have learned to apply manually in an automatic way. As a result, at that point we had an overview of the technical knowledge to apply general purpose computations with graphics, which was useful in both projects.

9.2 GPU4S: Porting the Fast-Fourier Transform

One we had an understanding of performing general purpose computations with OpenGL ES 2 and its successors, it was time to start porting a benchmark from the GPU4S benchmark suite [31], the Fast Fourier Transform (FFT).

We have selected this particular benchmark for various reasons. First of all, some of the other algorithmic building blocks used in the GPU4S suite for benchmarking like matrix multiplication were already implemented by [3], while the complexity of some others, eg. Finite Impulse Response (FIR) was low for an undergraduate thesis and in addition not much performance gain was expected by porting it in the GPU. On the other hand, we wanted to select a computationally intensive benchmark which could provide great speedups compared to the GPU. Moreover, FFT was one of the few algorithms ported in GPUs in the early versions of GPGPU computing [3] before the appearance of CUDA and OpenCL. As a consequence, this allowed us to have an initial version to start porting, instead of developing all the necessary and complex code from scratch.

The FFT algorithm is an operation used extensively in the aerospace domain to process data, but also in satelite communications for send and receive operations. For this reason it has been selected for inclusion in the GPU4S benchmark suite developed in the project with same name funded by the European Space Agency (ESA), after a survey performed across multiple aerospace domains with the contribution of the project partner, Airbus Defence and Space, Toulouse.

The baseline code we used for our implementation was implemented in OpenGL 1.3 in 1998 using C++ [29]. As a consequence, the code was using an extremely old version of the desktop OpenGL API, which meant that we had to apply a step-by-step approach in order to modernise it and make it work in newer versions.

In particular, porting the application from desktop OpenGL 1.3 to OpenGL SC 2 could not be performed at once, since the API has been radically changed from version to version. That means that porting the application directly to OpenGL SC 2.0 would require massive changes without being able to test its functionality until every part of the application was ported. Moreover, in such case if

the functionality was not equivalent with the initial version, the complexity of identifying the error would be immense.

For this reason, we decided to port the application from OpenGL 1.3 to OpenGL 2, then in OpenGL ES 2 and finally in OpenGL SC 2.0 and Brook Auto as it is shown in Figure 2, following the evolution of the OpenGL APIs, based on the analysis of the graphics APIs we have performed as part of this thesis, which is presented in the next Section.

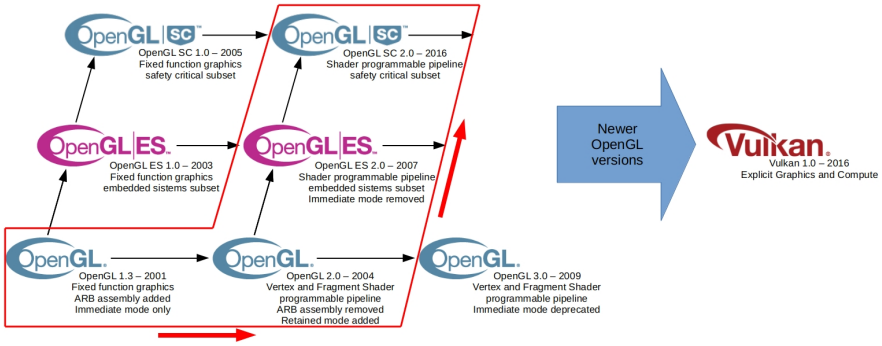
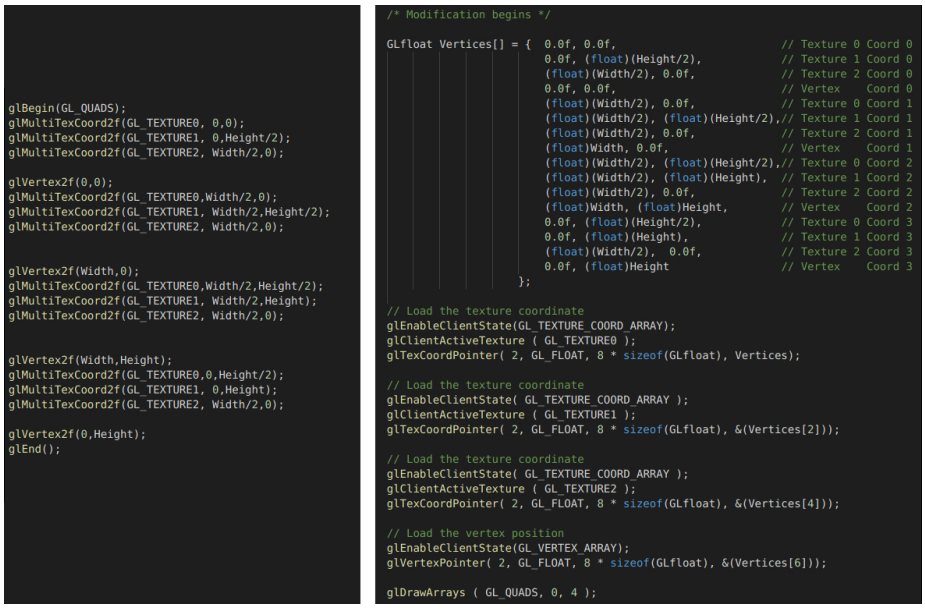


Figure 2: The steps in the porting path between OpenGL versions we have decided to take in the FFT project.

The first main objective was to replace the deprecated immediate mode of OpenGL commands, which is not available in OpenGL ES 2 and SC 2, with its equivalent retained mode calls. In particular, this required to remove all glBegin and glEnd calls from the application. The reason is that even though newer desktop versions of OpenGL can still use this deprecated functionality to specify polygons or textures in the screen, embedded and safety critical subsets do not support this.

In our porting we took advantage of the fact that desktop OpenGL versions still allow the use of immediate mode even when other parts of the application can use retained mode. This way, we were able to gradually port our application by performing smaller changes and verify that the functionality of the application remained identical to the original application. This justifies our decision to perform the porting to OpenGL 2 first instead of OpenGL ES 2, in which this would only be possible when the entire application would have been ported.

In total, the application contained 7 immediate mode sections. Figure 3 shows an example of the conversion required. In particular, within a glBegin/glEnd section, the programmer specifies the vertex coordinates and texture coordinates one by one. To convert such section in retained mode, the vertex coordinates and the texture coordinates have to be placed in arrays which have to be registered for use with OpenGL using the calls glVertexPointer/glTexCoordPointer, and specify low level accessing details for the values contained in these arrays ie.



```
glBegin(GL_QUADS);
glMultiTexCoord2f(GL_TEXTURE0, 0,0);
glMultiTexCoord2f(GL_TEXTURE1, 0,Height/2);
glMultiTexCoord2f(GL_TEXTURE2, Width/2,0);

glVertex2f(0,0);
glMultiTexCoord2f(GL_TEXTURE0,Width/2,0);
glMultiTexCoord2f(GL_TEXTURE1, Width/2,Height/2);
glMultiTexCoord2f(GL_TEXTURE2, Width/2,0);


glVertex2f(Width,0);
glMultiTexCoord2f(GL_TEXTURE0,Width/2,Height/2);
glMultiTexCoord2f(GL_TEXTURE1, Width/2,Height);
glMultiTexCoord2f(GL_TEXTURE2, Width/2,0);


glVertex2f(Width,Height);
glMultiTexCoord2f(GL_TEXTURE0,0,Height/2);
glMultiTexCoord2f(GL_TEXTURE1, 0,Height);
glMultiTexCoord2f(GL_TEXTURE2, Width/2,0);

glVertex2f(0,Height);
glEnd();
```

```
/* Modification begins */

GLfloat Vertices[] = {  0.0f, 0.0f,                         // Texture 0 Coord 0
                        0.0f, (float)(Height/2),            // Texture 1 Coord 0
                        (float)(Width/2), 0.0f,             // Texture 2 Coord 0
                        0.0f, 0.0f,                         // Vertex    Coord 0
                        (float)(Width/2), 0.0f,             // Texture 0 Coord 1
                        (float)(Width/2), (float)(Height/2),// Texture 1 Coord 1
                        (float)(Width/2), 0.0f,             // Texture 2 Coord 1
                        (float)Width, 0.0f,                 // Vertex    Coord 1
                        (float)(Width/2), (float)(Height/2),// Texture 0 Coord 2
                        (float)(Width/2), (float)(Height),  // Texture 1 Coord 2
                        (float)(Width/2), 0.0f,             // Texture 2 Coord 2
                        (float)Width, (float)Height,        // Vertex    Coord 2
                        0.0f, (float)(Height/2),            // Texture 0 Coord 3
                        0.0f, (float)(Height),              // Texture 1 Coord 3
                        (float)(Width/2),  0.0f,            // Texture 2 Coord 3
                        0.0f, (float)Height                 // Vertex    Coord 3
                      };

// Load the texture coordinate
glEnableClientState(GL_TEXTURE_COORD_ARRAY);
glClientActiveTexture ( GL_TEXTURE0 );
glTexCoordPointer( 2, GL_FLOAT, 8 * sizeof(GLfloat), Vertices);

// Load the texture coordinate
glEnableClientState( GL_TEXTURE_COORD_ARRAY );
glClientActiveTexture ( GL_TEXTURE1 );
glTexCoordPointer( 2, GL_FLOAT, 8 * sizeof(GLfloat), &(Vertices[2]));

// Load the texture coordinate
glEnableClientState( GL_TEXTURE_COORD_ARRAY );
glClientActiveTexture ( GL_TEXTURE2 );
glTexCoordPointer( 2, GL_FLOAT, 8 * sizeof(GLfloat), &(Vertices[4]));

// Load the vertex position
glEnableClientState(GL_VERTEX_ARRAY);
glVertexPointer( 2, GL_FLOAT, 8 * sizeof(GLfloat), &(Vertices[6]));

glDrawArrays ( GL_QUADS, 0, 4 );
```

Figure 3: A comparison between an immediate mode code excerpt from the baseline GPUFFTW and the corresponding retained mode excerpt in our GPUFFTW port in OpenGL 2.0. The OpenGL ES 2 and OpenGL SC 2 versions are slightly different in the fact that the word "Client" is removed from the API calls.

how many coordinates are provided for each coordinate, how values are packed etc. Such conversion is not trivial, since it required very careful consideration of the values, as well as figuring out certain aspects of the functionality of some calls which are not identical with new OpenGL versions, by trial and error. The reason is that since immediate functionality is depricated but still usable in modern OpenGL version and the initial OpenGL version of our baseline application was too old, there was no documentation or tutorials available on the details of such conversion between Open GL 1.3 to OpenGL 2.

Moreover, one of the immediate sections generated the coordinates at runtime, using a loop. Such conversion in retained mode was particularly challenging, since it required to fill all the generated data within arrays, before submitting the drawing call, thus creating a trade-off of higher memory consumption in the newer API version.

Another of the problems we had was that the application was using some functions not included in the core OpenGL 1.3 library on the FFT code. For this reason, we had to manually add the function prototype and obtain the function pointer from the dynamic library for each required function.

This happens because OpenGL vendors, mainly Nvidia and AMD, usually im-

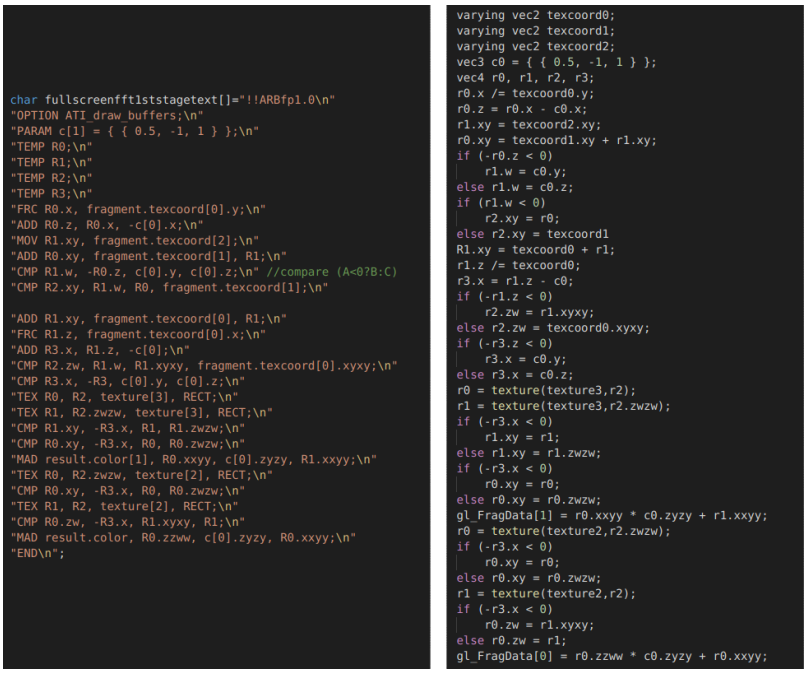
plement extra functionalities on their graphics cards. These utilities are called extensions, and initially only work in their GPU brands, getting a company related prefix like NV or ATI. These features are later brought for discussion in the Khronos committee and are initially accepted as ARB (Architecture Review Board [37]) extensions, and if there is an agreement and support from all vendors they are included in subsequent OpenGL versions as core functions, which belong officially to the standard. Our baseline GPUFFT version was using several Nvidia extensions because it was designed to work with these cards and it was quite challenging to find out the corresponding extensions that made it as ARB or core functions, since our ultimate goal was to port the code to the core OpenGL SC 2.

This issue appeared during the initial stages of the porting, when we faced problems running the original version of the program on the development laptop, which was equipped with an Intel GPU, and therefore didn't support the NVIDIA extensions used by the program. For this reason we had to run it on an Nvidia GPU to get this support, in particular the Xavier automotive platform.

As already mentioned, a significant problem we realised working with such an extremely old version of OpenGL, it was its difficulty to find information regarding specific functions and extensions. There are a lot of broken urls and pages, incomplete forum posts with scarce information and cheatsheets that explain the very basics of its functionality. In summary, most of the time used to port this application from 1.3 to newer versions was utilized in searching information about this older and rough around the edges OpenGL version. However, this was a highly educational task for me, since I obtained a very clear idea about the evolution and refinement of the API, and understand the reason why some API calls got removed or changed in next versions. For example, the glTexCoordPointer we mentioned above, was introduced in OpenGL 2, but was removed later and therefore is not present in the ES and SC versions. The reason is that glVertexPointer provides the same functionality, and both arrays contain coordinates either for vertices or textures. Besides, some applications use a single array to contain both. Therefore, there is no need for including two calls for the same functionality, since they can be differentiated easily with a specification call before calling them on whether they are applied on a texture or vertex attribute.

In addition to the GPU port, an equivalent CPU-only version of the application was also developed, in order to be able to check that the values generated using this GPU software were correct, without any kind of strange behavior.

Finally, another challenging part of our porting was related to the fact that OpenGL before version 2 had a fixed function pipeline, meaning that the transformations for the geometry and the assignment of pixels colors and depth were built-in to the hardware, and could not be modified/programmed. However, OpenGL 1.3 featured ARB extensions implementing the first generation of programmable vertex and fragment shaders, (called *programs* at that time) and had very limited functionality in comparison to the capabilities of modern shaders.



```
char fullscreenfft1ststagetext[]="!!ARBfp1.0\n"
"OPTION ATI_draw_buffers;\n"
"PARAM c[1] = { { 0.5, -1, 1 } };\n"
"TEMP R0;\n"
"TEMP R1;\n"
"TEMP R2;\n"
"TEMP R3;\n"
"FRC R0.x, fragment.texcoord[0].y;\n"
"ADD R0.z, R0.x, -c[0].x;\n"
"MOV R1.xy, fragment.texcoord[2];\n"
"ADD R0.xy, fragment.texcoord[1], R1;\n"
"CMP R1.w, -R0.z, c[0].y, c[0].z;\n" //compare (A<0?B:C)
"CMP R2.xy, R1.w, R0, fragment.texcoord[1];\n"

"ADD R1.xy, fragment.texcoord[0], R1;\n"
"FRC R1.z, fragment.texcoord[0].x;\n"
"ADD R3.x, R1.z, -c[0];\n"
"CMP R2.zw, R1.w, R1.xyxy, fragment.texcoord[0].xyxy;\n"
"CMP R3.x, -R3, c[0].y, c[0].z;\n"
"TEX R0, R2, texture[3], RECT;\n"
"TEX R1, R2.zwzw, texture[3], RECT;\n"
"CMP R1.xy, -R3.x, R1, R1.zwzw;\n"
"CMP R0.xy, -R3.x, R0, R0.zwzw;\n"
"MAD result.color[1], R0.xxyy, c[0].zyzy, R1.xxyy;\n"
"TEX R0, R2.zwzw, texture[2], RECT;\n"
"CMP R0.xy, -R3.x, R0, R0.zwzw;\n"
"TEX R1, R2, texture[2], RECT;\n"
"CMP R0.zw, -R3.x, R1.xyxy, R1;\n"
"MAD result.color, R0.zzww, c[0].zyzy, R0.xxyy;\n"
"END\n";
```

```
varying vec2 texcoord0;
varying vec2 texcoord1;
varying vec2 texcoord2;
vec3 c0 = { { 0.5, -1, 1 } };
vec4 r0, r1, r2, r3;
r0.x /= texcoord0.y;
r0.z = r0.x - c0.x;
r1.xy = texcoord2.xy;
r0.xy = texcoord1.xy + r1.xy;
if (-r0.z < 0)
    r1.w = c0.y;
else r1.w = c0.z;
if (r1.w < 0)
    r2.xy = r0;
else r2.xy = texcoord1
R1.xy = texcoord0 + r1;
r1.z /= texcoord0;
r3.x = r1.z - c0;
if (-r1.z < 0)
    r2.zw = r1.xyxy;
else r2.zw = texcoord0.xyxy;
if (-r3.z < 0)
    r3.x = c0.y;
else r3.x = c0.z;
r0 = texture(texture3,r2);
r1 = texture(texture3,r2.zwzw);
if (-r3.x < 0)
    r1.xy = r1;
else r1.xy = r1.zwzw;
if (-r3.x < 0)
    r0.xy = r0;
else r0.xy = r0.zwzw;
gl_FragData[1] = r0.xxyy * c0.zyzy + r1.xxyy;
r0 = texture(texture2,r2.zwzw);
if (-r3.x < 0)
    r0.xy = r0;
else r0.xy = r0.zwzw;
r1 = texture(texture2,r2);
if (-r3.x < 0)
    r0.zw = r1.xyxy;
else r0.zw = r1;
gl_FragData[0] = r0.zzww * c0.zyzy + r0.xxyy;
```

Figure 4: A comparison between a fragment shader from the baseline GPUFFTW from OpenGL 1.3 written in ARB assembly vs the port GLSL in OpenGL 2. The difference in OpenGL ES 2 and OpenGL SC 2 is that the shader can only have a single output value ($gl\_FragData[0]$ or $gl\_FragColor$), therefore the code has to be split in two separate shaders.

As shown in Figure 4, the fragment programs in OpenGL 1.3 were implemented in a very low level language, called ARB assembly, which was a vendor neutral assembly language. This represents an immense complexity compared to GLSL (OpenGL Shading Language) in which modern shaders are programmed. This added a new layer of intricacy in the port, which was more and more time consuming.

Our baseline contained six fragment shaders and , which we ported two of them in their equivalent GLSL. Again, as in the case of immediate mode porting, we were able to take advantage of the fact that modern desktop OpenGL are still compatible with older OpenGL versions and allows part of the applications to be written using ARB assembly and others in GLSL. This way, we were able to port the application in small parts and be able to test after each one, whether the original behaviour of the application is retained.

In fact, we managed to port successfully two shaders, reaching very close to produce a complete port. However, we faced an impossible problem to overcome in one of the shaders, because it was using more than one textures. In that first

version of programmable shaders, textures were accessed directly by referring to their texture unit id in the assembly language. However, this is not the case in GLSL, where the textures have a distinct named texture id which needs to be linked with the corresponding array containing its vertices. Unfortunately, after almost two weeks trying possible workarounds for that shader together with the director, it was not possible to reproduce the same behaviour with the original application and in order to avoid compromising the thesis, we agreed to move in the next project. We suspect that a simple additional, deprecated OpenGL call in version 2.0 is missing, in order to make it work, since that happened several times during the porting process. After that, the conversion to OpenGL ES 2 and OpenGL SC 2 and of course Brook Auto would be straight forward, since the most significant features of the ES and SC version (retained mode and GLSL) were already implemented. As we describe in the next Section in our analysis, their limitations with respect to the OpenGL 2 version are replacement of QUADS primitives with triangles, normalised coordinates and texture values and the absence of the possibility to read directly values from a texture not bound in the framebuffer.

Finally, it is worth noting that the baseline software was not extremely complex in its structure and it was accompanied by a quite detailed documentation in its related publication [30]. However, the extremely old OpenGL version in which it was written combined with a novel optimisation proposed in the same publication in order to optimally exploit the graphics texture cache, added additional complexity to the porting process.

In conclusion, due to the scarce availability of information and the software special behaviour, the porting was lengthy and very challenging, however we managed to bring it very close to completion, as it is indicated by the fact that the software behaves identically with the original version, with the exception of a shader which was not possible to port. In the future we are going to resume this work to complete it.

9.3 Porting the ADS Application

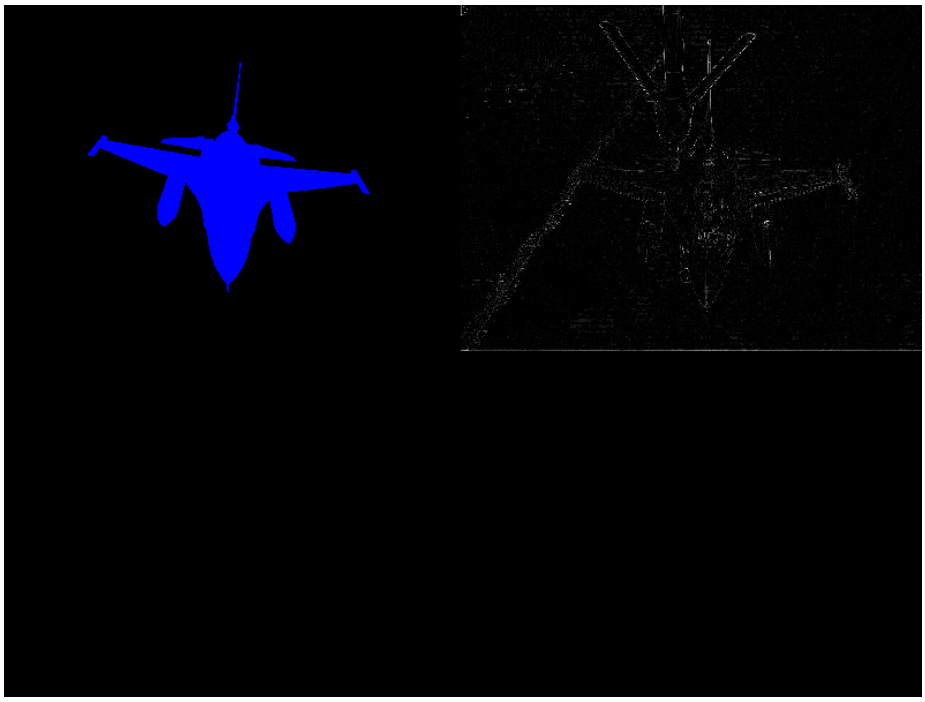


Figure 5: Screenshot of our port of the ADS application in OpenGL SC 2.

As we mentioned earlier, working in this project it was initially the first choice of the two possible paths of the internship. However, the delay in the delivery of the application has shifted our project towards the GPU4S project and the FFT application, but when we faced an impasse close to its completion, we switched to this project, since the application has become available.

The baseline software implements a GPU application implemented in Vulkan, which resembles the automatic air-to-air-refueling capabilities developed and recently demonstrated by Airbus [38].

The display is divided into four regions, although it currently only uses two of them, the two upper ones. Future application versions provided by Airbus are expected to use the other ones, too. In the upper left, the program displays a polygonal view of a plane, rotating as time passes. In the upper right region there is the post processed image of a video frame, which displays the silhouette of a plane and the border between land and sea as a result of an edge detection compute shader. Figure 6 shows a screenshot of our ported version of the application in OpenGL SC 2, which is identical to the original application.

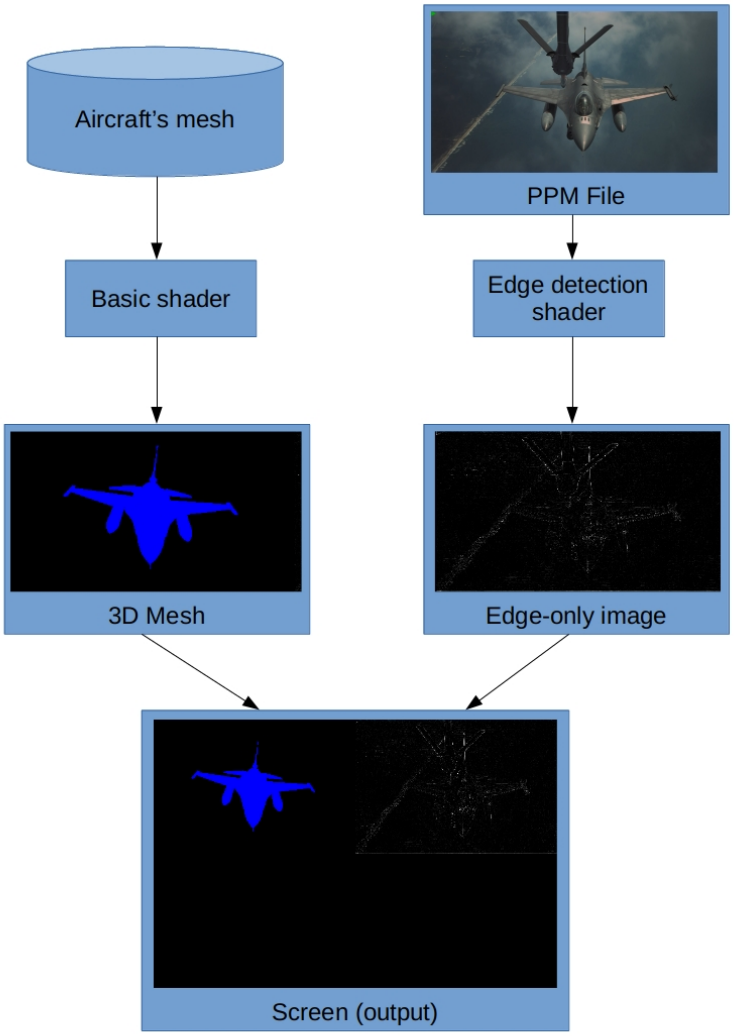


Figure 6: Flowchart of our OpenGL SC 2 port of the application.

The application has been provided by Airbus Defence and Space in Getafe to the project partners CoreAVI and BSC, for testing and demonstration purposes of their technology and tooling in the framework of the collaboration.

9.3.1 Porting the application to OpenGL ES 2

Similar to the FFT project, we decided to perform the porting in steps, following the next closer graphics API in the OpenGL evolution as shown in Figure 7, in order to take a gradual approach, which has been shown beneficial in the GPU4S project.

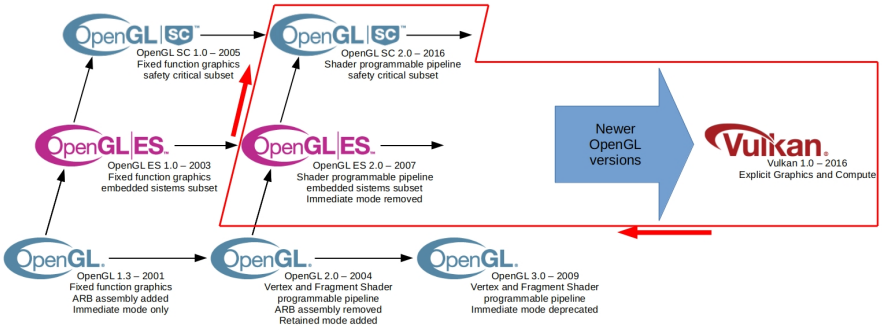


Figure 7: The steps in the porting path between OpenGL versions we have decided to take in our porting of the ADS application.

The first thing we did was read the the code and figure out how they printed the two different images in the same window. We found found that they used a color framebuffer with z-buffer (or depth buffer) activated and another one that was only a color framebuffer, so they used different framebuffers and different shaders to draw in the screen.

Once we figured out how they printed the output, replicating it in OpenGL ES 2 was pretty straightforward. Two viewports were used to do this, so each part of the screen had its own position data.

First, the postprocessed video frame of the plane was drawn in the upper right corner of the screen. In fact, this is accomplished using a Vulkan compute shader, whose outcome is used in a framebuffer for display. However, OpenGL ES 2 and OpenGL SC 2, only support graphics operations so we had to apply the techniques we learned during the bootstraping of the project, following the methods introduced by [3].

In the current version of the application, this frame is read from a jpeg encoded image. Because OpenGL ES 2 does not have any utilities to read from an image file, we took the decision to convert the JPG to a PPM format. PPM is a format that is a container of raw RGB data with some extra headers. After reading this image file, a mock-up fragment shader was used to create a template edge detection shader so a result similar to the Vulkan application could be seen.

Then, we decided to take on the left side of the screen, the rotating 3D plane.

This part is composed purely by graphics computations, in a vertex and framgment shader, so this code was converted to the OpenGL ES 2 graphics operations. In this aspect we used the knowledge we obtained during the project bootstrapping phase, when we were introduced to OpenGL ES 2. In particular, we have reused the knowledge from the 3D rotating cube example in the OpenGL ES 2 tutorials. After understanding how the representation of a 3D object worked in this language, we managed to display the plane in the screen.

After having a template edge detector on the upper right side and the 3D plane at the left side separately, we tried to display them together, but we faced a problem. When we tried to draw the 3D plane after the photograph, the program crashed. Because we didn't understand what was happening there, a lot of tests were done until we finally found the solution. The solution was, for each frame, to disable the vertex arrays for each of the parts of the screen as soon as they were not used. For example, after printing the 3D plane in the screen, disabling the array with the information about the vertices and their output order. In this debugging progress we found very useful the skills and the methodology we developed during our porting of the FFT code, which allowed us to have a structured approach towards the solution.

After that, the two parts of the screen were outputting their respective data. But there was still work to do.

The next step was to tackle the edge detection algorithm. This algorithm is executed in a compute shader, a type of Vulkan shader that does not generates a screen output. This one in particular stores the output information on a texture so it can be printed in the pertinent framebuffer.

Because OpenGL ES 2 does not have a compute shader type, we used instead a fragment shader to make the mathematical operations, which then prints the output directly into the screen. To be able to do this, we decided to separate the 3D mesh vertex and fragment shaders from the photography shaders.

The algorithm, then, works as a convolution of two 3 by 3 matrices for each pixel in the screen. The first one has the average value of the three color channels (red, green and blue) for the pixel in the actual position and its contiguous. The second matrix, which is called a *kernel* in GPU compute terminology, has some parameters to amplify or reduce the value on those pixels using a convolution with the input image.

Once we had this code implemented, we realised that the output from this port was different from the Vulkan code because the screen had some white pixels and vertical and horizontal lines scattered across the image. After a couple of days, we identified the solution comparing it to the mock-up edge-detection previously done. The difference was that in OpenGL ES 2, the texture position values in the shaders are normalized, going from 0 to 1 as pointed out by [3] which we were studying in the project bootstraping, while we where trying to access this positions in a matrix equivalent to the number of pixels of the image, in this case from 0 to 1920 or 1080 as in the case of the GLSL code of Vulkan in

the original application. To normalize the value, then, we divided the number of pixels in one of the axis by one because we wanted to access only one of the pixels in the texture.

After that, some minor tweaks about the rotation and position of the plane model were needed and finally, the project in OpenGL ES 2 was completed.

9.3.2 Porting the application to the OpenGL SC 2 version

Because OpenGL SC 2 is a subset of OpenGL ES 2, the changes we had to do for the full software to be completely ported were minimal. In fact, we only had to change the generation and loading process of the shaders, which is the main difference between the two standards, as we describe in the analysis we performed and presented in the next Section.

The capabilities to compile at runtime any kind of shader have been removed in OpenGL SC 2 for predictability reasons, so they can only be read them from a file in a precompiled form. Because shader compilation may fail at runtime, or can be a expensive procedure that can affect the timing of the system, they are precompiled and stored in a binary file in order to make sure there are no compilation errors for any reason.

Since we wanted to implement an OpenGL SC 2 version that is also compatible with OpenGL ES 2 implementations which support the option to obtain the precompiled binary, we had to access this functionality which is not available in the core OpenGL ES 2 through an extension. In this task we also reused the knowledge we got about adding the extension function prototypes and obtaining the function pointers implementing the extensions by quering the library from our work with the FFT application. This way we managed to create a binary for each pair of a vertex and a fragment shader from OpenGL ES 2. Note that the same functionality in OpenGL SC 2 environments like the AMD 8860 and AMD 9171 with the CoreAVI driver, this is simply achieved by invoking their supplied offline compiler. In the binary file we create, the information we store are, in the following order: the binary format, the size of the binary and the binary data. So in the SC version, then, we read that binary with all its information.

Note that the binary format is an encoding done by the driver of the GPU. Because it is an internal encoding, we can only get the binary format (which is an integer) and encode and decode it with a function, which has the parameters specified in the last parameter.

As we describe in our analysis in the next Section, the other difference of OpenGL SC 2 with respect to OpenGL ES 2 is the absence of calls to destroy/release dynamically allocated memory for textures, framebuffers etc. However, since our application is never ending, we haven't included such features in our OpenGL ES 2 port, so we didn't need to make any additional changes for the OpenGL SC 2.

9.3.3 Porting the application to Brook Auto

Brook Auto [4] is a certification-friendly general purpose language for general purpose GPU computing, based on graphics and particularly OpenGL ES 2/SC 2. As such, it can only be used for compute tasks, not for graphics, although it supports interoperability with graphics.

However, its graphics interoperability with OpenGL ES 2 had not been tested before this project and for this reason we decided to port first only the functionality related to the compute part: the edge-detection algorithm. Since Brook Auto has has similarities with Nvidia's CUDA API, which I learned during my internship's training by attending the PUMPS summerschool, the porting has been completed quickly and only small problems appeared during its implementation.

To start the porting, we prepared the data to be obtained by the GPU. In our case, the image was read from a PPM file like in OpenGL SC. The next step was to copy that information from the CPU to the GPU, which was done with a function call. After that, the GPU executes some functions, called kernels, which implement the main functionalities the code is expected to perform in the accelerator, which in this application was the execution of the edge-detection.

After that, the code is copied back to the CPU to execute some final code before the program ends, which in our case was storing the code in an image to be able to check the result.

During the implementation, we found a bug in Brook Auto's code generation, which we reported and was fixed. Because Brook Auto is a source to source compiler which converts code written in Brook Auto to GLSL ES, it lets you explore the output code. We noticed that the code generation omitted the final *clamp* function we specified in the edge detection kernel, in order to highlight the edges when they were above a threshold. After adding this function call, the obtained result was identical with the Vulkan compute shader.

It is worth noting that writing the Brook Auto version was significantly easier that the OpenGL ES 2 and OpenGL SC 2 code. This was not only because we didn't have to deal with low level operations like texture allocations, copies between them and the framebuffer and shader compilation or loading and linking, but also because within the kernel we could use normal C indexing like in Vulkan and not the counter intuitive normalised ones, that cost us some debugging time in the OpenGL ES 2 version.

After that, we devoted some time to integrate the Brook Auto code with our OpenGL ES 2 port of the application, but since it was an untested functionality and we already implemented several tasks we deemed as optional in the planning of the thesis, we decided to devote the remaining time on the document writing.

9.3.4 Evaluation

After having the Airbus application ported in three different APIs, two of which safety-critical ones, we performed an evaluation of the developed solutions.

In terms of performance, the original application is working in real-time, that is the screen is constantly updating without any visible lag. The same happens in the OpenGL ES 2 and OpenGL SC 2 versions, where there is no visible difference compared with the original application. The Brook Auto version is not integrated with the graphical part of the application, but it is executed instantaneously and since its underlying implementation is on OpenGL ES 2/OpenGL SC 2, we expect a similar performance.

However, in addition to performance, which is not very applicable in such an application, we have evaluated other metrics, mainly regarding the productivity of the ported solutions, by tracking our development time and measuring the amoung of the resulting code. Table 4 shows the detailed information of this data, while Figure 8 offers a visual representation that shows better the relative comparison between the metrics for each version.

Programming language	Vulkan (original code)	OpenGL SC 2	OpenGL SC 2 (edgeDetection)	Brook Auto (edgeDetection)
Development time (days)	31	17	9	2.5
Lines of code (approx)	4000	1400	1200	160

Table 4: Development time and lines of code of each of the versions of the original Airbus application used in this project.

We notice that the development time of the OpenGL SC 2 version (including also the time it took us to produce the intermediate OpenGL ES 2 version) was shorter than the original application written in Vulkan, which we estimate in 31 days. Despite the fact that the original version was produced by another developer so no direct comparison can be applied, one of the reasons is that the application has been developed from scratch, while in our case we had a baseline version which we ported to another graphics API. Moreover, the complexity of Vulkan is higher than the one of OpenGL SC 2, since it is a lower-level API and as such, each OpenGL SC 2 call corresponds to multiple Vulkan calls. This is reflected in the number of lines of code required to implement the same functionality, which is almost 3 times bigger.

Moreover, we can also see that the compute part of the application, the edge detection functionality took more that half of our development time of the OpenGL SC 2 version and accounted for 85% of its code. This is expected since performing computations in graphics APIs by using the methods of [3] is more complex than performing graphics operations such as visualisation tasks.

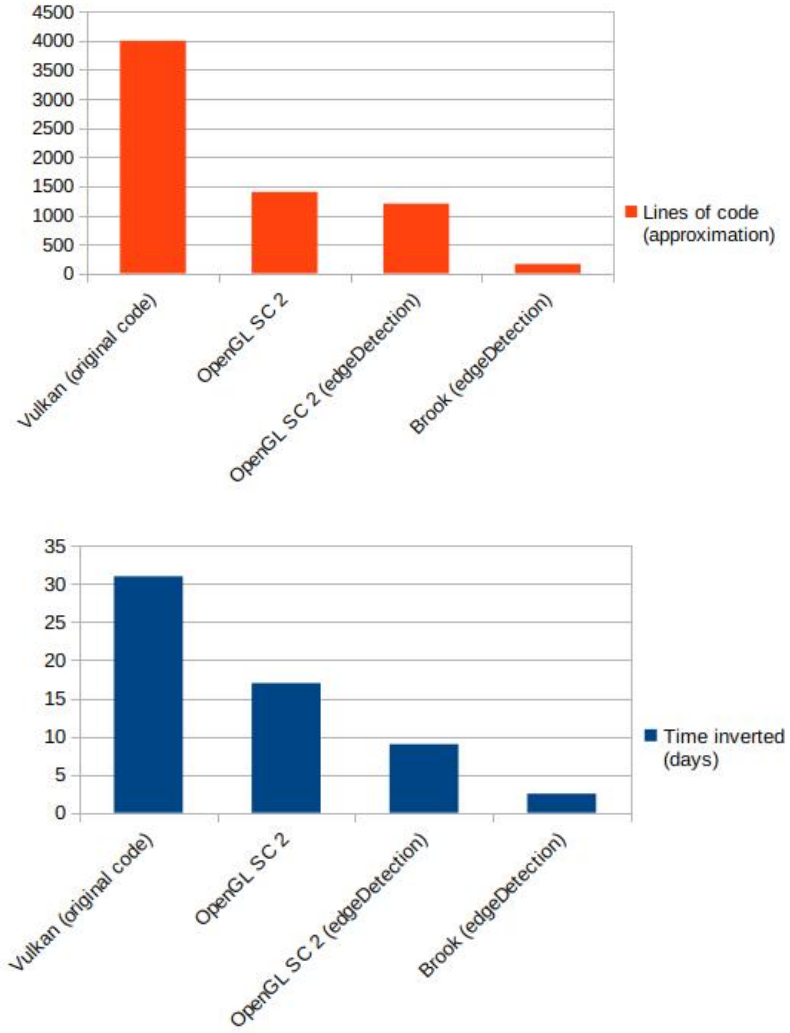


Figure 8: Comparison between lines of code and development time for each versions the Airbus application, the original and the ones we ported in this project.

Finally, the Brook Auto version provided the most impressive results in terms of productivity, since the development time of the compute part was 3.5 times less than the corresponding OpenGL SC 2 implementation and was achieved with 7.5 times less code.

It is important noting that the cost reduction expected in an industrial environment is much larger that the saving of the development cost, especially in the

safety-critical domain, in which verification and certification accounts for the largest portion of the product cost. In particular, less amount of written code is translated to fewer bugs, faster debugging and also faster verification and certification, since all of them scale linearly with the code size and programming language complexity.

10 Analysis of Graphics-based APIs

In the previous Section we described the project evolution and the problems we faced with the porting of the two applications in different graphics APIs. In this Section we provide the outcome of our analysis of the differences between the various versions of the graphics APIs used in this project, with particular emphasis on the safety critical ones. This way, the reader can better understand the rationale behind our decisions during the software development.

This analysis has been based on our preliminary study at the project bootstraping phase and also contains the differences that we discovered during the actual development of the project.

10.1 The OpenGL API evolution

The OpenGL graphics API has been around for almost 3 decades, featuring several versions. Some of the versions had small modifications over the previous one, while others represented more substantial changes from the previous edition.

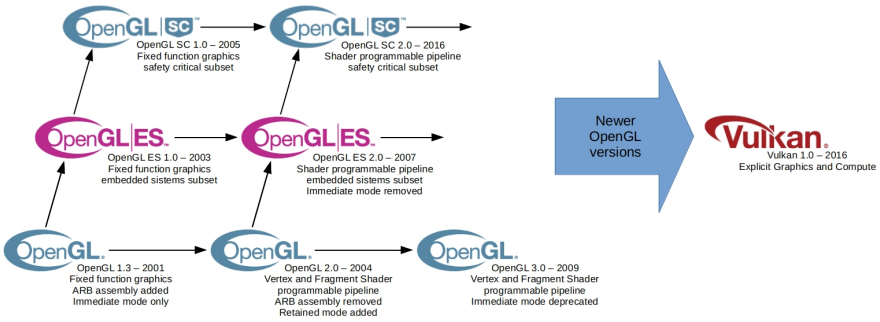


Figure 9: Hierarchical view of the evolution of various versions of OpenGL and their differences.

In Figure 9 we can see a visual representation of the relationship between the main OpenGL versions and their variants, specifically focusing on the versions used in this project.

In the lowermost line we have the regular (desktop) versions of OpenGL which have been the main representative versions of the API. These versions have been the forerunner editions of the more specialised versions for embedded systems which were based on the stable features of the API.

Above the desktop version, we have the Embedded Systems version (ES), which has been based on cutout desktop versions, in order to provide small footprint

implementations and a reasonable feature list for less powerful embedded processors.

On top we find the Safety Critical versions (SC), which are further refinements of the corresponding ES versions.

Finally, Vulkan is the latest addition to the OpenGL family, which combines the capabilities of graphics with compute, in single API.

Before we get into the details of the evolution of these 4 main lines of APIs and their relation, it is worth explaining how the OpenGL API is defined.

OpenGL, similar to other standardisation activities like C++, OpenMP etc, is driven by a board of members, which are stakeholders in the graphics domain. Mainly the members are GPU hardware vendors, however there are also members from software side, eg. from gaming or Virtual Reality (VR) companies. Whenever a new feature is proposed, the committee discusses it and drafts its specification. Whenever a feature is unanimously approved, it is added in the next OpenGL version. This means that all members which provide hardware or software versions of the OpenGL API are going to implement this feature, whose functionality is going to be identical to the specification. That is if an end user decides to change a GPU hardware or GPU driver, the functional behaviour of her/his software is going to be identical. In order to guarantee this, there are *conformance* which check whether a given implementation completely supports all the mandatory features of the standard.

In addition to the core OpenGL features, GPU design companies frequently implement additional features called *extensions*, which are hardware accelerated in their devices. Initially extensions are only supported by the design company which added them first and are usually propose them for addition in the next version. Historically, the OpenGL Architecture Review Board (ARB) [37] was voting for extensions to be added in the next version. These features, whose identification was prefixed with `ARB_` or `EXT_` in modern versions, describe optional functionality which is not part of the core OpenGL specification, but still it is a standardised feature that is supported by more than one vendor.

In the evolution of GPUs and their 3 distinct lines of APIs (desktop, embedded and safety critical), we have noticed the same abrupt change between the versions 1 and 2, although such a change took place with some years of difference between them. This has been with the fixed-function hardware in the first generation of each domain, which was gradually replaced by programmable stages in the graphics pipeline, such as the vertex and fragment processing.

In the following subsections we examine in more detail the differences between the different OpenGL versions we have used in this project.

10.2 Differences between OpenGL 1.3 and 2

The jump from OpenGL 1.3 to 2 was quite important because they practically defined the modern approach by OpenGL to graphics rendering. An exhaustive list of differences can be found in [34]. However, below we only focus on features we encountered in our porting endeavour, as well as the relevant features that are useful for implementing general purpose computations with graphics.

- OpenGL 1.3 is a fixed function graphics API. This does mean that, even though OpenGL rendering pipelines are very similar in all its versions, older ones had built-in math operators instead of programs defined by the user. As extracted from [33], the fixed pipeline was a set of user configured matrices and other configurations but it lacked the flexibility of user programs, namely vertex and fragment shaders, which were added in OpenGL 2.
- The OpenGL 1.3 introduced the had the precursor of vertex and fragment shaders, in the form of the `ARB_vertex_program` and `ARB_fragment_program` extensions. These two extensions added some flexibility specially in texture rendering but were very limited. Both programmable stages are used in ARB assembly, which as a very low level language is harder to learn, develop and debug than GLSL, OpenGL's Shader Language which is based on C.
- OpenGL 1.3 was programmed in immediate mode, which uses the calls `glBegin` and `glEnd` to define a block of commands that are sent to the GPU one by one. In OpenGL 2 it is possible to program in either immediate or retained mode. In the latter, the information is grouped together in arrays, which allows their submission to the GPU for drawing at once. However, such arrays need to be bound to *attributes*, which are different types of variables that will be used by the shaders, before the drawing function is called.

10.3 OpenGL 2 vs OpenGL 2 ES

Because OpenGL 2 ES hardware was more constrained in size and power that the standard version, some features that were considered unnecessary or deprecated were removed. Similaly with the previous pair, we will only focus on the ones that were relevant to the development of this project. A exhaustive list of the differences can be found in [35], while an exhaustive list of the differences relevant to general purpose computations is presented in [3].

- The most important difference was that immediate mode was removed from OpenGL 2 ES, which is the capability to program using glBegin and glEnd syntax as mentioned above. Because of this, any code from any version prior to OpenGL 2 could not work in the ES version.

- OpenGL 2 ES can only draw up up to 3-vertex polygons, meaning triangles. Because triangles can draw every other polygon with no exception, the option to draw quads is removed.
- Another difference is that, in OpenGL Shader Language (GLSL) for the ES version, the textures and the coordinates stored in the GPU are normalised, that is in the range [0-1]. In comparison, in standard GLSL, you have the option to choose how your information has to be stored. This difference was the main reason why general purpose computations were not performed in embedded GPUs until [3] solved this issue by proposing encoding and decoding mathematical trasnformations implemented in the GLSL, and in the floating point case also in the CPU side.
- Finally, OpenGL ES 2 allows only one output per fragment shader, unlike OpenGL 2 which is only limited by the number of Multiple Render Targets (MRTs) supported by the target GPU. In order to obtain the same functionality, a kernel using multiple outputs like the example in Figure 3 has to be split in one shader per output.

10.4 OpenGL 2 ES vs OpenGL SC 2

OpenGL SC 2 is targeted to critical systems, so it has to meet their certification requirements. For that reason, only the basic functionalities remain and the safety critical guidelines must be followed. A complete list of information can be found in [36].

- One of the most important functionalities that have been removed are the tools to compile shaders at runtime ("on-line"), because there is a high possibility that the compilation will fail at runtime. We also have to take into account that this process is usually very expensive in time, which is very important in the realm of real-time systems, and that the compiler code needs to be certified so it can be used in a critical system with the rest of the code.
- Some data structures have been reduced to its very core and also restrictions are applied to ensure the resilience and the correct execution of the program. For example, the absence of calls to destroy/release dynamically allocated memory for textures, framebuffers etc.

10.5 OpenGL vs Vulkan

The main differences we identified while working with both APIs in the context of our project are:

- Vulkan is a new API for hardware accelerated graphics (and general computation) which is more low-level oriented, having more control over the

execution of the program in the accelerator. For this reason, the resulting code has an extension quite large in comparison with OpenGL.

- Apart from vertex and fragment shaders, it also has compute shaders, which executes mathematical operations without the need to output anything to screen.
- In Vulkan, shaders are compiled into SPIR-V, an intermediate language developed by Khronos Group, which then is compiled by the driver in the GPU.
- Because it is a newer API build from scratch, only a smal percentage of low-end GPUs has support for it, greatly limiting the usability for applications in critical systems.

10.6 OpenGL SC 2 vs Brook Auto

Brook Auto is a subset of Brook, a high level programming language for general purpose programming in GPUs, fitted for critical systems in the automotive domain. This are the differences I have found working in this project.

- Brook Auto uses a high level programming language very similar to CUDA, which is based in *kernels*, functions that are executed in the accelerator. The programmer does not need to compile and link the kernels, set up uniforms, bind attributes etc, since Brook takes care of these tasks.
- Memory accesses within *kernels* are performed in usual integer arithmetic similar to C, unlike the counter-intuitive normalised coordinate form of OpenGL SC 2.
- In addition to *kernels* the programmer is only required to copy memory from the CPU to GPU and vice versa. There is no need to take care of textures, framebuffers or complicated format conversions to deal with normalised texture values.
- Brook Auto has high productivity in comparison with OpenGL SC 2 because the quantity of the code written by the programmer is inferior, for the same functionality.
- Brook Auto generates a OpenGL SC 2 GPU code and host (CPU) code in C, which can be read, modified and certified.
- Brook Auto is Open Source and has a small compiler, making it easy to maintain and capable of removing functions not needed by the software in development.
- It is still in alpha version, so more work needs to be done to reach a production level quality as OpenGL already has. Activities like this project to use industrial code with it can help for it to mature further.

10.7 Brook Auto vs CUDA

Brook Auto is based on original Brook [32], a language developed at Standford University to facilitate general purpose GPU programming. Before Brook, general purpose programming was only possible using graphics APIs, in the same way that we applied them in the context of this thesis for safety critical systems. However, as we have experienced first hand, this task is very challenging.

Brook has been the basis of CUDA and was also used by ATI/AMD, before the appearance of OpenCL, in the form of Brook+. For this reason the differences between Brook and CUDA are minimal, which makes porting of software from one language to the other very easy:

- Brook does not use pointers in the GPU, instead it provides an opaque type called stream denoted by <> for accessing GPU memory within a kernel.
- Brook supports transfers between the CPU and the GPU with steamWrite and streamRead methods, instead of the cudaMemCopy provided by CUDA.
- Brook Auto differs from Brook and CUDA in the fact that it enforces the programmer to provide upperbounds in the loops used in the kernel, in order to prevent infinite loops. It also enforces a single output per kernel, following a limitation from its implementation with OpenGL ES 2 and OpenGL SC 2. An exchaustive list of Brook Auto's properties can be found in [4].

11 Conclusions

In this project, worked with GPU accelerated general purpose software written in graphics APIs for safety critical systems like automotive, avionics and space domains. We contributed in two different research projects by porting two applications in several APIs and we analysed their differences.

With the porting of the FFT application for the GPU4S (GPU for Space) project we have learned, first and foremost, that applications written in an obsolete programming language require a lot of time and effort to make them work in a modern environment. However, they are still better starting point than developing a complex general purpose computing software from scratch on graphics API, as we have learned in our training period during the project boostraping. Also, working with this application helped us to learn the specifics of OpenGL ES 2 and the difficulties to port a software to this specialised version, which has been proved harder than we anticipated. In any case, all this previous work with the FFT software helped us to reach the completion of the next project more rapidly.

The Airbus Defence and Space project, on the other hand, was more straightforward in its development. Even though Vulkan was a technology we never used before, we could extract the basic functionalities of the software and adapt them to the requirements of OpenGL ES 2 and SC 2, thanks to the advanced experience with graphics task we obtained from the beginning of this thesis. As we said before, working with the FFT application helped to have more knowledge about these OpenGL versions before starting with the project, so it streamlined the implementation of Airbus's software.

In addition, working with Brook Auto provided to be an easier task than what we expected. Being a high level programming language helps automatizing a great quantity of code, which means, less error prone implementations with high productivity in the critical systems domain.

In conclusion, this project made us realize the importance of a good working solution for the GPU software development for critical systems in order to increase the productivity and obtain the required safety certifications.

11.1 Future improvements

There are some parts of the project that could be expanded or finished if we had more time:

- The porting of the FFT algorithm could be completed with more time, since the project reached very close to completion. Having more information available on older OpenGL versions would be very useful in this aspect.

- Brook Auto, in theory, has the necessary tools for interoperability with OpenGL SC 2. In this project we just scratched the surface, but with more time we could implement the full OpenGL SC 2 application with the compute part programmed in Brook Auto working under the same context.
- We only attacked one of the software of the GPU4S suite, the FFT, but other software within this set of programs can also be ported.
- Optimizations could not be performed to the actual code, so a study can be done with the implementation we produced to further enhance the capabilities of the software, especially in the FFT application.

Appendix

In this section we provide the FFT modifications we performed for the GPU4S project. Note that since the baseline code from [29] contains around 3000 lines of code spread across 11 files, we only present the the files which we modified in order to port the application from OpenGL 1.3 to OpenGL 2.

A GPUFFTW modified code

A.1 ported_shader.h

```
//--- Original ---//

char copyfptext[]="!!ARBfp1.0\n"
"OPTION ATI_draw_buffers;\n"
"PARAM c[1] = { program.local[0] };\n"
"TEX result.color[1], fragment.texcoord[0], texture[3], RECT;\n"
"TEX result.color, fragment.texcoord[0], texture[2], RECT;\n"
"END\n";

//--- Port ---//

uniform sampler2D texture0;
uniform sampler2D texture1;

varying vec2 texcoord0;

vec3 c0 = { { 0.5, -1, 1 } };
gl_FragData[1] = texture(texture3,texcoord0.xy);
gl_FragData[0] = texture(texture2,texcoord0.xy);


//--- Original ---//

char fullscreenfft1ststagetext[]="!!ARBfp1.0\n"
"OPTION ATI_draw_buffers;\n"
"PARAM c[1] = { { 0.5, -1, 1 } };\n"
"TEMP R0;\n"
"TEMP R1;\n"
"TEMP R2;\n"
"TEMP R3;\n"
"FRC R0.x, fragment.texcoord[0].y;\n"
"ADD R0.z, R0.x, -c[0].x;\n"
"MOV R1.xy, fragment.texcoord[2];\n"
```

```
"ADD R0.xy, fragment.texcoord[1], R1;\n"
"CMP R1.w, -R0.z, c[0].y, c[0].z;\n" //compare (A<0?B:C)
"CMP R2.xy, R1.w, R0, fragment.texcoord[1];\n"

"ADD R1.xy, fragment.texcoord[0], R1;\n"
"FRC R1.z, fragment.texcoord[0].x;\n"
"ADD R3.x, R1.z, -c[0];\n"
"CMP R2.zw, R1.w, R1.xyxy, fragment.texcoord[0].xyxy;\n"
"CMP R3.x, -R3, c[0].y, c[0].z;\n"
"TEX R0, R2, texture[3], RECT;\n"
"TEX R1, R2.zwzw, texture[3], RECT;\n"
"CMP R1.xy, -R3.x, R1, R1.zwzw;\n"
"CMP R0.xy, -R3.x, R0, R0.zwzw;\n"
"MAD result.color[1], R0.xxyy, c[0].zyzy, R1.xxyy;\n"
"TEX R0, R2.zwzw, texture[2], RECT;\n"
"CMP R0.xy, -R3.x, R0, R0.zwzw;\n"
"TEX R1, R2, texture[2], RECT;\n"
"CMP R0.zw, -R3.x, R1.xyxy, R1;\n"
"MAD result.color, R0.zzww, c[0].zyzy, R0.xxyy;\n"
"END\n";

//--- Port ---//

varying vec2 texcoord0;
varying vec2 texcoord1;
varying vec2 texcoord2;

// "PARAM c[1] = { { 0.5, -1, 1 } };\n"
vec3 c0 = { { 0.5, -1, 1 } };

vec4 r0, r1, r2, r3;

// "FRC R0.x, fragment.texcoord[0].y;\n"
r0.x /= texcoord0.y;

// "ADD R0.z, R0.x, -c[0].x;\n"
r0.z = r0.x - c0.x;

// "MOV R1.xy, fragment.texcoord[2];\n"
r1.xy = texcoord2.xy;

// "ADD R0.xy, fragment.texcoord[1], R1;\n"
r0.xy = texcoord1.xy + r1.xy;

// "CMP R1.w, -R0.z, c[0].y, c[0].z;\n"
if (-r0.z < 0)
```

```
    r1.w = c0.y;
else r1.w = c0.z;

// "CMP R2.xy, R1.w, R0, fragment.texcoord[1];\n"
if (r1.w < 0)
    r2.xy = r0;
else r2.xy = texcoord1

// "ADD R1.xy, fragment.texcoord[0], R1;\n"
R1.xy = texcoord0 + r1;

// "FRC R1.z, fragment.texcoord[0].x;\n"
r1.z /= texcoord0;

// "ADD R3.x, R1.z, -c[0];\n"
r3.x = r1.z - c0;

// "CMP R2.zw, R1.w, R1.xyxy, fragment.texcoord[0].xyxy;\n"
if (-r1.z < 0)
    r2.zw = r1.xyxy;
else r2.zw = texcoord0.xyxy;

// "CMP R3.x, -R3, c[0].y, c[0].z;\n"
if (-r3.z < 0)
    r3.x = c0.y;
else r3.x = c0.z;

// "TEX R0, R2, texture[3], RECT;\n"
r0 = texture(texture3,r2);

// "TEX R1, R2.zwzw, texture[3], RECT;\n"
r1 = texture(texture3,r2.zwzw);

// "CMP R1.xy, -R3.x, R1, R1.zwzw;\n"
if (-r3.x < 0)
    r1.xy = r1;
else r1.xy = r1.zwzw;

// "CMP R0.xy, -R3.x, R0, R0.zwzw;\n"
if (-r3.x < 0)
    r0.xy = r0;
else r0.xy = r0.zwzw;

// "MAD result.color[1], R0.xxyy, c[0].zyzy, R1.xxyy;\n" (A*B+C)
gl_FragData[1] = r0.xxyy * c0.zyzy + r1.xxyy;
```

```
// "TEX R0, R2.zwzw, texture[2], RECT;\n"
r0 = texture(texture2,r2.zwzw);

// "CMP R0.xy, -R3.x, R0, R0.zwzw;\n"
if (-r3.x < 0)
    r0.xy = r0;
else r0.xy = r0.zwzw;

// "TEX R1, R2, texture[2], RECT;\n"
r1 = texture(texture2,r2);

// "CMP R0.zw, -R3.x, R1.xyxy, R1;\n"
if (-r3.x < 0)
    r0.zw = r1.xyxy;
else r0.zw = r1;

// "MAD result.color, R0.zzww, c[0].zyzy, R0.xxyy;\n"
gl_FragData[0] = r0.zzww * c0.zyzy + r0.xxyy;
```

A.2 GPUFFT.h

```
/*****************************************************************************\

Copyright 2019 Barcelona Supercomputing Center and
Universitat Politecnica de Catalunya.

Contact:
M. Benito or L. Kosmidis
Department of Computer Architecture - Operating Systems
marc.benito@bsc.es - leonidas.kosmidis@bsc.es

Copyright 2005 The University of North Carolina at Chapel Hill.
All Rights Reserved.

Permission to use, copy, modify and distribute this software and its
documentation for educational, research and non-profit purposes, without
fee, and without a written agreement is hereby granted, provided that the
above copyright notice and the following three paragraphs appear in all
copies. Any use in a commercial organization requires a separate license.

IN NO EVENT SHALL THE UNIVERSITY OF NORTH CAROLINA AT CHAPEL HILL BE LIABLE
TO ANY PARTY FOR DIRECT, INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL
DAMAGES, INCLUDING LOST PROFITS, ARISING OUT OF THE USE OF THIS SOFTWARE AND
ITS DOCUMENTATION, EVEN IF THE UNIVERSITY OF NORTH CAROLINA HAVE BEEN
ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.


Permission to use, copy, modify and distribute this software and its
documentation for educational, research and non-profit purposes, without
fee, and without a written agreement is hereby granted, provided that the
above copyright notice and the following three paragraphs appear in all
copies.

THE UNIVERSITY OF NORTH CAROLINA SPECIFICALLY DISCLAIM ANY WARRANTIES,
INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND
FITNESS FOR A PARTICULAR PURPOSE.  THE SOFTWARE PROVIDED HEREUNDER IS ON AN
"AS IS" BASIS, AND THE UNIVERSITY OF NORTH CAROLINA HAS NO OBLIGATION TO
PROVIDE MAINTENANCE, SUPPORT, UPDATES, ENHANCEMENTS, OR MODIFICATIONS.


--------------------------------
|Please send all BUG REPORTS to:  |
|                                 |
|   geom@cs.unc.edu               |
```

```
/                                 /
----------------------------------


The authors may be contacted via:

US Mail:          N. Govindaraju or D. Manocha
                    Department of Computer Science
                    Sitterson Hall, CB #3175
                    University of North Carolina
                    Chapel Hill, NC 27599-3175

\**************************************************************************/

//#include <arbfprog.h>
#include <shahelp.h>
enum FFT_ERROR {FFT_SUCCESS, FFT_ARRAY_TOO_BIG,
                                  FFT_ARRAY_TOO_SMALL, FFT_NON_POWER_OF_TWO};


inline int comparePtr(const void *i1, const void *i2);

class GPUFFTW{
public: //methods
  //ctor
  /*
   * Parameters:
   */
  GPUFFTW();

  //dtor
  ~GPUFFTW();

  /* Returns max array size that can be handled depending on video RAM */
  int maxSize(){
    return(MAX_SIZE);
  }


  /* Sorts an input array of (key,pointer) tuples on the GPU.
   * WARNING: The contents of the array will be overwritten
   *          with the sorted result.
   * Parameters:
   *   array: array of tuples to sort
   *   size: length of array
   * Return value:
```

```
     *   error code. FFT_SUCCESS is returned in case of no error.
     *   Use errorToString() to print formatted string of error code.
     */
    FFT_ERROR cgpufft1d(float *array, float  *ptrs,
                                          int size, int sign, int isreal=0);
    FFT_ERROR scgpufft1d(float *real,  int size, int sign);

    FFT_ERROR setFFTParams(int size);

    /* Upload float data to Video Memory.
     *
     * Parameters:
     *   array: Input data. All the required parameters not explicit here are
     *          supposed to have been set by a preceeding call to setSortParams().
     * Return Value:
     *   None
     *
     * Assumption: All the pointers are 32-bit (float * and void *)
     *
     */
    void uploadData(void *array);


    /* Read data back from GPU to CPU memory
     *
     * Parameters:
     *   array: an array of scalars or (key,pointer) tuples depending
     *          on whether tuples was set to FALSE or TRUE in the call
     *      to setSortParams()
     * Return value:
     *   None
     *
     */
    void readbackData(void *array);

    /* Print a formatted error string for errors returned from sort() */
    char *FFTErrorString(FFT_ERROR error);
    FFT_ERROR cGPUFFTW2d(float *real, float  *imag, int arraywidth,
                                       int arrayheight, int sign, int isreal);
    void CheckErrors();

  private: //methods
    inline void PingPong();
    void PingPong2to4();
    void PingPong4to2();
    void cgpufft1d( float sign, int usedfor2dfft=0 );
```

```
private: //data
  int W, H, MAX_SIZE, MIN_SIZE;
  int Width, Height, N, LOGN;
  unsigned int fb;
  unsigned int textureid[6];//last two as temporary buffers


  // ARBFProg copyfp;
  ShaHelp copyfp;
  ARBFProg fullscreenrowwidthfp;
  ARBFProg fftstages_twicerowwidth_to_n_fp;
  ARBFProg fftstages3_to_rowwidthfp;
  ARBFProg fullscreenfft1ststage;
  ARBFProg fullscreenfft2ndstagefp;
  int TILE_SIZE;
  int source, target;
  int vCoord, texCoord0;
};
```

A.3 GPUFFT.cpp

```
/*****************************************************************************\

Copyright 2019 Barcelona Supercomputing Center and
Universitat Politecnica de Catalunya.

Contact:
M. Benito or L. Kosmidis
Department of Computer Architecture - Operating Systems
marc.benito@bsc.es - leonidas.kosmidis@bsc.es

Copyright 2005 The University of North Carolina at Chapel Hill.
All Rights Reserved.

Permission to use, copy, modify and distribute this software and its
documentation for educational, research and non-profit purposes, without fee,
and without a written agreement is hereby granted, provided that the above
copyright notice and the following three paragraphs appear in all copies.
Any use in a commercial organization requires a separate license.

IN NO EVENT SHALL THE UNIVERSITY OF NORTH CAROLINA AT CHAPEL HILL BE LIABLE TO
ANY PARTY FOR DIRECT, INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES,
INCLUDING LOST PROFITS, ARISING OUT OF THE USE OF THIS SOFTWARE AND ITS
DOCUMENTATION, EVEN IF THE UNIVERSITY OF NORTH CAROLINA HAVE BEEN ADVISED OF
THE POSSIBILITY OF SUCH DAMAGES.

Permission to use, copy, modify and distribute this software and its
documentation for educational, research and non-profit purposes, without fee,
and without a written agreement is hereby granted, provided that the above
copyright notice and the following three paragraphs appear in all copies.

THE UNIVERSITY OF NORTH CAROLINA SPECIFICALLY DISCLAIM ANY WARRANTIES,
INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND
FITNESS FOR A PARTICULAR PURPOSE.  THE SOFTWARE PROVIDED HEREUNDER IS ON AN
"AS IS" BASIS, AND THE UNIVERSITY OF NORTH CAROLINA HAS NO OBLIGATION TO
PROVIDE MAINTENANCE, SUPPORT, UPDATES, ENHANCEMENTS, OR MODIFICATIONS.


--------------------------------
|Please send all BUG REPORTS to:  |
|                                |
|   geom@cs.unc.edu              |
|                                |
--------------------------------
```

```
The author may be contacted via:

US Mail:          N. Govindaraju
                    Department of Computer Science
                    Sitterson Hall, CB #3175
                    University of North Carolina
                    Chapel Hill, NC 27599-3175

\**************************************************************************/

#include <stdlib.h>
#include <stdio.h>
#include <assert.h>
#include <math.h>

#include <GL/glut.h>
#include <shader.h>
#include <defines.h>
#include <GPUFFTW.h>

#ifdef _WIN32
typedef BOOL (*NvCplGetDataIntType)(long, long*);
#endif

GLenum buffers1[] = {GL_COLOR_ATTACHMENT0_EXT, GL_COLOR_ATTACHMENT2_EXT};
GLenum buffers2[] = {GL_COLOR_ATTACHMENT1_EXT, GL_COLOR_ATTACHMENT3_EXT};


GLenum buffers41[] = {GL_COLOR_ATTACHMENT0_EXT, GL_COLOR_ATTACHMENT2_EXT,
                      GL_COLOR_ATTACHMENT1_EXT, GL_COLOR_ATTACHMENT3_EXT};
GLenum buffers42[] = {GL_COLOR_ATTACHMENT1_EXT, GL_COLOR_ATTACHMENT3_EXT,
                      GL_COLOR_ATTACHMENT0_EXT, GL_COLOR_ATTACHMENT2_EXT};
GLenum buffers4[] = {GL_COLOR_ATTACHMENT0_EXT, GL_COLOR_ATTACHMENT1_EXT,
                     GL_COLOR_ATTACHMENT2_EXT, GL_COLOR_ATTACHMENT3_EXT};



GPUFFTW::GPUFFTW(){

  //Create OpenGL context
  glutInitDisplayMode( GLUT_SINGLE | GLUT_RGBA );
  glutCreateWindow( "NVToolkit" );

  InitFBOExts();
```

```
  fullscreenrowwidthfp.Load("fullscreenrowwidthFFTptr", fullscreenrowwidthfptext);
  fftstages_twicerowwidth_to_n_fp.Load("maxptrrg", fftstages_twicerowwidth_to_n_fptext);
  fftstages3_to_rowwidthfp.Load("minptrrg", fftstages3_to_rowwidthfptext);
  fullscreenfft1ststage.Load("fullscreenfft1ststage", fullscreenfft1ststagetext);
  fullscreenfft2ndstagefp.Load("fullscreenfft2ndstagefp",fullscreenfft2ndstagefptext);
  copyfp.Load("copy", copyfptext);
  // copyfp.Load();


  glGenFramebuffersEXT(1, &fb);

  glGenTextures(1, &textureid[0]);
  glGenTextures(1, &textureid[1]);
  glGenTextures(1, &textureid[2]);
  glGenTextures(1, &textureid[3]);
  glGenTextures(1, &textureid[4]);
  glGenTextures(1, &textureid[5]);

  bool VRAMdetected=false;
  long int videoMemorySize;
#ifdef WIN32
  // Load NVCPL library
    HINSTANCE hLib = ::LoadLibrary("NVCPL.dll");
    if (hLib == 0) {
        printf("Unable to load NVCPL.dll\n");
        //return -1;
    }
#ifdef GPU_DEBUG
    printf("\nNVCPL.dll successfully loaded\n\n");
#endif
  NvCplGetDataIntType NvCplGetDataInt =\
  (NvCplGetDataIntType)::GetProcAddress(hLib, "NvCplGetDataInt");
  if (NvCplGetDataInt == 0)
       printf("- Unable to get a pointer to NvCplGetDataInt\n");

    if (NvCplGetDataInt(2, &videoMemorySize) == FALSE){
          printf("Unable to retrieve Video Memory Size!\n");
  }
    else{
#ifdef GPU_DEBUG
          printf("Detected Video Memory Size = %d MBytes\n", videoMemorySize);
#endif
      VRAMdetected=true;
  }
```

```
  // Free NVCPL library
    ::FreeLibrary(hLib);
#endif

  if(!VRAMdetected){
    printf("WARNING: Video Memory Size could not be detected. Assuming 128 MB.\n");
    videoMemorySize=128;
  }

  N=videoMemorySize<<16;
  //LOGN represents Log(texture width* texture height)
  //rather than log of number of elements.
  for(LOGN=0;N>>LOGN;LOGN++);
  LOGN=LOGN-4;//added for MRTs
  LOGN--;//added for temporary buffers
  W = (1 << (LOGN - (int) (LOGN/2)) );
  H = (1 << (int) (LOGN/2));

#ifdef GPU_DEBUG
  printf("Largest texture which can be allocated = %d * %d\n",W,H);
#endif

  glActiveTextureARB( GL_TEXTURE2_ARB );
  glBindTexture(GL_TEXTURE_RECTANGLE_NV, textureid[0]);

  glTexImage2D(GL_TEXTURE_RECTANGLE_NV, 0, GL_FLOAT_RGBA32_NV,
                 W, H, 0, GL_RGBA, GL_FLOAT, NULL);

  glActiveTextureARB( GL_TEXTURE2_ARB );
  glBindTexture(GL_TEXTURE_RECTANGLE_NV, textureid[1]);

  glTexImage2D(GL_TEXTURE_RECTANGLE_NV, 0, GL_FLOAT_RGBA32_NV,
                 W, H, 0, GL_RGBA, GL_FLOAT, NULL);

  glActiveTextureARB( GL_TEXTURE3_ARB );
  glBindTexture(GL_TEXTURE_RECTANGLE_NV, textureid[2]);

  glTexImage2D(GL_TEXTURE_RECTANGLE_NV, 0, GL_FLOAT_RGBA32_NV,
                 W, H, 0, GL_RGBA, GL_FLOAT, NULL);


  glActiveTextureARB( GL_TEXTURE4_ARB );
  glBindTexture(GL_TEXTURE_RECTANGLE_NV, textureid[3]);

  glTexImage2D(GL_TEXTURE_RECTANGLE_NV, 0, GL_FLOAT_RGBA32_NV,
```

```
                    W, H, 0, GL_RGBA, GL_FLOAT, NULL);


  glActiveTextureARB( GL_TEXTURE2_ARB );
  glBindTexture(GL_TEXTURE_RECTANGLE_NV, textureid[4]);

  glTexImage2D(GL_TEXTURE_RECTANGLE_NV, 0, GL_FLOAT_RGBA32_NV,
                  W, H, 0, GL_RGBA, GL_FLOAT, NULL);

  glActiveTextureARB( GL_TEXTURE3_ARB );
  glBindTexture(GL_TEXTURE_RECTANGLE_NV, textureid[5]);

  glTexImage2D(GL_TEXTURE_RECTANGLE_NV, 0, GL_FLOAT_RGBA32_NV,
                  W, H, 0, GL_RGBA, GL_FLOAT, NULL);


  GLuint error=glGetError();
  if(error!=GL_NO_ERROR){
    printf("Error while creating two textures of size %d * %d.
            Cannot proceed further.\n",W,H);
  }
#ifdef GPU_DEBUG
  else{
    printf("Successfully created two textures of size %d * %d. \n",W,H);
  }
#endif

  MAX_SIZE = 4*W*H; // Dependent on the available video memory ON THE GPU.
  MIN_SIZE = 16;   //MIN_SIZE should not be less than 16 so that data is
                   //packed correctly in the FFT routine
  //This is for cache-optimization. TILE_SIZE should be a power of 2
  //and the performance improvement based on TILE_SIZE
  //can be dependent on the underlying graphics processor
  TILE_SIZE=64;

  glBindFramebufferEXT(GL_FRAMEBUFFER_EXT, fb);

  glFramebufferTexture2DEXT(GL_FRAMEBUFFER_EXT,
                                  GL_COLOR_ATTACHMENT0_EXT,
                                  GL_TEXTURE_RECTANGLE_NV, textureid[0], 0);

  glFramebufferTexture2DEXT(GL_FRAMEBUFFER_EXT,
                                  GL_COLOR_ATTACHMENT1_EXT,
                                  GL_TEXTURE_RECTANGLE_NV, textureid[1], 0);

  glFramebufferTexture2DEXT(GL_FRAMEBUFFER_EXT,
```

```
                                    GL_COLOR_ATTACHMENT2_EXT,
                                    GL_TEXTURE_RECTANGLE_NV, textureid[2], 0);

  glFramebufferTexture2DEXT(GL_FRAMEBUFFER_EXT,
                                    GL_COLOR_ATTACHMENT3_EXT,
                                    GL_TEXTURE_RECTANGLE_NV, textureid[3], 0);

  glViewport(0,0,W, H);
  glMatrixMode(GL_PROJECTION);
  glLoadIdentity();
  gluOrtho2D(0, W, 0, H);
  glMatrixMode( GL_MODELVIEW );
  glLoadIdentity();

  glBindFramebufferEXT(GL_FRAMEBUFFER_EXT, 0);
}


GPUFFTW::~GPUFFTW(){
  glDeleteTextures(1, &textureid[0]);
  glDeleteTextures(1, &textureid[1]);
  glDeleteFramebuffersEXT(1, &fb);
  glutDestroyWindow(glutGetWindow());
}

void GPUFFTW::CheckErrors(){
  GLenum error = glGetError();
  if( error != GL_NO_ERROR ) {
    fprintf( stderr, "\nGL Error: %s\n", gluErrorString( error ) );
    assert(0);
  }
}


/** Prints a formatted error message for error codes
    returned from setFFTParams()
*/
char *GPUFFTW::FFTErrorString(FFT_ERROR error){
    char *err = new char[1000];
    sprintf(err,"GPUFFTW: ");
    switch(error){
      case FFT_SUCCESS: sprintf(err,"No error.");
      break;
      case FFT_ARRAY_TOO_BIG: sprintf(err,"Not enough video RAM to FFT.
                                    Maximum array size is %d", MAX_SIZE);
      break;
```

```
      case FFT_ARRAY_TOO_SMALL: sprintf(err,"Input Array is too small.
                                  Minimum size: %d",MIN_SIZE);
      break;
      case FFT_NON_POWER_OF_TWO: sprintf(err,"Non power of two array
                                  - not handled");
      break;
      default: sprintf(err,"Unknown error code. This should not happen.");
      break;
    }
    return(err);
}


/** Sets internal parameters for performing fft on a given sized array.
    Returns the actual number of fft elements
*/
FFT_ERROR GPUFFTW::setFFTParams(int size){

  N=size;

  if(N<MIN_SIZE) return(FFT_ARRAY_TOO_SMALL);
  if(N>MAX_SIZE) return(FFT_ARRAY_TOO_BIG);

  //LOGN represents Log(texture width* texture height)
  //rather than log of number of elements.
  for(LOGN=0;N>>LOGN;LOGN++);
  LOGN=LOGN-3;
  Width = (1 << (LOGN - (int) (LOGN/2)));
  Height = (1 << (int) (LOGN/2));

  TILE_SIZE=64;
  if(TILE_SIZE > Width) TILE_SIZE=Width;
  if(TILE_SIZE > Height) TILE_SIZE=Height;

#ifdef GPU_DEBUG
  printf("\nGPUFFTW: Size of array = %d. Width = %d, height = %d,
  LOGN=%d, Tile Size = %d\n",size,Width,Height,LOGN,TILE_SIZE);
#endif

  return(FFT_SUCCESS);
}
#include "stopwatch.hpp"
FFT_ERROR GPUFFTW::cgpufft1d(float *real, float  *imag,
                             int size, int sign, int isreal){
  if(size < MIN_SIZE){
    return (FFT_ARRAY_TOO_SMALL);
```

```
}
//amount of data to be ffted on the GPU
int gpuSize=0;

//Find largest power of 2 smaller than size
int logSize=0;
for(;size>>logSize;logSize++);
logSize--;

gpuSize=1<<logSize;
int cpuSize=size-gpuSize;

if(cpuSize==0){
  FFT_ERROR err = setFFTParams(gpuSize);
  if(err!=FFT_SUCCESS) return(err);

  glActiveTextureARB( GL_TEXTURE3_ARB );
  glBindTexture(GL_TEXTURE_RECTANGLE_NV, textureid[3]);
  glTexSubImage2D(GL_TEXTURE_RECTANGLE_NV, 0, 0, 0,
                  Width, Height, GL_RGBA, GL_FLOAT, imag);


  uploadData(real);
  Stopwatch bitonic_timer;
  glFinish();
  bitonic_timer.Start();
  //for(int dummy=0;dummy<100;dummy++)

  if(isreal){
    //rearrange
  }
  cgpufft1d(sign);
  if(isreal){
    //rearrange
  }
  glFinish();
  bitonic_timer.Stop();
  printf("time:%4.5f \n", bitonic_timer.GetTime());


  readbackData(imag);

  glBindFramebufferEXT(GL_FRAMEBUFFER_EXT, fb);
  if(target==1)
    glReadBuffer( GL_COLOR_ATTACHMENT1_EXT);
  else
```

```
      glReadBuffer( GL_COLOR_ATTACHMENT0_EXT);

    glReadPixels(0, 0, Width, Height, GL_RGBA, GL_FLOAT, real);

    glBindFramebufferEXT(GL_FRAMEBUFFER_EXT, 0);

    if(sign==-1){
      for(int d=0;d<size;d++){
        real[d]/=size;
        imag[d]/=size;
      }

    }

  }
  else return(FFT_NON_POWER_OF_TWO);

  return(FFT_SUCCESS);
}


FFT_ERROR GPUFFTW::scgpufft1d(float *real, int size, int sign){
  float *reals = new float[size/2];
  float *imags = new float[size/2];
  FFT_ERROR result;


  unsigned long i,i1,i2,i3,i4,np3;
  float c1=0.5,c2,h1r,h1i,h2r,h2i;
  //Double precision for the trigonometric recurrences.
  double wr,wi,wpr,wpi,wtemp,theta;
  //Initialize the recurrence.
  theta=3.14159265358979323846/(double) (size>>1);
  int nsize=size/2;

  //convert the 1D real FFT into a complex FFT of half the size
  //this can be speeded up using a fragment program and is slow on CPUs
  if (sign == 1) {
    c2 = -0.5;
    for(i=0;i<nsize;i++){
      reals[i] = real[2*i];
      imags[i] = real[2*i+1];

    }
    result=cgpufft1d(reals,imags, size/2, sign, 1);
    for(i=0;i<nsize;i++){
```

```
    real[2*i] = reals[i];
    real[2*i+1] = imags[i];

  }
  delete reals;
  delete imags;
}
else {
  c2=0.5; //Otherwise set up for an inverse trans
  theta= -theta; //form.
}
wtemp=sin(0.5*theta);
wpr = -2.0*wtemp*wtemp;
wpi=sin(theta);
wr=1.0+wpr;
wi=wpi;
np3=size+3;
for (i=2;i<=(size>>2);i++) { //Case i=1 done separately below.
  i4=1+(i3=np3-(i2=1+(i1=i+i-1)));
  h1r= c1*(real[i1]+real[i3]); //The two separate transforms are sep
  h1i= c1*(real[i2]-real[i4]); //arated out of real.
  h2r = -c2*(real[i2]+real[i4]);
  h2i=c2*(real[i1]-real[i3]);
  //Here they are recombined to form the true transform
  //of the original real real.
  real[i1]=h1r+wr*h2r-wi*h2i;
  real[i2]=h1i+wr*h2i+wi*h2r;
  real[i3]=h1r-wr*h2r+wi*h2i;
  real[i4] = -h1i+wr*h2i+wi*h2r;
  wr=(wtemp=wr)*wpr-wi*wpi+wr; //The recurrence.
  wi=wi*wpr+wtemp*wpi+wi;
}
if (sign == 1) {
    //Squeeze the first and last real together to get them all
    //within the original array.
  real[0] = (h1r=real[0])+real[1];
  real[1] = h1r-real[1];
}
else {
  real[0]=c1*((h1r=real[0])+real[1]);
  real[1]=c1*(h1r-real[1]);

  //convert the 1D real FFT into a complex FFT of half the size
  //this can be speeded up using a fragment program and is slow on CPUs

  for(i=0;i<nsize;i++){
```

```
      reals[i] = real[2*i];
      imags[i] = real[2*i+1];

    }
    result=cgpufft1d(reals,imags, size/2, sign,1);
    for(i=0;i<nsize;i++){
      real[2*i] = reals[i];
      real[2*i+1] = imags[i];

    }
    delete reals;
    delete imags;

  } //case isign=-1.

  return result;
}


FFT_ERROR GPUFFTW::cGPUFFTW2d(float *real, float  *imag,
        int arraywidth, int arrayheight, int sign, int isreal){

  return FFT_ARRAY_TOO_SMALL;//2d FFT not incorporated in the library yet
                      //write the transpose function and use the 1-D FFT code

  if(arraywidth < MIN_SIZE || arrayheight<MIN_SIZE){
    return (FFT_ARRAY_TOO_SMALL);
  }
  //amount of data to be ffted on the GPU
  int gpuWidth=0;

  //Find largest power of 2 smaller than size
  int logWidth=0;
  for(;arraywidth>>logWidth;logWidth++);
  logWidth--;

  gpuWidth=1<<logWidth;
  int cpuSize=arraywidth-gpuWidth;

  if(cpuSize==0){
    //FFT_ERROR err = setFFTParams(gpuSize);
    Width= arraywidth/4;
    Height = arrayheight;
    logWidth-=2;
    LOGN=logWidth;
```

```
glActiveTextureARB( GL_TEXTURE3_ARB );
glBindTexture(GL_TEXTURE_RECTANGLE_NV, textureid[3]);
glTexSubImage2D(GL_TEXTURE_RECTANGLE_NV, 0, 0, 0,
                Width, Height, GL_RGBA, GL_FLOAT, imag);


uploadData(real);
Stopwatch bitonic_timer;
glFinish();
bitonic_timer.Start();
//for(int dummy=0;dummy<100;dummy++)

if(isreal){
  //rearrange
}
cgpufft1d(sign,1);
if(isreal){
  //rearrange
}

//transpose
Width=arrayheight/4;
Height = arraywidth;
int logHt=0;
for(;arrayheight>>logHt;logHt++);
logHt--;
LOGN=logHt-2;
cgpufft1d(sign,1);

//transpose again
Width= arraywidth/4;
Height = arrayheight;


glFinish();
bitonic_timer.Stop();
printf("time:%4.5f \n", bitonic_timer.GetTime());


readbackData(imag);

glBindFramebufferEXT(GL_FRAMEBUFFER_EXT, fb);
if(target==1)
  glReadBuffer( GL_COLOR_ATTACHMENT1_EXT);
else
```

```
      glReadBuffer( GL_COLOR_ATTACHMENT0_EXT);

    glReadPixels(0, 0, Width, Height, GL_RGBA, GL_FLOAT, real);

    glBindFramebufferEXT(GL_FRAMEBUFFER_EXT, 0);

    if(sign==-1){
      for(int d=0;d<Width;d++)
        for(int k=0;k<Height;k++){
          real[d]/=(arraywidth*arrayheight);
          imag[d]/=(arraywidth*arrayheight);;
        }
    }

  }
  else return(FFT_NON_POWER_OF_TWO);

  return(FFT_SUCCESS);
}


/** Send data to the GPU from the CPU */
void GPUFFTW::uploadData(void *array){
  //Upload into data into texture
  glActiveTextureARB( GL_TEXTURE2_ARB );
  glBindTexture(GL_TEXTURE_RECTANGLE_NV, textureid[1]);
  glTexSubImage2D(GL_TEXTURE_RECTANGLE_NV, 0, 0, 0,
                  Width, Height, GL_RGBA, GL_FLOAT, array);


}
/** Readback the data from the GPU to CPU */
void GPUFFTW::readbackData(void *data){
  glBindFramebufferEXT(GL_FRAMEBUFFER_EXT, fb);
  if(target==1)
    glReadBuffer( GL_COLOR_ATTACHMENT3_EXT);
  else
    glReadBuffer( GL_COLOR_ATTACHMENT2_EXT);

  glReadPixels(0, 0, Width, Height, GL_RGBA, GL_FLOAT, data);

  glBindFramebufferEXT(GL_FRAMEBUFFER_EXT, 0);
}

// Swaps the texture to "render from" and the texture to "render to"
inline void GPUFFTW::PingPong(){
```

```
  int temp=source;
  source = target;
  target = temp;

  if(target==0){
    //glDrawBuffer( GL_COLOR_ATTACHMENT0_EXT);
    glDrawBuffers(2, buffers1);
    glActiveTextureARB( GL_TEXTURE2_ARB );
    glBindTexture(GL_TEXTURE_RECTANGLE_NV, textureid[1]);
    glActiveTextureARB( GL_TEXTURE3_ARB );
    glBindTexture(GL_TEXTURE_RECTANGLE_NV, textureid[3]);

  }
  else{
    //glDrawBuffer( GL_COLOR_ATTACHMENT1_EXT);
    glDrawBuffers(2, buffers2);
    glActiveTextureARB( GL_TEXTURE2_ARB );
    glBindTexture(GL_TEXTURE_RECTANGLE_NV, textureid[0]);
    glActiveTextureARB( GL_TEXTURE3_ARB );
    glBindTexture(GL_TEXTURE_RECTANGLE_NV, textureid[2]);

  }
}



inline void GPUFFTW::PingPong2to4(){
  int temp=source;
    source = target;
    target = temp;
    if(target==0){
      glFramebufferTexture2DEXT(GL_FRAMEBUFFER_EXT,
                                GL_COLOR_ATTACHMENT0_EXT,
                                GL_TEXTURE_RECTANGLE_NV, textureid[0], 0);


      glFramebufferTexture2DEXT(GL_FRAMEBUFFER_EXT,
                                GL_COLOR_ATTACHMENT1_EXT,
                                GL_TEXTURE_RECTANGLE_NV, textureid[4], 0);


      glFramebufferTexture2DEXT(GL_FRAMEBUFFER_EXT,
                                GL_COLOR_ATTACHMENT2_EXT,
                                GL_TEXTURE_RECTANGLE_NV, textureid[2], 0);
```

```
      glFramebufferTexture2DEXT(GL_FRAMEBUFFER_EXT,
                                GL_COLOR_ATTACHMENT3_EXT,
                                GL_TEXTURE_RECTANGLE_NV, textureid[5], 0);

      glDrawBuffers(4, buffers41);
      glActiveTextureARB( GL_TEXTURE2_ARB );
      glBindTexture(GL_TEXTURE_RECTANGLE_NV, textureid[1]);
      glActiveTextureARB( GL_TEXTURE3_ARB );
      glBindTexture(GL_TEXTURE_RECTANGLE_NV, textureid[3]);

    }
    else{
      glFramebufferTexture2DEXT(GL_FRAMEBUFFER_EXT,
                                GL_COLOR_ATTACHMENT0_EXT,
                                GL_TEXTURE_RECTANGLE_NV, textureid[4], 0);

      glFramebufferTexture2DEXT(GL_FRAMEBUFFER_EXT,
                                GL_COLOR_ATTACHMENT1_EXT,
                                GL_TEXTURE_RECTANGLE_NV, textureid[1], 0);

      glFramebufferTexture2DEXT(GL_FRAMEBUFFER_EXT,
                                GL_COLOR_ATTACHMENT2_EXT,
                                GL_TEXTURE_RECTANGLE_NV, textureid[5], 0);

      glFramebufferTexture2DEXT(GL_FRAMEBUFFER_EXT,
                                GL_COLOR_ATTACHMENT3_EXT,
                                GL_TEXTURE_RECTANGLE_NV, textureid[3], 0);


      glDrawBuffers(4, buffers42);
      glActiveTextureARB( GL_TEXTURE2_ARB );
      glBindTexture(GL_TEXTURE_RECTANGLE_NV, textureid[0]);
      glActiveTextureARB( GL_TEXTURE3_ARB );
      glBindTexture(GL_TEXTURE_RECTANGLE_NV, textureid[2]);

    }
}

inline void GPUFFTW::PingPong4to2(){
  if(target==0){
      glFramebufferTexture2DEXT(GL_FRAMEBUFFER_EXT,
                                GL_COLOR_ATTACHMENT1_EXT,
                                GL_TEXTURE_RECTANGLE_NV, textureid[1], 0);
      glFramebufferTexture2DEXT(GL_FRAMEBUFFER_EXT,
                                GL_COLOR_ATTACHMENT3_EXT,
                                GL_TEXTURE_RECTANGLE_NV, textureid[3], 0);
```

```
      glDrawBuffers(2, buffers1);
      glActiveTextureARB( GL_TEXTURE2_ARB );
      glBindTexture(GL_TEXTURE_RECTANGLE_NV, textureid[4]);
      glActiveTextureARB( GL_TEXTURE3_ARB );
      glBindTexture(GL_TEXTURE_RECTANGLE_NV, textureid[5]);

    }
    else{
      glFramebufferTexture2DEXT(GL_FRAMEBUFFER_EXT,
                                GL_COLOR_ATTACHMENT0_EXT,
                                GL_TEXTURE_RECTANGLE_NV, textureid[0], 0);


      glFramebufferTexture2DEXT(GL_FRAMEBUFFER_EXT,
                                GL_COLOR_ATTACHMENT2_EXT,
                                GL_TEXTURE_RECTANGLE_NV, textureid[2], 0);

      glDrawBuffers(2, buffers2);
      glActiveTextureARB( GL_TEXTURE2_ARB );
      glBindTexture(GL_TEXTURE_RECTANGLE_NV, textureid[4]);
      glActiveTextureARB( GL_TEXTURE3_ARB );
      glBindTexture(GL_TEXTURE_RECTANGLE_NV, textureid[5]);

    }

}



#define BLOCK_SIZE 64
double pi= 3.14159265358979323846;
void GPUFFTW::cgpufft1d( float sign, int usedfor2dfft ){
  int i,j,z;
  //Go into FBO mode, bind the uploaded data as the source texture
  glBindFramebufferEXT(GL_FRAMEBUFFER_EXT, fb);
  glDrawBuffer( GL_COLOR_ATTACHMENT0_EXT);
  glColorMask(~0,~0,~0,~0);
  glEnable(GL_TEXTURE_RECTANGLE_NV);
  glActiveTextureARB( GL_TEXTURE2_ARB );
    glBindTexture(GL_TEXTURE_RECTANGLE_NV, textureid[1]);

  glActiveTextureARB( GL_TEXTURE3_ARB );
    glBindTexture(GL_TEXTURE_RECTANGLE_NV, textureid[3]);
```

```
source=0;
target=1;
//the first two stages are special stages since the FFT is performed
//on chunks of size 4
glActiveTextureARB( GL_TEXTURE2_ARB );
PingPong();
fullscreenfft1ststage.Bind();

GLfloat Vertices[] = {  0.0f, 0.0f,            // Texture 0 Coord 0
            0.0f, (float)(Height/2),          // Texture 1 Coord 0
            (float)(Width/2), 0.0f,           // Texture 2 Coord 0
            0.0f, 0.0f,              // Vertex    Coord 0
            (float)(Width/2), 0.0f,           // Texture 0 Coord 1
            (float)(Width/2), (float)(Height/2),// Texture 1 Coord 1
            (float)(Width/2), 0.0f,        // Texture 2 Coord 1
            (float)Width, 0.0f,           // Vertex    Coord 1
            (float)(Width/2), (float)(Height/2),// Texture 0 Coord 2
            (float)(Width/2), (float)(Height),  // Texture 1 Coord 2
            (float)(Width/2), 0.0f,        // Texture 2 Coord 2
            (float)Width, (float)Height,    // Vertex    Coord 2
            0.0f, (float)(Height/2),          // Texture 0 Coord 3
            0.0f, (float)(Height),            // Texture 1 Coord 3
            (float)(Width/2),  0.0f,          // Texture 2 Coord 3
            0.0f, (float)Height          // Vertex   Coord 3
          };

// Load the texture coordinate
glEnableClientState(GL_TEXTURE_COORD_ARRAY);
glClientActiveTexture ( GL_TEXTURE0 );
glTexCoordPointer( 2, GL_FLOAT, 8 * sizeof(GLfloat), Vertices);

// Load the texture coordinate
glEnableClientState( GL_TEXTURE_COORD_ARRAY );
glClientActiveTexture ( GL_TEXTURE1 );
glTexCoordPointer( 2, GL_FLOAT, 8 * sizeof(GLfloat), &(Vertices[2]));

// Load the texture coordinate
glEnableClientState( GL_TEXTURE_COORD_ARRAY );
glClientActiveTexture ( GL_TEXTURE2 );
glTexCoordPointer( 2, GL_FLOAT, 8 * sizeof(GLfloat), &(Vertices[4]));

// Load the vertex position
glEnableClientState(GL_VERTEX_ARRAY);
glVertexPointer( 2, GL_FLOAT, 8 * sizeof(GLfloat), &(Vertices[6]));
```

```
glDrawArrays ( GL_QUADS, 0, 4 );

PingPong();
fullscreenfft2ndstagefp.BindProg();

GLfloat Vertices2[] = {  0.0f, 0.0f,          // Texture 0 Coord 0
             0.0f, (float)(Height/2),          // Texture 1 Coord 0
            (float)(Width/2), 0.0f,            // Texture 2 Coord 0
            1.0f, (float) cos(-sign*2*pi/(4.0)),// Texture 3 Coord 0
            0.0f, (float) sin(-sign*2*pi/(4.0)),// Texture 4 Coord 0
            0.0f, 0.0f,              // Vertex    Coord 0
            (float)(Width/2), 0.0f,            // Texture 0 Coord 1
            (float)(Width/2), (float)(Height/2),// Texture 1 Coord 1
            (float)(Width/2), 0.0f,        // Texture 2 Coord 1
            1.0f, (float) cos(-sign*2*pi/(4.0)),// Texture 3 Coord 1
            0.0f, (float) sin(-sign*2*pi/(4.0)),// Texture 4 Coord 1
            (float)Width, 0.0f,          // Vertex    Coord 1
            (float)(Width/2), (float)(Height/2),// Texture 0 Coord 2
            (float)(Width/2), (float)(Height),  // Texture 1 Coord 2
            (float)(Width/2), 0.0f,        // Texture 2 Coord 2
            1.0f, (float) cos(-sign*2*pi/(4.0)),// Texture 3 Coord 2
            0.0f, (float) sin(-sign*2*pi/(4.0)),// Texture 4 Coord 2
            (float)Width, (float)Height,    // Vertex    Coord 2
            0.0f, (float)(Height/2),           // Texture 0 Coord 3
            0.0f, (float)(Height),             // Texture 1 Coord 3
            (float)(Width/2),  0.0f,           // Texture 2 Coord 3
            1.0f, (float) cos(-sign*2*pi/(4.0)),// Texture 3 Coord 2
            0.0f, (float) sin(-sign*2*pi/(4.0)),// Texture 4 Coord 2
            0.0f, (float)Height          // Vertex   Coord 3
          };

// Load the texture coordinate
glEnableClientState(GL_TEXTURE_COORD_ARRAY);
glClientActiveTexture ( GL_TEXTURE0 );
glTexCoordPointer( 2, GL_FLOAT, 12 * sizeof(GLfloat), Vertices2);

// Load the texture coordinate
glEnableClientState( GL_TEXTURE_COORD_ARRAY );
glClientActiveTexture ( GL_TEXTURE1 );
glTexCoordPointer( 2, GL_FLOAT, 12 * sizeof(GLfloat), &(Vertices2[2]));

// Load the texture coordinate
glEnableClientState( GL_TEXTURE_COORD_ARRAY );
glClientActiveTexture ( GL_TEXTURE2 );
glTexCoordPointer( 2, GL_FLOAT, 12 * sizeof(GLfloat), &(Vertices2[4]));
```

```
// Load the texture coordinate
glEnableClientState( GL_TEXTURE_COORD_ARRAY );
glClientActiveTexture ( GL_TEXTURE3 );
glTexCoordPointer( 2, GL_FLOAT, 12 * sizeof(GLfloat), &(Vertices2[6]));

// Load the texture coordinate
glEnableClientState( GL_TEXTURE_COORD_ARRAY );
glClientActiveTexture ( GL_TEXTURE4 );
glTexCoordPointer( 2, GL_FLOAT, 12 * sizeof(GLfloat), &(Vertices2[8]));

// Load the vertex position
glEnableClientState(GL_VERTEX_ARRAY);
glVertexPointer( 2, GL_FLOAT, 12 * sizeof(GLfloat), &(Vertices2[10]));

glActiveTexture ( GL_TEXTURE2 );

glDrawArrays ( GL_QUADS, 0, 4 );


//row stages
int t;

TILE_SIZE=BLOCK_SIZE/2;
if(TILE_SIZE > Height) TILE_SIZE=Height;
if(TILE_SIZE > Width) TILE_SIZE=Width;

//these stages are implemented by drawing column aligned quads
for(t=3; t<=(LOGN+2);t++){

  int quad_width = (1<< (t-1-2));
  if(quad_width==Width) break;

  int num_quads = Width/quad_width;
  int s=0;
  int tex_s=-quad_width;

  float angle1= 2*pi/(1<<t);
  float angle2 = angle1 *(1<<(t-1));

  angle2=-angle2;
  angle1=-angle1;

  fftstages3_to_rowwidthfp.BindProg();

  PingPong2to4();
```

```
s=0;
int h=TILE_SIZE;
h = quad_width < TILE_SIZE ? TILE_SIZE : Height;
int num_splits= Height/h;
int ht=0;
int l=quad_width <TILE_SIZE? TILE_SIZE/quad_width:1;

int num_quad_splits = quad_width < TILE_SIZE? Width/TILE_SIZE : num_quads;

for(int g=0; g< num_splits;g++){
    s=0;
    int tex_s=-quad_width;

    for(int d=0; d< num_quad_splits;d++)
    {
      for(int start=0; start<l; start++){
        float ang_diff = 2*angle1;
        float ang_diff_x = 2*angle1;

        if((l==1 && d%2==0) || (l>1 && start%2==0)){//addition
          tex_s+=quad_width;

          float angle10 = 0-ang_diff;
          float angle11 = angle1-ang_diff;
          float angle12 = 2*angle1-ang_diff;
          float angle13 = 3*angle1-ang_diff;

          GLfloat Vertices3[] = {  // Texture 0 Coord 0
  (float)tex_s, (float)(ht/2), (float)((ht+Height)/2),
  // Texture 1 Coord 0
  (float)(tex_s+Width/2), (float)(sign*angle10), (float)(sign*angle11),
  // Texture 2 Coord 0
  (float)(sign*angle12), (float)(sign*angle13), 0.0f,
  // Vertex    Coord 0
  (float)s, (float)ht, 0.0f,
  // Texture 0 Coord 1
  (float)(tex_s+quad_width), (float)(ht/2), (float)((ht+Height)/2),
  // Texture 1 Coord 1
  (float)(tex_s+quad_width+Width/2),
  (float)(sign*(angle2-4*angle1+ang_diff_x)),
  (float)(sign*(angle2-3* angle1+ang_diff_x)),
  // Texture 2 Coord 1
  (float)(sign*(angle2-2*angle1+ang_diff_x)),
  (float)(sign*(angle2-angle1+ang_diff_x)), 0.0f,
  // Vertex    Coord 1
```

```
(float)(s+quad_width), (float)ht, 0.0f,
//Texture 0 Coord 2
(float)(tex_s+quad_width), (float)((ht+h)/2),
(float)((ht+h+Height)/2),
// Texture 1 Coord 2
(float)(tex_s+quad_width+Width/2),
(float)(sign*(angle2-4*angle1+ang_diff_x)),
(float)(sign*(angle2-3* angle1+ang_diff_x)),
// Texture 2 Coord 2
(float)(sign*(angle2-2*angle1+ang_diff_x)),
(float)(sign*(angle2-angle1+ang_diff_x)), 0.0f,
// Vertex   Coord 2
(float)(s+quad_width), (float)(ht+h), 0.0f,
// Texture 0 Coord 3
(float)tex_s, (float)((ht+h)/2), (float)((ht+h+Height)/2),
// Texture 1 Coord 3
(float)(tex_s+Width/2), (float)(sign*angle10), (float)(sign*angle11),
// Texture 2 Coord 3
(float)(sign*angle12), (float)(sign*angle13), 0.0f,
// Vertex   Coord 3
(float)s, (float)(ht+h), 0.0f
        };

        // Load the texture coordinate
        glEnableClientState(GL_TEXTURE_COORD_ARRAY);
        glClientActiveTexture ( GL_TEXTURE0 );
        glTexCoordPointer( 3, GL_FLOAT, 12 * sizeof(GLfloat),
                           Vertices3);

        // Load the texture coordinate
        glEnableClientState( GL_TEXTURE_COORD_ARRAY );
        glClientActiveTexture ( GL_TEXTURE1 );
        glTexCoordPointer( 3, GL_FLOAT, 12 * sizeof(GLfloat),
                           &(Vertices3[3]));

        // Load the texture coordinate
        glEnableClientState( GL_TEXTURE_COORD_ARRAY );
        glClientActiveTexture ( GL_TEXTURE2 );
        glTexCoordPointer( 3, GL_FLOAT, 12 * sizeof(GLfloat),
                           &(Vertices3[6]));

        // Load the vertex position
        glEnableClientState(GL_VERTEX_ARRAY);
        glVertexPointer( 3, GL_FLOAT, 12 * sizeof(GLfloat),
                        &(Vertices3[9]));
```

```
          glActiveTexture ( GL_TEXTURE2 );

          glDrawArrays ( GL_QUADS, 0, 4 );

        }
        s+=quad_width;
      }
    }
    ht+= h;
  }

copyfp.BindProg();
PingPong4to2();

ht=0;
for(z=0; z< num_splits;z++){
  s=0;

  for(int d=0; d< num_quad_splits;d++)
  {
    for(int start=0; start<l; start++)
    {
      // printf ("ht: %d / s: %d\n", ht, s);

      if((l==1 && d%2==1) || (l>1 && start%2==1)) //addition
      {

        GLfloat Vertices4[] = {
        (float)(s-quad_width), (float)ht,  // Texture 0 Coord 0
        (float)s, (float)ht,           // Vertex    Coord 0
        (float)s, (float)ht,             // Texture 0 Coord 1
        (float)(s+quad_width), (float)ht,  // Vertex    Coord 1
        (float)s, (float)(ht+h),       // Texture 0 Coord 2
        (float)(s+quad_width), (float)(ht+h),// Vertex    Coord 2
        (float)(s-quad_width), (float)(ht+h),// Texture 0 Coord 3
        (float)s, (float)(ht+h)         // Vertex   Coord 3
                  };


        // Use the program object
          glUseProgram ( copyfp.getProgId() );
        printf("Location Texture0: %d / Location Vertex: %d\n",
        copyfp.getLocTexture0(), copyfp.getLocVertex());

        // Load the texture coordinate
        glVertexAttribPointer ( copyfp.getLocTexture0(), 2,
```

```
            GL_FLOAT, GL_FALSE, 4 * sizeof(GLfloat), &(Vertices4) );

            // glEnableVertexAttribArray ( copyfp.getLocTexture0() );

            glActiveTexture ( GL_TEXTURE0 );

               glBindTexture ( GL_TEXTURE_RECTANGLE_NV, textureid[0] );

            // Load the vertex position
            glVertexAttribPointer ( copyfp.getLocVertex(), 2,
            GL_FLOAT, GL_FALSE, 4 * sizeof(GLfloat), &(Vertices4[2]) );

            glDrawArrays ( GL_QUADS, 0, 4 );

          }
          s+=quad_width;

        }



      }

      ht+= h;

    }

    // glActiveTexture ( GL_TEXTURE2 );



}
if(usedfor2dfft){
  glBindFramebufferEXT(GL_FRAMEBUFFER_EXT, 0);
  return;
}
TILE_SIZE=BLOCK_SIZE;
if(TILE_SIZE > Width) TILE_SIZE=Width;
if(TILE_SIZE > Height) TILE_SIZE=Height;
//this single stage is the transition
//from column-aligned quads to row-aligned quads
if(t<=LOGN+2){
  fullscreenrowwidthfp.BindProg();

  int quad_width=Width;
```

```
float angle1= 2*pi/(1<<t);
float angle2 = angle1 *(1<<(t-1));

angle2=-angle2;
angle1=-angle1;
float ang_diff = 2*angle1;
float ang_diff_x = 2*angle1;
PingPong();

GLfloat Vertices5[] = {
// Texture 0 Coord 0
0.0f, 0.0f,
// Texture 1 Coord 0
0.0f, (float)(Height/2),
// Texture 3 Coord 0
(float)(sign*(0-ang_diff)), (float)(sign*(angle1-ang_diff)),
// Texture 4 Coord 0
(float)(sign*(2*angle1-ang_diff)), (float)(sign*(3*angle1-ang_diff)),
// Vertex    Coord 0
0.0f, 0.0f,
// Texture 0 Coord 1
(float)(Width), 0.0f,
// Texture 1 Coord 1
(float)Width, (float)(Height/2),
// Texture 3 Coord 1
(float)(sign*(angle2-4*angle1+ang_diff_x)),
(float)(sign*(angle2-3* angle1+ang_diff_x)),
// Texture 4 Coord 1
(float)(sign*(angle2-2*angle1+ang_diff_x)),
(float)(sign*(angle2-angle1+ang_diff_x)),
// Vertex    Coord 1
(float)Width, 0.0f,
// Texture 0 Coord 2
(float)(Width), (float)(Height/2),
// Texture 1 Coord 2
(float)(Width), (float)(Height),
// Texture 3 Coord 1
(float)(sign*(angle2-4*angle1+ang_diff_x)),
(float)(sign*(angle2-3* angle1+ang_diff_x)),
// Texture 4 Coord 1
(float)(sign*(angle2-2*angle1+ang_diff_x)),
(float)(sign*(angle2-angle1+ang_diff_x)),
// Vertex    Coord 2
(float)Width, (float)Height,
// Texture 0 Coord 3
0.0f, (float)(Height/2),
```

```
  // Texture 1 Coord 3
  0.0f, (float)(Height),
  // Texture 3 Coord 3
  (float)(sign*(0-ang_diff)), (float)(sign*(angle1-ang_diff)),
  // Texture 4 Coord 3
  (float)(sign*(2*angle1-ang_diff)), (float)(sign*(3*angle1-ang_diff)),
  // Vertex   Coord 3
  0.0f, (float)Height
            };


  // Load the texture coordinate
  glEnableClientState( GL_TEXTURE_COORD_ARRAY );
  glClientActiveTexture ( GL_TEXTURE0 );
  glTexCoordPointer( 2, GL_FLOAT, 10 * sizeof(GLfloat), &(Vertices5));

  // Load the texture coordinate
  glEnableClientState( GL_TEXTURE_COORD_ARRAY );
  glClientActiveTexture ( GL_TEXTURE1 );
  glTexCoordPointer( 2, GL_FLOAT, 10 * sizeof(GLfloat), &(Vertices5[2]));

  // Load the texture coordinate
  glEnableClientState( GL_TEXTURE_COORD_ARRAY );
  glClientActiveTexture ( GL_TEXTURE3 );
  glTexCoordPointer( 2, GL_FLOAT, 10 * sizeof(GLfloat), &(Vertices5[4]));

  // Load the texture coordinate
  glEnableClientState( GL_TEXTURE_COORD_ARRAY );
  glClientActiveTexture ( GL_TEXTURE4 );
  glTexCoordPointer( 2, GL_FLOAT, 10 * sizeof(GLfloat), &(Vertices5[6]));

  // Load the vertex position
  glEnableClientState(GL_VERTEX_ARRAY);
  glVertexPointer( 2, GL_FLOAT, 10 * sizeof(GLfloat), &(Vertices5[8]));

  glActiveTexture ( GL_TEXTURE2 );

  glDrawArrays ( GL_QUADS, 0, 4 );

  t++;
}


//perform the remaining stages using row-aligned quads
for(;t<=(LOGN+2);t++){
```

```
int quad_height = (1<<(t-1))/(Width*4);

int num_quads = Height/quad_height;
int s=0;
int tex_s=-quad_height;
fftstages_twicerowwidth_to_n_fp.BindProg();
PingPong2to4();

float angle1= 2*pi/((1<<t));
// printf ("angle1: %f\n", angle1);
float angle2 = angle1 *(1<<(t-1));
// printf ("angle2: %f\n", angle2);
angle1=-angle1;
angle2=-angle2;
int w=TILE_SIZE;
w = quad_height < TILE_SIZE ? TILE_SIZE : Width;

int num_splits= Width/w;
int wt=0;

int l=quad_height <TILE_SIZE? TILE_SIZE/quad_height:1;
int num_quad_splits = quad_height < TILE_SIZE? Height/TILE_SIZE : num_quads;

for(z=0; z< num_splits;z++){
  s=0;
  int tex_s=-quad_height;

  for(int d=0; d< num_quad_splits;d++)
  {

    for(int start=0; start<l; start++){
      float ang_diff_y = 0.5*(4*Width*angle1);//divide by zero issue???
      float ang_diff_x = 2*angle1 ;

      float offset=0;

        if((l==1 && d%2==0) || (l>1 && start%2==0)){//addition
          tex_s+=quad_height;

          GLfloat Vertices6[] = {
// Texture 0 Coord 0
(float)wt, (float)tex_s,
// Texture 1 Coord 0
(float)wt, (float)(tex_s+Height/2),
// Texture 3 Coord 0
(float)(sign*(wt*angle1*4-ang_diff_x- ang_diff_y+offset)),
```

```
(float)(sign*(wt*angle1*4+angle1-ang_diff_x- ang_diff_y+offset)),
// Texture 4 Coord 0
(float)(sign*(wt*angle1*4+2*angle1-ang_diff_x- ang_diff_y+offset)),
(float)(sign*(wt*angle1*4+3*angle1-ang_diff_x - ang_diff_y+offset)),
// Vertex    Coord 0
(float)wt, (float)s,
// Texture 0 Coord 1
(float)(wt+w), (float)tex_s,
// Texture 1 Coord 1
(float)(wt+w), (float)(tex_s+Height/2),
// Texture 3 Coord 1
(float)(sign*((wt+w)*angle1*4-4*angle1+ang_diff_x-ang_diff_y+offset)),
(float)(sign*((wt+w)*angle1*4-3* angle1+ang_diff_x-ang_diff_y+offset)),
// Texture 4 Coord 1
(float)(sign*((wt+w)*angle1*4-2*angle1+ang_diff_x-ang_diff_y+offset)),
(float)(sign*((wt+w)*angle1*4-angle1+ang_diff_x-ang_diff_y+offset)),
// Vertex    Coord 1
(float)(wt+w), (float)s,
// Texture 0 Coord 2
(float)(wt+w), (float)(tex_s+quad_height),
// Texture 1 Coord 2
(float)(wt+w), (float)(tex_s+quad_height+Height/2),
// Texture 3 Coord 1
(float)(sign*((quad_height-1)*Width*angle1*4 + (wt+w)*angle1*4
                        -4*angle1+ang_diff_x+ang_diff_y+offset)),
(float)(sign*((quad_height-1)*Width*angle1*4 + (wt+w)*angle1*4
                          -3*angle1+ang_diff_x+ang_diff_y+offset)),
// Texture 4 Coord 1
(float)(sign*((quad_height-1)*Width*angle1*4 + (wt+w)*angle1*4
                        -2*angle1+ang_diff_x+ang_diff_y+offset)),
(float)(sign*((quad_height-1)*Width*angle1*4 + (wt+w)*angle1*4
                        -angle1+ang_diff_x+ang_diff_y+offset)),
    // Vertex    Coord 2
(float)(wt+w), (float)(s+quad_height),
    // Texture 0 Coord 3
(float)wt, (float)(tex_s+quad_height),
// Texture 1 Coord 3
(float)wt, (float)(tex_s+quad_height+Height/2),
// Texture 3 Coord 3
(float)(sign*((quad_height-1)*Width*angle1*4+wt *4*angle1
                           -ang_diff_x+ ang_diff_y+offset)),
(float)(sign*((quad_height-1)*Width*angle1*4+wt *4*angle1
                           +angle1-ang_diff_x+ ang_diff_y+offset)),
// Texture 4 Coord 3
(float)(sign*((quad_height-1)*Width*angle1*4+wt*4*angle1
                           +2*angle1-ang_diff_x+ ang_diff_y+offset)),
```

```
(float)(sign*((quad_height-1)*Width*angle1*4+wt*4*angle1+3*angle1
                            -ang_diff_x+ang_diff_y+offset)),
// Vertex   Coord 3
(float)wt, (float)(s+quad_height)
          };


// Load the texture coordinate
glEnableClientState( GL_TEXTURE_COORD_ARRAY );
glClientActiveTexture ( GL_TEXTURE0 );
glTexCoordPointer( 2, GL_FLOAT, 10 * sizeof(GLfloat), &(Vertices6));

// Load the texture coordinate
glEnableClientState( GL_TEXTURE_COORD_ARRAY );
glClientActiveTexture ( GL_TEXTURE1 );
glTexCoordPointer( 2, GL_FLOAT, 10 * sizeof(GLfloat), &(Vertices6[2]));

// Load the texture coordinate
glEnableClientState( GL_TEXTURE_COORD_ARRAY );
glClientActiveTexture ( GL_TEXTURE3 );
glTexCoordPointer( 2, GL_FLOAT, 10 * sizeof(GLfloat), &(Vertices6[4]));

// Load the texture coordinate
glEnableClientState( GL_TEXTURE_COORD_ARRAY );
glClientActiveTexture ( GL_TEXTURE4 );
glTexCoordPointer( 2, GL_FLOAT, 10 * sizeof(GLfloat), &(Vertices6[6]));

// Load the vertex position
glEnableClientState(GL_VERTEX_ARRAY);
glVertexPointer( 2, GL_FLOAT, 10 * sizeof(GLfloat), &(Vertices6[8]));

glActiveTexture ( GL_TEXTURE2 );

glDrawArrays ( GL_QUADS, 0, 4 );

      }
      s+=quad_height;
    }

  }


  wt+=w;
}

copyfp.BindProg();
```

```
PingPong4to2();
wt=0;

for(z=0; z< num_splits;z++){
  s=0;
  int tex_s=-quad_height;

  for(int d=0; d< num_quad_splits;d++)
  {

    for(int start=0; start<l; start++){
      float ang_diff_y = 0.5*(4*Width*angle1);//divide by zero issue???
      float ang_diff_x = 2*angle1 ;

      float offset=0;

      if((l==1 && d%2==1) || (l>1 && start%2==1)){//addition

        GLfloat Vertices7[] = {
        (float)wt, (float)(s-quad_height),    // Texture 0 Coord 0
        (float)wt, (float)s,            // Vertex    Coord 0
        (float)(wt+w), (float)(s-quad_height),  // Texture 0 Coord 1
        (float)(wt+w), (float)s,        // Vertex    Coord 1
        (float)(wt+w), (float)s,        // Texture 0 Coord 2
        (float)(wt+w), (float)(s+quad_height),  // Vertex    Coord 2
        (float)wt, (float)s,          // Texture 0 Coord 3
        (float)wt, (float)(s+quad_height)    // Vertex   Coord 3
                  };

        // Load the texture coordinate
        glEnableClientState( GL_TEXTURE_COORD_ARRAY );
        glClientActiveTexture ( GL_TEXTURE0 );
        glTexCoordPointer( 2, GL_FLOAT, 4 * sizeof(GLfloat), Vertices7);

        glEnableClientState(GL_VERTEX_ARRAY);
        glVertexPointer( 2, GL_FLOAT, 4 * sizeof(GLfloat), &(Vertices7[2]));

        glDrawArrays ( GL_QUADS, 0, 4 );

      }
      s+=quad_height;
    }

  }

  wt+=w;
```

```
    }

  }

  glBindFramebufferEXT(GL_FRAMEBUFFER_EXT, 0);
}
```

A.4 shahelp.h

```
/*****************************************************************************\

Copyright 2019 Barcelona Supercomputing Center and
Universitat Politecnica de Catalunya.

Contact:
M. Benito or L. Kosmidis
Department of Computer Architecture - Operating Systems
marc.benito@bsc.es - leonidas.kosmidis@bsc.es

Copyright 2005 The University of North Carolina at Chapel Hill.
All Rights Reserved.

Permission to use, copy, modify and distribute this software and its
documentation for educational, research and non-profit purposes, without
fee, and without a written agreement is hereby granted, provided that the
above copyright notice and the following three paragraphs appear in all
copies. Any use in a commercial organization requires a separate license.

IN NO EVENT SHALL THE UNIVERSITY OF NORTH CAROLINA AT CHAPEL HILL BE LIABLE
TO ANY PARTY FOR DIRECT, INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL
DAMAGES, INCLUDING LOST PROFITS, ARISING OUT OF THE USE OF THIS SOFTWARE AND
ITS DOCUMENTATION, EVEN IF THE UNIVERSITY OF NORTH CAROLINA HAVE BEEN
ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.


Permission to use, copy, modify and distribute this software and its
documentation for educational, research and non-profit purposes, without
fee, and without a written agreement is hereby granted, provided that the
above copyright notice and the following three paragraphs appear in all
copies.

THE UNIVERSITY OF NORTH CAROLINA SPECIFICALLY DISCLAIM ANY WARRANTIES,
INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND
FITNESS FOR A PARTICULAR PURPOSE.  THE SOFTWARE PROVIDED HEREUNDER IS ON AN
"AS IS" BASIS, AND THE UNIVERSITY OF NORTH CAROLINA HAS NO OBLIGATION TO
PROVIDE MAINTENANCE, SUPPORT, UPDATES, ENHANCEMENTS, OR MODIFICATIONS.


--------------------------------
|Please send all BUG REPORTS to:  |
|                                 |
|   geom@cs.unc.edu               |
```

```
/                                /
--------------------------------


The authors may be contacted via:

US Mail:          N. Govindaraju
                    Department of Computer Science
                    Sitterson Hall, CB #3175
                    University of North Carolina
                    Chapel Hill, NC 27599-3175

\**************************************************************************/

/**************************************************************************\

This API will help to port old OpenGL to newer versions

\**************************************************************************/

#ifndef _SHAHELP_H
#define _SHAHELP_H

#include <string.h>
#ifdef WIN32
#include <windows.h>
#else
#include <GL/gl.h>
#include <GL/glx.h>
#endif
#include <GL/glu.h>
class ARBFProg{
public:
  ARBFProg(){}
  ~ARBFProg();

  void Load(const char* progname, char* prog);
  void Bind();
  void BindProg();
  void Release();

  GLuint prog_id;
};

class ShaHelp{
public:
```

```
  ShaHelp(){}
  ~ShaHelp();

  void Load(const char* progname, char* prog);
  void Bind();
  void BindProg();
  void Release();
  GLuint getProgId();
  GLint getLocVertex();
  GLint getLocTexture0();

  GLuint prog_id;
  GLint locVertex, locTexture0;
  GLint locSampler2, locSampler3;

};

#endif
```

A.5 shahelp.cpp

```
/*****************************************************************************\

Copyright 2019 Barcelona Supercomputing Center and
Universitat Politecnica de Catalunya.

Contact:
M. Benito or L. Kosmidis
Department of Computer Architecture - Operating Systems
marc.benito@bsc.es - leonidas.kosmidis@bsc.es

Copyright 2005 The University of North Carolina at Chapel Hill.
All Rights Reserved.

Permission to use, copy, modify and distribute this software and its
documentation for educational, research and non-profit purposes, without
fee, and without a written agreement is hereby granted, provided that the
above copyright notice and the following three paragraphs appear in all
copies. Any use in a commercial organization requires a separate license.

IN NO EVENT SHALL THE UNIVERSITY OF NORTH CAROLINA AT CHAPEL HILL BE LIABLE
TO ANY PARTY FOR DIRECT, INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL
DAMAGES, INCLUDING LOST PROFITS, ARISING OUT OF THE USE OF THIS SOFTWARE AND
ITS DOCUMENTATION, EVEN IF THE UNIVERSITY OF NORTH CAROLINA HAVE BEEN
ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.


Permission to use, copy, modify and distribute this software and its
documentation for educational, research and non-profit purposes, without
fee, and without a written agreement is hereby granted, provided that the
above copyright notice and the following three paragraphs appear in all
copies.

THE UNIVERSITY OF NORTH CAROLINA SPECIFICALLY DISCLAIM ANY WARRANTIES,
INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND
FITNESS FOR A PARTICULAR PURPOSE.  THE SOFTWARE PROVIDED HEREUNDER IS ON AN
"AS IS" BASIS, AND THE UNIVERSITY OF NORTH CAROLINA HAS NO OBLIGATION TO
PROVIDE MAINTENANCE, SUPPORT, UPDATES, ENHANCEMENTS, OR MODIFICATIONS.


--------------------------------
|Please send all BUG REPORTS to:  |
|                                 |
|   geom@cs.unc.edu               |
```

```
/                                /
--------------------------------


The authors may be contacted via:

US Mail:          N. Govindaraju
                    Department of Computer Science
                    Sitterson Hall, CB #3175
                    University of North Carolina
                    Chapel Hill, NC 27599-3175

\**************************************************************************/
#include <stdio.h>

#ifdef WIN32
#include <windows.h>
#else
#include <GL/glx.h>
#endif
#include <GL/glu.h>
#include <shahelp.h>
typedef void (APIENTRY * PFNGenProgramsARB) (GLsizei n, unsigned int *ids);
typedef void (APIENTRY * PFNLoadProgramARB)
    (int target, unsigned int id, GLsizei len, const unsigned char *program);
typedef void (APIENTRY * PFNGLProgramStringARB)
    (GLenum target, GLenum format, GLsizei len, const void* string);
typedef void (APIENTRY * PFNGLBindProgramARB) (GLint target, GLuint id);
typedef void (APIENTRY * PFNGLDeleteProgramsARB) (GLsizei n, const GLuint *);
/* Modifications start */
typedef GLuint (APIENTRY * PFGLCreateShader) (GLenum type);
typedef void (APIENTRY * PFGLShaderSource) (GLuint shader, GLsizei count,
    const GLchar *const*string, const GLint *length);
typedef void (APIENTRY * PFGLCompileShader) (GLuint shader);
typedef void (APIENTRY * PFGLGetShaderiv)
    (GLuint shader, GLenum pname, GLint *params);
typedef GLuint (APIENTRY * PFGLCreateProgram) (void);
typedef void (APIENTRY * PFGLAttachShader) (GLuint program, GLuint shader);
typedef void (APIENTRY * PFGLBindAttribLocation)
    (GLuint program, GLuint index, const GLchar *name);
typedef void (APIENTRY * PFGLLinkProgram) (GLuint program);
typedef void (APIENTRY * PFGLGetProgramiv)
    (GLuint program, GLenum pname, GLint *params);
typedef GLint (APIENTRY * PFGLGetAttribLocation)
    (GLuint program, const GLchar *name);
typedef GLint (APIENTRY * PFGLGetUniformLocation)
```

```
    (GLuint program, const GLchar *name);
/* Modifications end */


#define GL_FRAGMENT_PROGRAM_ARB 0x8804
#define GL_MAX_TEXTURE_COORDS_ARB 0x8871
#define GL_MAX_TEXTURE_IMAGE_UNITS_ARB 0x8872
#define GL_FRAGMENT_PROGRAM_BINDING_ARB 0x8873
#define GL_PROGRAM_ERROR_STRING_ARB 0x8874

PFNGenProgramsARB glGenProgramsARB;
PFNLoadProgramARB glLoadProgramARB;
PFNGLBindProgramARB glBindProgramARB;
PFNGLProgramStringARB glProgramStringARB;
PFNGLDeleteProgramsARB glDeleteProgramsARB;
/* Modifications start */
PFGLCreateShader glCreateShader;
PFGLShaderSource glShaderSource;
PFGLCompileShader glCompileShader;
PFGLGetShaderiv glGetShaderiv;
PFGLCreateProgram glCreateProgram;
PFGLAttachShader glAttachShader;
PFGLBindAttribLocation glBindAttribLocation;
PFGLLinkProgram glLinkProgram;
PFGLGetProgramiv glGetProgramiv;
PFGLGetAttribLocation glGetAttribLocation;
PFGLGetUniformLocation glGetUniformLocation;
/* Modifications end */
#define GL_PROGRAM_FORMAT_ASCII_ARB 0x8875


void InitARBFP(){
#ifdef WIN32
glGenProgramsARB = (PFNGenProgramsARB) wglGetProcAddress("glGenProgramsARB");
glLoadProgramARB= (PFNLoadProgramARB) wglGetProcAddress("glLoadProgramARB");
glBindProgramARB = (PFNGLBindProgramARB) wglGetProcAddress("glBindProgramARB");
glProgramStringARB = (PFNGLProgramStringARB)
    wglGetProcAddress("glProgramStringARB");
glDeleteProgramsARB = (PFNGLDeleteProgramsARB)
    wglGetProcAddress("glDeleteProgramsARB");
/* Modifications start */
glCreateShader = (PFGLCreateShader) glXGetProcAddressARB("glCreateShader");
glShaderSource = (PFGLShaderSource) glXGetProcAddressARB("glShaderSource");
glGetShaderiv = (PFGLGetShaderiv) glXGetProcAddressARB("glGetShaderiv");
glCreateProgram = (PFGLCreateProgram) glXGetProcAddressARB("glCreateProgram");
glAttachShader = (PFGLAttachShader) glXGetProcAddressARB("glAttachShader");
```

```
glBindAttribLocation =
    (PFGLBindAttribLocation) glXGetProcAddressARB("glBindAttribLocation");
glLinkProgram = (PFGLLinkProgram) glXGetProcAddressARB("glLinkProgram");
glCompileShader = (PFGLCompileShader) glXGetProcAddressARB("glCompileShader");
glGetProgramiv = (PFGLGetProgramiv) glXGetProcAddressARB("glGetProgramiv");
glGetAttribLocation =
    (PFGLGetAttribLocation) glXGetProcAddressARB("glGetAttribLocation");
glGetUniformLocation =
    (PFGLGetUniformLocation) glXGetProcAddressARB("glGetUniformLocation");
/* Modifications end */

#else
glGenProgramsARB = (PFNGenProgramsARB)
    glXGetProcAddressARB((const GLubyte *) "glGenProgramsARB");
glLoadProgramARB = (PFNLoadProgramARB)
    glXGetProcAddressARB((const GLubyte *) "glLoadProgramARB");
glBindProgramARB = (PFNGLBindProgramARB)
    glXGetProcAddressARB((const GLubyte *) "glBindProgramARB");
glProgramStringARB = (PFNGLProgramStringARB)
    glXGetProcAddressARB((const GLubyte *) "glProgramStringARB");
glDeleteProgramsARB = (PFNGLDeleteProgramsARB)
    glXGetProcAddressARB((const GLubyte *)"glDeleteProgramsARB");
/* Modifications start */
glCreateShader = (PFGLCreateShader) glXGetProcAddressARB
                                    ((const GLubyte *) "glCreateShader");
glShaderSource = (PFGLShaderSource) glXGetProcAddressARB
                                    ((const GLubyte *) "glShaderSource");
glCompileShader = (PFGLCompileShader) glXGetProcAddressARB
                                    ((const GLubyte *) "glCompileShader");
glGetShaderiv = (PFGLGetShaderiv) glXGetProcAddressARB
                                    ((const GLubyte *) "glGetShaderiv");
glCreateProgram = (PFGLCreateProgram) glXGetProcAddressARB
                                    ((const GLubyte *) "glCreateProgram");
glAttachShader = (PFGLAttachShader) glXGetProcAddressARB
                                    ((const GLubyte *) "glAttachShader");
glBindAttribLocation = (PFGLBindAttribLocation) glXGetProcAddressARB
                                    ((const GLubyte *) "glBindAttribLocation");
glLinkProgram = (PFGLLinkProgram) glXGetProcAddressARB
                                    ((const GLubyte *) "glLinkProgram");
glGetProgramiv = (PFGLGetProgramiv) glXGetProcAddressARB
                                    ((const GLubyte *) "glGetProgramiv");
glGetAttribLocation = (PFGLGetAttribLocation) glXGetProcAddressARB
                                    ((const GLubyte *) "glGetAttribLocation");
glGetUniformLocation = (PFGLGetUniformLocation) glXGetProcAddressARB
                                    ((const GLubyte *) "glGetUniformLocation");
/* Modifications end */
```

```
#endif
}


/* Original functions */

ARBFProg::~ARBFProg(){
  glDeleteProgramsARB(1,&prog_id);
}

void ARBFProg::Load(const char* progname, char* prog){
    glEnable(GL_FRAGMENT_PROGRAM_ARB);
  glGenProgramsARB( 1, &prog_id );
    glBindProgramARB(GL_FRAGMENT_PROGRAM_ARB, prog_id);
  glProgramStringARB(GL_FRAGMENT_PROGRAM_ARB,
                     GL_PROGRAM_FORMAT_ASCII_ARB, strlen(prog), prog);
  GLenum error = glGetError();
  if( error != GL_NO_ERROR )
    fprintf( stderr, "ERROR\n%s\n", gluErrorString( error ) );
#ifdef GPU_DEBUG
    else
    printf("Loaded program [%s] (id %i) successfully", progname, prog_id);
#endif
  }

void ARBFProg::Bind(){
  glEnable( GL_FRAGMENT_PROGRAM_ARB );
  glBindProgramARB(GL_FRAGMENT_PROGRAM_ARB, prog_id);

  GLenum error = glGetError();
  if( error != GL_NO_ERROR )
    fprintf( stderr, "ERROR - Bind()\n%s progid: %d\n",
                     gluErrorString( error ),prog_id );

}

void ARBFProg::BindProg(){
  glBindProgramARB(GL_FRAGMENT_PROGRAM_ARB, prog_id);

}

void ARBFProg::Release(){
  glDisable( GL_FRAGMENT_PROGRAM_ARB );
  GLenum error = glGetError();
  if( error != GL_NO_ERROR )
    fprintf( stderr, "ERROR - Release()\n%s\n", gluErrorString( error ) );
```

```
}

/* Ported functions */

ShaHelp::~ShaHelp(){
  glDeleteProgramsARB(1,&prog_id);
}

void ShaHelp::Load(const char* progname, char* prog){

  const GLchar* const vShaderStr[] =  {
      "attribute vec4 vPosition;    \n"
      "attribute vec2 vtexCoord;    \n"
      "varying vec2 ftexCoord;      \n"
      "void main()                  \n"
      "{                            \n"
      "   gl_Position = vPosition;  \n"
      "   ftexCoord = vtexCoord;    \n"
      "}                            \n"
  };

  const GLchar* const fShaderStr[] = {
      "precision highp float;\n"
      "varying vec2 ftexCoord;\n"
      "uniform sampler2D texture2;\n"
      "uniform sampler2D texture3;\n"
      "void main()                                        \n"
      "{                                                  \n"
      " gl_FragData[1] = texture2D(texture3,ftexCoord);\n"
      " gl_FragData[0] = texture2D(texture2,ftexCoord);\n"
      "}                                                  \n"
  };

  GLuint vshader, fshader;
  GLint compiled;
  GLuint programObject;
  GLint linked;

  // Create the shader object
  vshader = glCreateShader ( GL_VERTEX_SHADER );
  fshader = glCreateShader ( GL_FRAGMENT_SHADER );

  // Load the shader source
  glShaderSource ( vshader, 1, vShaderStr, NULL );
  glShaderSource ( fshader, 1, fShaderStr, NULL );
```

```
  // Compile the shader
  glCompileShader ( vshader );
  glCompileShader ( fshader );

  // Check the compile status
  glGetShaderiv ( vshader, GL_COMPILE_STATUS, &compiled );
  printf("Vshader compiled? %d - ", compiled);
  glGetShaderiv ( fshader, GL_COMPILE_STATUS, &compiled );
  printf("Fshader compiled? %d\n", compiled);

  // Create the program object
  programObject = glCreateProgram ( );

  if ( programObject == 0 )
    printf("Program Object is 0");

  glAttachShader ( programObject, vshader );
  glAttachShader ( programObject, fshader );

  // Bind vPosition to attribute 0
  //glBindAttribLocation ( programObject, 0, "vPosition" );

  // Link the program
  glLinkProgram ( programObject );

  // Check the link status
  glGetProgramiv ( programObject, GL_LINK_STATUS, &linked );
  printf("Program linked? %d\n", linked);

  prog_id = programObject;

  locVertex = glGetAttribLocation ( prog_id, "vPosition" );
  locTexture0 = glGetAttribLocation ( prog_id, "vtexCoord" );
  printf("locVertex: %d / locTexture0: %d\n", locVertex, locTexture0);

  locSampler2 = glGetUniformLocation ( prog_id, "texture2" );
  locSampler3 = glGetUniformLocation ( prog_id, "texture3" );
  printf("locSampler2: %d / locSampler3: %d\n", locSampler2, locSampler3);

  }

void ShaHelp::Bind(){
  printf("ShaHelp Bind\n");
}
```

```
void ShaHelp::BindProg(){
  glBindProgramARB(GL_FRAGMENT_PROGRAM_ARB, prog_id);

}

void ShaHelp::Release(){
  printf("ShaHelp Release\n");
}

GLuint ShaHelp::getProgId(){
  return prog_id;
}

GLint ShaHelp::getLocVertex(){
  return locVertex;
}

GLint ShaHelp::getLocTexture0(){
  return locTexture0;
}
```

B Gantt diagram

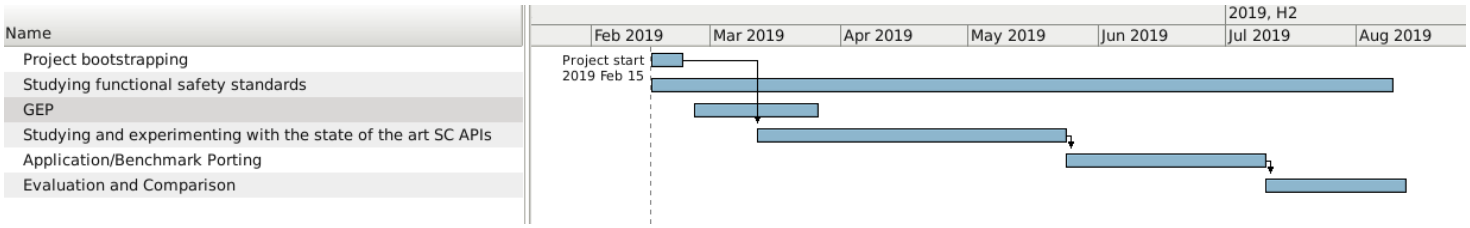


Figure 10: Project's schedule Gantt chart

References

[1] F. Wartel, L. Kosmidis, C. Lo , B. Triquet, E. Quinones, J. Abella, A. Gogonel, A. Baldovin, E. Mezzetti, L. Cucu, T. Vardanega and F. Cazorla, *"Measurement-Based Probabilistic Timing Analysis: Lessons from an Integrated-Modular Avionics Case Study"*, Proceedings of the 8th IEEE International Symposium on Industrial Embedded Systems: SIES, 2013

[2] L. Kosmidis, J. Lachaize, J. Abella, O. Notebaert, F. Cazorla and D. Steenari, *"GPU4S: Towards Embedded GPUs in Space"*, DAta Systems In Aerospace (DASIA) 2019

[3] M. Trompouki and L. Kosmidis, *"Towards General Purpose Computations on Low-End Mobile GPUs"*, Design, Automation & Test in Europe Conference & Exhibition (DATE), 2016

[4] M. Trompouki and L. Kosmidis, *"Brook Auto: High-Level Certification-Friendly Programming for GPU-powered Automotive Systems"*, Proceedings of the 55h Annual Design Automation Conference, 2018

[5] M. Trompouki and L. Kosmidis, *Optimisation opportunities and evaluation for GPGPU applications on low-end mobile GPUs*, Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017

[6] M. Trompouki and L. Kosmidis, *Brook GLES Pi: democratising accelerator programming*, HPG '18 Proceedings of the Conference on High-Performance Graphics, 2018

[7] F. Wartel, L. Kosmidis, A. Gogonel, A. Baldovin, Z. Stephenson, B. Triquet, E. Quiñones, C. Lo, E. Mezzetti, I. Broster, J. Abella, L. Cucu-Grosjean, T. Vardanega and F. Cazorla, *Timing analysis of an avionics case study on complex hardware/software platforms*, Design, Automation & Test in Europe Conference & Exhibition, 2015

[8] L. Kosmidis, C. Maxim, V. Jegu, F. Vatrinet and F. Cazorla, *Industrial Experiences with Resource Management under Software Randomization in ARINC653 Avionics Environments*, Proceedings of the International Conference on Computer-Aided Design, 2018

[9] S. Alcaide, L. Kosmidis, H. Tabani, C. Hernandez, J. Abella, F. Cazorla, *Safety-Related Challenges and Opportunities for GPUs in the Automotive Domain*, IEEE Micro, 2018

[10] L. Kosmidis, R. Vargas, D. Morales, E. Quiñones, J. Abella and F. Cazorla, *TASA: toolchain-agnostic static software randomisation for critical real-time systems*, 35th International Conference on Computer-Aided Design, 2016

[11] O. Notebaert, J. Franklin, V. Lefftz, J. Moreno, M. Patte, M. Syed and A. Wagner, *Way Forward for High Performance Payload Processing Development*, Data Systems in Aerospace (DASIA), 2012

[12] ISO 26262-1:2011, Road vehicles — Functional safety

[13] RTCA/DO-178B, "Software Considerations in Airborne Systems and Equipment Certification"

[14] OpenGL SC, The Kronos Group, https://www.khronos.org/openglsc/

[15] Vulkan, The Kronos Group, https://www.khronos.org/vulkan/

[16] Vulkan SC, The Kronos Group, https://www.khronos.org/vulkansc

[17] Dell Latitude 7490, https://www.dell.com/en-uk/work/shop/port%C3%A1tiles/latitude-14-7490-laptop/spd/latitude-14-7490-laptop?~ck=bt

[18] NVIDIA GTX 1050 Ti, https://www.nvidia.com/en-gb/geforce/products/10series/geforce-gtx-1050/

[19] NVIDIA GTX 1080 Ti, https://www.nvidia.com/en-gb/geforce/products/10series/geforce-gtx-1080-ti/

[20] AMD Radeon E9171 MCM, https://www.techpowerup.com/gpu-specs/radeon-e9171-mcm.c3028

[21] AMD Radeon E8860, https://www.techpowerup.com/gpu-specs/radeon-e8860.c2550

[22] NVIDIA Jetson TX2, https://www.nvidia.com/en-us/autonomous-machines/embedded-systems-dev-kits-modules/?section=jetsonTX2

[23] NVIDIA Jetson AGX Xavier, https://www.nvidia.com/en-us/autonomous-machines/embedded-systems-dev-kits-modules/#JetsonAGXxavierModule

[24] HiKey 970, https://www.96boards.org/product/hikey970/

[25] Green 500, https://www.top500.org/green500/

[26] Emission factors related to electrical energy: the electrical mix, http://canviclimatic.gencat.cat/en/redueix_emissions/com-calcular-emissions-de-geh/factors_demissio_associats_a_lenergia/

[27] A. Munshi, D. Ginsburg and D. Shreiner, *OpenGL® ES 2.0 Programming Guide*

[28] A. Munshi, D. Ginsburg and D. Shreiner, http://opengles-book.com/es2/index.html

[29] N. Govindaraju and D. Manocha, *GPUFFTW: High Performance Power-of-Two FFT Library using Graphics Processors*, http://gamma.cs.unc.edu/GPUFFTW/index.html

[30] N. Govindaraju and D. Manocha, *Cache-efficient numerical algorithms using graphics hardware*, In Journal of Parallel Computing archive, Volume 33, Issue 10-11, Pages 663-684, November, 2007

[31] L. Kosmidis et al., *GPU4S: Embedded GPUs for Space*, in *Digital System Design (DSD) Euromicro Conference*, 2019.

[32] Ian Buck, Tim Foley, Daniel Horn, Jeremy Sugerman, Kayvon Fatahalian, Mike Houston, and Pat Hanrahan. *Brook for GPUs: Stream Computing on Graphics Hardware*. In ACM SIGGRAPH 2004 Papers (SIGGRAPH '04)

[33] Fixed Function Pipeline, `https://www.khronos.org/opengl/wiki/Fixed_Function_Pipeline`

[34] Common Profile Specification 2.0.25 (Difference Specification) (November 2, 2010) (Annotated), `https://www.khronos.org/registry/OpenGL/specs/es/2.0/es_cm_spec_2.0.pdf`

[35] The OpenGL Graphics System: A Specification (Version 2.0 - October 22, 2004), `https://www.khronos.org/registry/OpenGL/specs/es/2.0/es_cm_spec_2.0.pdf`

[36] OpenGL SC Version 2.0.0 (Full Specification) (April 19, 2016), `https://www.khronos.org/registry/OpenGL/specs/sc/sc_spec_2.0.pdf`

[37] Khronos, About the OpenGL ARB "Architecture Review Board", `https://www.opengl.org/archives/about/arb`

[38] Airbus, Airbus performs world's first automatic air-to-air refuelling contact with large aircraft receiver, press release, July 2018, `https://www.airbus.com/newsroom/press-releases/en/2018/07/Airbus-performs-worlds-first-automatic-air-to-air-refuelling-contact-with-large-aircraft.html`

[39] CoreAVI, Vulkan Graphics and Compute `https://www.coreavi.com/product_category/safety-critical-graphics-and-compute`