USSE Performance and Shader Optimizations: Tutorial

PlayStation®Vita Programmer Tool Runtime Library Release 3.50

Document serial number: 000004892117

Table of Contents

About This Document **4**
Purpose 4
Audience 4
Related Documentation 4
1 Introduction to USSE **5**
Introduction to USSE Features 5
Programmable Data Sequencer (PDS) 5
Unified Store 5
USSE Multi-threading Modes 6
USSE Execution 6
2 Performance Considerations **8**
Secondary and Primary Programs 8
Secondary Programs 8
Primary Programs 9
Uniform Data 9
Register-Based Uniforms 9
Memory-Based Uniforms 10
Pre-fetching Uniforms into SA Registers 10
Loading Order of Uniforms 10
Recommendations 10
Interpolant Performance 11
Cost of Interpolation 12
Cost of Interpolation Transfer 12
Recommendations 13
Texture Fetch Performance 13
Dependent vs. Non-dependent Texture Reads 14
Recommendations 20
Branch Performance 20
Recommendations 23
Loops: Special Branch Case 24
Static Loops Using Constant Iterations 24
Using Variable Loop Iterations 24
Recommendations 26
Register Spilling and Its Effects 27
3 Shader Optimizations **28**
Compiler Flags 28
fastmath 28
fastprecision 28
#pragma position_invariant 29
Instructions Implemented in Software 30
Normalize 30
Saturate 30
Swizzle 30
Encoding Functions as a Texture 31
Vectorization 31

Variable Precision....32
Precision in Vertex Shaders....32
Precision in Fragment Shaders....32
4 psp2shaderperf Tool....33
High-Level Shader Analysis and Symbols....33
Disassembly....34
5 Best Practices....35
Convert Dependent to Non-dependent Texture....35
Vectorize Your Code....35
Use Register-Based Uniform Buffers When Possible....35
Limit the Number of Interpolants....35
Avoid Branches or Loops....35

About This Document

Purpose

This document provides shader-optimization test cases and performance statistical analysis for PlayStation®Vita to help you determine bottlenecks/limitations in your applications. It also explains the complexities related to shader programming on PlayStation®Vita, and provides effective tips for tuning and optimizing shaders to improve your application's performance. Much of the data used in this document was gathered using psp2cgc, psp2shaderperf, and Razor for PlayStation®Vita (Razor).

Audience

This document is intended for application developers whose main focus is PlayStation®Vita GPU development, with the goal of optimizing graphics performance.

Related Documentation

Refer to the following related documents for additional details:

- *libgxm Overview* – provides an overview of the libgxm graphics library and how function calls interact with the underlying GPU.
- *libgxm Reference* – provides detailed reference descriptions of each of the libgxm API functions and data structures.
- *GPU User's Guide* – provides a description of the PlayStation®Vita GPU (SGX543MP4+), with a detailed overview of the USSE architecture.
- *Shader Compiler User's Guide* – provides a description of the PlayStation®Vita shader compiler and its usage.
- *psp2shaderperf User's Guide* – provides an overview of the psp2shaderperf tool, which provides static analysis of compiled shader programs.
- *Performance Analysis and GPU Debugging* – provides detailed descriptions of Razor GPU performance analysis and debugging features.

1 Introduction to USSE

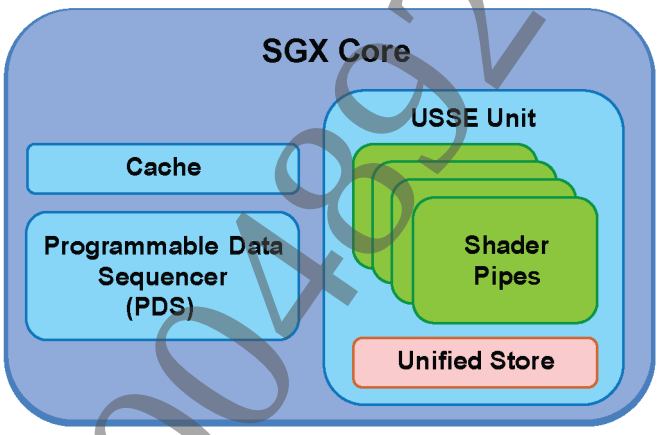
The PlayStation®Vita GPU is based on the fully programmable shader unit Universal Scalable Shader Engine (USSE), also known as *Unified Shader*, which provides flexibility and high performance for both vertex and fragment shaders. However, you can optimize your shader programs for both vertex and fragment processing in order to achieve substantial performance improvements and optimal efficiency from the USSE.

This chapter begins with an introduction to the extensive features of the PlayStation®Vita USSE unit, and then provides an overview of vertex and fragment processing.

Introduction to USSE Features

SGX is based on a unified shader architecture that can handle both vertex and fragment processing. The most important feature of this architecture is the fully programmable USSE unit, which has a highly optimized instruction set for processing both vertices and pixels efficiently. Each USSE unit contains four execution pipes, each with its own pool of active threads. These threads can be any mixture of vertex and fragment jobs, allowing vertex and fragment processing to be performed simultaneously. Workloads are first distributed between the cores and then assigned to the USSE pipes.

Figure 1 SGX Core Overview



The unified nature of the USSE unit ensures that pixels and vertices can be processed together with each sharing the USSE's resources and as a result maximizes resource usage. However, after either the vertex or pixel processing finishes, all USSE resources are available to the remainder of the other process, thus maximizing performance. This means that optimizing code that is not the main bottleneck may still result in a performance increase.

Programmable Data Sequencer (PDS)

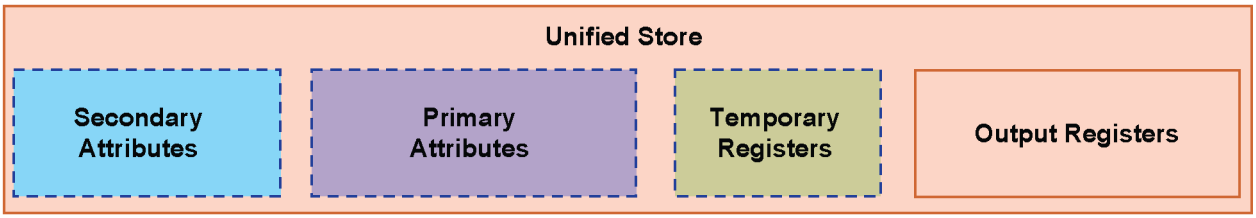
Another important component of the SGX pipeline is the PDS (Programmable Data Sequencer) unit, which is responsible for controlling and submitting work to the USSE, pre-fetching data, and handling USSE resource allocation prior to the USSE execution stage. Decoupling task generation and part of the data fetch from the execution stage allows much better pipelining of the data fetches and data usage, and also saves some valuable USSE cycles. Where a *task* is a workload made up of a collection of either vertices or fragments sharing the same shader.

Unified Store

Each USSE pipe contains a dedicated local memory (Unified Store) which hosts the primary and secondary attributes, as well as temporary and output registers. Unified Store is dynamically partitioned into these. This on-chip memory allows the USSE to minimize the need to read and write to external memory, thus minimizing latency and bandwidth consumption. Once the USSE has finished writing a set

of output registers, they are read by later units which then release them. For fragment processing, this is the Pixel Back End. For vertex processing, the Tiling Accelerator is responsible for this. A detailed description about the USSE's data flow is provided in the *GPU User's Guide.*

Figure 2 Different Register Types in Unified Store



The USSE relies on the unified store for both vertex and fragment processing phases. The unified store holds all the secondary attribute (`sa`), primary attribute (`pa`), temporary (`r`), and output (`o`) registers, with each register being 32 bits wide. So each register can hold, e.g. 1 `float`, 2 `half floats`, or 4 `uchar` when executing shader code. These registers are actually a chunk of embedded RAM and thus are subject to banking and related bank conflicts, which the compiler will try to avoid.

The partitioning of the unified store into primary attributes, secondary attributes, and temporary registers is dynamically controlled by the graphics driver and hardware. Note that most instructions are limited to 64 bits per operand from the unified store, which only allows `float2` or `half4` operations.

Additionally, the USSE also uses three 128-bit wide general-purpose internal (GPI) registers labeled `i0`, `i1` **and** `i2`, which are not located in the Unified Store. There are no bandwidth restrictions on GPI registers, allowing the hardware to fetch more than 128 bits per instruction. This in turn allows the hardware to execute `float3/4` operations in one cycle.

Furthermore, the GPI bandwidth does not interfere with Unified Store bandwidth, making it possible to read 256 bits from GPIs and 128 bits from Unified Store in the same instruction. These are *real* registers and are thus free of any bank clashes that can occur with registers in the Unified Store.

For additional details about the registers available to shader programs, refer to the *psp2shaderperf User's Guide.*

USSE Multi-threading Modes

After the input data is written to the registers in the unified store, the USSE will start processing tasks (see section Programmable Data Sequencer (PDS)). These tasks are split up into separate threads and executed on the USSE, in a multi-thread like way. The USSE executes these tasks in either of the two threading modes: per-instance or parallel mode. In either case, each thread has its own program counter (PC).

In per-instance mode, each thread has a single data instance, whereas in parallel mode each thread has 4 data instances, potentially providing four times better latency hiding through four times higher occupancy. Occupancy is a measure of how many active data instances are on the pipe in relation to the maximum number of active data instances that the pipe can hold. So if a workload suffers from stalls due to external dependencies, improving the occupancy may allow the pipe to execute instructions from additional data instances.

USSE Execution

Because both vertices and fragments can be processed by the USSE, there is overlapping execution for the two pipelines with dynamic load balancing across the USSE pipes. The PDS will generate the tasks, which are divided into threads that consist of data instances, where a data instance is a single vertex or fragment. Each task is made up of 1 to 16 individual data instances sharing the same shader; these are divided into threads and run either in per-instance or parallel mode.

The PDS then inserts these tasks into the USSE's task queue, which itself is 16 tasks deep. This allows the USSE to begin execution, leaving the PDS to carry on generating and submitting newer tasks. This decoupling of task submission and execution helps to minimize latency within the USSE pipe.

The USSE also has a thread manager with 16 thread slots. If the thread manager has any empty slots, the USSE will look through the task queue for tasks and will place their data instances into the empty thread slots and schedule them for execution. The order of execution between threads does not matter, because they do not share any data. So the USSE is free to execute them in any order, and does not have to follow the submission order of the PDS. For each USSE pipe, the thread manager can make 4 threads active at a time, which as mentioned earlier will run either in per-instance or parallel threading modes.

After a thread becomes active, it will remain active until processing for its up to four data instances has been completed or has stalled. If there are enough threads ready for execution, the ALU (Arithmetic Logic Unit) should be able to round robin between active threads, thus keeping the pipeline busy and helping to hide any latency. However, if not enough threads are ready for execution, USSE stalls can occur; you should be able to notice such stalls in Razor's metrics group 8.

If an active thread is running in parallel mode, each of its data instances will execute and consume a cycle. This means that if a thread has four valid data instances, it will consume up to four cycles before the next active thread can be swapped onto the ALU. So, parallel mode actually does not imply performing simultaneous execution of the four data instances, however it saves time on instruction decoding which allows each instruction to be shared across all valid instances.

USSE execution follows this policy for all the threads and will continue until all processing is complete. A detailed description about the USSE's execution model is provided in the SDK document *GPU User's Guide.*

2 Performance Considerations

The SGX core was designed to prevent USSE stalls due to memory problems as much as possible. This is accomplished by performing memory accesses outside of the shader cores, where possible. However, despite the PlayStation®Vita USSE being highly efficient, it can be a bottleneck. The bottlenecks at different stages in the pipeline can force USSE tasks to go into waiting mode. This could result in threads not being ready, causing the ALUs to be idled and thus affecting overall USSE performance.

The kinds of bottlenecks that can limit performance include:

- Register pressure
- Starving for work and back-pressure
- Memory stall, for example due to USSE-issued texture fetches

Additionally, if the shader runs in per-instance mode instead of parallel mode (for example because of a dynamic branch), performance can be further lowered because of low occupancy by the data instances in this mode.

Most of these bottlenecks are due to shader complexity and inefficient shader operations. So simplifying and optimizing your shader code is an important step toward improving USSE performance. This chapter examines some key factors that could have a large impact on shader performance. It begins by introducing the secondary and primary programs and discussing how they relate to you shader code.

Secondary and Primary Programs

Shaders can store/compute two types of values: one that will remain constant for all the data instances, and a second that changes for each data instance. Using a single shader to compute both types of values would result in redundant values computed for all instances. For example, when computing a clip-space matrix in vertex programs, the computed matrix will always remain the same across the entire batch, and it would be waste of time to compute it for every vertex.

For the USSE architecture, the PlayStation®Vita shader compiler solves this problem by dividing the shader instructions into two programs: secondary and primary programs. Division of the shader program is done by the shader compiler, which simplifies the computation by generating the secondary and primary programs.

Secondary Programs

Secondary programs are similar in concept to DirectX9 pre-shaders, but instead of computations occurring on the CPU side, secondary programs will be executed by the USSE. Even though they are referred to as being secondary they are always executed before the primary program. Their job is to simplify the primary program by computing data that will stay constant across all data instances within a batch, and handling the loading of memory-based uniforms (Please see the Uniform Data section) fetched from an offset computed in the secondary program.

Ideally, secondary programs would only need to run once for each draw call; however due to multiple cores/pipes and the tile-based rendering architecture, the secondary programs can run multiple times for both the processing phases:

- For vertex processing, depending on triangle distribution among the cores, the secondary program may need to run independently for each USSE pipe.
- For fragment processing, when a fragment shader state or uniform data changes, the secondary program of a new fragment program is run independently for each pipe processing the draw call. So for each tile that the draw touches, there can be up to 4 invocations of the secondary program that will fill up the unified store for each active pipe.

The secondary programs do not always exist, but will be generated by the compiler if code can be extracted out of the primary program. When a secondary program exists, it will be placed on the USSE

task queue in its own task by the PDS unit. It includes the instructions that, when executed, will compute and store values that are constant for all the data instances as SAs in the unified store. The SAs are thus updated only once by the secondary programs and are referenced for multiple data instances during the primary program execution later.

Primary Programs

The primary program forms the core of your shader program and performs the bulk of the processing. The PDS will issue the primary program as a task, and places it in the task queue with a dependency that enforces it to be executed only after the task made up of the secondary program (if any) is executed. The primary program consists of instructions/values that are not shared across data instances and are executed independently for each vertex or pixel. The program will use the secondary and primary-attribute registers in the unified store, and store back the intermediate results either as temporary or primary attributes. The final results of the primary programs are written to the output registers. Note that this is only done by the shader compiler if `__nativecolor` is used. Otherwise the run-time shader patcher adds the final code that writes to the output registers.

Uniform Data

Uniform variables represent data passed by the user that will remain constant for all vertices or fragments processed as part of a draw call. The libgxm rendering API exposes two types of uniform buffers for holding uniform data: user-declared uniform buffers (declared in shader code), and the default uniform buffer. Details about these buffers are provided in the *libgxm Overview*.

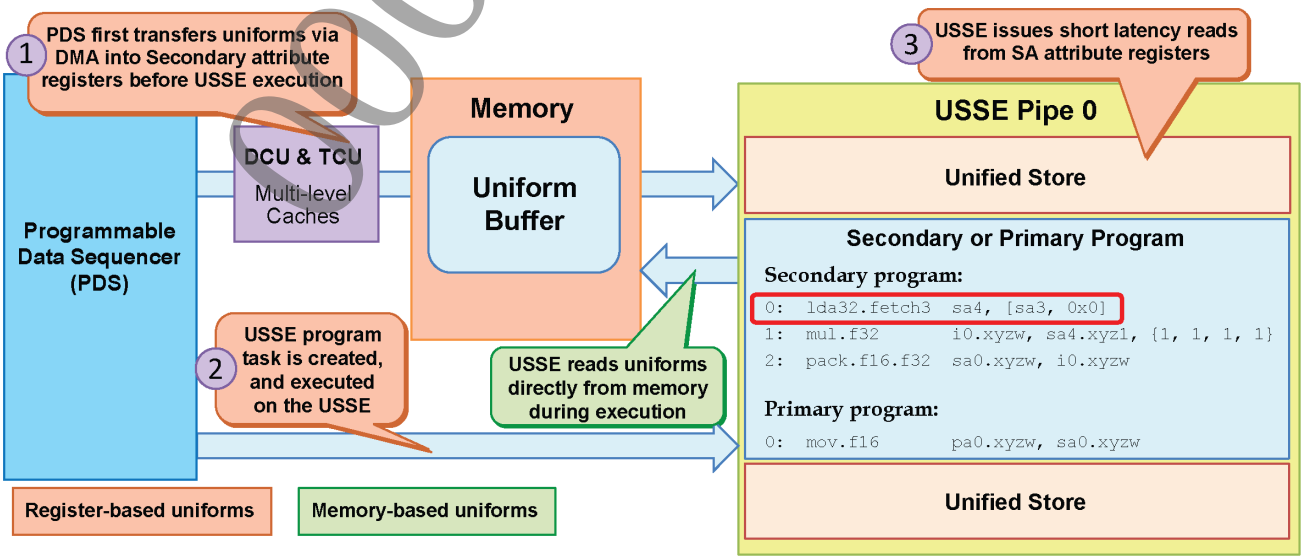
PlayStation®Vita can upload uniform data into the GPU via two paths (see Figure 3):

- PDS – The PDS can issue a DMA transfer directly from memory into SA registers when processing a group of data instances. Buffers are loaded before USSE execution and are referred to as "register based".
- USSE – The USSE itself can load data from memory in either secondary or primary program. Buffers are loaded during shader execution and are referred to as "memory based".

Register-Based Uniforms

The register-based uniform fetch is the fastest way of loading to the unified store because SA registers are filled before the USSE shader program thread (which was submitted by the PDS) is started, and short latency reads are then performed from the registers to fetch data during USSE execution. Each uniform component (if loaded) will be assigned to consecutive registers, assuming no registers are required for alignment.

Figure 3 Overview of Uniform Buffer Upload to Unified Store for USSE Processing



Memory-Based Uniforms

The memory-based uniform fetches are issued by the USSE via the **LDA** instruction, where each load can handle 64 bytes per instruction. To minimize the amount of DMA by a primary program, the compiler will try to issue these load instructions in the secondary program, but if a shader goes over the limit of secondary-attribute registers, loads will be forced into the primary programs.

Pre-fetching Uniforms into SA Registers

Default uniforms are either stored as registers in the unified store or placed in memory depending on the available space in the SA block in the unified store. By default, the user-defined uniform buffers are placed in main memory, but you can use **`#pragma register_buffer`** to force the buffers to be pre-fetched by the PDS and to place them directly into SA registers.

Loading Order of Uniforms

If user-defined uniform buffers are set to be placed in registers, they will be favored to be placed first into the SA registers before the default uniform buffer. If there is still register space available, default uniforms will be loaded, where the attribute allocation will be decided based on usage frequency within the shader. If any remaining uniforms do not fit within the 128 secondary attribute limit, the compiler will demote them to memory-based uniforms and they will be loaded during primary program execution; accessing them will result in increased bandwidth and latency costs.

Recommendations

For user-defined uniform buffers placed in memory, the read is issued by the USSE either from the secondary or primary program. Performing direct memory accesses from within secondary or primary programs incurs significantly higher latency than accesses to the registers loaded via DMA issued by the PDS unit, but if memory-based uniforms get loaded in the secondary programs, the impact will be less than loading primary programs. Uniforms will be loaded by the primary program when using dynamic indexing to fetch data from user-defined uniform buffers placed in memory.

Table 1 Uniform Buffer Setup and Efficiency

Uniform buffer setup	Type of operations for data fetch	Efficiency
Default uniform buffer (default behavior) User-defined uniform buffer with **`#pragma register_buffer`**	PDS: Fetches uniforms from memory through DCU before USSE execution.	Best because no USSE processing is involved. DMA latency increases as the number of uniforms increase.
User-defined uniform buffer (default behavior)	USSE: In secondary program via LDA instruction.	Resultant cycles increase as LDA instruction count increases; better performance when compared to fetching in primary program. **But USSE will only read in necessary uniforms.**
Uniforms reach 128 SA registers limit Dynamically indexed data from user-defined uniform buffer with **`#pragma memory_buffer`**	USSE: In primary program via LDA instruction.	Worst, because the fetch occurs for each data instance. Highly impacted as LDA count increases. **But USSE will only read in necessary uniforms.**

However, issuing load instructions from the USSE offers an advantage when performing sparse fetches from a large buffer; it allows the user to be selective and to read only the data that is needed. This is significantly more memory efficient than loading in the entire buffer, which occurs when issuing a load

from PDS and results in an unnecessarily high usage of bandwidth and registers, potentially stalling the USSE.

Usage Model

Normally the setup of uniform buffer depends on your usage, so if you have small set of uniforms that are frequently used it might be fast to have them pre-fetched into registers, whereas if you have large buffers that are sparsely used it might be efficient to load them straight from memory.

For example, when performing skinning, use a memory-based buffer and only load in those skin matrices that are required by the USSE. This will be significantly more efficient than simply loading in the entire matrix buffer.

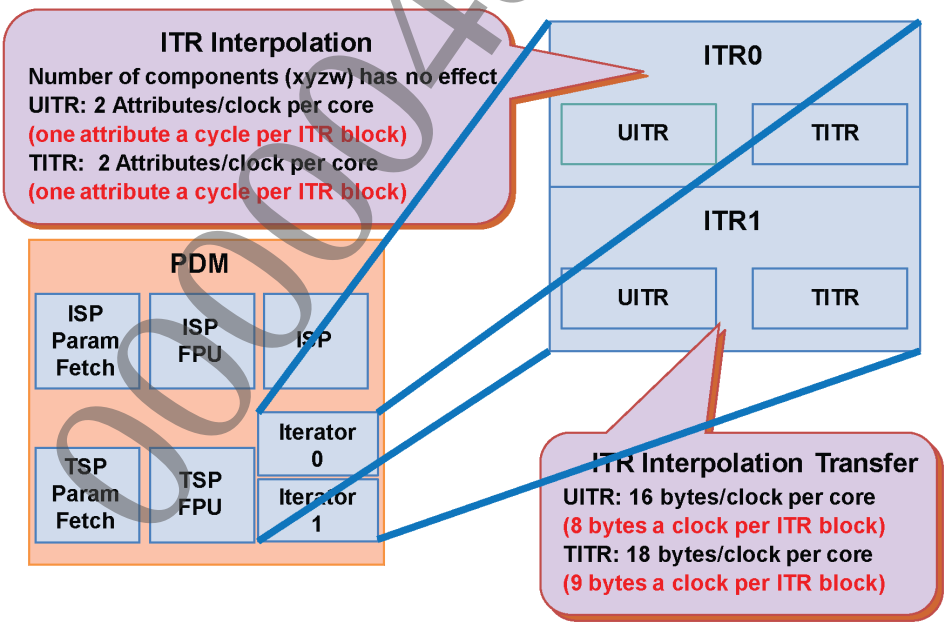
Interpolant Performance

Interpolants are used as inputs in the fragment program for performing per-fragment operations. These include color, texture coordinates, window position coordinates, and others. For an extensive list, refer to the *Shader Compiler User's Guide*.

Because these interpolants are exclusive to each fragment, they are always stored in primary-attribute registers when the PDS primary program is executed. The attribute values output by the vertex program are per-vertex; however after each vertex's data corresponding to the triangles is processed into tiles and rasterized into fragments, these attribute values are interpolated across the triangle, producing a corresponding attribute value per-fragment. The computation generates per-fragment data by using the XY position outputs from the vertex shader (fetched from the parameter buffer over the Texture and Shader setup Processor (TSP) unit and the plane equation generated by the TSP FPU.

Iterator (ITR) blocks are responsible for generating per-fragment color and texture-coordinate interpolants for visible primitives. There are two ITR blocks within a core ITR 0 and ITR 1, each shared by two USSE pipes. Interpolation within these blocks is always performed at 24-bit float precision.

Figure 4 Iterator (ITR) Block Diagram



Each iterator block is further divided into two sub-blocks:

- UITR is responsible for the interpolation of all attributes that will be directly forwarded to the USSE.
- TITR is responsible for the interpolation of texture coordinates that will be forwarded to the Texture Address Generator (TAG) blocks for non-dependent texture reads.

An ITR block has two processing costs associated with it: first, the cost of performing interpolation on attributes; second, the cost of transferring the computed interpolated value out to the TAG or USSE unit.

Cost of Interpolation

The cost of performing interpolation depends on the number of interpolators, not on how many components each interpolator has (*xyzw*). So if ITR input becomes a bottleneck, it is beneficial to pack multiple components into single interpolators (for example pack two `half2` interpolants into one `half4` interpolator). The ITR block on each core can compute 2 attributes per clock, where 1 attribute can be up to a 4-components vector. However, note the following limitations:

- Packing benefits relate only to interpolation cost, and not to the transfer cost of getting the results.
- Packing in texture coordinates can change non-dependent texture reads to dependent reads.
- Each core's ITR block feeds four USSE pipes, so every pipe is supplied with 0.5 attributes/clock.

Cost of Interpolation Transfer

The cost of transferring interpolators depends on the data size of each interpolator. Each interpolator transfer takes 1 cycle / 64 bits, so a transfer of fp16x4 will take one cycle but an fp32x4 transfer will take 2 cycles. Thus it can be beneficial to reduce the data type size to a minimum to increase the transfer rate, if it becomes the bottleneck.

Table 2 Improving Transfer Rate by Replacing Full-precision Interpolants with Half-precision Interpolants

Full float	Half float

`float2 Texcoord : TEXCOORD0;`
`float3 WorldNormal : TEXCOORD1;`
`float3 WorldPos : TEXCOORD2;`
	`half2 Texcoord : TEXCOORD0;`
	`half3 WorldNormal : TEXCOORD1;`
	`half3 WorldPos : TEXCOORD2;`

This was tested via a simple example using a 7 cycle fragment shader with three interpolants, first at full float, and then using half floats.

Analysis

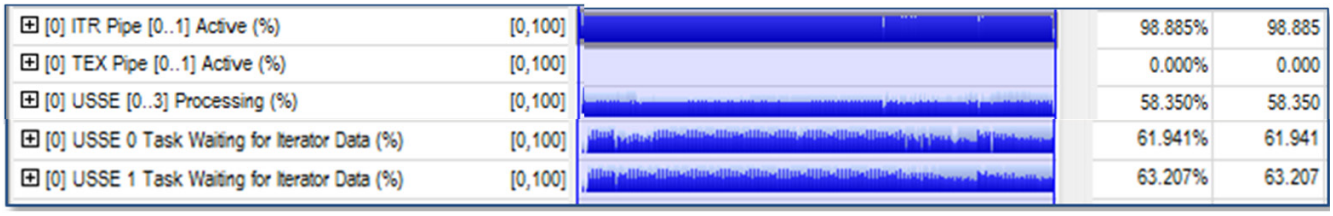
When using full float, the Razor capture (**metrics group 0**), showed that the ITR activity was approximately 98.9%. Over the course of a whole frame, it took roughly 3 ms to complete. Also if you look at the Razor capture (**metrics group 19**), you can see that 62.95% of the time when the scheduler wanted to issue a task, it was blocked from executing by an iterator dependency. This actually leads to the USSE running out of work for 41.65% of the time, with processing occurring for only 58.35%, as can be noted from the USSE Processing metric in the Razor capture.

Table 3 Effects of Lowering the Interpolant Precision

	Full float Core 0	Half float Core 0
Average Fragment Activity	3.073 ms	2.55 ms
Average ITR Pipe Active	97.87%	73.86%
Average Task Waiting for Iterator	62.95%	4.67%
Vertex Shader Cycles	18	18
Fragment Shader Cycles	6	7

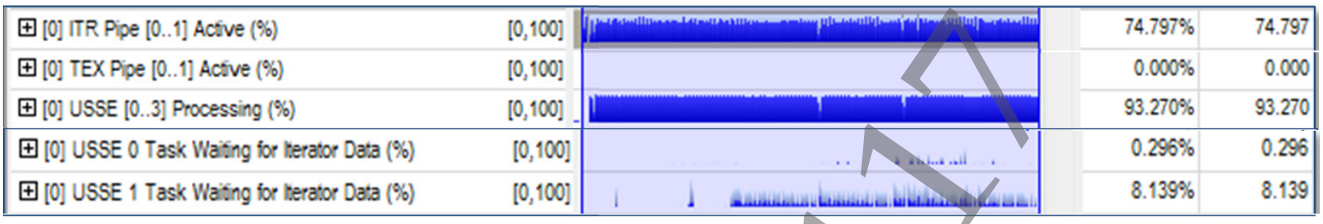
Note that even though the task was blocked, the USSE could still be busy working on other threads. Because the task manager issues many more tasks than the USSE can work on, it causes the task manager to effectively suffer from back-pressure. So, even though this causes tasks to wait, there is not always a complete stall in the pipeline because the USSE has enough work, unless an insufficient number of threads are available.

Figure 5 Metrics When Using Full-float Interpolants in Fragment Shader



The next test uses half-float inputs inside of the fragment program. Reducing the precision causes ITR activity to drop by roughly 24%, as shown by the Razor captures in Figure 6.

Figure 6 Metrics When Reducing Precision of Interpolants to half float in Fragment Shader



The time the task needed to wait on the iterator unit was also greatly reduced, to an average value of 4% (with the fragment processing time dropping to 2.5 ms), saving over half a ms for the whole frame.

Recommendations

When investigating interpolant performance, it is important to profile in Razor to ensure that the ITR unit is actually the bottleneck. To do this, you can use Razor capture (metrics group 0), which will provide details on ITR activity.

- Demote precision of interpolants: If the iterator unit activity is high, demoting the precision of interpolants can help to minimize their impact on the bus.
- Reduce the number of interpolants: Reducing the number of interpolants by packing can additionally reduce interpolation time, although most shaders should be able to hide this latency.

When making these changes, make sure you do not generate any unnecessary instructions when unpacking interpolants in your fragment shaders, because it may outweigh the gains. Secondly, you should avoid packing texture coordinates into *zw* components of an interpolant, because you will lose non-dependent texture reads. Finally, be careful not to use interpolants if they are unnecessary, like for the case when vertex program outputs interpolants that the fragment program does not read. This is because the increase of ITR activity can reduce the efficiency of an executing task if the fragment shader is not large enough to hide the latency.

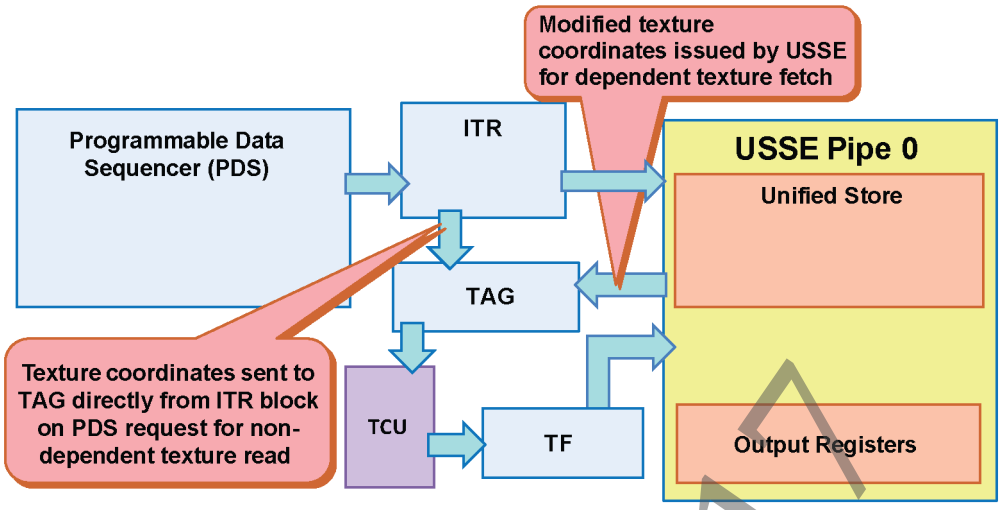
Texture Fetch Performance

Texture fetch operations could be one of the main bottlenecks in an application and can have a large impact on the USSE's performance due to high latency. A number of factors should be considered for reducing the texture related performance penalty; these factors are related to efficient texture memory bandwidth usage and latency hiding. Some of these factors that influence performance are:

- Dependent or non dependent reads
- Memory location/layout
- Texture format
- Texture compression/mipmaps
- Filtering mode
- Texture count

Note: This section only covers dependent and non-dependent texture reads, because everything else is out of scope for the USSE pipeline.

Figure 7 Texture Fetch Pipeline Within a SGX Core



For the PlayStation®Vita GPU, the texture pipeline consists of multiple stages to handle fetching and filtering of texture data before passing it to the USSE (as shown in Figure 7):

- There is the ITR unit with UITR or TITR blocks that compute the interpolated texture coordinates used per-pixel, as has been discussed in the previous section on interpolators.
- There are the TAG blocks that receive the texture coordinates from the ITR or USSE, along with the corresponding state information, from which the TAG unit generates the texture lookup addresses. Also, if mip-mapping is enabled, the derivative corresponding to the texture coordinates of the pixels in the 2x2 quad is computed and the mipmap level is set. The TAG unit then sends the relevant information to the Texture Cache Unit (TCU) for performing texture fetch operation.
- The TCU is a dedicated data cache, responsible for reducing the memory bandwidth and latency of texture lookup operations.
- After the fetch is complete, the Texture Filter (TF) blocks will receive the texel data from the TCU and the texture states and filter coefficients from the TAG unit, and will use this information to format, expand, and filter the texture lookup pass The filtered texture values are passed to the USSE.

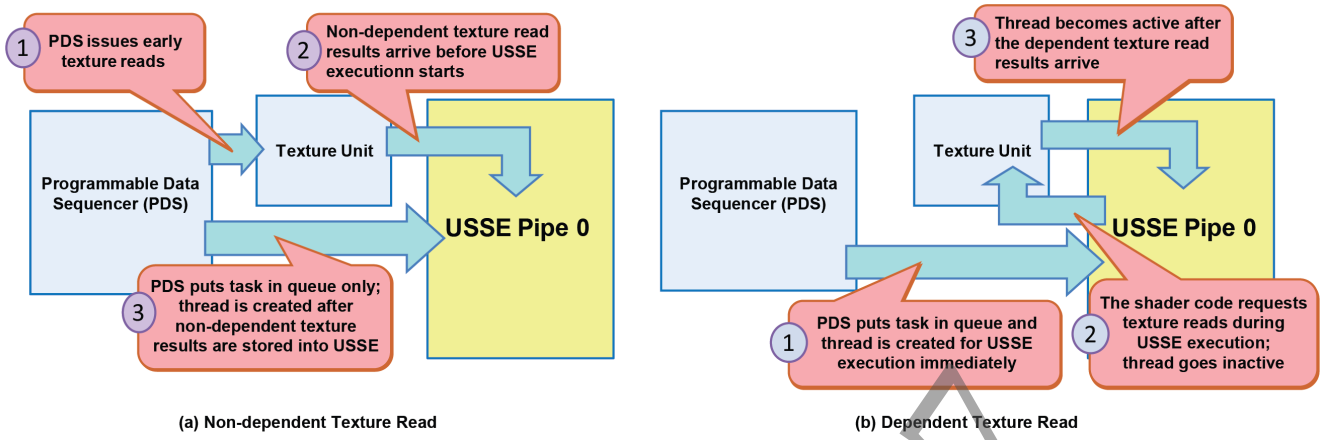
Dependent vs. Non-dependent Texture Reads

There are two types of texture fetching available: non-dependent and dependent reads. Their usage depends on whether the request is issued early by PDS (before USSE processing starts to hide latency) or in more standard way by the shader core itself during USSE processing. The setup for both types of texture fetching is shown in Figure 8.

In the case of non-dependent textures, the texture coordinates will be computed by TITR with values passed directly to the TAG blocks, and fetched texture values are stored into PA registers before USSE execution starts. These reads are just shader inputs, like interpolants, which can be set up before the thread (and its temporary registers) is ever allocated.

In the case of dependent texture reads, the texture-coordinates are computed by the shader program and then passed to TAG blocks using a USSE instruction. The destination register of that instruction can be either a PA or temporary register.

Figure 8 Pipeline for Non-dependent and Dependent Texture Fetches



A non-dependent texture read is issued only when using unconditional reads (not fetched within a branch), with unmodified texture coordinates. Also, the fragment program expects that only specific components be used with specific functions. Table 4 shows a list of function calls with specific components that result in a non-dependent texture fetch.

Table 4 List of Non-dependent Texture Read Function Call Cases

Function Call
`tex1D(texture1DMap, texCoord.x)`
`tex2D(texture2DMap, texCoord.xy)`
`tex2Dproj(texture2DMap, texCoord.xyw)`
`texCUBE(textureCubeMap, texCoord.xyz)`
`texCUBE(textureCubeMap, texCoord.xyzw)`

Non-dependant texture reads have the following advantages:

- Texture fetches occur before the USSE thread is created, which means that the hardware can simultaneously process the non-dependent reads of more data instances than the USSE can hold.
- The number of shader cycles is reduced because there is no instruction needed to fetch the texture within the fragment program.
- The need for additional attribute registers to store/compute texture coordinates is eliminated in some cases, reducing the register pressure that would have occurred if the USSE had to issue a fetch.
- The SA usage is reduced, since no control words need to be stored for that texture.

In the case of dependent textures, the texture coordinates are modified within the shader code and issued by the USSE. Some of the cases that can result in dependent reads are:

- Modifying texture coordinates within the fragment program
- Texture fetching in a conditional block
- Using any texture coordinate swizzles other than those given in Table 4
- Vertex shader texture fetching (such fetches originate from the USSE and so are always dependent reads)

Because dependent texture reads are issued from the USSE during execution, threads can enter waiting mode for the texture lookup results. Moreover, if there are not enough threads to hide texture fetch latency, the USSE may stall; you will be able to see this in Razor, corresponding to the "USSE * Stalls Due to Dependent Texture" metrics in the metric group 8.

Test Case 1: Swizzled texture coordinates

This test case demonstrates a simple case of performance difference between non-dependent and dependent texture fetch. As shown in the Figure 9, the non-dependent texture fetch is issued by using *.xy* components for the texture coordinates, which store the texture fetch in a primary attribute; in this case,

the primary program has 2 instruction to execute. If the texture coordinates are instead stored in the *.zw* component (also shown in Figure 9) of an interpolant in a vertex shader, a dependent texture fetch is issued that increases the instruction count in the primary program to 4; this includes an instruction to move the texture coordinates to the .xy component and instructions related to performing the texture lookup.

Figure 9 Performance Comparison for a Simple Case of Non-Dependent and Dependent Texture Reads

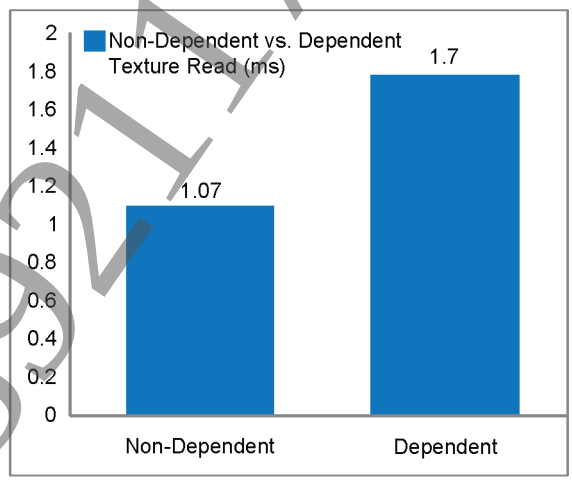
```
half4 main(half4 vColor : COLOR0,
        half4 vTexCoord : TEXCOORD0,
        uniform sampler2D modelTex : TEXUNIT0
    ): COLOR
    {
        half4 fragmentCol = tex2D(modelTex, vTexCoord.xy);    // non-dependent read
     // half4 fragmentCol = tex2D(modelTex, vTexCoord.zw) ;   // dependent read
        return fragmentCol * vColor;
    }
```

Non-Dependent read Primary program:

```
pa2=tex2D<half4>(modelTex, TEXCOORD0.xy)
0:  mul.f16  pa0.xyzw, pa2.xyzw, pa0.xyzw
1:  mov.f32  o0.xy, pa0.xy
```

Dependent read Primary program:

```
0:  mov.f16    pa2.xy, pa2.zw
1:  tex2D.f16  pa2, pa2.xy, sa2
2:  mul.f16    pa0.xyzw, pa2.xyzw, pa0.xyzw
3:  mov.f32    o0.xy, pa0.xy
```

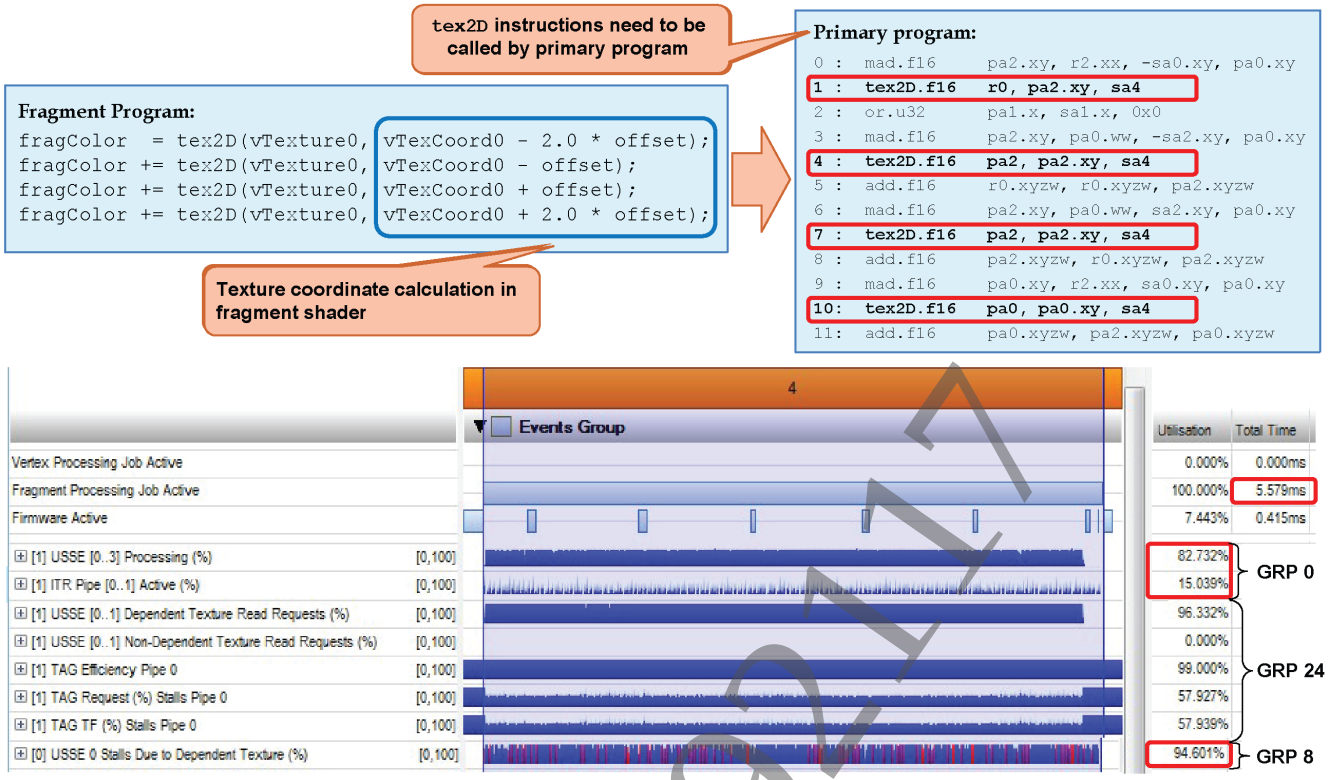


You can see that non-dependent texture fetch has a lower cost in comparison to dependent texture fetch, which can be attributed to the increased number of instructions as well as an increase in stalls due to texture fetch latency during USSE execution.

Test Case 2: Converting dependent to non-dependent texture fetch

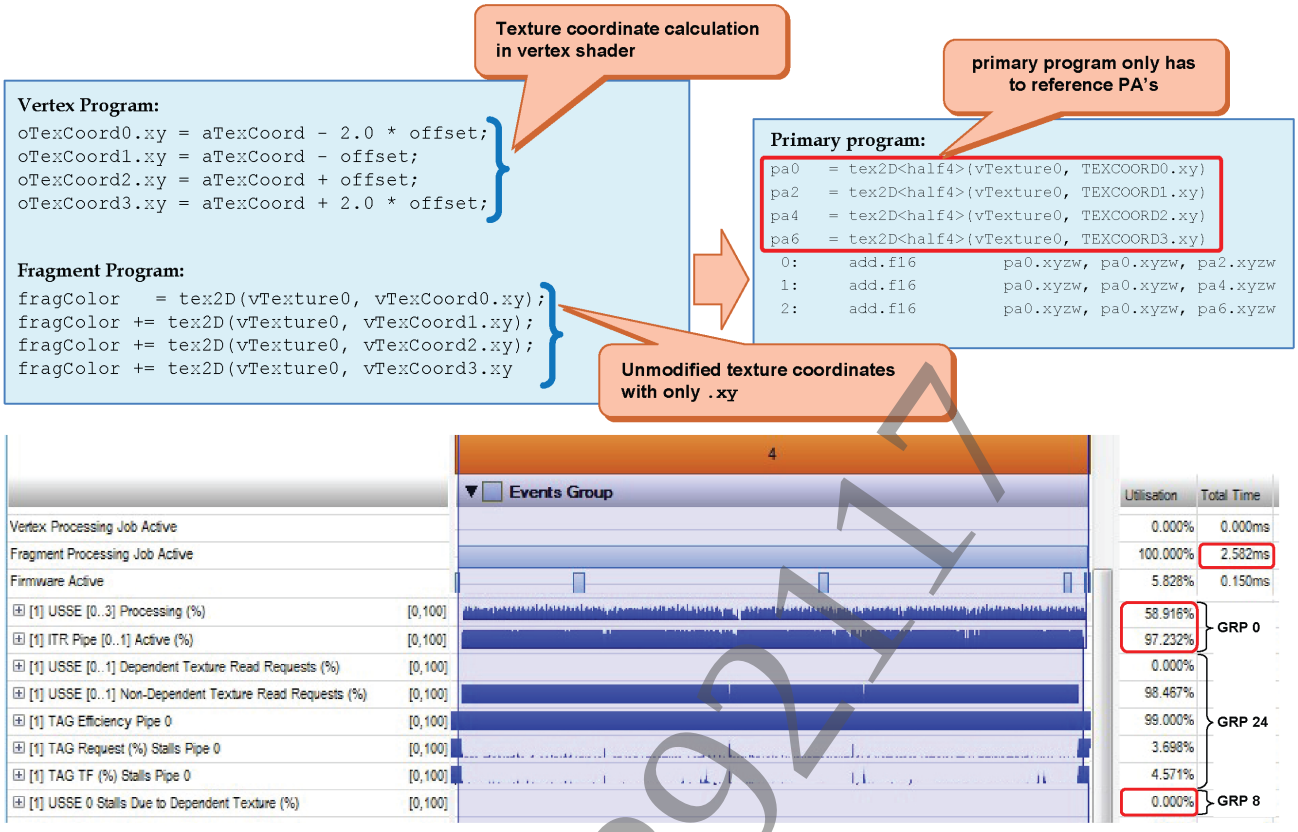
This test case also demonstrates the advantages of using non-dependent texture fetches, and shows how this can be achieved by moving texture-coordinate related calculation to the vertex program. As you can see in Figure 10, performing texture fetches with modified texture coordinates in a fragment program results in explicit texture fetch instructions in the fragment's primary program. In addition to the high instruction count, texture fetches within the shader result in a high USSE-stall percentage, as can be seen in the Razor capture (metric group 8) at the bottom of Figure 10. This is one of the major bottlenecks associated with dependent reads and consequently reduces the performance of the USSE pipes.

Figure 10 Useful Metrics When Performing Dependent Texture Reads, Captured Using Razor



Converting all dependent reads to non-dependent reads offers a practical way to remove the inefficiencies associated with dependent reads and to reduce the bottlenecks that cause the USSE pipes to stall. This can be effectively done by moving the texture coordinate calculations to the vertex program before passing it onto the fragment program, as shown in Figure 11.

Figure 11 Useful Metrics When Above Texture Reads Converted to Non-dependent Texture Reads



In the fragment program, the coordinates are simply passed through to the texture, and if the *.xy* component is not modified, a non-dependent texture fetch will be issued by the PDS; in this case, the compiler will not generate any code for the texture fetch in the primary program. When the program is executed, it only needs to reference texture data that will be stored in primary attributes and used during the pixel's execution. Looking at the Razor metrics (metric group 8), you can see that the texture fetch associated USSE stalls are reduced to 0 and the performance is drastically improved with the frame time reduced from 5.58 ms to 2.58 ms.

However, note that this change might sometimes result in another unit becoming a bottleneck, in this case the ITR unit. You can see from the Razor captures (metric group 0) for the two cases that the ITR activity has drastically increased for non-dependent texture. This is mainly because as the texture coordinates are computed in the vertex shader the number of iterators passed to the fragment shader increases, resulting in higher ITR activity.

So although performance has improved you can see another unit has become a bottleneck. That is why, whenever making such conversions, you need to profile and carefully study the results, taking into consideration the number of attributes being passed from the vertex shader to the fragment shader as well as parameter buffer usage.

Case of shadow reads

Both projective texture lookup and depth testing will occur when shadow mapping is implemented using `tex2Dproj()` with all 4 *.xyzw* components as texture coordinates. However, the texture fetch in this case is issued as a dependent read. The shader code `shadowTerm = f1tex2Dproj(shadowMap, texCoord.xyzw)` will generate code for dividing `texCoord.xy` by `texCoord.w`, and the depth test is performed by comparing against `texCoord.z/texCoord.w`. The shadow-map texture reads can be converted to non-dependent reads by shader code that implements shadow comparison explicitly as:

```
shadowTerm = (f1tex2Dproj(shadowMap, texCoord.xyw) >= (texCoord.z/texCoord.w))
```

However, note that shadow comparison by the shader code will give a binary result, and so should only be performed if a hard shadow edge is acceptable. Also, you will miss out on the very efficient PCF code

implementation that generates a dependent neighborhood fetch and does a vector PCF operation. You can achieve the same visual quality by doing a PCF operation; however, this would be inefficient because it would involve 4 non-dependent point sample reads as well as passing the texture coordinate to get a fractional value, and then doing the PCF arithmetic in shader code. In contrast, a hardware implementation can process 4 neighborhood texels at the same time, which is efficient.

Test Case 3: Cost of texture fetch when using control flow

If a texture fetch is based on a condition, it will not be rescheduled across the branch. This opens up the opportunity to control the path on which a texture fetch will be issued.

So for example when using discard, you can completely avoid a texture fetch if you place it after the discard operation, thus reducing memory bandwidth usage and improving shader performance. However, if the texture fetch is non-dependent, the fetch within the branch condition is demoted to a dependent texture read. So unless your discard test kills enough fragments, non-dependent texture fetch should be issued before the branch to reduce both instructions and stalls.

Table 5 Simple Example When Using Discard

```
half4 main(half4 vColor : COLOR0,
       float4 texCoord : TEXCOORD0,
       uniform sampler2D modelTex : TEXUNIT0): COLOR
   {
       half4 fragmentCol = tex2D(modelTex, texCoord.xy) * vColor;     // non-dependent
                                                                      // read
       // half4 fragmentCol = tex2D(modelTex, texCoord.zy) * vColor; // dependent read
       if(vColor.a > 0.5)
          discard;

       return fragmentCol;
   }
```

Non-Dependent Read		Dependent Read	
Estimated cost:	4 cycles, parallel mode	**Estimated cost:**	6 cycles, parallel mode
Register count:	4 PAs, 0 temps, 5 SAs *	**Register count:**	5 PAs, 0 temps, 9 SAs *
Texture reads:	1 non-dependent,	**Texture reads:**	0 non-dependent,
Dependent:	0 unconditional, 0 conditional	**Dependent:**	0 unconditional, 1 conditional

Samplers:

```
TEXUNIT0  = modelTex
```

Iterators:

```
pa0   = (half4) COLOR0
```

Primary program:

```
pa2       =       tex2D<half4>(modelTex,
TEXCOORD0.xy)
0: cmp.le.f16  p1, pa0.w, sa4.x
1: p1 kill
2: nop
3: mul.f16  pa0.xyzw, pa2.xyzw, pa0.xyzw
```

Samplers:

```
TEXUNIT0  = modelTex
```

Iterators:

```
pa0   = (half4) COLOR0
pa2   = (float3) TEXCOORD0
```

Primary program:

```
0: cmp.le.f16  p1, pa0.w, sa8.x
1: p1 kill
2: nop
3: mov.f32  pa2.xy, pa2.zy
4: tex2D.f16  pa2, pa2.xy, sa4
5:  mul.f16        pa0.xyzw,  pa2.xyzw,
pa0.xyzw
```

However, note that if a non-dependent texture fetch is issued within a branch it is demoted to a dependent texture fetch with instructions added in the primary program instead of being pre-fetched by PDS. This will increase the number of registers, because it will now need to store texture coordinates for lookup and any additional computation results. So this is something that should be avoided.

Additionally, because any dynamic branch will force the program into per-instance mode, additional USSE instructions will need to be computed for gradient computation to handle mip-selection, because

the hardware cannot use implicit gradients if the data instances in a 2x2 stamp have potentially divergent control flow.

Recommendations

Texture fetch can indeed be a potential bottleneck in the rendering pipeline, forcing the USSE unit to wait for texture data. However, the PlayStation®Vita GPU architecture has excellent latency hiding mechanisms that can effectively reduce the associated performance penalty. To facilitate this, the following recommendations should be considered:

- It is always a good idea to convert all dependent reads to non-dependent reads to reduce the load on the USSE where possible. Look for opportunities to:
  - Eliminate packing of texture coordinates in the .zw components of an interpolator.
  - Move texture-coordinate computations to the vertex program with no modifications in the fragment program.
  - Avoid non-dependent texture fetch within a static branch.
- Minimize texture fetches within a dynamic branch to avoid the shader computing the gradients explicitly, unless many fragments can be skipped. See the following section for more details.

Branch Performance

Both static and dynamic branching is supported in PlayStation®Vita GPU shader code, although it incurs an overhead penalty and when combined with memory accesses (other than non-dependent texture read) can drastically hurt performance. However, if used intelligently it can reduce the USSE workload by skipping expensive code paths that do not need to be executed, and in some cases can result in saving significant processing time.

Static flow control can be used for cases where the same block of code is to be disabled/enabled for all the data instances in a draw call – for example, discarding lighting computation for all vertices/fragments if a light is disabled, or skipping skinning computations if animation is disabled. It can also be used to combine multiple effects into one single shader; and then, depending on the required feature, the value of the uniform to be used for a conditional check can be set to get that behavior.

Dynamic flow control is useful in skipping shader instructions at each data-instance level, based on the computed data-dependent value. Although it causes some inefficiency because different vertices/fragments can take different code paths, in some cases it can provide better performance. For example, it can be used for early-out optimizations, such as skipping character bone animation if skin weight is zero, skipping lighting computation for vertices/fragments facing away from light, and applying soft shadow filtering only on penumbra regions of the shadow.

The higher cost of dynamic branches can be attributed to two sources:

- Lower occupancy makes USSE issued memory reads more costly. Shaders that do not have these will pay significantly less for dynamic branches.
- Invalid data instances are more costly than in parallel mode, as they still need to be computed fully. This needs some more testing on our side, however.

To determine the effects of branches on shader performance, four tests were performed as described below.

Test 1: Simple lighting scenario with a single static branch

This test shows the overhead cost associated with a static branch. The test compares a simple lighting scenario, where first shader without any branch is used. The shader has 28 instructions performing lighting calculations, which are always executed for all the valid pixels for the draw call.

Next, a branch is added across the lighting computation based on a uniform bool variable (see the left side of Figure 12 that can be toggled by the user. This allows the user to disable/enable the lighting if required. However, adding the branch generates additional instructions for setting up the branch conditions. The

performance cost is the cost of the branch (3 instructions) plus the cost of the instructions. The right side of Figure 12 compares the costs of using no branch versus a static branch; it shows an increase of ~0.43 ms.

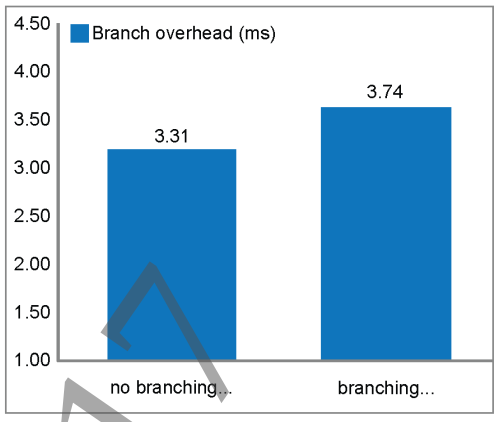
Figure 12 A Simple Test to Determine Overhead Cost of a Static Branch

```
half4 fragColor = meshColor;
if(lightEnabled)    // Global static branch executed if
                    uniform bool lightEnabled is true
{
    // Set up diffuse component computations and
    ambient color term Ca

        ...

    // Compute diffuse color term Cd
    // Compute the specular color term Cs

    fragColor.rgb *= (Ca + Cd + Cs);
}
```



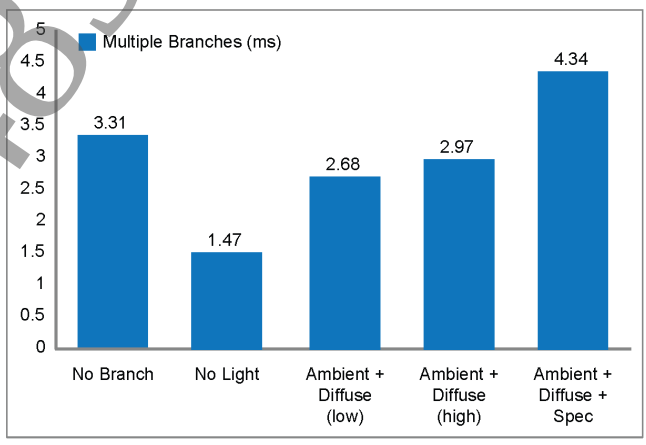
Test 2: Multiple branches to handle lighting

The case in Test 1 above is extended to include multiple branches, such as two static branches (**Branch 1 & 3**) and one dynamic branch (**Branch 2**), as shown in Figure 13.

Figure 13 Use of Multiple Branches to Control Different Effects; Resulting Changes in Performance

```
half4 fragColor = tex2D(modelTex, vTexCoord.xy);
if(lightEnabled)    // Branch 1 (static) executed
                    if lighting enabled
{
    // Set up diffuse component computations
    and ambient color term Ca
        ...
    half diffuseCoeff = dot(N, lightDirection);
        ...
    if(diffuseCoeff > 0.0)  // Branch 2 (dynamic)
                executed if fragment faces light
    {
        // Compute diffuse color term Cd

        if(specularEnabled)  // Branch 3 (static)
            executed if specular component enabled
        {
            // Compute the specular color term Cs
        }
    }
    fragColor.rgb *= (Ca + Cd + Cs);
}
```



The test is aimed at finding the performance cost of a shader with multiple branches. Different conditions are set to enable/disable the branches and to check the performance. The Branch 1 condition is a static branch that can be toggled by the user. If the condition is not satisfied (false), all lighting computations are excluded with the resultant cost shown by the second bar in Figure 13. If true, the ambient lighting is computed and the condition for the dynamic branch (Branch 2) to compute the diffuse term is set. In this case, the condition is that the primitives must face the light (N.L > 0.0); if true, instructions are executed for the fragments that satisfy it.

The cost of performance depends on the number of fragments that face the light, which varies as light is moved. Performance will improve as the number of fragments that are excluded (based on the dynamic condition) increases. As shown by the third and fourth bars in Figure 13, the costs for the following two cases were recorded: one with a low number of fragments facing the light, and the second with a higher number.

Branch 3 is another static branch that, if true, results in specular highlights being computed for the light. This can also be toggled by the user, and if enabled will result in the object rendered with both ambient

and diffuse as well as specular light components. The cost for this case is shown in the last bar in Figure 13.

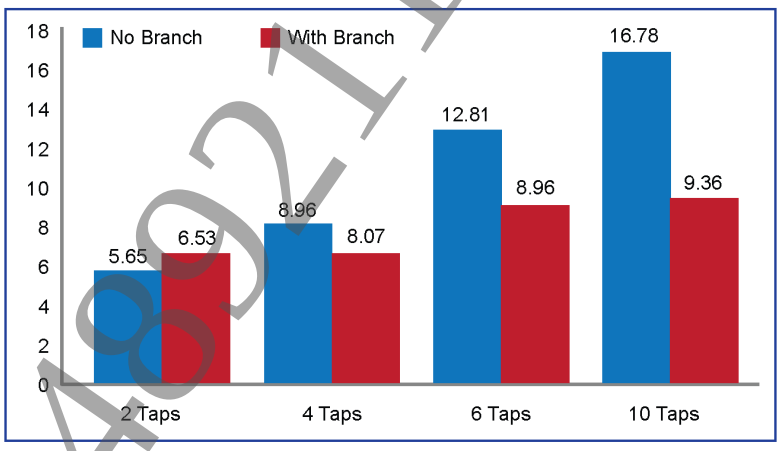
Test 3: Dynamic branching for soft shadows

This test involved soft shadow computation to demonstrate that it can be advantageous to use dynamic branch to skip expensive instructions for a large number of data instances that are not affected. The branch is used to apply soft-shadow computation only to the shadow boundary to give a smoother transition between shadowed and unshadowed regions (the penumbra region).

The shadow penumbra region is computed using a partial set of samples and then conditional checking of this result for each fragment. The partial set includes two neighboring samples and the average value for the shadow tests is used for the condition, although for better accuracy more samples could be considered. If the average value is either 0 (all samples in shadow) or 1 (all samples not in shadow), the fragment is excluded from further processing for soft shadows. If the average value is between 0.0 and 1.0, the fragment lies on the shadow boundary, and additional samples are used to compute the soft-shadow value for the penumbra region.

Figure 14 Use of Dynamic Branch to Skip Pixels not Lying on Penumbra Region of the Shadow

Number of samples used for soft-shadow computation	No Branch (ms)	With Dynamic Branch (ms)
2 (**NC**: 2)	5.65	6.53
4 (**NC**: 2, **C**: 2)	8.96	8.07
6 (**NC**: 2, **C**: 4)	12.81	8.96
10 (**NC**: 2, **C**: 8)	16.78	9.36



NC – Non-Conditional texture fetch, with all fragments undergoing shadow checks to determine penumbra region.
C – Conditional texture fetch; only fragments in penumbra region undergo the additional operations.

Figure 14 shows the performance cost for different sample counts used for soft-shadow computation with and without a dynamic branch. Note that with dynamic branch the shader runs in per-instance mode with additional gradient instructions (`ddx`, `ddy`) to compute the derivatives for the shadow texture reads in the shader. The additional cost for this case is thus due to the increase in instructions because of both the branch and the derivative calculation.

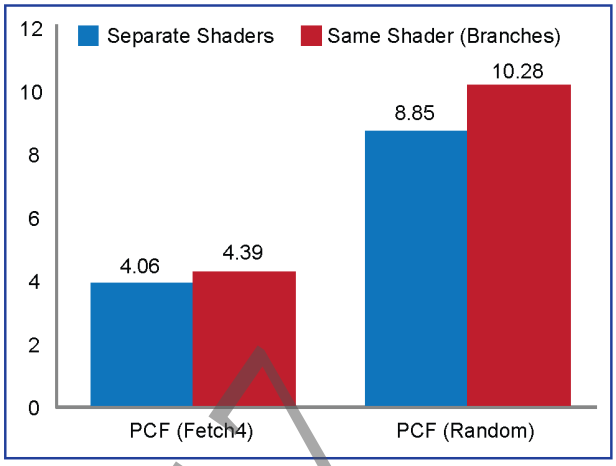
As you see for the case with only 2 NC, using a dynamic branch adds to the overhead cost. However, for more than 2 samples the branch overhead balances out and you can see there is a drastic improvement in performance when using the dynamic branch because the expensive computation is performed only for the fragment in the shadow's penumbra region.

Test 4: Comparing cost of multiple effects in single shader vs. multiple shaders

This test involved a comparison of costs for (a) complex effects combined within a single shader using branches versus (b) using the same blocks of code in different shaders switched at runtime. This test used two different blocks of code to show ways of sampling for performing PCF for soft shadows. The function `EffectPCFFetch4()` uses a 4 component projected query that fetches 4 neighboring samples over a 2x2 grid and performs a vector PCF operation, whereas the function `EffectPCFRandomFetch()` manually uses 4 different randomly offset samples to perform PCF.

Figure 15 Compare Performance of Multiple Shaders vs. Combined Effects within Single Shader

```
if(effectCondition)
{
    // Executes nearest PCF Filtering
    EffectPCFFetch4() ;
}
else
{
    // Executes random offset PCF Filtering
    EffectPCFRandomFetch() ;
}
```



In the case of using a single shader, the two effect functions are separated by a static branch (shown in Figure 15) based on the uniform parameter toggled by the user to achieve the desired effect. In the case of multiple shaders, the two effects are implemented as two separate shaders; switching between them occurs at run time by attaching/detaching, with draw call for rendering.

When evaluating the performance cost, you can see that the separate PCF effect with random offsets is slower than the regular PCF using a 2x2 grid, as shown by the blue bars in Figure 15. However, when combined into a single shader using a branch, both of the effects are impacted and the cost is relatively higher because of the overhead related to the branch instructions as shown by the red bars in Figure 15. Thus implementing all the effects as one combined shader and enabling/disabling the desired effects will result in a penalty.

Recommendations

Although static branches run in parallel mode, their usage does have some inefficiency, and the following recommendations are therefore suggested:

- Static branches should be avoided if possible because they cause additional overhead due to their handling of compare and branch instructions.
- Because each data instance will always take the same path, it would be better to split the shader into multiple variations, and to switch during runtime.
- You generally should benchmark thoroughly when deciding whether to put multiple paths into one shader (depending on your scenario), although excluding static branches is most efficient.

For dynamic branches, the following recommendations are suggested:

- Dynamic branches can be useful; in certain circumstances, they can provide benefits by allowing the program to skip expensive code paths, and instead take simpler ones, across many data instances. However, the effectiveness of dynamic branches depends on your rendering scenario; you must set up efficient conditions and organize code properly for the branches to be useful.
- Shaders with dynamic branches run in per-instance mode with a lower occupancy; this means neighboring pixels in a 2×2 quad may go down different paths. This can affect the results of instructions that require neighboring values, such as gradient calculation during texture lookup. You should therefore avoid such instructions until a large number of data instances are expected to be skipped.
- Similar to static branches, the results should be tested and investigated in Razor to benchmark and evaluate the benefits that the dynamic branch can provide.

Loops: Special Branch Case

The PlayStation®Vita GPU shader compiler supports static as well as dynamic loops for both vertex and fragment shaders. It automatically generates an efficient set of instructions depending on the shader code inside the block and the type of the number of iterations for the loop. The compiler also supports pragmas to control the loop code being generated:

```
#pragma loop (unroll: always)    // loop will always be unrolled
#pragma loop (unroll: never)     // loop will never be unrolled
#pragma loop (unroll: default)  // compiler decision to unroll or not
```

Static Loops Using Constant Iterations

When using a constant value for loop iteration, the PlayStation®Vita GPU shader compiler can, by default, unroll the shader code, eliminating overhead due to branches. Additionally, if the code within the loop can be optimized, the shader compiler intelligently schedules the instructions for best resource usage.

Using Variable Loop Iterations

If iteration is set as a uniform or varying, the loop pragmas do not work (issuing warnings) and the loop is never unrolled. This implies lesser optimization opportunities for the loop block in comparison to unrolled loops, potentially resulting in extra cycles. Moreover, for varying or dynamically computed iterations, the shader runs in per-instance mode, which might result in stalls. The loop performance is also highly dependent on the amount of memory access within the loop, similar to texture fetching and uniform data loads.

For loop flow control, two tests were performed.

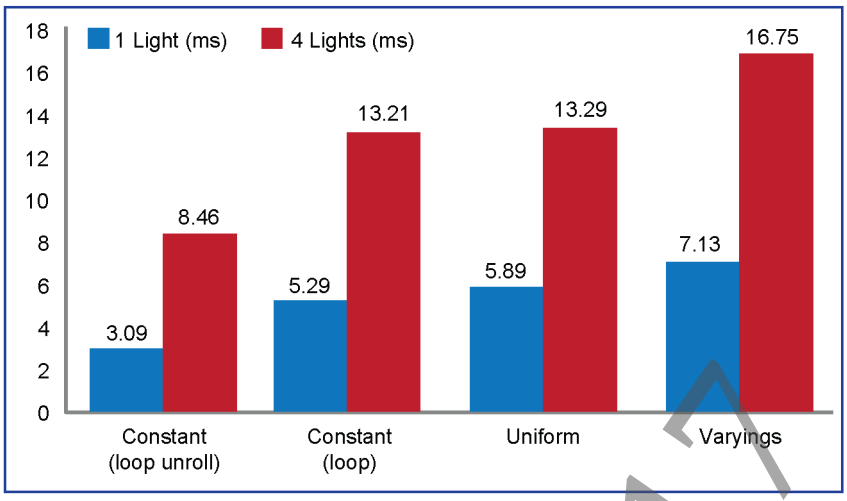
Test 1: Multiple lights with constant, uniform/varying iterations

This test involved a simple lighting scenario with multiple lights, mainly to evaluate loop behavior/performance for different types of iterations. First, the test used constant iteration for loop count, for which the compiler unrolled the shader code as well as included optimized instructions. However, the number of instructions as well as the number of registers that were used increased as the number of lights increased.

To determine loop overhead, `#pragma loop (unroll: never)` was used to disable unrolling. This resulted in additional branch instructions, although the number of registers used was constant and did not vary with the increase in light count (unlike the "flattened case" mentioned above). The runtime performance saw a drop for the branch case, which can be attributed to the increased number of instruction because of the branch.

When using a uniform value for the iterations, the iteration count is fetched and initialized within the secondary program before the main primary program. Statically, however, the compiler adds flow control instructions within the primary program, which also occurs with a constant-iteration loop running in parallel mode.

Figure 16 Performance Comparison When Using Different Iteration Types



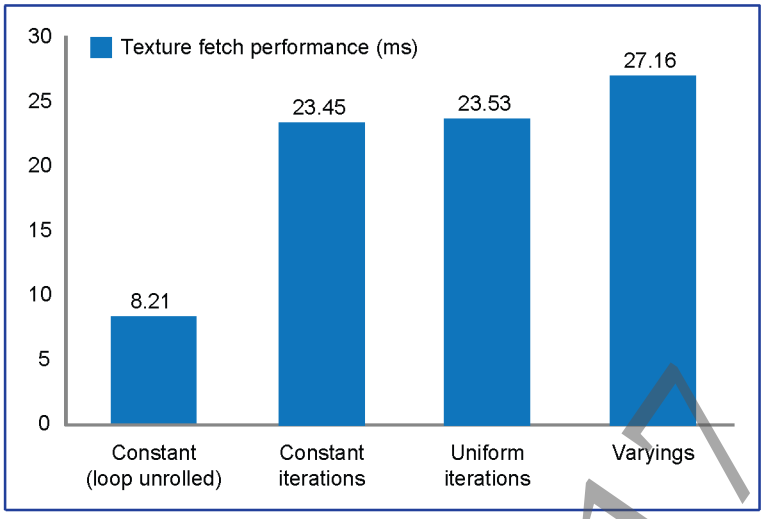
Iteration count →	1		4		
Iteration Type	**Time (ms)**	**Instruction Count**	**Time (ms)**	**Instruction Count**	**Processing Mode**
Constant (loop unroll)	3.09	16 cycles	8.46	37 cycles	Parallel mode
Constant (loop)	5.29	24.75 cycles [Excluding loop: 11.25, In loop: 13.5 cycles]	13.21	24.75 cycles [Excluding loop: 11.25, In loop: 13.5 cycles]	Parallel mode
Uniform	5.89	27.75 cycles [Excluding loop: 14.25, In loop: 13.5 cycles]	13.29	27.75 cycles [Excluding loop: 14.25, In loop: 13.5 cycles]	Parallel mode
Varyings	7.13	31 cycles [Excluding loop: 16, In loop: 15 cycles]	16.75	31 cycles [Excluding loop: 16, In loop: 15 cycles]	Per-instance mode

When using a varying type value for iterations, the iteration count is fetched within the primary program, which adds a few extra cycles for the initialization. The instruction count in the loop within the primary program is the same as for constant iteration with an unrolled loop. However, the shader runs in per-instance mode, and this results in a performance drop in comparison to the usage of either constant or uniform iterations.

Test 2: Loops with texture reads

This test involved a shadow-map scenario to evaluate loop performance when using texture fetches. The loop that was used has texture fetches from a shadow map for computing soft-shadow contribution using a box filter. The shadow-map texture fetches also involve computing gradients using the lookup values from neighboring fragments. When using constant iterations (with unroll disabled) and uniform iterations, the gradient computation is implicitly done because the shader runs in parallel mode for both cases, with similar performance.

Figure 17 Performance Comparison When Using Texture Fetch Within a Loop



Constant (loop unroll)	Constant iterations	Uniform iterations	Varyings
8.21	**23.45**	**23.53**	**27.16**
1 non dependent, 4 unconditional	1 non-dependent, 1 conditional	1 non-dependent, 1 conditional	1 non-dependent, 1 conditional
SP: 6 cycles, **PP**: 28 cycles	**SP**: 2 cycles, **PP**: 28 (1 loop: 21 cycles)	**SP**: 4 cycles, **PP**: 31 (1 loop: 21 cycles)	**SP**: 2 cycles, **PP**: 33 (1 loop: 22 cycles)

SP: Secondary Program, **PP:** Primary Program. Loop count = 4

But within the dynamic loop, the shader runs in per-instance mode and thus additional instructions for gradient computation are added in the shader code, using extra USSE cycles. Therefore, performance is impacted most severely when using dependent texture reads within a dynamic loop.

Also note that when setting `#pragmas` to always unroll, it usually results in an increase in register usage with each loop iteration and can result in shader spilling, causing a drastic performance drop.

Recommendations

Using loops can not only change the number of instructions but also the mode of execution, which can drastically affect the shader performance depending on the rendering scenario. Therefore, as with branches, try to avoid using loops in first place. If for some reason you must use them, follow these guidelines:

- Use constant iterations where possible and let the compiler decide on unrolling. However, when unrolling keep loop count under control, because each iteration will result in an increase in register usage and eventually shader spilling.
- Uniform iterations can be used; however, loops cannot be unrolled so you will incur some overhead cost due to the added branch instructions.
- Varyings or dynamically computed iteration values should be avoided in all cases because they force the shader to run in per-instance mode.
- If possible, non-dependent texture reads should be avoided in any loop because of the similar penalties incurred when fetching within static or dynamic branches.

Register Spilling and Its Effects

Complex shaders can result in excessive register usage per data instance and in some cases may also exceed the allocated limit. On the SGX, the hardware is limited to allocating up to 64 primary attributes and 64 temporary registers. If the compiler's allocations exceed the limit for any of these registers (for example, by allocating a 65th primary attribute), the register will be written to memory instead of using the unified store. This process is called *spilling* and allows a shader to execute, even though it has gone beyond its register allocation limits.

If a shader runs out of registers, you will be notified by the compiler with an error. However, by using pragmas you can either demote the error to a warning that the shader will be slow during execution or you can disable the error notification completely, allowing you to run the shader. The primary attributes and temporary registers for each data instances that are over the hardware limit will be stored in the system memory and fetched when required.

However, because primary-attribute registers are unique to every data instance, this will lead to high memory usage. The allocators used are those that you provide to libgxm. In addition, because these attribute registers are required in memory, data cache thrashing will increase due to additional STA and LDA instructions being inserted into your shaders to handle these stores and loads during runtime. This can significantly affect performance, with a speed difference typically 10 to 20 times worse.

Due to the large overhead, spilling should be used for development purposes only – for example when porting heavy shaders from other platforms that are more likely to spill. To avoid the drastic impact on performance due to spilling, such shaders should be optimized to remove the chance of spilling.

3 Shader Optimizations

This chapter describes details about different shader optimizations that can help reduce the number of execution instructions and improve performance. The methods range from compiler options that generate different shader output, to usage of vectorization techniques.

Compiler Flags

The psp2cgc compiler supports many low-level optimization tricks and can produce code that can help maximize shader performance. The compiler can generate an optimized version of a program based on the compiler flags, resulting in more efficient programs and leading to substantial performance improvements.

fastmath

The `fastmath` option performs basic simplification of expressions, at the potential cost of small deviations in the floating-point results. This allows for constant folding, and also allows the compiler to perform many types of transformations to reduce overall instruction costs. By default "`fastmath`" is enabled in the compiler.

In a very simple example, instead of adding the value A three times, the compiler could simply multiply it by three. A few other examples are included in Table 6:

Table 6 Examples of Expression Optimized by -fastmath Compile Flags

Expression to Evaluate	Optimized Version with fastmath
`A + A + A`	`A * 3`
`A + B - B`	`A`
`A * B / B`	`A`
`A * A + A * B * C + D`	`A * ( A + B * C ) + D`

To further illustrate the first example, if `A` is added three times without `fastmath`, the compiler will perform 2 ADD instructions to calculate the sum. By using the `fastmath` flag, the entire operation can be transformed into a single MUL instruction, saving 1 cycle as shown in Table 7.

Table 7 Use of nofastmath/fastmath Compile Flags for Performing (A+A+A)

-nofastmath	-fastmath

`# 2 cycles`
`# 2 PA registers`
`# 3 SA registers`
	`# 1 cycles - saved 1 cycle`
	`# 2 PA registers`
	`# 4 SA registers`
	`sa2 = {3}`

Primary program:
`0: add.f16 r0.xyzw, pa0.xyzw, pa0.xyzw`
`1: add.f16 pa0.xyzw, r0.xyzw, pa0.xyzw`
	Primary program:
	`0: mul.f16 pa0.xyzw, pa0.xyzw, sa2.zzzz`

fastprecision

When `fastprecision` is used, the compiler will allow the demotion of precision to half floats for an entire expression if parts of the expression are at half precision, or the result of the expression is stored at half precision.

But if a shader's inputs and outputs are full float, this can actually have a negative side effect because the compiler will first need to convert the full-precision floating types to half-precision floating types via packing instructions, so that it can perform the operations at half. In some cases, this may result in an overall increase in instructions and cycle counts.

As an example of using the fast precision, Table 8 shows a program that uses full floating-point arithmetic. Because this fragment program's inputs are coming in as full float, by enabling the compiler flag the shader instruction count actually increased by 2 instructions. So even though the compiler was able to optimize some arithmetic, the additional cycles for converting from 32 bit caused the program to be slower.

Table 8 Disassembly When Using -nofastprecision/-fastprecision Compile Flags

-nofastprecision	-fastprecision
Primary program: pa4 = tex2D<float4>(Texture, TEXCOORD0.xy) 0 : nop 1 : mov.f32 i0.xyz, sa0.xyz 2 : mad.f32 i0.xyz, -pa8.xyz, {1, 1, 1}, i0.xyz 3 : mov.f32 i1.xyz, pa0.xyz 4 : dot.f32 pa0.x, i1.xyz, i1.xyz 5 : rsq.f32 pa0.x, pa0.x 6 : mad.f32 i1.xyz, pa0.xxx, i1.xyz, {0, 0, 0} 7 : dot.f32 pa0.-y, i0.xyz, i0.xyz 8 : rsq.f32 pa0.-y, pa0.-y 9 : mad.f32 i0.xyz, pa0.yyy, i0.xyz, {0, 0, 0} 10 : dot.f32 pa0.x, i1.xyz, i0.xyz 11 : max.f32 i0.xyz, pa0.xxx, sa8.yyy 12 : mad.f32 i0.xyz, pa4.xyz, i0.xyz, {0, 0, 0} 13 : mad.f32 i0.xyz, sa4.xyz, i0.xyz, {0, 0, 0} 14 : mov.f32 i0.---w, pa2.---x **15 : pack.f16.f32 pa0.xyzw, i0.xyzw**	**Secondary program:** 0 : pack.f16.f32 sa4.xyz, sa4.xyz **Primary program:** pa4 = tex2D<half4>(Texture, TEXCOORD0.xy) 0 : nop 1 : mov.f32 i0.xyz, sa0.xyz 2 : mad.f32 i0.xyz, -pa6.xyz, {1, 1, 1}, i0.xyz 3 : mov.f32 i1.xyz, pa0.xyz 4 : dot.f32 pa2.x, i1.xyz, i1.xyz 5 : rsq.f32 pa2.x, pa2.x **6 : pack.f16.f32 pa2.x, pa2.x** **7 : pack.f16.f32 pa0.xyz, i1.xyz** 8 : mul.f16 r0.xyz, pa2.xxx, pa0.xyz **9 : pack.f16.f32 pa6.xyz, i0.xyz** 10 : dot.f32 pa2.-y, i0.xyz, i0.xyz 11 : rsq.f32 pa2.-y, pa2.-y **12 : pack.f16.f32 pa0.x, pa2.y** 13 : mul.f16 pa0.xyz, pa6.xyz, pa0.xxx 14 : dot.f16 pa0.x, r0.xyz0, pa0.xyz1 15 : max.f16 pa0.xyz, pa0.xxx, sa8.zzz 16 : mul.f16 pa0.xyz, pa4.xyz, pa0.xyz 17 : mul.f16 pa0.xyz, pa0.xyz, sa4.xyz

Recommendations

When your shader program's inputs and outputs are already half precision, you can use the fast-precision flag to enable the shader to benefit from optimizations achieved by using half expressions.

However, for most shaders it is recommended to manually select the right precision as needed, and not rely on the compiler, to avoid any accidental packs that can increase the number of instructions.

#pragma position_invariant

Although using the `-fastmath` and `-fastprecision` compiler options can provide some useful optimizations, they can sometimes result in artifacts when calculating output position in a vertex program.

These options can cause the compiler to generate different permutations of code when computing output positions. Although the `-fastmath` option results in some ULPs (unit of least precision) error, `-fastprecision` can result in different LSBs (least significant bits) for output positions. The precision errors that are introduced can lead to visual artifacts.

In such cases, use the `#pragma position_invariant <func>` to force the compiler to generate a position-invariant vertex shader. This will ensure that no error-introducing optimizations are performed for computations affecting the `POSITION` output semantic.

Instructions Implemented in Software

The SGX does not have any efficient implementation support for `normalize()` and `saturate()` standard library calls. Additionally, the use of swizzling is not fully supported, and can result in additional instructions.

Normalize

For `normalize()`, the SGX needs to perform normalization with 3 instructions: one for dot product, one for reciprocal square root, and one for a `MUL`. This will explicitly cost you 3 cycles (unless you can fuse `MUL` with an addition to form a `MAD`), so use this function only when needed.

```
half3 Normal = normalize( Input.WorldNormal.xyz );
```

Primary program:

```
 0:    dot.f16     pa0.w, pa0.xyz, pa0.xyz
 1:    rsq.f16     pa0.w, pa0.w
 2:    mul.f16     pa0.xyz, pa0.www, pa0.xyz
```

Saturate

For `saturate()`, the compiler will generate a `MIN` and `MAX` instruction to perform clamping between 0 and 1.

But instead of calling `saturate()`, you can actually further reduce this to one cycle, because you can manually clamp with either `MIN` or `MAX` if you already know that the value will be clamped within one of the two boundaries.

```
half3 NdotL = saturate( dot(Normal, LightDir) );
```

Primary program:

```
0:    dot.f16     pa0.x, pa0.xyz, r0.xyz
1:    min.f16     pa0.x, pa0.x, sa12.x
2:    max.f16     pa0.x, pa0.x, sa10.x
...
```

Swizzle

Swizzling may result in additional instructions because the hardware does not fully support arbitrary swizzling in some operands and instructions.

Inspect your disassembly carefully because this issue could be adding additional instructions, especially when porting shaders from other platforms. Performing texture fetch with swizzled texture coordinates offers a simple example of this issue, as shown below:

```
half4 texCol = tex2D(texMap, texCoord.zw);
```

Primary program:

```
0:    mov.f16     pa0.xy, pa0.zw
1:    tex2D.f16   pa0, pa0.xy, sa4
```

Unused Channels

Do not write unnecessary values into unused channels, for example via setting alpha to 1 in a color. On the SGX, these operations require extra cycles because the compiler will actually generate instructions to pack in the values even though they are not used.

Encoding Functions as a Texture

You can also use texture lookups to remove instructions completely from your shaders.

Because the SGX has the advantage of non-dependent textures, you can move complex functions outside of the shader by encoding them into textures and can then use texture query format for format conversion when doing texture lookup. The functions can be encoded either as 1D texture for one-variable functions, 2D texture for two-variable functions, or as cube maps for the functions whose inputs are direction vectors , and then, based on the coordinates, the function value can be retrieved per fragment at run time. Some uses include pre-calculated specular highlights encoded in a cube-map texture, normalization cube maps, among others.

Vectorization

The USSE is a SIMD (single instruction, multiple data) processor, so it is necessary to optimize shaders in ways to take advantage of vectorization. This makes it possible to use the same instructions on multiple items in parallel, saving more time than scalar processing.

Good candidates for vectorization include lighting calculations that compute scalar values, or areas of code that contain the same sequence of operations with scalar data.

Table 9 Disassembly Showing Reduction in Instructions When Using Vector Processing Instead of Scalar Processing

Scalar Processing	Vector Processing

`half NdotL0 = saturate(dot(N,L0));`
`half NdotH0 = saturate(dot(N,H0));`
`half NdotL1 = saturate(dot(N,L1));`
`half NdotH1 = saturate(dot(N,H1));`
	`half4 dots = saturate(half4( dot(N,L0),`
	`dot(N,H0),`
	`dot(N,L1),`
	`dot(N,H1)));`

`5 : dot.f16 pa0.x, r0.xyz0, sa18.xyz1`
`6 : min.f16 pa0.x, pa0.x, sa12.x`
`7 : max.f16 pa2.x, pa0.x, sa10.x`
`8 : dot.f16 pa0.x, r0.xyz0, sa16.xyz1`
`9 : min.f16 pa0.x, pa0.x, sa12.x`
`10: max.f16 pa0.x, pa0.x, sa10.x`
`11: dot.f16 pa0.x, r0.xyz0, sa14.xyz1`
`12: min.f16 pa0.x, pa0.x, sa12.x`
`13: max.f16 pa0.x, pa0.x, sa10.x`
`14: dot.f16 pa0.x, r0.xyz0, sa2.xyz1`
`15: min.f16 pa0.x, pa0.x, sa12.x`
`16: max.f16 pa0.x, pa0.x, sa10.x`
	`5 : dot.f16 pa0.x, r0.xyz0, sa16.xyz1`
	`6 : dot.f16 pa0.-y, r0.xyz0, sa18.xyz1`
	`7 : dot.f16 pa0.--z, r0.xyz0, sa14.xyz1`
	`8 : dot.f16 pa0.---w, r0.xyz0, sa2.xyz1`
	`9 : min.f16 pa0.xyzw, pa0.xyzw, sa12.xxxx`
	`10: max.f16 pa0.xyzw, pa0.xyzw, sa10.xxxx`

Table 9 shows a lighting calculation that outputs a series of scalar types. As discussed previously, the `saturate()` function is not efficient on the SGX, and you could end up with a disassembly like the one shown in the left column of Table 9 when clamping your lighting computation. In this case the compiler would generate 8 instructions for the `saturate()` calls alone and 4 instructions for the dot product: 12 cycles in total.

But instead of calling the `saturate()` function for every individual scalar value, you could actually optimize this to a single `saturate()` function call for a vector, and reduce the cycle count from 12 to 6, doubling the throughput. Also, because these values are normalized, you only need to clamp in one direction, and could change the saturate call to a single max function, reducing the cycle count to 5 cycles.

When vectorizing, you may want to avoid the following scalar instructions: exponent, logarithmic, reciprocal, and reciprocal square root. These can only be computed in scalar form.

You may also want to avoid certain standard functions in Cg that can only compute single components. Using such a function, like `pow()`, will generate code for `LOG`, `MUL`, and `EXP`, and because the call is scalar only, it will become more difficult to vectorize. Instead you many want to consider other ways to work around them, in order to keep as much code in a vectorized form as possible.

Note that the ideal vector width is usually 64 bits, so `half4/float2` operations are most efficiently performed as vector operations. Vector `float4` operations are possible, but can suffer from register pressure/bandwidth limitations.

Variable Precision

Variable precision is an important way to help reduce the number of instructions in your shaders, but it is also possible that it could increase the number of instructions if not used correctly.

Precision in Vertex Shaders

At the machine code level, all vertex shader outputs are required to be at full floating-point precision. While it is legal to declare an output as a half type, the compiler will silently convert the value to float before passing it on to the Tiling Engine. This means that though inputs and shader computations can happen in half-precision floats, the compiler will insert pack instructions to promote half values back to float before passing the result back to the hardware.

In this case, inserting half-precision floats in your vertex shader may cost additional cycles, but in other cases the gains can outweigh the additional pack instructions, so it is important to profile your shaders with psp2shaderperf.

Full float	Half float (legacy)	Half float (preferred)

`float2 Texcoord : TEXCOORD0;`
`float3 WorldNormal : TEXCOORD1;`
`float3 WorldPos : TEXCOORD2;`
	`half2 Texcoord : TEXCOORD0;`
	`half3 WorldNormal : TEXCOORD1;`
	`half3 WorldPos : TEXCOORD2;`
		`float2 Texcoord : TEXCOORD0_HALF;`
		`float3 WorldNormal : TEXCOORD1_HALF;`
		`float3 WorldPos : TEXCOORD2_HALF;`

If you need to leave your vertex shaders in full float, use the `TEXCOORD#_HALF` flag, which will not have any effect on performance for both vertex and fragment shading, but will allow you to reduce parameter buffer usage thus reducing the chance of a partial render.

Precision in Fragment Shaders

For fragment shaders these points should be considered:

- Fragment processing usually outweights vertex processing, so superfluous packs are relatively more expensive.
- Use half-precision floats in your shaders, and use full floats only if it is really necessary.
- If you need format conversion, work out the exact points where you want these to happen and verify with psp2shaderperf that no other pack instructions remain.
- You can get interpolants in either full floats or half precision, by declaring them in these formats in the shader. The conversion is done by UITR.

4 psp2shaderperf Tool

The psp2shaderperf tool allows disassembly of both primary and secondary programs, as well as providing static performance data on pre-patched shaders to help you optimize shaders. It provides high-level analysis and performance warnings. For runtime-patched shaders, use the Razor capture.

To generate the disassembly and the symbols corresponding to a shader, the psp2shaderperf tool uses the `-disasm` and `-symbols` options. The generated output contains the following information:

- High-level shader analysis
- List of Symbols
- Disassembly

High-Level Shader Analysis and Symbols

The psp2shaderperf output provides an array of metrics and information that you can use to measure the performance of your shaders as well as an overview of USSE register resources. Items provided include the following (the numbers below are keyed to Figure 18):

(1) Details about estimated cycles, along with cycles in a loop (if any), where basically a single USSE pipe executes 1 instruction per cycle.

(2) The USSE thread mode in which the shader is executed. Loops over uniforms will be executed in parallel mode, unless other branches require per-instance mode.

(3) A count of unified-store register allocations, for attributes and temps.

(4) The number of textures sampled; whether they are non-dependent or dependent; and whether the dependent fetch is in conditional or unconditional code path.

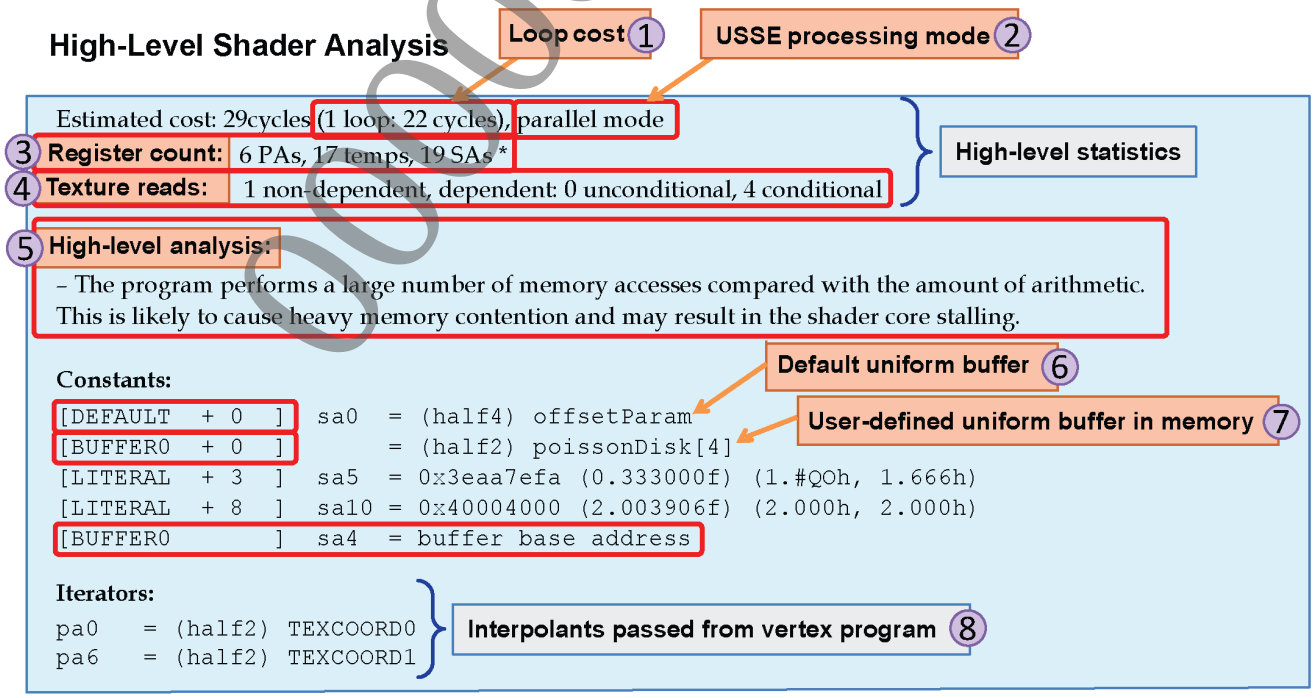
(5) A high-level analysis stating any causes that can cause a performance drop.

(6) A list of all constants or uniforms set by the user and used in the shader that are assigned to secondary-attribute registers (this includes the default uniform buffers).

(7) User-defined uniform buffers, literals, and base addresses for memory-based buffers.

(8) All interpolants used by the fragment program that are stored as primary attributes.

Figure 18 High-Level Shader Analysis and List of Symbols Provided by the psp2shaderperf Tool



Disassembly

The second part of the psp2shaderperf output is the disassembly for both secondary and primary programs, listing all related instructions that get executed. Some of the items provided by the disassembly include (the numbers below are keyed to Figure 19):

(1) Each instruction, post-fixed with a precision type that indicates the type of precision for the arithmetic (usually `.f32` or `.f16`).

(2) Pack instructions (whenever a format conversion is required, pack instructions will be inserted; for example in Figure 19 pack is inserted for converting from 16 bits to 32 bits).

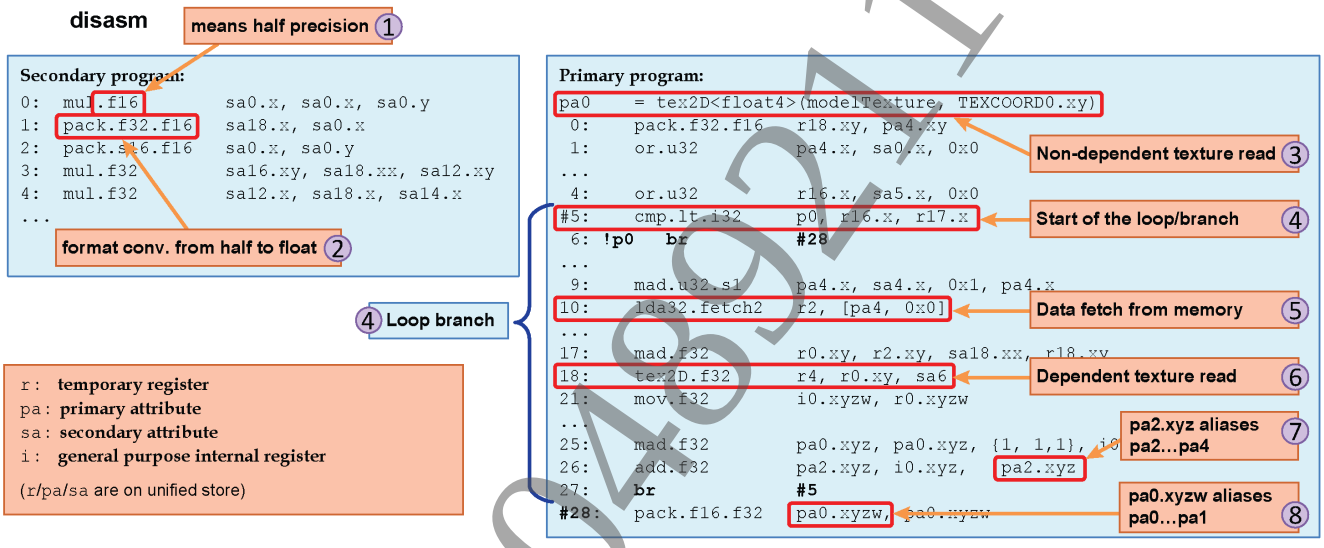
(3) Any non-dependent texture fetches going directly into the primary attributes.

(4) Any loops or branches in the code (shown within a block, as in the example).

(5) Any data fetches directly from the memory (`"lda"` instructions) and writes to memory due to spilling (`"sta"` instructions).

(6) Any dependent texture fetches occur within the primary program.

Figure 19 Disassembly for Both Secondary and Primary Programs



When `.f32`and `.f16` values are sourced, different aliasing rules apply (the numbers below are keyed to Figure 19):

(7) For `.f32`, each 32-bit float occupies a register. So `pa2.`*`xyz`* will alias to `pa2`, `pa3`, and `pa4`.

(8) For `.f16`, the aliasing rules will change, with two 16-bit values occupying a single register. So the operand `pa0.`*`xyzw`* is aliased to `pa0` and `pa1`, where `pa0` will map to *xy*, and `pa1` will alias to *zw*.

For a detailed overview of the tool, refer to the *psp2shaderperf User's Guide*.

5 Best Practices

This chapter provides a summary of important recommendations, discussed throughout this document, that can significantly alleviate bottlenecks associated with the USSE execution pipeline, lead to efficient shader optimizations, and above all improve overall shader performance. In your programs, you should use psp2shaderperf and Razor GPU traces to determine any bottlenecks; you should also adhere to the following suggestions (if applicable).

Convert Dependent to Non-dependent Texture

Convert dependent reads to non-dependent texture reads, for good performance. Wherever possible the texture-coordinate modifications should be moved to the vertex program and unmodified texture coordinates should be used in the shader. Avoid storing texture coordinates in swizzled form, such as is done with *.zy* components for the `tex2D()` function. Also, avoid including non-dependent texture fetches within branches because these are demoted to dependent texture fetches.

Vectorize Your Code

Vectorize your code everywhere possible, to improve throughput. Cutting cycles will make a difference. When considering what to vectorize, look for opportunities in your shader code where you can perform the same sequence of operations for different sets of scalar data. Also check for any scalar functions that are used multiple times. If you compute the result for some lanes differently from others, use masks to only compute specific lanes within a vector.

Use Register-Based Uniform Buffers When Possible

Use register-based uniform buffers when possible, to ensure that data is pre-fetched by the PDS prior to shader execution. To achieve optimal efficiency, if you have small set of uniforms that are frequently used, you should usually pre-fetch them into registers. However, if you have large buffers that are sparsely used, avoid loading the entire buffer through the PDS; instead, load them straight from the memory, which is more efficient.

Limit the Number of Interpolants

Limit the number of interpolants, and reduce precision when ITR activity is high. Where possible you should pack the interpolants in a vertex program to reduce the count. Do not use interpolants if they are unnecessary, because this will reduce the efficiency if the fragment shader is not large enough to hide the latency. Also, if you use reduced precision in your fragment program, do not generate any unnecessary instructions when unpacking interpolants in your program; this may outweigh any other gains.

Avoid Branches or Loops

Static branches should be avoided. Although the shader runs in parallel mode, the additional overhead associated with the branch instruction will result in lower performance. In general, you should prefer splitting into multiple shaders instead of combining into a single shader.

Only use dynamic branches to skip expensive code paths across many data instances. Avoid instructions within the branch that could drastically impact performance when running in per-instance mode, such as texture fetches that perform gradient computations.

For both static and dynamic branches, use Razor to ensure good performance; otherwise avoid branching completely.