Thesis no: MECS-2015-08

An Asynchronous Event Communication Technique for Soft Real-Time GPGPU Applications

Alexander Vestman

Faculty of Computing
Blekinge Institute of Technology
SE–371 79 Karlskrona, Sweden

This thesis is submitted to the Faculty of Computing at Blekinge Institute of Technology in partial fulfillment of the requirements for the degree of Master of Science in Engineering: Game and Software Engineering. The thesis is equivalent to 20 weeks of full-time studies.

Contact Information:
Author(s):
Alexander Vestman
E-mail: alve10@student.bth.se

University advisor:
Prof. Håkan Grahn
Department of Computer Science & Engineering

Faculty of Computing
Blekinge Institute of Technology
SE–371 79 Karlskrona, Sweden

Internet : www.bth.se
Phone : +46 455 38 50 00
Fax : +46 455 38 50 57

Abstract

Context. Interactive GPGPU applications requires low response time feedback from events such as user input in order to provide a positive user experience. Communication of these events must be performed asynchronously as to not cause significant performance penalties.
Objectives. In this study the usage of CPU/GPU shared virtual memory to perform asynchronous communication is explored. Previous studies have shown that shared virtual memory can increase computational performance compared to other types of memory.
Methods. A communication technique that aimed to utilize the performance increasing properties of shared virtual memory was developed and implemented. The implemented technique was then compared to an implementation using explicitly transferred memory in an experiment measuring the performance of the various stages involved in the technique.
Results. The results from the experiment revealed that utilizing shared virtual memory for performing asynchronous communication was in general slightly slower than- or comparable to using explicitly transferred memory. In some cases, where the memory access pattern was right, utilization of shared virtual memory lead to a 50% reduction in execution time compared to explicitly transferred memory.
Conclusions. A conclusion that shared virtual memory can be utilized for performing asynchronous communication was reached. It was also concluded that by utilizing shared virtual memory a performance increase can be achieved over explicitly transferred memory. In addition it was concluded that careful consideration of data size and access pattern is required to utilize the performance increasing properties of shared virtual memory.

Keywords: GPGPU, Asynchronous communication, Shared memory.

Contents

Abstract **i**

1 Introduction **1**
1.1 Motivation . . . 1
1.2 Problem Statement . . . 3
1.3 Aim . . . 3
1.4 Approach . . . 4
1.5 Thesis Outline . . . 4

2 Background & Related Work **6**
2.1 Related Work . . . 6
2.2 Communication Requirements . . . 7
2.2.1 Events . . . 7
2.2.2 Latency and Performance . . . 7
2.2.3 Host Originating Events . . . 8
2.2.4 Device Originating Events . . . 9
2.3 CPU/GPU Shared Memory . . . 10
2.3.1 Unified Memory . . . 10
2.3.2 Device/Host Memory Coherency . . . 10
2.4 OpenGL . . . 11
2.4.1 Memory Allocation . . . 11
2.4.2 Synchronization Primitives . . . 11

3 Asynchronous GPGPU Communication **13**
3.1 Transfer Buffer . . . 13
3.1.1 Segmentation . . . 13
3.1.2 Segment Swapping . . . 14
3.2 Event Staging . . . 16
3.2.1 Event Type Representation . . . 16
3.2.2 Packages . . . 16
3.2.3 Staging on the Host . . . 17
3.2.4 Staging on the Device . . . 18
3.3 Interpretation . . . 20
3.3.1 Interpretation on the Host . . . 20

3.3.2 Interpretation on the Device 20
3.3.3 Event Culling 21

4 Experimental Method **23**
4.1 Experiment Testbed 23
4.1.1 Worst Case Scenario 26
4.1.2 Best Case Scenario 26
4.2 Measurements 26
4.2.1 Device Interpretation 27
4.2.2 Device Staging 27
4.2.3 Host Staging 27
4.3 Test Execution 28
4.4 Method Evaluation 29

5 Results **30**
5.1 Array of Structures 31
5.2 Structure of Arrays 33

6 Result Analysis & Discussion **35**
6.1 Requirement Fulfillment 35
6.2 Analysis 35

7 Conclusions and Future Work **37**

References **39**

Appendix A Host Side Code **41**
A.1 Common Implementation Code 41
A.2 Array of Structures Specific Code 42
A.3 Structure of Arrays Specific Code 43

Appendix B Device Side Code **44**
B.1 Array of Structure Device Interpretation 44
B.2 Structure of Arrays Device Interpretation 45
B.3 Common Device Staging 45

Chapter 1

Introduction

1.1 Motivation

Interactive soft real-time graphical applications such as games or computer-aided design programs requires a high frame-rate in order to be utilized with a positive user experience. Such real-time applications often exhibit a characteristic behavior of demanding an increasing need for computational power with each new generation of software. But increasing the computational requirements entails a lower frame-rate since a larger amount of data is processed per time unit with the same computational speed, thus making the application usage experience worse.

The dilemma of reduced frame-rate when increasing workload can be solved with two distinct solutions, or a combination of them. Increasing the computational power of the platform running the application, or modifying the techniques and algorithms utilized by the application. Joselli et al [8] explores a combination of these two options by utilizing the processing power of the graphical processing unit (GPU) to execute general application logic by implementing an entire game in GPU executed code.

Despite that the general processing on GPU (GPGPU) approach employed by Joselli et al [8] does not utilize the central processing unit (CPU) to process data, dependencies on primary memory bound data still exist. In particular these dependencies exists on data such as events, e.g user input, or other data which can not be directly retrieved on the GPU, and therefore has to be communicated to the GPU. Likewise must GPU resident data and generated events that the CPU executed application logic is dependent on be transferred from the GPU to the CPU. In Joselli et al [8], communication is performed sporadically as the communicated data primarily stems from user input.

Communicating and transferring data entails accessing GPU memory for either writing or reading. Accessing memory that is currently in use by the rendering pipeline with the CPU requires that a synchronization between the GPU and CPU occurs in order to avoid manipulating data that is currently in use by the graphics pipeline. A synchronization flushes the rendering pipeline until the accessed memory buffer is not in use before the CPU can either perform writing or reading operations on the memory. This can entail a significant stall in exe-

cution on the CPU. Likewise can synchronization induce performance losses on the GPU, as after a flush the rendering pipeline may not contain enough work to fully utilize the rendering pipeline until it can be filled again; this is referred to as pipeline stalling.

In order to not incur performance losses associated with synchronization and pipeline stalling, the communication between the CPU and GPU must be performed asynchronously, that is, concurrently with GPU and CPU execution. Existing works in the area of asynchronous data transferring shows that performance can be gained by using a non-blocking transfer method [13]. Hao-Wei et al [13] presents a technique that achieves asynchronous transfers by deferring data transferring and kernel dispatches by utilizing task graphs. By deferring and reordering transfers and dispatches, the ability of modern GPUs to compute and asynchronously transfer memory at the same is fully utilized, thus increasing throughput. However, the delay of the defer induced by the technique presented by Hao-Wei et al [13] should be taken into account before communication of time critical data such as events is implemented with the technique. Shneiderman and Plaisant [14] expresses the importance of very fast response times of user input events and other occurrences that can be perceived by a user in order to provide a positive user experience

Another way to perform asynchronous communication is to utilize a circular buffers as described by Everitt and McDonald [4] and Hrabcak and Masserann [7]. Hrabcak and Masserann [7] presents several methods of achieving asynchronous bi-directional transferring utilizing circular buffers and traditional transferring operations. The technique proposed by Everitt and McDonald [4] also utilizes a circular buffer allocated with what they call persistently mapped buffers, which are GPU/CPU shared virtual memory. This memory type, which is also known as unified memory, does not require explicit transferring as the methods presented by Hrabcak and Masserann [7] does; instead the task of transferring data is delegated to the graphics driver and DMA engine.

Landaverde et al [12] studied the effect of utilizing unified memory and showed that in some cases this memory type performed better compared to explicitly transferred memory. Everitt and McDonald [4] also shows that unified memory can increase performance as less application/driver interaction is performed. However, Landaverde et al [12] also showed that utilizing unified memory can decrease performance in cases such as memory intensive applications.

This thesis presents an implementation utilizing shared virtual memory to facilitate communication between the CPU and the GPU. The performance effects of using shared virtual memory to perform the communication is explored and analyzed.

1.2 Problem Statement

Joselli et al [8] mentions that CPU/GPU communication in their implementation is kept to a minimum in order to not cause performance issues, e.g pipeline stalling and synchronizations. The problem faced in this thesis is to perform communication in a GPGPU application without inducing the performance penalties of synchronizations and pipeline stalling, while also taking into account the need of low latency communication posed by user input and other time critical data that can be perceived by users. An attempt to solve the problem is made by utilizing shared virtual memory to take advantage of the performance increase over traditional transferring methods as documented by Everitt and McDonald [4] and Landaverde et al [12].

However, as Landaverde et al [12] also points out, using shared virtual memory can reduce performance compared to explicitly transferred memory types depending on how the memory is accessed from the GPU. The performance loss of using shared virtual memory is particularly evident in cases of memory intensive- or instruction intensive applications according to Landaverde et al [12]. Thus, the problem dealt with in this thesis also involves comparing the implementation used to solve the above mentioned problem to methods using explicitly transferred memory to verify that using shared virtual memory can increase performance when communicating data in an asynchronous and low latency fashion.

From the posed problem stems these research questions that this thesis attempts to answer:

RQ1 Is it possible to communicate events between the GPU and the CPU in the context of GPGPU soft real-time applications without causing pipeline stalling or synchronization utilizing shared virtual memory?

RQ2 What are the performance characteristics of utilizing shared virtual memory to perform asynchronous communication compared to existing asynchronous transfer methods using explicitly transferred memory?

1.3 Aim

Rather than focusing on high throughput asynchronous transferring akin to previous works such as Hao-Wai et al [13] and Wang et al [17], this thesis aims to investigate low latency asynchronous transferring. Low latency asynchronous transferring is deliberately chosen as main goal since communication of time critical data such as events between the CPU and the GPU can facilitate further development of interactive GPGPU applications.

Although other works has focused on asynchronous transferring and shared virtual memory, no approach to performing low latency asynchronous communication utilizing shared virtual memory catering to the needs of interactive GPGPU

applications has been made. This thesis intends to present such a technique in order to hopefully improve on performance and response times over existing methods of performing communication in GPGPU applications.

1.4 Approach

In order to conclude on the research questions with empirical data a communication technique based on the usage of shared virtual memory was created. Two different implementations based on the communication technique presented in chapter 3 were created to explore the usage of shared virtual for communication purposes. The goal of utilizing two implementations was to test two different GPU accesses patterns to achieve utilization of the performance increasing properties of shared virtual memory mentioned by Landaverde et al [12].

In addition to the two shared virtual memory implementations, two reference implementations with the same memory access patterns were also created. These reference implementations were based on one of the OpenGL asynchronous buffer transfer techniques utilizing explicitly transferred memory presented by Hrabcak and Masserann [7].

All implementations utilizes OpenGL to allocate-, transfer-, and manipulate GPU memory. OpenGL can, as Direct3D, both perform rendering and computation without the need for additional libraries. As Direct3D 11 lacks the ability to allocate shared virtual memory it could not be utilized to answer the posed research questions, and since Direct3D 12 is not officially released, it was deemed to unreliable to be used in this study. Thus, to keep the implementations as simple as possible, and still maintain the ability to perform rendering, OpenGL 4.4 was chosen over Direct3D 11, Direct3D 12, and traditional compute libraries such as CUDA.

After the implementations were finished and verified to be working correctly, a series of tests were performed on each implementation to gather measurements of how they behaved during various levels of utilization. The measurements gathered from the implementations were then compared, discussed, and used to conclude on the research questions.

1.5 Thesis Outline

The rest of this thesis is divided into chapters, each with a specific focus. In chapter *2 Background & Related Work*, related works and references are discussed followed by the requirements imposed on the communication technique stemming from the time critical aspect of the problem statement and the usage of shared virtual memory. Chapter *3 Asynchronous GPGPU Communication*, details the different stages that constitutes the communication technique from both a usage and implementation perspective.

In chapter *4 Experimental Method* the details regarding experiment execution, test cases and which measurements were taken and how they were gathered is presented and discussed. Chapter *5 Results* presents the results gathered by the experiment which is followed by chapter *6 Result Analysis & Discussion* which discusses the results in relation to related work and the requirements posed in chapter 2. In the last chapter *7 Conclusions and Future Work*, the results and analysis are put into perspective of the research questions to form a collusion and to perceive what the results indicate in general.

Lastly, two appendices containing source code from the implementation are included in the thesis. Appendix A and Appendix B contains the source code of critical parts of the CPU and GPU side of the implemented technique respectively.

Chapter 2

Background & Related Work

2.1 Related Work

Both GPU computation APIs such as CUDA [3] and graphics libraries such as OpenGL [5] provides asynchronous data transfer functionality that enables retrieval of data while computation occurs. However, although being asynchronous, the functionality of the API functions can still cause serialization between application and graphics card drivers as described by Everitt and McDonald [4]. To avoid driver induces pipeline stalling, Everitt and McDonald [4] thus proposes a more low level solution that employs host/device shared memory where avoiding concurrent access is handled by the client application rather than by the driver.

Hrabcak and Masserann [7] presents several methods of creating asynchronous transferring between the GPU and CPU using OpenGL. The techniques presented by Hrabcak and Masserann [7] utilizes both multiple buffers and asynchronous API functionality to avoid concurrent memory access. By comparing the different methods in their experiment, they conclude that utilizing multiple buffers to achieve asynchronous transferring is viable from a performance perspective compared to using asynchronous API functions.

In [17] Wang et al developed GDM, a device memory manager that utilizes host side virtual memory staging areas to store data before transfer. The staging areas are utilized to allow a large amount of memory to be utilized by allowing data unused by a kernel to be swapped out when space contention occurs. The staging areas also enables asynchronous transfers as data can be temporarily stored in the staging area before being transferred, thus eliminating the need to immediately complete a transfer upon request. Before a kernel is launched the data it is referencing is asynchronously transferred while the previous kernel is executing. Although focusing on handling very large sets of data rather than event communication, Wang et al [17] provides insight into how asynchronicity can be achieved without reordering transfers.

The Non-blocking buffer (NBB) technique presented by Kim [9] facilitates asynchronous event communication between two processes in a real-time application. Although not focusing on CPU to GPU communication, the NBB technique achieves the desired goal of low latency asynchronous communication. In similar

to many of the asynchronous transfer techniques presented in [7], NBB uses a circular buffer with the addition of two counters for keeping track of buffer usage. Due to this similarity to existing asynchronous GPU transfer techniques, the non-blocking buffer technique was chosen as a base for the technique presented in this thesis.

2.2 Communication Requirements

Before the proposed technique is presented the requirements imposed on the technique is discussed. With insight into the requirements, a better understanding of the limitations and decisions made in the development of the technique will be procured while reading chapter 3.

2.2.1 Events

In order to discern the usage domain of the technique presented in chapter 3, the category of data intended to be communicated, namely events, must be defined. An event refers to an action or message stemming from a source that is to be communicated to one or more destinations within applications with the possible intent of causing a change to the application state.

Events are classified into types that reflect their origin, e.g an event deriving from a press of a key on a keyboard is often referred to as a key event. Events can also originate from within an application itself, e.g an event is generated containing results that is to be communicated to other parts of an application once a task completes. Event types are thus a concept used to separate events between each other based on point of origin and intended usage. Using types, constructs such as event handlers and listeners can be used to direct communication to intended destinations.

Events often contain data that details the communicated action further. Examples of this is a value of a key being pressed in a key event, or results from a completed task. Events can thus be viewed as atomic pieces data that are only relevant in the context of their origin. To clarify, events are viewed as data that can not be split up into several parts, and that origin, or type, of the event is important in the interpretation of said data. This implies that events are data, but that the data must be handled in such a way that it does not violate the atomicity or context of the event. Further considerations around events exists which are discussed below.

2.2.2 Latency and Performance

According to Shneiderman and Plaisant [14], the acceptable latency from event generation to perceived effect depends on the expectations of the user on the

particular event type. Distinguishing which events a user expects to be completed quickly in order to prioritize certain events is a particularly difficult challenge, and much outside the scope of the technique presented in this paper. Shneiderman and Plaisant [14] mentions that a user's previous experiences, individual personality differences and task differences also influences the expected response time, making the task even more difficult. Because of the inability to prioritize, the technique presented in this thesis is limited to treating all events as equally important in fulfilling the expected response time requirement.

The expected response time for events can vary between a few milliseconds to several seconds depending on the complexity the user perceives [14]. Since events are not prioritized based on their expected response times the communication and execution of all events must be completed within the shortest response time requirement time span. To clarify, if events can not be selectively prioritized, the lowest expected response time must be upheld for all events in order to not cause a missed response requirement. According to Shneiderman and Plaisant [14], the response time must in some cases be within a one tenth of a second time span in order to provide a positive user experience.

Due to the requirement of upholding the shortest expected response time for all events, simultaneous communication of multiple events poses a requirement of efficiency. The requirement implies that the time spent on communicating an arbitrary amount of events must not cause a breach of the response time requirement for any of the communicated events. In practice this implies that communication of an event can not be performed in such a way that the overhead cost of communicating the event prohibits further communication to uphold the expected response time requirement. This requirement can of course not be upheld for an infinite amount of events, but the technique must take communication efficiency and response latency into account in order to facilitate communication of more than a handful of events without reducing the user experience.

2.2.3 Host Originating Events

The types of event that are generated in a host process can originate from two different sources, either from the user or the application. User input events such as keyboard and mouse input events are generated by the operative system or drivers before the application can process them. Thus only user input events that are supported by the operative system or input device driver can be managed by the application. This limitation constrains user input events to the limits of the operative systems event handling procedures.

In contrast to events that originates from user input, application logic originating events are not restricted to the operative systems limitations. The diversity of possible events originating from application logic entails that no finite set of events can cover all potential events generated by an arbitrary application. Further, the types of events originating from application logic varies between ap-

plications and even events with similar origins can have different specifications in both latency and data size in different applications. Thus, in order to not restrict the technique by a finite set of events with minor variations, the technique is required to support any arbitrary event.

2.2.4 Device Originating Events

The device differs from the host by that all generated events only originates from application logic. Like with the application logic originating events generated on the host, events generated by the application logic on the device can exhibit the same diversity of event types. The requirements for the events from the device are thus similar to that of the host. But because the device logic is executed on the GPU, limitations to how events can be communicated are incurred. These limitations are discussed further below.

Accessing the file system (FS) from the GPU has been proven to be possible by using an abstraction of GPU memory [15]. In [15], Silberstein et al presented GPUfs, an abstraction for accessing the file system on the GPU. Since file system access is provided by the host through the abstraction layer, rather than through native hardware access by the GPU, the data communicated via the file system must thus be handled by the CPU beforehand. Data would thus be communicated GPU-CPU-FS-CPU when utilizing the file system to perform communication between the GPU and CPU. Thus, utilizing the file system to communicate within the same application causes redundant communication as the CPU handles the data twice.

In [10], Kim et al present GPUnet, a networking API for the GPU. GPUnet enables network communication for GPUs by utilizing peer-to-peer (P2P) direct memory access over the PCI-express buss to transfer data between the network interface controller (NIC) and the GPU. Utilizing the P2P direct memory access (DMA) transfer, the GPU can with initial setup by the CPU, initiate network traffic to for example other CPUs or GPUs. Kim et al [10] mentions that utilizing DMA in this manner can eliminate much of the overhead cost otherwise associated with transferring data residing in GPU memory over network as the otherwise necessary GPU-CPU transfer is removed. However, as P2P DMA can only access devices connected to the PCI-express buss, writing into operative system interprocess communication (IPC) sockets can not be accomplished as they reside in main memory. Communication using sockets on the GPU must thus be performed with a loop-back network socket, which implies that an extra copy by the DMA engine must be performed compared to standard transferring methods and that the data must travel twice over the PCI-express buss.

Due to the above mentioned extra transferring steps involved when utilizing the file system and network sockets to perform communication, ordinary message passing techniques such as using the operative system IPC sockets or file sockets are disadvantageous as low latency communication is desired. Thus, in-

stead of utilizing a socket based communication scheme, this thesis presents a communication technique based on shared virtual memory.

2.3 CPU/GPU Shared Memory

2.3.1 Unified Memory

In CUDA 6.0, unified memory was introduced as an alternative to mechanisms such as pinned host memory, known as zero-copy memory, and serves to simplify memory transferring by removing the need for explicit copy operations otherwise associated with non-mapped memory. Similar memory mechanisms can also be found in OpenCL 2.0 and onwards in the form of shared virtual memory (SVM) allocations. The main benefit of memory with virtual memory is to achieve simpler programs by allowing the host and device to utilize the same pointers and simplify memory management.

In [12], Landaverde et al showed that utilizing CUDA unified memory can in some scenarios increase performance, particularly when smaller data sets are used as input and when output is generated concurrently by kernels. Landaverde et al [12] further points out that the performance implications of using unified memory strongly depends on memory access patterns of kernels, where a subset access of data using multiple consecutive kernels before performing a write or accessing more memory provided a performance increase. Thus, by utilizing unified memory to perform communication, both a performance increase and an ease of use can be attained over explicitly transferred memory.

Due to the property of having a unified address space, shared virtual memory also enables the host and the device to read data that the other processor has written without performing an explicit memory transfer, pointer patching or other data transformation. From an application point of view, accessing the memory is the same as accessing regular system memory and the memory is likewise perceived as a standard memory from the device. As seen in section 3.2, this allows simple copy operations and memory access on the host and device during the writing stages.

2.3.2 Device/Host Memory Coherency

Unified memory provides memory coherency between the CPU and the GPU by migrating memory from the system memory to the device memory and vice versa. However, coherency can only be guaranteed by excluding access from all but one of the processors sharing the memory. As such, in both OpenCL and CUDA, coarse-grained coherency for shared memory between host and device is only guaranteed at certain synchronization points, e.g after kernel execution completion and at kernel dispatch, where the migration of data occurs. An impli-

cation of the mutual exclusive ownership is that once a kernel is executing using shared virtual memory, accessing the same memory region may, as in the case of OpenCL, cause stalling until the kernel completes, or in the case of CUDA cause segmentation fault. Thus, in order to uphold coherency and asynchronicity, multiple shared memory regions must be utilized when performing communication utilizing shared memory.

2.4 OpenGL

As this thesis utilizes OpenGL to allocate-, manage-, and manipulate GPU memory, some of the OpenGL functionality utilized in the experiment implementation is detailed in this section to provide a better insight into the presented work. In particular, the utilized functionality for synchronization and memory allocation is detailed.

2.4.1 Memory Allocation

Memory regions allocated through OpenGL functionality are assigned to opaque identifiers called buffer objects which each corresponds with single a consecutive region of GPU memory. Buffer objects are created with the function `glGenBuffers` independently of the actual memory allocation of memory which is performed with `glBufferStorage` or `glBufferData`. When allocating memory with `glBufferData`, and in particular with `glBufferStorage`, it is possible to specify whether to allocate the memory on the host or the device, and if any special attributes should be given to the memory, e.g mappable or host-device coherent.

Everitt and McDonald [4] utilizes the functionality of OpenGL to specify memory attributes to allocate SVM in their attempt to reduce application/driver interaction overhead when transferring data to- and from the GPU. As SVM can be written and read without using memory manipulation function such as `glBufferSubData`, interaction between the OpenGL driver and application is reduced. But removing the need to use memory manipulation functions removes the ability for OpenGL to track buffer object dependencies and thus guarantee correct behavior when altering data from the CPU concurrently with accessing it from the GPU.

2.4.2 Synchronization Primitives

OpenGL utilizes an asynchronous execution model with the use of a command queue, and as such commands waiting to be processed, e.g rendering, can refer to memory regions that subsequent commands alter the contents of. In order to avoid altering memory contents prematurely, thus affecting previously issued

commands, OpenGL keeps a reference count of buffer objects accessed by each command. As some OpenGL functions, e.g `glBufferSubData`, only returns once the changes have actually been written into memory, manipulating buffer objects that are referred to by queued commands will block the host until those commands have been executed.

When SVM is used, OpenGL can not prevent manipulation of buffer objects that are referenced by queued commands as no OpenGL function is utilized. Instead, the application itself is responsible for not manipulating memory referenced by queued commands. Everitt and McDonald [4] suggests avoiding manipulation of referenced data by using sync objects created with `glFenceSync`. When sync object is created it is inserted into the command queue and is executed after all previously issued commands are complete. By calling `glClientWaitSync`, a sync objects completion can be asserted, and thus the completion of all previously queued commands.

Since the implicit transferring that keeps SVM coherent between the CPU and the GPU occurs immediately after dispatch execution completion, synchronization primitives can be used to implement mutual exclusion. By knowing that a command that writes to-, or reads from a SVM region has finished, it is asserted that the data can be read-, and written to by the CPU due to the coherency guarantees of SVM.

Chapter 3
Asynchronous GPGPU Communication

3.1 Transfer Buffer

In this section the usage and segmentation of shared virtual memory is discussed and detailed. The discussed material consolidates the transfer buffer, which is a fundamental part in the asynchronous GPGPU event communication technique presented in this thesis.

3.1.1 Segmentation

Fig. 3.1 depicts the transfer buffer from a usage perspective. From a usage perspective the transfer buffer is considered to consist of equally sized segments of shared virtual memory in a ring buffer configuration. The overall design of the data structure is inspired by the non-blocking buffer (NBB) proposed by Kim [9] and the multi buffer techniques presented by Hrabcak and Masserann [7]. As such, each segment of the transfer buffer is treated similarly to the buffer slots discussed by Kim [9]. However, instead of using an acknowledge counter to indicate the number of elements written into the buffer as proposed by Kim [9], each segment has a counter of the number of elements written into it. To clarify, for each segment there is a counter, called write head, of the number of elements written into it, while the transfer buffer also utilizes counters to indicate how many segments has been accessed by each processor. Both the segments and the counters are allocated using the shared virtual memory in order to provide both the host and device access to the values.

In order to avoid data races, the memory of each segment is only accessed by one processor at a time, thus each segment is fully read and written to before the next segment is accessed. This is performed by utilizing mutual exclusion for operations accessing the shared memory to make sure that no concurrent access is performed. The mutual exclusion method is further discussed in section 3.1.2. The details of how the segments are filled and processed can be found in section 3.2 and section 3.3 respectively.

As seen in Fig. 3.1, each segment is exclusively in one of three states, CPU active, GPU active, or queued to be active on either processor. In each state ex-

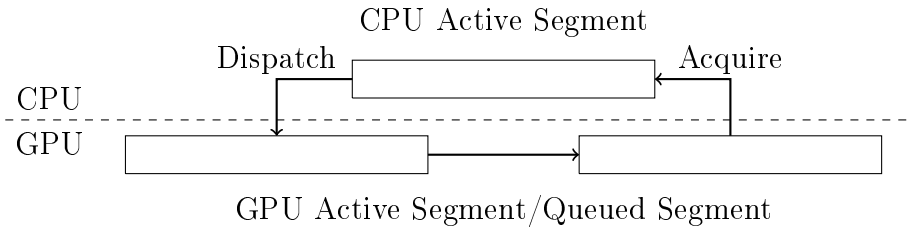


Figure 3.1: Conceptual view of the transfer buffer. Each segment progresses along the arrow when a state change occurs. Conceptually, all segments but the CPU active one are viewed as in the GPU pipeline.

cept the queued state, the data in a segment is read (interpreted) and overwritten (staged) with new events.

In Fig. 3.1, three segments can be seen. These three segments are only conceptual, the actual numbers of segments required depends on the execution timing of the device and the host process. Song and Choi [16], and Chen and Burns [2] discusses how different execution timings among the involved processes can effect the number of required buffer slots in a circular buffer technique in order to avoid contention issues.

3.1.2 Segment Swapping

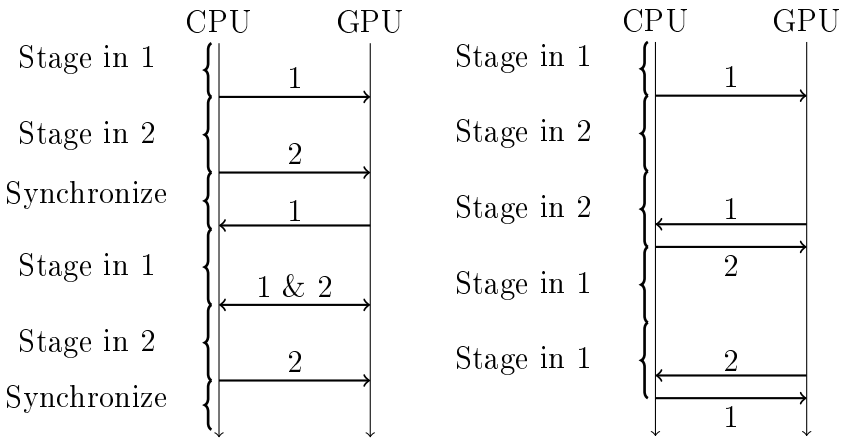
Both Chen and Burn [2], and Song and Choi [16] depends on requirements of the scheduler in order to guarantee that the slots in the ring buffer suffice. Since the GPU and the CPU are two separate processors, their schedulers are also different. Instead of relying on properties of the CPU and the GPU schedulers to not cause erroneous segment accesses, each segment of the transfer buffer is protected by mutual exclusion. The mutual exclusion serves to guard the host from accessing the mapped memory segments concurrently used, or queued to be used, by the device. Since the mutual exclusion is of segment granularity, an equal amount of fences as segments is required to be stored on the host. Mutual exclusion is created by inserting a fence immediately after dispatches of kernels accessing a segment and using a host-device synchronization function to guarantee mutual exclusion. If all device commands prior to the fence is completed, the kernel accessing the segment guarded by the mutual exclusion has completed and due to the coherent property of SVM no data race occurs if the memory segment is read.

In the case where the host outperforms the device, a request to access a segment owned by the device can occur. Thus, in order to avoid and explicit synchronizing a fence querying function is utilized to check the completeness status of the fence prior to calling the actual synchronization function on the

fence. Using a try-synchronize scheme thus prevents the host from stalling due to segment swapping.

Once the mutual exclusion is unlocked the segment can be accessed by the host and becomes the new CPU active segment. When this occurs the previously active CPU segment's status is set to queued and is dispatched to the device. Thus a swap of active segments has occurred.

An implication of this method of swapping is that if no unlocked segment can be acquired, and the currently active CPU segment is dispatched to the device, no staging can be performed until a segment is freed by the device. In this case a synchronization is necessary in order to acquire an active segment for the CPU. An example of this scenario is given in Fig. 3.2a, where the duration for staging and interpreting a segment on the host and device diverge and a dispatch and acquire always occur after staging on the CPU. To avoid synchronizations when



(a) Swapping after staging. (b) Swapping after acquiring.

Figure 3.2: Process views of two different approaches to handling divergent execution speeds on the host and the device. In this example the transfer buffer is split into two segments. Arrows indicate that the buffer segment is queued to the side which it points. In Fig. 3.2a the host process synchronizes until an active segment is acquired. In Fig. 3.2b the host process stages multiple times into the same segment until a new segment can be acquired, thus not needing to synchronize with the device process.

acquiring a new CPU active segment, the currently active CPU segment is used for staging multiple times in the case where no new segment can be acquired. As such, the CPU active segment is dispatched to the device only when another segment can be acquired. Thus no synchronizations for segment acquisition have to be performed. Fig. 3.2b depicts this method compared to synchronizing with the GPU to acquire a segment.

3.2 Event Staging

As mentioned in chapter 2, application logic can generate a vast number of different events. It was also reasoned that the transferring technique could not be constructed to only handle a finite set of events. By doing so, the technique would limit the possible event types that could be communicated, thus limiting any implementation of the technique. Further, the posed requirement also dictates that the algorithm is required to support any arbitrary event to not incur such limitation.

To facilitate the usage of any arbitrary event, and at the same time uphold the requirement to not impose any major limitation, the staging of events, on both the host and the device, is event type agnostic. That is, that all events are handled the same during preparation for transfer regardless of their origin. It also implies that staging is not dependent on specific event attributes, such as memory usage or data layout. Event agnosticism thus allows the technique to view events as plain memory. Properties stemming from event type are thus disregarded as events are viewed as a consecutive series of bytes.

3.2.1 Event Type Representation

Although the origin of the data is disregarded during staging; event types are, as seen in section 3.3, required to perform other stages of the technique. Abstracting events to bytes of memory discards the type information as only the values contained in the event are exposed as memory. Preservation of the event type for interpretation purposes is performed by saving the event type as data along with the other values of the event.

By letting each event type be assigned a unique integer number, the types can, like the event itself, be abstracted as memory. Using an integer for type representation, n allocated bits of memory can represent 2^n different integer numbers. A numerous amount of events can thus be uniquely represented by a relatively few number of bits.

3.2.2 Packages

Before staging, the integer type representation and the event data is aggregated into a data structure called an event package. The type information is laid out before the event data in order to ease the type distinction during the interpretation stage. Besides the event data and type information, padding can also be added to packages for alignment purposes. In Fig. 3.3 the layout of memory of the aggregation of event data, type representation and padding, that is, the event package data structure is described.

The package abstraction is utilized during device staging and during one of the host staging passes. Utilizing the memory consecutiveness provided by a

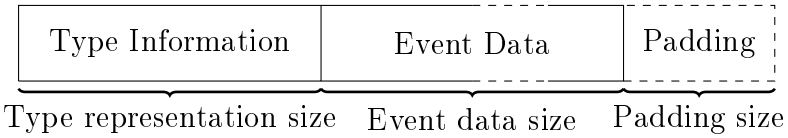


Figure 3.3: The memory layout of an event package. The sizes indicates how many bytes are used for storage. Dashed lines indicates that the storage requirement is of variable size.

package, staging an event can performed by linearly copying the contents of a package into the transfer buffer segment that is available for either the GPU or the CPU. Some details regarding staging differs the staging on the host and the device. These differences are described in the sections below.

3.2.3 Staging on the Host

The data structure created by the host staging pass effects how interpretation is performed on the device. In this section the methods of creating two different data structures for device interpretation are presented. One data structure utilizes the package abstraction while the other one utilizes a structure of arrays layout. Both methods of data structure creation can be utilized together with the other stages of the technique, but display different behaviors based on the characteristics of the transferred events as seen in chapter 5.

Array of Structures

When utilizing the first data structure called array of structures, which is based on the package concept, reading locations during interpretation are inferred by index since no offset information is contained in the package abstraction. Thus, to enable interpretation, packages are placed at regularly spaced intervals by incrementing the write head of the segment with an access interval size. The interval, or rather access offset, is equal the largest sized package, or in other words, the size of the largest event data staged added to the size of type representation. The final position of the write head then provides the location in the segment where the staged data ends. Packages can thus be read during device interpretation by accessing the segment at intervals up to the size of the write head.

The padding in the package abstraction is used to align packages to the access offset and the size of the padding required to align the a package depends on the size of the event's data. The padding size of a package is equal to the largest event data, subtracted with the size of the current package's event data.

When the size of the padding has been determined for a package, it is written into the active transfer buffer segment, directly behind any previous packages.

This procedure is repeated for all events that is to be transferred from the host to the device when utilizing the array of structures host staging method.

Structure of Arrays

Fig. 3.4 depicts the second data structure based on a structure of arrays (SoA) layout. Creating a structure of arrays layout with a single buffer is performed by utilizing two write heads, one writing from the beginning and one from the end. Thus, by viewing each end of the segment as an individual array, a two typed structure of arrays data structure can be created.

The two data types that an event is broken down to, the type representation and event data, are separated and stored in each end of the double ended buffer. Event data is stored at the end of the segment using a reversely traversing write head and type representations are stored together with offsets in an interleaved fashion from the beginning of the segment. The offsets indicate where in the segment the data corresponding to the event type/offset pair is located in the segment.

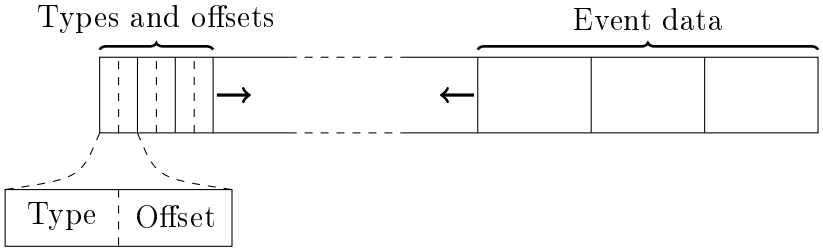


Figure 3.4: The structure of arrays memory layout. The type and offsets of each event are packed together. The offset indicates where in the segment the data corresponding to the type is located. The initial event data storage offset begins at the end of the segment and is decremented by the size of the event data to be stored. The arrows indicate the direction of writing for the two types of data.

Since explicit positions for event data are stored in the type/offset pairs, event data is not required to be aligned to access intervals. Events with data of different size can thus be staged without the use of padding and since no padding is utilized for the type/offset pairs nor the event data, the whole segment can utilized for actual information.

3.2.4 Staging on the Device

Unlike staging on the host, staging on the device is performed by a compute kernel, thus multiple work items are potentially staging into the segment simultaneously. An individual event is however only staged by one work item, that is, that staging

of a particular event is not performed by more than one work item at the same time.

As explained in section 3.3, the interpretation stage on the host accesses memory linearly from a single process. In contrary to the device interpretation's aligned packages, the host interpretation is based on the packages being tightly packed in memory. Padding is thus not added to the packages when staging on the device. And since no padding is used, neither is there any unused memory between each package for the memory to be tightly packed. However, unused memory can still exist between the end of the last package and the end of the segment.

Fig. 3.5 depicts the method used to tightly store packages in the transfer buffer. To keep track of which addresses are available for use, a write head that points to the first free address of the transfer buffer segment is used. Every address larger than the write head, until the end of the transfer buffer segment, is considered a free address. By using the write head, a work item obtains an address where subsequent addresses form a consecutive series of bytes that is used for package storage. To facilitate further staging, the write head is increased by the size of the package that is to be written. The write head then points to a new series of consecutive bytes that can be used for staging another event, either by the same work item, or another.

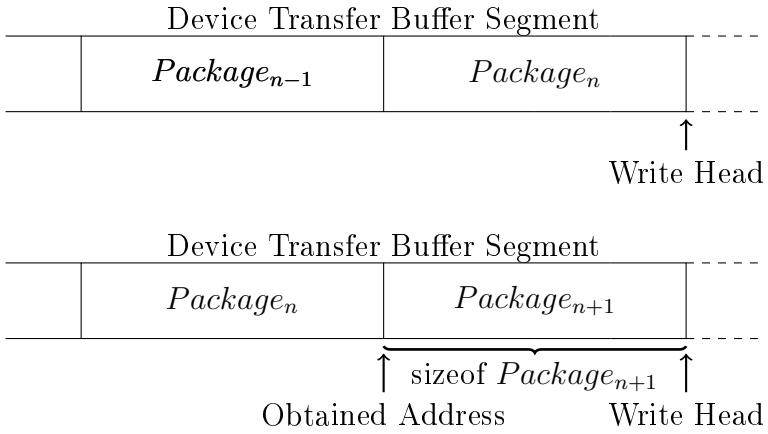


Figure 3.5: Packages tightly packed together in the transfer buffer segment. The write head is always positioned at first unused memory address in the transfer buffer segment. When staging an event, the write head position is used to allocate memory to store the package. The write head is then increased by the package size.

As staging in parallel entails that work items may access the write head simultaneously, data races can occur when incrementing and reading is performed at the same time. To avoid a data race, an atomic fetch and add operation is

used on the write head when staging on the device. As such incrementing and retrieving the writing position in the segment is done atomically.

3.3 Interpretation

3.3.1 Interpretation on the Host

Accessing the data from the CPU active segment is performed by starting from the offset into the transfer buffer given by the segment's position in the ring buffer structure of the transfer buffer. The segment size in conjunction with the index of the active segment gives the offset into the transfer buffer. Each byte can then be accessed sequentially by reading from the offset until the end of the staged data. In [9], Kim uses an update counter that is shared between processes to indicate the quantity of data stored in the buffer. Similarly the amount of data in a segment is provided by the write head used during staging on the device. The memory range beginning at the start of the segment and ending at the offset indicated by the write head covers all the data stored in the segment.

Extracting data from a package is performed by reading the event type stored at the beginning of every package and performing a corresponding action to that type. Using the event type representation, the size of the event data can be inferred by the use of a lookup table containing the size of each event type's data. The data of the event stored in the package can then be extracted. Once the data in a package has been responded to, the next package in the segment is processed.

Since packages are laid out in a tight continuous fashion in the segment, a package's offset can be computed by taking the current reading offset into the segment and incrementing with the current package's size. This process continues until the reading offset becomes equal to the write head and when this occurs the write head is reset to zero to prepare for host staging.

3.3.2 Interpretation on the Device

The many core architecture and memory hierarchies employed in modern GPUs poses a significant challenge in the interpretation on the device. To not breach the efficiency requirement, both the memory hierarchy and the parallel processing model is taken into account during the interpretation process.

In [11], Kirk and Hwu describes two behaviors of memory access that can be exhibited by algorithms in a parallel execution environment such as a GPU. The two behaviors are the scatter behavior, and the gather behavior. Kirk and Hwu [11] defines the scattering behavior as when a piece inseparable data has an effect that is distributed to several locations in the global device memory. An implication of utilizing an algorithm with a scattering access pattern is that different work items can write to the same locations, forcing the need for very fine

grained work item synchronization or the usage of a large amount of atomic variables. As such, utilizing an algorithm with a scattering access pattern drastically impacts performance in both CUDA based applications [11] and OpenCL based applications [1].

The gathering access pattern defined by Kirk and Hwu [11] is the opposite of the scattering pattern. In the gathering access pattern, each location in memory collects the pieces of data that effects them. A gathering access pattern algorithm thus avoids the need for synchronization by assigning each potential target of the input data to a work item and letting that work item update that target. For example: in the case of event communication, each kernel will read through the packages or type/offset pairs in the GPU active segment and only act upon the events that effects the part/memory of the application that the work item is processing.

The device interpretation is thus performed with a gathering memory access pattern to avoid the poor performance otherwise caused by the massive scale of synchronization that would be required. Besides synchronization avoidance, a gathering algorithm has another benefit. An algorithm with a gathering access pattern facilitates usage of work group shared memory [11]. By gathering events into shared memory the amount of global memory accesses is reduced as subsequent reads of events can be performed from shared memory. However, storing every event present in a segment into work group shared memory can be both unnecessary and detrimental to performance due to exhaustion of shared memory resources. Event culling is thus performed in order to reduce the number of events stored in each work group's shared memory.

3.3.3 Event Culling

Using a gathering access patterned algorithm entails that each work item is not necessarily effected by each input. Potentially, each work item may only effected by a subset of the communicated events. The distinction between events that effect a particular work item and those who does not can be made on event types. Each work item is thus assigned a list of event types that effects the application part, or memory, that the work item updates or processes. The granularity of the work item division in regards to the application logic thus dictates how many events can be culled for a particular work item.

The list of event types that effects a work item is represented as a bit mask constructed from the identification values of the events. The bit mask is created by taking two by the power of the type representation value of the event types that effects a particular work item and summarizing the results.

The effect of an event on a work item can be determined by performing a bitwise AND operation where the operators are the bit mask and the binary representation of two by the power of the event type identification value. If the result of the bitwise operation is a non-zero value, the n:th bit in the bit mask

is set to one, where n is the type identification value. Utilizing a bit mask to store the list thus enables determining whether or not an event has an effect or not with a single operation regardless of the number of event types a work item is effected by.

The same method of utilizing a bit mask to cull events is also utilized on a work group level. This group shared bit mask is created by aggregating the bit masks of the work items in the group. By performing an atomic bitwise OR operation on the shared bit mask and the work item's own, the aggregated bit mask is constructed. This aggregated bit mask thus contains the set of events that effect at least one work item within the work group.

With the ability to determine whether or not an event is effecting at least one work item in a group, a work item can cull events that does not effect any work item in its group by not copying it into shared memory. Since each work item can cull on behalf of the work group, each event in the GPU active segment only needs to be accessed once per group. If the package based data structure is utilized and an event is found to effect the work group, the package is copied into group shared memory. However, if the structure of arrays host staging method is utilized an additional read from the segment is required to retrieve the event data from the position given by the offset value.

Copying into the shared memory follows the same rules as host staging and either the package based- or structure of array data structure can be employed for shared memory copying. When utilizing the array of structures data structure, each package is aligned to the access offset to facilitate a uniformly sized read and in similarity to host staging, a group shared atomic variable is utilized keep track of a write head. When utilizing the structure of arrays data structure for shared memory copying, two write heads are utilized in similarity to the host staging counterpart. The cooperative gathering is performed until all events in the GPU active segment has been processed by the work group.

Once the culling is completed each work item can begin to process the events stored in shared memory. The bit mask effect determining process is utilized on the events in shared memory using the work item's bit mask. Any non-zero result is then acted upon in an application dependent manner. Immediately after the culling is completed for all groups the staging process can begin since all relevant events are copied into shared memory.

Chapter 4

Experimental Method

4.1 Experiment Testbed

The technique described in chapter 3 was implemented in order to examine the usability of shared memory interprocess communication for soft real-time GPGPU applications. Both variants of the host staging step was created as separate implementations. The staging variation writing the packages directly into memory is referred to as array of structure (AoS), since the packages, or structures, are placed linearly in a consecutive piece of memory. The other staging variation is referred to as structure of arrays (SoA), as each segment, consists of a structure of two logical arrays of consecutive memory. Measurements were then taken from the implementations in areas relating to the stages of the technique to expose the performance characteristics of the implemented techniques. The results were based on measurements of 1000 iterations of each implementation configuration to increase the statistical significance of the results.

The implementations were created using C++11 with OpenGL 4.4 as GPU access API. The device staging and device interpret stages of the technique were implemented with OpenGL compute shaders. Details of critical parts of the implementations can be found in appendix A for the host side stages, and in appendix B for the device side stages.

In order to compare the usage of shared virtual memory to facilitate transfer to existing methods, an additional means of transfer was implemented based on the methods proposed by Hrabcak and Masserann [7]. One of the methods proposed by Hrabcak and Masserann utilizes a ring buffer structure similar to the one presented in this thesis and was therefore chosen as a reference implementation. The reference implementation does not utilize shared memory but instead uses the OpenGL function `glBufferSubdata` to write into device memory and `glGetBufferSubData` to retrieve data. The technique was among the faster methods of performing asynchronous buffer transfer according to the results presented by Hrabcak and Masserann [7].

The reference method does not utilize `GL_PERSISTENT_BIT | GL_COHERENT_BIT` for buffer storage creation, but instead only uses `GL_DYNAMIC_STORAGE_BIT` as buffers in the reference technique are not intended for mapping nor to be allo-

cated in zero-copy memory. `GL_COHERENT_BIT` can thus not be used to provide coherency between the host and the device and therefore explicit memory barriers must be utilized since the buffers being bound to `GL_SHADER_STORAGE_BUFFER` target are written to by shader programs [6, p. 142-148]. Coherency for reading the device memory buffer is as such provided by a use of `glMemoryBarrier`. In order to provide an asynchronous reading of data written to by the device, the implemented reference method must also explicitly copy data from the device memory buffers to host memory buffers using `glCopyBufferSubData` as reading directly from the device causes synchronization [7]. After copying the device memory buffer to the host memory buffer, `glGetBufferSubData` is called in order to retrieve the data.

To investigate the performance characteristics of the implementations a testbed application was developed. The testbed allows setting various parameters of the implementations such as event size and event culling granularity to facilitate a thorough testing procedure. The modifiable parameters that was used during the testing procedure is described below.

Padding The number of bytes added to the event data in order to increase data size. The parameter is labeled as p in the result graphs.

Host-to-device Events The number of events staged on the host in each segment before it is dispatched. The parameter is labeled e in the result graphs.

Device-to-host Events The occurrence of an event being staged on the device. Every o:th work item stages an event on the device. The parameter is labeled as o in the result graphs.

Culling Bit Mask The value of the bit mask used by each thread to cull events. The parameter dictates if the implementation behaves in the worst or best case scenario configuration.

Combinations of these parameters were used during different executions in order to test the implementations in various situations. During both of the scenarios, the values of the testing parameters were varied in order to evaluate how the implementation behaves during different kinds of work loads, both in transfer and execution. The variations of the parameters used to create the test combinations are detailed and explained below.

Padding In order to test the implementations with differently sized events the event data sizes are mutable. The size of the transferred events are varied between a size of 16 and 256 bytes, where 4 bytes are reserved for the type representations. In the test result graphs this is reflected as 0 and 60 additional 32-bit integers.

Host-to-device Events The number of staged events on the host side was varied with the values 0 and 100. Staging 0 events on the host was done to provide an indication of how the implementation behaves when only communicating from the host. Staging 100 events was chosen to create a large enough sample set to distinguish differences between the two memory layout variations described in section 3.2.3 during device interpretation.

Device-to-host Events The host-to-event occurrence parameter is similarly to the host-to-device event parameter varied to either stage 100 or 0 events on the device. The device-to-host events parameter is thus varied between 100 and 100000 to stage either 100 or 0 events on the device.

Culling Bit Mask The value of the culling bit mask used for culling is varied between 1 and 1023 to create two scenarios. These two scenarios are discussed further below.

In total 16 different combinations can be made with the above mentioned variables. However, in addition to these eight tests, a ninth test was performed with the host-to-device events parameter set to 1000 with a padding size of 0 in order to test the writing capabilities of the implementations further. The writing test was performed in both culling modes and as such, a total of 18 test cases are performed per implementation.

Besides the above mentioned variable parameters, a few constant parameters were used. The implementation utilized three segments, or buffers depending on the implementation, where each was 40000 bytes in size, allowing to stage all events for all the parameter combinations. All kernel dispatches was performed with 10000 work items divided into 16x1x1 work groups, creating a total of 625 work groups for each dispatch.

The 16x1x1 work group size was used to divide the work items into a fairly large amount of work groups as to not create a trivial work load for the GPU to process and to enable concurrent fetching of memory and execution. As work group sizes directly affect both memory access pattern and shared local memory usage, utilizing other work group sizes could affect the performance of the implementations for better or for worse. However, if the performance increasing subset data access pattern of SVM mentioned by Landaverde et al [12] can be achieved at one work group size, it could reasonably be achieved at other sizes as well. Also as the performance increasing subset data access of SVM refers to the joint memory access by all work items to the global memory structure, the individual work group's memory access should not affect the results beyond what it does normally in the sense of coalesced reads and writes. As such, only one work group size was utilized in the experiment.

The work group shared memory was limited to storing 100 copies of events, which was enough to store all events transferred in the test cases, except for the write test in the worst case culling mode. Once 100 copies was stored by a

work group into shared memory, any further events were disregarded from device interpretation in that work group.

Event types were varied by assigning each staged event's type an incrementing value modulated by ten. Each time an event was staged the assignment value was incremented by one, thus ten different event types were utilized in the implementations.

4.1.1 Worst Case Scenario

The worst case scenario is intended to produce the behavior that the implementation would exhibit in an application were device event culling is not effectively utilized. The culling bit mask parameter is as such set to a value 1023, thus making the culling test fail to reduce the transferred data set. The worst case scenario was constructed to investigate if any of the implementations behaved differently, from a perspective of each other or from different parameter settings, during low utilization of culling.

4.1.2 Best Case Scenario

The best case scenario is on the other hand intended to induce the behavior of the implementations that would occur if an application utilized a very fine grained culling, e.g one event type per work group. The bit mask was as such set to 1 while retaining the 10 different event type in order to cull away 90% of the transferred data.

4.2 Measurements

The experiment primarily focuses on investigating the performance of event interpretation and data transferring both from-, and to the GPU. As such the execution time, on both the host and the device, of selected areas in the implementations are measured. The areas of interest relates to the different stages detailed in chapter 3, primarily the device interpret, device staging and host staging implementations are measured. Also, in regards to the requirement of response time detailed in section 2.2, the round trip time (RTT), or the time from dispatching a segment to the acquisition of the same segment on the host, for each segment is measured. The RTT can thus be viewed as the minimal duration a response from the device that is the result from events staged in the segment on the host priorly. The RTT can as such be seen as the shortest possible time a user can perceive a reaction in the case of host-device-host communication. The value of RTT can thus be compared against the expected response time requirement.

Measurements of elapsed time on the host is performed using the C++ standard library `std::chronos::high_resolution_clock`. Timestamps before and

after host staging, and before dispatching to device interpret and after acquiring the segment again was taken. These timestamps were then used the calculate the duration of the operations.

Measurements on the device are performed using OpenGL timer query objects [6, p. 45-46]. Calls to `glBeginQuery` are made just before calls to dispatches and a call to `glEndQuery` is perform immediately after. Once the fence used to avoid concurrent memory access is signaled, the result of the operations are retrieved via `glGetInteger`.

4.2.1 Device Interpretation

Device interpretation comprises the reading of data from memory and copying events that pass the bit mask test to work group shared memory. Measurements of device interpretation can be used to perceive different behaviors regarding global memory access among the implementations. Varying the levels of culling and data further provides information regarding the behaviors of the implementations.

4.2.2 Device Staging

Device staging is composed of writing to global memory and utilizing atomic fetch-and-add function on globally shared memory to retrieve the writing positions. Device staging is completed once the values written by the device are visible to the host. In the case of the reference implementation this criteria includes the memory barrier and the asynchronous copying to host memory. Measurements of device staging can be used to perceive the effects and behaviors of using coherent zero-copy memory in various levels of memory utilization compared to explicitly copying the memory and using memory barriers.

4.2.3 Host Staging

Host staging consists of writing data into the device memory, being either buffers or shared memory depending on the implementation. Host staging measurements begin right before the writing commences and ends as soon as the written values are readable by the device. The measurements of host staging is taken as an aggregated time of all writes into a segment, that is, the timer is started before the first event is staged, and stopped after the last event is completed. Like with device staging, measurements of host staging can be used to perceive the effects of the different memory usage models among the implementations.

4.3 Test Execution

The parameter combinations described in the section above is utilized to run each implementation 18 times. Between each execution the application is completely closed in order to eliminate potential performance degradations to continuous execution and issues relating to persistent state between tests. Each execution performs 1000 iterations of communication, that is, 1000 segments are dispatched and that each stage is repeated an equal amount of times. The measurements of each implementation, for each configuration is thus taken 1000 times, and an average of the execution time and standard error is taken as a result. After each iteration has completed the results of the measurements is written to file.

In order to simplify and reduce the human interaction with testing, and the errors that come with it, the testing procedure was automated by the use of a script. The script keeps track of, and creates, combinations of parameters using interleaved loops over the variations that was previously described. Each implementation was executed with every parameter combination, and likewise with the additional write test. Each execution's result was written to a separate file for each parameter variation and implementation combination. Once the execution was completed, the results in each data file was used to create histogram graphs using GNUplot. The results of each implementation were divided into the two graphs based on the two scenarios and were further divided by being grouped together by the area of interest, thus creating four groups of histograms in each plot.

Operative System	Debian 7 Wheezy 64-bit
CPU	Intel(R) Core(TM) i5-4670K
RAM	8GB
GPU	NVIDIA GeForce GTX 760
GPU Driver	346.47 Linux-x86_64

Table 4.1: The hardware specifications of the computer executing the tests.

All tests were performed on a computer with the specifications found in table 4.1. Before the testing procedure was commenced the computer was restarted to free any locked resources and reduce the number of running programs to a minimum. After the reboot only a minimal amount of programs necessary to run the implementations was started, namely an X11 server and client. As such the amount of graphical programs was limited to the bare necessity and the implementations themselves.

During the testing no rendering was performed as to not interfere with the transferring and filling the OpenGL pipeline with commands not related to transferring. V-sync and back buffer swapping was disabled as to not cause stalling on the host while waiting for the device to catch up.

4.4 Method Evaluation

Assuring that the implementations, and by association the measurements, are valid the implementations' functionality was asserted before measurements were taken. The implementations rendered geometry whose position was updated in the interpretation stage when events of the right type and event data of specific values was processed. The position of the rendered geometry was changed by communicating key events from the host and when certain key code values was found during device interpretation the position of the rendered geometry was changed. By writing an event that caused the geometry to move to different locations in the segment and asserting the behavior was unchanged the functionality of the implementations was asserted. The rendering of this geometry was then disabled as previously mentioned and the reaction to the key events removed from the compute shader.

Likewise was events created on the device and transferred over to the host in order to assert functionality. Events were staged with types equal to the index of the work item modulated by 10 and in the event data the work item index was stored. The values of the events, both type and data, was then asserted by modulating the data value and comparing it to the event type value. This procedure was repeated for all events retrieved by the host interpretation stage. If one of the events did not produce the correct values a debug assertion was triggered, thus halting the program. If the values were correct, nothing happened and the device staging and host interpretation of the implementation was deemed to function correctly. After the implementation was deemed to function correctly the assertion code was removed as not to impede the measurement values' validity.

After it was deemed that the implementations were functioning correctly, the asynchronicity of the implementations were asserted. This was performed by looking at the number of failed segment acquires that was made during the 1000 iterations of each parameter combination. After staging events after acquiring a segment, the host was busy waiting on the next acquire by calling `glClientWaitSync` with the timeout parameter set to 0, thus if the returned result was `GL_TIMEOUT_EXPIRED` the segment acquisition was counted as failed. If no failed segment acquisitions were made during execution, especially during tests with large amounts of events for device interpretation, the host and device must have implicitly synchronized prior to calling `glClientWaitSync`. Such cases were found and corrected, and once all tests were performed, all of the test cases on all implementations achieved failed acquisitions, thus asserting that all the implementations were fully asynchronous.

Chapter 5

Results

The results attained from the implementations discussed in the previous chapter is divided into separate sections. Each section contains two histogram with the results from the worst and best case scenario respectively. The histograms contains both the results of the shared virtual memory (SVM) implementations and reference implementations. Each histogram are grouped by the used memory layouts, either array of structures (AoS) or structure of arrays (SoA) facilitate comparison between the SVM- and the reference implementations. The results from the reference implementation is postfixed with *ref* and have a crossed pattern in the same color as the corresponding result from a SVM implementation.

The results in the histograms are the mean execution times and standard deviations of 1000 iterations of each combination of the testing parameters discussed in section 4.1. Each result is labeled after the parameters used when running the testbed application and the meaning behind the labels: p, e and o can be found in the chapter 4.

The y-axis of the histograms are broken into two subranges, one from 0 to 0.5 milliseconds, and the other from 0.5 to 25 milliseconds. This was done to show results that otherwise was to small to perceive using a single y-axis scale histogram. The black line drawn in staples for results larger than 0.5 millisecond indicate the break in the y-axis scale.

5.1 Array of Structures

As seen in Fig. 5.1 and Fig. 5.2, the difference in RTT between worst and best case when using smaller events in the SVM implementation, presented in green bars, is about 0.1 millisecond despite the tenfold difference in processed data. When utilizing larger events, depicted by the purple bars in Fig. 5.1 and Fig. 5.2, the difference in RTT between the worst and best case is likewise not proportional to the difference of processed data. Using larger events also increases the device interpretation time despite that no events are transferred from the host, as seen on the blue bars in Fig. 5.1 and Fig. 5.2.

Despite the differences in event sizes, device staging execution time is similar among the different configurations in the SVM implementation. Host staging does likewise not differentiate much between the different sizes and in all configurations the host staging execution time is almost immeasurable small.

As seen in both Fig. 5.1 and Fig. 5.2, the device staging execution time is for the reference implementation is shorter than in the SVM implementation and does not differentiate much between the configurations. The SVM implementation does likewise not differentiate much in device staging between configurations, but the results are larger in the cases where device to host communication is present compared to the reference implementation. However, when host staging is not performed, the SVM implementation presents a shorter host staging execution time than the reference implementation as seen in Fig. 5.1 and Fig. 5.2.

The execution time results of the reference implementation are, as seen in Fig. 5.1 and Fig. 5.2, slightly shorter than the results attained from the SVM array of structures implementation with the exception of the write test. The device interpretation results of the write test for the reference implementation, depicted in crossed cyan bars in Fig. 5.1 and Fig. 5.2, should be noted as they are shorter, both compared to the other small event size configurations and in comparison to the SVM implementation. Due to the longer host stage time, the RTT of the write test for the reference implementation is in both cases significantly larger than in the SVM implementation.

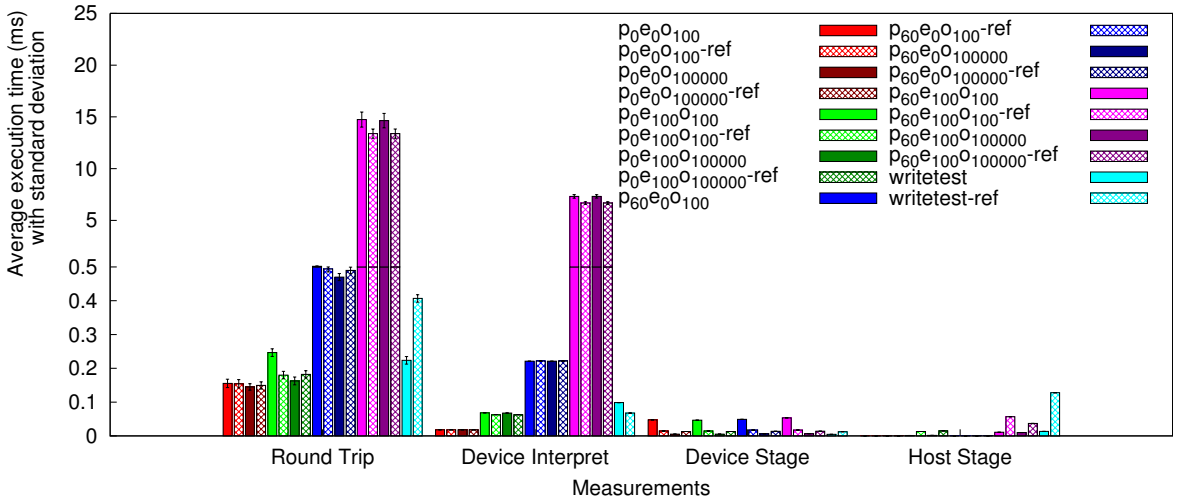


Figure 5.1: Best case scenario, array of structures layouts

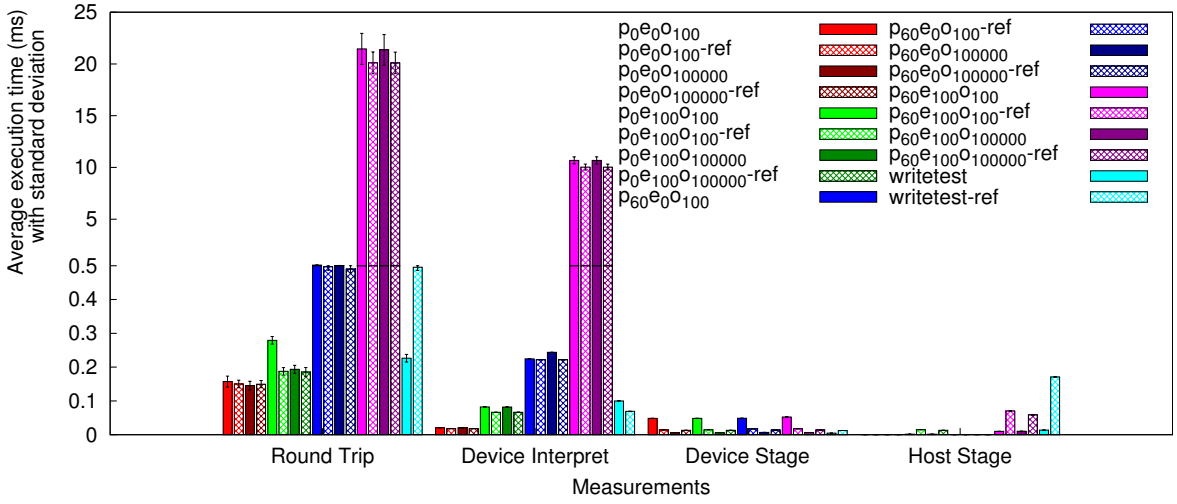


Figure 5.2: Worst case scenario, array of structures layouts

5.2 Structure of Arrays

A comparison of the results in Fig. 5.4 to the results in Fig. 5.2, shows that the worst case results for both of the SVM implementations are similar. However, a big difference in results between the two SVM implementations occurs when using larger event sizes and communication is performed host to device, which is depicted by the purple bars in Fig. 5.1 and Fig. 5.3. As seen in Fig. 5.3, the RTT in the best case scenario when using large events is under 5 milliseconds, which is significantly less than the result attained with the same configuration in the array of structures implementation, as seen in Fig. 5.1.

The results of both the SVM- and reference implementation in Fig. 5.3 and Fig. 5.4 shows the same characteristics as the results in Fig. 5.1 and Fig. 5.2. The device staging execution time is short and undifferentiating between the configurations. The host staging execution time of the reference implementation in both Fig. 5.3 and Fig. 5.4 are longer compared to the corresponding results of the SVM implementation and is even longer than the array of structure reference implementation seen in Fig. 5.1 and Fig. 5.2.

Comparing the results for using large events and performing communication, depicted by the purple bars in Fig. 5.3, shows that the RTT for larger events in the best case scenario is shorter in the SVM implementation. However, the RTT for the SVM implementation when using large events and performing communication is longer compared to the reference implementation in the worst case scenario as seen when comparing the solid and crossed purple bars Fig. 5.4.

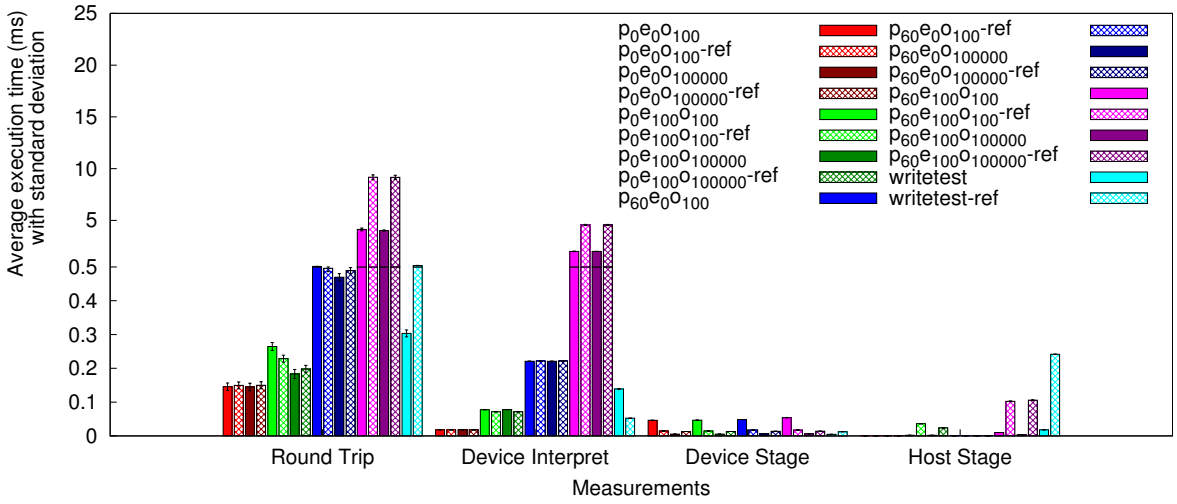


Figure 5.3: Best case scenario, structure of arrays layout

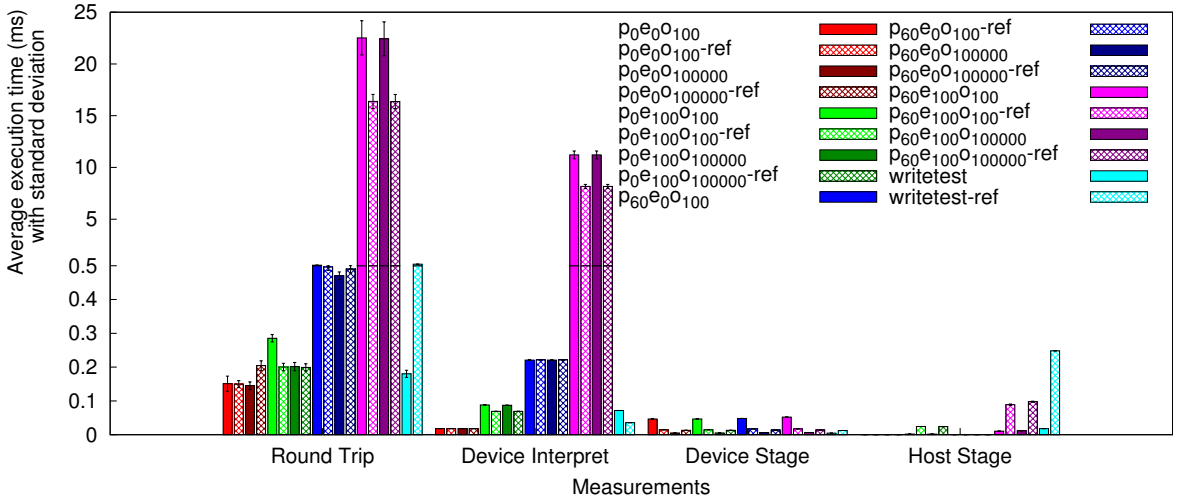


Figure 5.4: Worst case scenario, structure of arrays layout

Chapter 6

Result Analysis & Discussion

6.1 Requirement Fulfillment

From the results presented in chapter 5 a number of observations relating to the requirements posed in section 2.2, can be made. The expected response time requirement stems from the claim by Shneiderman and Plaisant [14] that user experience is strongly connected to rapid feedback in an interactive application. In the results attained from the shared virtual memory (SVM) implementations, the longest round trip time (RTT) was 22 milliseconds which can be seen in Fig. 5.2. Although being a significantly long RTT compared to some of the other RTTs, it is less than the 100 milliseconds mentioned by Shneiderman and Plaisant [14] as shortest required response time for interactive applications.

The other requirement posed in section 2.2 is the arbitrary event requirement, which states that the expected response time requirement is to be upheld when communicating events of both arbitrary size and number. In the experiment, these aspects were tested and as previously mentioned the longest RTT was shorter than the time limit proposed by Shneiderman and Plaisant [14]. It can thus be concluded that the requirements are upheld when communicating 100 instances of data sized between 4 and 256 bytes, and 1000 instances with the size of 4 bytes, when performing the experiment with the hardware setup discussed in chapter 4.

6.2 Analysis

Comparing the results of the SVM implementations to the results of the reference implementations presented in Fig. 5.1, Fig. 5.2, Fig. 5.3 and Fig. 5.4, shows that the reference methods in general executes faster than their SVM counterparts with some exceptions. As the only difference between the respective implementations is the utilized memory type, the difference in results stems from the usage of shared virtual memory.

By comparing the SVM implementation and reference implementation results in Fig. 5.1 and Fig. 5.2, it can be observed that the reference implementation

utilizing an array of structures host staging method is slightly faster when performing two way communication and when utilizing larger event data. However, the array of structures SVM implementation achieved a lower RTT than the reference implementation during the write test and a similar RTT when small event data was utilized.

A major characteristical difference between the SVM implementations and their respective reference implementations is the difference in the host and device staging results. Staging on the host is faster in both of the SVM implementations compared to their respective reference implementation. This finding conforms with the results presented by Everitt and McDonald [4] that utilizing SVM allows for an order of magnitude more transferring in the same amount of time compared to utilizing explicitly transferred memory.

Another characteristical difference between the SVM- and reference implementations are the device staging execution times seen in Fig. 5.1, Fig. 5.2, Fig. 5.3 and Fig. 5.4. The device staging execution times of the reference implementations are uniformly lower than in the SVM implementations when device to host communication is performed. However, when no device to host communication is performed, the SVM implementations are faster. Since device staging execution time does not increase relatively to the size of the larger data written by the device for both types of implementations, it appears that writing to memory from the device consists entirely of an overhead cost. In the case of the SVM implementations, the overhead is approximately 0.05 milliseconds. The overhead cost, although small, causes both of the SVM implementations to have a slightly longer RTT in the small sized, two way communication configurations compared to the reference implementations despite having comparable host staging and device interpreting execution times.

Examining and comparing the results of both implementations in Fig. 5.3 it can be observed that in the best case scenario the RTT of the SVM implementation is lower than in the reference implementation in the large event configurations. However, in the worst case scenario, the reference implementation possess a lower RTT in the large event configurations, as seen by comparing the solid and crossed purple bars in Fig. 5.4. The difference in device interpretation execution time between accessing the entirety and a subset of the transferred memory from the device conforms with the findings and conclusions of Landaverde et al [12] that in the case of subset access utilizing SVM can increase performance. As the device interpretation execution time is dependant on subset access, the granularity of the device side culling, and thus by extension, the application division on a GPGPU level, becomes important in the case of utilizing SVM for low response time CPU-GPU communication.

Chapter 7
Conclusions and Future Work

In this thesis a technique for performing low response time communicating between the CPU and GPU utilizing the properties of shared virtual memory was presented. The presented technique was implemented and compared to an alternate version of the technique employing explicitly controlled memory in an experiment measuring the execution times of the implementations in various configurations. The results attained from the experiment was sufficient to answer both of the research questions.

The first research question was: *is it possible to communicate events between GPU and CPU in the context of GPGPU soft real-time applications without causing pipeline stalling or synchronization utilizing shared memory?* From the results attained from the experiment it was observed that it was possible to utilize shared virtual memory for real-time communication between the CPU and GPU, and that shared virtual memory can be utilized without any implicit or explicit synchronizations occurring. Although it was possible to achieve asynchronous transferring with shared virtual memory, the experiment revealed performance differences between the usage of shared virtual memory and explicitly transferred memory that can be used to answer the second research question.

The second research question was: *what are the performance characteristics of utilizing shared memory to perform asynchronous communication compared to existing asynchronous transfer methods using explicitly transferred memory?* The results attained from the experiment showed that the overall execution and round trip times was slightly longer when utilizing shared virtual memory than when explicitly transferred memory was used. The slightly higher round trip times of using shared virtual memory stems from an overhead latency when accessing the memory from the device. However, in areas such as writing from the host, the shared virtual memory implementations achieved lower execution times. Utilizing shared virtual memory can also significantly improve device side performance when only a subset of the virtual memory is accessed by the device. In a best case scenario, where only a subset of the memory was accessed a 5 millisecond reduction, which was a decrease of about 50% compared to the reference implementation's total round trip time, was measured when utilizing shared virtual memory. In a worst case scenario, where all memory was accessed,

the same implementation displayed an increase by 6 millisecond in the round trip time compared to the reference implementation that was slower in the best case scenario.

The results indicate that with the implementation described in this thesis, shared virtual memory can be utilized to perform asynchronous communication faster than with explicitly transferred memory. The results further indicate that the subset access pattern, further described by Landaverde et al [12], achieved in a best case scenario, is the key factor to utilizing the performance increasing properties of shared virtual memory. However, when subset access is achieved, both a reduced host writing- and device reading duration can be had. Due to the importance of the subset access pattern, future works could study the effect of changing host staging order depending on device side usage in order to improve data locality.

As this thesis only utilized one set of hardware to perform the experiment, future works could expand the utilized systems and configurations further to form greater insight into the usage of shared virtual memory. In particular the usage graphics cards from several different vendors should be studied to perceive potential differences. Likewise should the effect of using accelerated processing units (APUs), e.g the Xbox One or PS4, also be studied due to see if the shared chip layout changes the effect of shared virtual memory communication.

With the knowledge that shared virtual memory can be used to improve communication performance, further work should be made into how to utilize the other aspects of shared virtual memory. Such aspects are among others that pointers can be shared between the host and the device, allowing for shared complex data structures, and that the shared virtual memory can be accessed as ordinary memory by the host. For example, an attempt to integrate the shared virtual memory communication into an existing framework or application to facilitate GPGPU computation with a minimal effect on the original code could be investigated.

Besides the aforementioned performance benefits, the usage of shared virtual memory comes with the benefits of less driver interaction, described by Everitt and McDonald [4], which can increase rendering performance in graphical applications. Studies into graphical applications utilizing GPGPU processing similarly to Joselli et al [8] could be performed to investigate if shared virtual memory communication to handle the required input events can improve the performance of such applications due to the decreased driver interaction. With increasingly powerful GPUs, it is possible that the number of graphical real-time applications that employ a GPGPU solution also increases, thus spurring the need for more research into the area of graphical GPGPU applications.

References

[1] Advanced Micro Devices Inc. AMD Accelerated Parallel Processing OpenCL Programming Guide. http://developer.amd.com/wordpress/media/2013/07/AMD_Accelerated_Parallel_Processing_OpenCL_Programming_Guide-rev-2.7.pdf, 2013.

[2] J. Chen and A. Burns. Loop-free asynchronous data sharing in multiprocessor real-time systems based on timing properties. In *Real-Time Computing Systems and Applications, 1999. RTCSA '99. Sixth International Conference on*, pages 236–246, 1999.

[3] NVIDIA Corporation. CUDA. https://developer.nvidia.com/cuda-zone, 2015.

[4] Cass Everitt and John McDonald. Beyond porting: How modern opengl can radically reduce driver overhead. https://www.youtube.com/watch?v=-bCeNzgiJ8I, 2014. Steam Dev Days 2014.

[5] The Khronos Group. OpenGL. https://www.opengl.org/, 2015.

[6] The Khronos Group. OpenGL core specification 4.4. https://www.opengl.org/registry/doc/glspec44.core.pdf, 2015.

[7] Ladislav Hrabcak and Arnaud Masserann. Asynchronous buffer transfers. In Patrick. Cozzi and Christophe. Riccio, editors, *OpenGL Insights*. CRC Press, Boca Raton, Fla., 2013.

[8] Mark Joselli, José Ricardo da Silva Junior, Marcelo Zamith, Esteban Clua, Mateus Pelegrino, Evandro Mendonça, and Eduardo Soluri. Techniques for designing GPGPU games. In *Games Innovation Conference (IGIC), 2012 IEEE International*, pages 1 – 5. IEEE, 2012.

[9] K.H. (Kane) Kim. A non-blocking buffer mechanism for real-time event message communication. *Real-Time Systems*, 32(3):197–211, 2006.

[10] Sangman Kim, Seonggu Huh, Xinya Zhang, Yige Hu, Amir Wated, Emmett Witchel, and Mark Silberstein. Gpunet: Networking abstractions for gpu programs. In *11th USENIX Symposium on Operating Systems Design and*

Implementation (OSDI 14), pages 201–216, Broomfield, CO, October 2014. USENIX Association.

[11] David Kirk and Wen-mei W. Hwu. *Programming massively parallel processors : a hands-on approach*. Morgan Kaufmann, San Francisco, Calif., 2010.

[12] R. Landaverde, Tiansheng Zhang, A.K. Coskun, and M. Herbordt. An investigation of unified memory access performance in cuda. In *High Performance Extreme Computing Conference (HPEC), 2014 IEEE*, pages 1–6, Sept 2014.

[13] Hao-Wei Liu, Hsien-Kai Kuo, Kuan-Ting Chen, and B.-C.C. Lai. Memory capacity aware non-blocking data transfer on gpgpu. In *Signal Processing Systems (SiPS), 2013 IEEE Workshop on*, pages 395–400, Oct 2013.

[14] Ben Shneiderman and Catherine Plaisant. *Designing the user interface : strategies for effective human-computer interaction*. Addison Wesley, Reading, Mass., 4. ed. edition, 2005.

[15] Mark Silberstein, Bryan Ford, Idit Keidar, and Emmett Witchel. Gpufs: Integrating a file system with gpus. *ACM Trans. Comput. Syst.*, 32(1):1:1–1:31, February 2014.

[16] Osok Song and Chong-Ho Choi. Wait-free data sharing between periodic tasks in multiprocessor control systems. *Control Engineering Practice*, 11(6):601 – 611, 2003.

[17] Kaibo Wang, Xiaoning Ding, Rubao Lee, Shinpei Kato, and Xiaodong Zhang. Gdm: Device memory management for gpgpu computing. In *The 2014 ACM International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '14, pages 533–545, New York, NY, USA, 2014. ACM.

Appendix A

Host Side Code

A.1 Common Implementation Code

```
bool EventHandler::AcquireSegment()
{
  unsigned int index = m_activeSegment % m_segmentCount;

  if(m_activeSegment >= m_segmentCount)
  {
    int acquireIndex = (m_activeSegment + 1) % m_segmentCount;
    GLenum syncStatus=glClientWaitSync(m_fences[acquireIndex],
    GL_SYNC_FLUSH_COMMANDS_BIT,0);

    if(syncStatus == GL_ALREADY_SIGNALED || syncStatus ==
    GL_CONDITION_SATISFIED){
      glDeleteSync(m_fences[acquireIndex]);

      GLint interpretDiff, stageDiff;
      std::chrono::duration<double> hostDiff;

      hostDiff = std::chrono::system_clock::now() - m_timers[acquireIndex];
      glGetQueryObjectiv(m_queries[acquireIndex*2U], GL_QUERY_RESULT,
    &interpretDiff);
      glGetQueryObjectiv(m_queries[acquireIndex*2U+1], GL_QUERY_RESULT,
    &stageDiff);

      DispatchSegment();
      m_activeSegment=m_activeSegment+1;
      Interpret();
    }
    else
    {
      m_missedSegmentAcquires=m_missedSegmentAcquires+1;
      return false;
    }
  }
  else
  {
    DispatchSegment();
    m_activeSegment=m_activeSegment+1;
    Interpret();
  }

  return true;
}
```

```
void EventHandler::DispatchSegment()
{
  unsigned int index=m_activeSegment%m_segmentCount;

  glBindBufferRange(GL_SHADER_STORAGE_BUFFER, 0, m_transferBuffer,
    index*m_segmentSize, m_segmentSize);
  glBindBufferRange(GL_ATOMIC_COUNTER_BUFFER, 0, m_writeHeadBuffer,
    index*sizeof(GLuint), sizeof(GLuint));

  m_timers[index]=std::chrono::system_clock::now();

  glUseProgram(m_interpretingShaderProgram);
  glUniform1i(m_eventSizeLocation, m_stagedEvents);
  glBeginQuery(GL_TIME_ELAPSED, m_queries[index*2U]);
  glDispatchCompute(m_entityCount/16, 1, 1);
  glEndQuery(GL_TIME_ELAPSED);

  reinterpret_cast<GLuint*>(m_writeHeadBufferPtr)[index]=0U;
  m_writeHead=0;
  m_stagedEvents=0;
  glUseProgram(m_stagingShaderProgram);
  glBeginQuery(GL_TIME_ELAPSED, m_queries[index*2U+1]);
  glDispatchCompute(m_entityCount/16, 1, 1);
  glEndQuery(GL_TIME_ELAPSED);

  m_fences[index]=glFenceSync(GL_SYNC_GPU_COMMANDS_COMPLETE, 0);
}
```

A.2 Array of Structures Specific Code

```
void EventHandler::Stage(unsigned int type
{
  unsigned int index=m_activeSegment%m_segmentCount;

  unsigned int offset=(sizeof(Event)+m_paddingSize)%(sizeof(GLfloat)*4);
  if(offset > 0)
  {
    offset=sizeof(GLfloat)*4-offset;
  }

  unsigned int packageSize=sizeof(Event)+m_paddingSize+offset;
  if(m_writeHead + packageSize < m_segmentSize)
  {
    event[0]=type;
    std::memcpy(reinterpret_cast<void*>(reinterpret_cast<unsigned
    char*>(m_transferBufferPtr)+m_segmentSize*index+m_writeHead),
    reinterpret_cast<void*>(event), sizeof(Event)+m_paddingSize);

    m_writeHead+=packageSize;
    m_stagedEvents=m_stagedEvents+1;
  }
}
```

A.3 Structure of Arrays Specific Code

```
1  void EventHandler::Stage(unsigned int type)
2  {
3    unsigned int index=m_activeSegment%m_segmentCount;
4
5    unsigned int offset=(sizeof(KeyEvent)+m_paddingSize)%(sizeof(GLfloat)*4);
6    if(offset > 0){
7      offset=sizeof(GLfloat)*4-offset;
8    }
9
10   unsigned int dataSize=sizeof(KeyEvent)+m_paddingSize+offset;
11   if(m_reverseWriteHead - dataSize > m_writeHead + sizeof(TypeOffsetPair)){
12     m_reverseWriteHead -= dataSize;
13
14     TypeOffsetPair t;
15     t.type=type;
16     event[0]=type;
17     t.offset=m_reverseWriteHead/dataSize;
18
19     std::memcpy(reinterpret_cast<void*>(reinterpret_cast<unsigned
       char*>(m_transferBufferPtr)+m_segmentSize*index+m_reverseWriteHead),
       reinterpret_cast<void*>(event), dataSize);
20
21     std::memcpy(reinterpret_cast<void*>(reinterpret_cast<unsigned
       char*>(m_transferBufferPtr)+m_segmentSize*index+m_writeHead),
       reinterpret_cast<void*>(&t), sizeof(TypeOffsetPair));
22
23     m_writeHead+=sizeof(TypeOffsetPair);
24     m_stagedEvents=m_stagedEvents+1;
25   }
26 }
```

Appendix B

Device Side Code

B.1 Array of Structure Device Interpretation

```
#define MAX_LOCAL_EVENT_SIZE 100
#define WORK_GROUP_WIDTH 16

layout(std140, binding = 0) buffer event_buffer_block
{
  event events[];
};

uniform int event_count;

layout(binding = 0, offset = 0) uniform atomic_uint global_write_head;

layout(local_size_x=WORK_GROUP_WIDTH, local_size_y=1, local_size_z=1) in;

shared uint local_write_head = 0;
shared uint local_selector = 0;

shared event local_events[MAX_LOCAL_EVENT_SIZE];

void main()
{
  uint id = gl_GlobalInvocationID.x;
  atomicOr(local_selector, WORK_ITEM_SELECTOR);

  barrier();
  for(uint i=gl_LocalInvocationID.x;i < event_count; i+=WORK_GROUP_WIDTH)
  {
    event e = events[i];
    uint index = 0;
    int mask = int(pow(2, e.type));
    if(index < MAX_LOCAL_EVENT_SIZE && bool(mask & local_selector))
    {
      index = atomicAdd(local_write_head, 1);
      if(index < MAX_LOCAL_EVENT_SIZE)
      {
        local_events[index] = e;
      }
    }
  }
}
```

B.2 Structure of Arrays Device Interpretation

```
#define MAX_LOCAL_EVENT_SIZE 100
#define WORK_GROUP_WIDTH 16

layout(std140, binding = 0) buffer event_data_buffer_block
{
  event_data event_datas[];
};

layout(std140, binding = 0) buffer event_type_offset_buffer_block
{
  event_type_offset event_type_offsets[];
};

uniform int event_count;

layout(binding = 0, offset = 0) uniform atomic_uint global_write_head;

layout(local_size_x=WORK_GROUP_WIDTH, local_size_y=1, local_size_z=1) in;

shared uint local_write_head = 0;
shared uint local_selector = 0;

shared event_data local_event_data[MAX_LOCAL_EVENT_SIZE];

void main()
{
  uint id = gl_GlobalInvocationID.x;
  atomicOr(local_selector, WORK_ITEM_SELECTOR);

  barrier();
  for(uint i=gl_LocalInvocationID.x;i < event_count; i+=WORK_GROUP_WIDTH)
  {
    event_type_offset t = event_type_offsets[i];
    int mask = int(pow(2, t.type));
    if(bool(mask & local_selector))
    {
      uint index = atomicAdd(local_write_head, 1);
      local_event_data[index] = event_datas[t.offset];
    }
  }
}
```

B.3 Common Device Staging

```
#define WORK_GROUP_WIDTH 16
layout(std140, binding = 0) buffer key_event_buffer_block
{
  event events[];
};

layout(binding = 0, offset = 0) uniform atomic_uint global_write_head;

layout(local_size_x=WORK_GROUP_WIDTH, local_size_y=1, local_size_z=1) in;

void main()
{
```

```
13    uint id = gl_GlobalInvocationID.x;
14    if(gl_GlobalInvocationID.x%DEVICE_TO_HOST_OCCURANCE == 0)
15    {
16      event e;
17      e.type = uint(0);
18      unsigned int write_head = atomicCounterIncrement(global_write_head);
19      events[write_head] = e;
20    }
21  }
```