# psp2shaderperf User's Guide



PlayStation®Vita Programmer Tool Runtime Library Release 3.50

# Table of Contents

# About This Document

This document describes psp2shaderperf, a tool provided as part of the SDK for performing static analysis of compiled shader programs.

## Conventions

The typographical conventions used in this guide are explained in this section.

### Notes

Additional advice or related information is presented as a 'note' surrounded by a box. For example:

**Note:** This is an additional note.

### Text

File names, source code and command-line text are formatted in a fixed-width font. For example:

```
host_tools\bin\psp2shaderperf.exe
```

## Errata

Any updates or amendments to this guide can be found in the release notes that accompany the release.

# 1 Introduction

## GXP Format

GXP is a file format for storing shader in a compiled form for use by libgxm.

Content of the GXP format:

- Compiled code to be run on USSE.
- Symbols for uniforms and input attributes.
- General information describing the shader execution model (e.g., parallel vs. per-instance).
- Internals information used by libgxm.

Tools are provided to inspect a GXP file:

- psp2cgnm allows to extract symbol information.
- psp2shaderperf allows to disassemble the compiled USSE code for the purpose of static performance analysis.

# 2 Using psp2shaderperf

psp2shaderperf is supplied as a command line utility for analyzing shaders, inspecting shader symbols and viewing the disassembled code.

The command line tool can be found inside the PlayStation®Vita SDK in the following folder:

```
host_tools\bin\psp2shaderperf.exe
```

## Command-line Usage

The basic syntax for viewing disassembly and symbols contained in a `.gxp` file is:

```
psp2shaderperf –disasm -symbols <input gxp file>
```

## Options

Table 1 describes the options that can be used with psp2shaderperf tool.

**Table 1 psp2shaderperf Options**

| Option | Description |
|---|---|
| `-disasm` | Prints the disassembly of the shader code to standard output. |
| `-symbols` | Prints symbols information to standard output. |
| `-stats` | Prints instruction statistics to standard output. |
| `-sdb <name>` | Name the SDB file associated with the input GXP file. |
| `-cachedir <dir>` | Name the directory where SDB files are stored. |

## Examples

The following command line:

```
psp2shaderperf -disasm -symbols shader.gxp
```

Generates the following output:

```
Estimated cost: 39 cycles, parallel mode
Register count: 13 PAs, 2 temps, 22 SAs *
Texture reads: 1 non-dependent, dependent: 2 unconditional,
0 conditional

High level analysis:
No warnings.

* Please refer to the Razor documentation for details regarding the
meaning of these numbers. Decreasing the number of registers used will
not necessarily increase performance

Constants:
[DEFAULT  + 0  ] sa0   =  (half3) ambientColor
[DEFAULT  + 2  ] sa2   =  (half3) lightColor
[DEFAULT  + 4  ] sa4   =  (float1) specularPower
[LITERAL  + 2  ] sa7   =  0x38003800 (0.000031f)  (0.500h, 0.500h)
[LITERAL  + 11 ] sa16  =  0xbc00bc00 (-0.007857f) (-1.000h, -1.000h)
[LITERAL  + 12 ] sa17  =  0x40004000 (2.003906f)  (2.000h, 2.000h)
[LITERAL  + 13 ] sa18  =  0x3ccccccd (0.025000f)  (-19.203h, 1.199h)
[LITERAL  + 14 ] sa19  =  0xbf800000 (-1.000000f) (0.000h, -1.875h)
[LITERAL  + 15 ] sa20  =  0x40000000 (2.000000f)  (0.000h, 2.000h)
[LITERAL  + 16 ] sa21  =  0x0        (0.000000f)  (0.000h, 0.000h)
```







Document serial number: 000004892117

```
Samplers:
TEXUNIT0  = albedoTex
TEXUNIT1  = heightTex
TEXUNIT2  = normalTex

Iterators:
pa0   = (float2) TEXCOORD0
pa6   = (float3) TEXCOORD1
pa10  = (float3) TEXCOORD2

Primary program:
pa2   = tex2D<float4>(heightTex, TEXCOORD0.xy)
 0 :     nop
 1 :     mov.f32        i0.xyz, pa6.xyz
 2 :     dot.f32        pa2.-y, i0.xyz, i0.xyz
 3 :     rsq.f32        pa8.-y, pa2.-y
 4 :     mad.f32        pa2.x, pa2.x, sa20.x, sa18.y
 5 :     mad.f32        pa4.xy, pa8.yy, i0.xy, {0, 0}
 6 :     mul.f32        pa2.xy, pa4.xy, pa2.xx
 7 :     mad.f32        pa0.xy, pa2.xy, sa18.xx, pa0.xy
 8 :     tex2D.f16      pa2, pa0.xy, sa8
 9 :     tex2D.f16      r0, pa0.xy, sa12
 10:     mov.f32        i0.xyz, pa10.xyz
 11:     dot.f32        pa0.x, i0.xyz, i0.xyz
 12:     rsq.f32        pa0.x, pa0.x
 13:     mad.f32        i0.xyz, pa0.xxx, i0.xyz, {0, 0, 0}
 14:     mov.f32        i1.x, pa8.y
 15:     mad.f32        i1.xyz, pa6.xyz, i1.xxx, i0.xyz
 16:     pack.f16.f32   pa4.xyz, i0.xyz
 17:     dot.f32        pa0.x, i1.xyz, i1.xyz
 18:     rsq.f32        pa0.x, pa0.x
 19:     mad.f32        i1.xyz, pa0.xxx, i1.xyz, {0, 0, 0}
 20:     mad.f16        pa2.xyzw, pa2.xyzw, sa16.zzzz, sa16.xxxx
 21:     dot.f16        pa0.x, pa2.xyz0, pa2.xyz1
 22:     rsq.f16        pa0.x, pa0.x
 23:     mul.f16        pa2.xyz, pa0.xxx, pa2.xyz
 24:     dot.f16        pa0.x, pa2.xyz0, pa4.xyz1
 25:     max.f16        pa4.xyz, pa0.xxx, sa20.zzz
 26:     pack.f16.f32   pa0.xyz, i1.xyz
 27:     dot.f16        pa0.x, pa2.xyz0, pa0.xyz1
 28:     max.f16        pa0.x, pa0.x, sa20.z
 29:     pack.f32.f16   pa0.x, pa0.x
 30:     log.f32        pa0.x, abs(pa0.x)
 31:     mul.f32        pa0.x, pa0.x, sa4.x
 32:     exp.f32        pa0.x, pa0.x
 33:     pack.f16.f32   pa0.x, pa0.x
 34:     mul.f16        pa0.xyz, pa0.xxx, sa6.zzz
 35:     mad.f16        pa2.xyzw, r0.xyzw, pa4.xyzw, pa0.xyzw
 36:     mul.f16        pa0.xyz, r0.xyz, sa0.xyz
 37:     mad.f16        pa0.xyzw, pa2.xyzw, sa2.xyzw, pa0.xyzw
 38:     mul.f16        pa0.xyzw, pa0.xyz1, {0.0039063, 0.0039063,
                        0.0039063, 0.0039063}
```

# 3 Reading psp2shaderperf Output

## High Level Shader Analysis

The first bit of information printed by psp2shaderperf is a set of high level statistics:

**Table 2 High Level Statistics**

| Name | Description |
|---|---|
| `cycle count` | An approximation of the cost in cycles of the analyzed shader on its longest path. |
| `Loop cycle count` | An approximation of the cost in cycles for each detected loop. |
| `Execution Mode` | Whether the shader will execute in parallel or per-instance mode. |
| `PA count` | Number of primary attribute registers used by the shader. |
| `Temp count` | Number of temporary registers used by the shader. |
| `SA count` | Number of secondary attribute registers used by the shader. |
| `Non-dep texture reads` | Number of non dependent texture reads. |
| `Unconditional texture reads` | Number of unconditionally executed texture reads. |
| `Conditional texture reads` | Number of conditionally executed texture reads. |

In addition psp2shaderperf will print any result obtained from a high level analysis of the code, currently one or more of these warnings can be printed:

**Table 3 High Level Analysis**

| Name | Description |
|---|---|
| `spilling` | The shader uses too many registers so it is forced to spill registers to memory, this will have a non negligible impact on performance and memory usage. |
| `indexable temps` | The shader uses a dynamic index to access a temporary array, this will cause the array to be relocated from register to memory effectively causing the same performance issue encountered with spilling shaders. |
| `complex derivatives` | The shader uses ddx/ddy inside a complex conditional block, this condition forces the compiler to insert sync-points between executing instances to make sure ddx/ddy results can be computed correctly. |
| `memory bound shader` | The shader contains a large number of memory transaction and not many ALU operations to balance with. The shader is likely to cause memory contention. |
| `misaligned output writes` | The shader has some of its output misaligned, this could cause redundant MOVs to be inserted. |

## Instruction Statistics

When psp2shaderperf is executed with the `-stats` command line it will print a set of statistics related to instruction usage in the primary program of the shader.

These statistics include:

- Number of ALU instructions
- Number of memory access instructions
- Number of texture queries
- Number of floating point instructions

- Number of integer instructions
- Number of type conversion instructions (pack)
- Number of MOV instructions
- Number of NOP instructions

## Symbols

psp2shaderperf will print a list of symbols contained in the GXP file, together with their register assignment, this print-out is aimed at simplifying the disassembly interpretation. For a more structured symbol extraction tool, psp2cgnm should be used.

Following is a brief explanation of the syntax used for symbols printing:

**Table 4 Symbols Syntax**

| Name | Description |
|---|---|
| `constant` | `[BUFFER + OFFSET] sa# = (TYPE) NAME` |
| `sampler` | `TEXUNIT# = NAME` |
| `iterator` | `pa# = (TYPE) TEXCOORD#` |
| `non-dependent texture read` | `pa# = tex2D<TYPE>(NAME, TEXCOORD#.xy)` |
| `input vertex attribute` | `pa# = (TYPE) NAME` |

## Disassembly Format

psp2shaderperf decodes USSE microcode into a form very similar to ARB/HLSL assembly, programmer used to read ARB assembly should find reading USSE disassembly quite familiar.

Some notable differences between ARB assembly and psp2shaderperf assembly syntax:

- The data format for operands and result of each instruction is usually explicit and indicated in a suffix after the opcode mnemonic; examples `.f32`, `.f16`.
- `Exp`/`Log`/`Rcp`/`Rsq` are all scalar instructions although the result can be replicated into multiple channels of the result.
- Integer operations are scalar instructions with the exception of multiply-add and shift operations, since they both support a 16-bit two-way operation variant.
- Floating point instructions have variants for half and full precision
- Type conversions are explicit. Conversions are achieved using pack instructions, where the suffix of the instruction indicates the destination and source format. In some cases it is possible to convert f16 values to f32 values without incurring the penalty of an additional instruction. See GPI Registers.
- Not all instructions support arbitrary swizzles on their operands. When a swizzle for a particular operand is not supported, the swizzle is usually hoisted into a previous `mov` instruction.
- The f16 variant of a `mad`, `mov`, or `movc` instruction, where the destination is the unified store, does not support an arbitrary write-mask. Only an `.xy`, `.zw` or `.xyzw` write-mask is supported. The compiler will generally insert either a `mul` by 1 or an `add` with 0 to simulate an unsupported write-mask.

When possible, the compiler will generate a form of co-issue instruction where the USSE can execute two specific instructions simultaneously. In this case psp2shaderperf will display two consecutive instructions, but they will be prefixed with the same cycle number. Additionally the second instruction is prefixed with a '+' symbol, for example:

```
10: rsq.f16 i0.x, i0.x
10: +mul.f16 pa4.xy, sa7.x, i1.x
```

## Mixed Mode Disassembly

psp2shaderperf will try to load an SDB file that matches the input GXP file in order to display source line associations in mixed mode.

If the SDB file cannot be automatically located it is possible to specify the filename or the location of it using the `-sdb` or `-cachedir` command line flags.

If an SDB file can be successfully loaded the disassembly will be interleaved with source code as in the following example:

```
0: cmp.le.f32   p1, pa0.y, sa4.x      // alpha_test_f.cg:13 discard albedo.g > 0.8f;
1: !p1 kill
2: nop
3: pack.f16.f32 pa0.xyz, pa0.xyz      // alpha_test_f.cg:11 float4 albedo =
                                         tex2D(albedoTex, vTexCoord);
4: mul.f16      pa0.xyzw, pa0.xyzh, {1, 1, 1, 1}
                                      // alpha_test_f.cg:9  uniform sampler2D
                                         (albedoTex) : COLOR
```

## Register Model

Although all registers (apart from one special case) are allocated from Unified Store, they are presented to the shader program in different banks. These will be labelled `pa`, sa, `r`, and `o` in the disassembly.

### Primary Attribute Registers

Primary attribute registers are labelled `pa` and are filled with the inputs of the program before the shader is run. However, they are read/write and may be used as temporary registers later in the program. Each primary attribute register is 32 bits in size.

In fragment programs that use MSAA, primary attribute registers are shared between all samples of the pixel. When a shader uses the programmable blending functionality, the compiler splits the program into two distinct phases: the first phase (pixel phase) runs once for all samples while the second phase (sample rate phase) runs for each sample. Since primary attribute regisers are shared by all samples, during the sample rate phase they become read-only registers. The compiler will generally use primary registers to pass parameters from the pixel phase to the sample rate phase.

### Secondary Attribute Registers

Secondary attribute registers are labelled `sa` and contain the uniform data and texture control words for the program. They are read-only in the primary program, but read/write in the secondary program, which may perform operations that only involve uniform data. Each secondary attribute register is 32 bits in size, and a maximum of 128 registers can be used as secondary attribute registers.

### Output Registers

Output registers are labelled `o` and are considered the outputs of the program. In vertex programs, these registers store the output attributes of the vertex and may be used as temporary registers during the program. In fragment programs, only `o0` and `o1` are available; these registers contain the current color value and can be considered as read/write. Each output register is 32 bits in size.

The shader compiler generates fragment shaders that write into the output registers only when the `__nativecolor` modifier is used, otherwise the color result is left in primary attributes; in this case, libgxm inserts the move of the result into output registers as part of the blending and/or writemask code that is appended at runtime by the shader patcher.

### Temporary Registers

Temporary registers are labelled `r` and are generic temporary registers allocated by the compiler to hold intermediate results of the shader computation. Each temporary register is 32 bits in size.







Document serial number: 000004892117

000004892117

When the program is split into two phases (usually when programmable blending is used), the content of temporary registers is lost across phase boundaries. Values that need to be preserved across phases will be allocated to primary attribute registers by the compiler.

### GPI Registers

GPI (general purpose internal) registers are labelled `i` and are a special case. These registers are not allocated from unified store and are 128 bits in size. There are 3 GPI registers, labelled `i0`, `i1` and `i2`.

GPI registers always store 32-bit floating point data, even when used in instructions that operate on 16-bit floating point data. This can sometimes be used to get free format conversion, for example in this instruction:

```
mul.f16 i0, pa0.xyzw, sa0.xyzw
```

In this case the multiplication is of type half an `pa0` and `sa0` contain half data, but when the result is written into `i0` it is automatically converted to a 32-bit floating point value.

GPI registers are generally used when float4 operations are required, since many instructions are limited to 64 bits per operand from unified store, which only allows for float2 operations. Since GPI registers do not use unified store, they always provide 128 bits per operand, allowing full float4 operations. GPI registers are not required for half4 operations, since this fits into the 64 bits per operand limitation of unified store.

### Instruction Syntax

Although most registers are 32 bits in size, many instructions write four components. For example with the following snippet of code:

```
mov.f32 i0, pa0.xyzw
```

The second argument covers more than one register, in this case it covers `pa0`, `pa1`, `pa2` and `pa3`, this if because of the swizzle and because data type is full precision 32-bit floating point.

In this second example:

```
mov.f16 r0, pa0.xyzw
```

The second argument covers only two primary attributes (`pa0` and `pa1`), this is due to data type being half precision 16-bit floating point so each register contains two coefficients of the vector.

### Non Dependent Texture Queries

psp2shaderperf will print *Non Dependent Texture Queries* as part of the instruction disassembly using the following format:

```
pa# = SamplerType<format>(sampler, TEXCOORD0.xy)
```

*`SamplerType`* will be either `tex2D` or `texCUBE`, one component textures (`sampler1D`) never generate a non dependent texture query.

The swizzle used on the texture coordinate also indicates the type of texture query, it can be `.xy` for a normal 2D query, `.xyz` for a query from a cube map or `.xyw` for a projected 2D query. Additionally the `_CENTROID` modifier will be added to the texture coordinate name if centroid sampling has to be used.

The *`format`* printed inside the angle bracket will be either `float`, `half` or `????`, the last is used when psp2shaderperf has no information regarding the result format of the texture query.

> **Note:** Even if Non Dependent Texture Queries are printed as part of the instruction disassembly, they do not account for instructions as the query is issued by the PDS, not by USSE itself.

# 4 DLL Interfaces

Dynamic Link Library (DLL) version of psp2shaderperf is provided.

## Files

The DLL versions of the tools are provided as the following sets of files (Table 5).

**Table 5 DLL Interface Files**

| File Name with Relative Path | Description |
|---|---|
| `sdk\host_tools\include\cgc\psp2shaderperf.h`<br>`sdk\host_tools\lib\psp2shaderperf.dll`<br>`sdk\host_tools\lib\psp2shaderperf.lib` | psp2shaderperf header file, DLL and import library for 32-bit Windows |
| `sdk\host_tools\include\cgc\psp2shaderperf.h`<br>`sdk\host_tools\lib.x64\psp2shaderperf.dll`<br>`sdk\host_tools\lib.x64\psp2shaderperf.lib` | psp2shaderperf header file, DLL and import library for 64-bit Windows |

The core function `scePsp2ShaderPerf` expect as input a pointer to a `ScePsp2ShaderPerfOptions` structure which has been first initialized using the function `scePsp2ShaderPerfInitializeOptions` and later filled with user options and a pointer to an in memory representation of a GXP file.

The output of `scePsp2ShaderPerf` is a pointer to a `scePsp2ShaderPerfOutput` structure which has been filled with the result of the analysis. Once data has been extracted from a `scePsp2ShaderPerfOutput` the structure should be destroyed by calling the function `scePsp2ShaderPerfDestroyOutput`.

## Sample Code

```
ScePsp2ShaderPerfOptions psp2ShaderPerfOptions;
ScePsp2ShaderPerfOutput const *psp2ShaderPerfOutput;

/* prepare psp2shaderperf options */
scePsp2ShaderPerfInitializeOptions(&psp2ShaderPerfOptions);
psp2ShaderPerfOptions.gxpData = userSuppliedGxpData;
psp2ShaderPerfOptions.disasm = 1;

/* run psp2shaderperf */
psp2ShaderPerfOutput = scePsp2ShaderPerf(&psp2ShaderPerfOptions);

/* ensure we have a result */
assert(psp2ShaderPerfOutput != NULL &&
       psp2ShaderPerfOutput->disassembly != NULL);

/* print the disassembly of the shader */
printf("%s\n", psp2ShaderPerfOutput->disassembly);

/* destroy shaderperf output */
scePsp2ShaderPerfDestroyOutput(psp2ShaderPerfOutput);
```