User Guide

AMD GPU Performance API

1 Introduction

The GPU Performance API (GPUPerfAPI, or GPA) is a powerful tool to help analyze the performance and execution characteristics of applications using the GPU.

This API:

- Supports DirectX10, DirectX11, and OpenGL on ATI Radeon™ 2000 series and newer graphics cards.
- Supports OpenCL on ATI Radeon™ 4000 series and newer graphics cards.
- Supports Microsoft Windows as a static library or as a dynamically loaded library.
- Supports Linux as a shared object library:
  - Targeting Ubuntu 10.04
  - OpenCL and OpenGL only
- Provides derived counters based on raw HW performance counters.
- Manages memory automatically – no allocations required.
- Requires ATI Catalyst™ driver 10.1 or later.

2 Usage

For DirectX10 and DirectX11, your application must run with administrator privileges, or UAC must be turned off so the counters can be accessed in the drivers.

2.1 Static Library

Using the static library option is great for applications that support a single API. To integrate GPUPerfAPI as a static library:

1. Include the header file `GPUPerfAPI.h`.
2. Link your application with the static library for your chosen API.
3. Use the functions directly to profile your application.

2.2 Dynamically Loaded Library

For applications that support multiple APIs, this approach ensures that you can easily profile each API.

1. Include the header file `GPUPerfAPI.h`.
2. Include the header file `GPUPerfAPIFunctionTypes.h`.
3. Define instances of each of the function types.
4. Call `LoadLibrary(...)` on the `GPUPerfAPI.dll` for your chosen API.
5. For each function in GPUPerfAPI, call `GetProcAddress(...)`.
6. Use the functions to profile your application.

2.3 Shared Object Library

For a shared-object library,

1. Include the header file `GPUPerfAPI.h`.
2. Include the header file `GPUPerfAPIFunctionTypes.h`.
3. Define instances of each of the function types.
4. Call `dlopen(...)` on `libGPUPerfAPICL.so` or `libGPUPerfAPIGL.so`.
5. For each function in GPUPerfAPI, call `dlsym(...)`.
6. Use the functions to profile your application.

2.4 Initializing the GPUPerfAPI

The API must be initialized before the rendering context or device is created so that the driver can be prepared for accessing the counters.

```
GPA_Status GPA_Initialize( );
```

After the context or device is created, the counters can be opened on the given context.

```
GPA_Status GPA_OpenContext ( void* context );
```

The supplied context must either point to a DirectX device, be the handle to the OpenGL rendering context, or the OpenCL command queue handle. The return value indicates whether or not the current hardware is supported by GPUPerfAPI. See Section 6, “API Functions,” page 14, for more information on individual entry points and return values.

2.5 Obtaining Available Counters

To determine the number of available counters, call:

```
GPA_Status GPA_GetNumCounters( gpa_uint32* count );
```

To retrieve the name of a counter, call:

```
GPA_Status GPA_GetCounterName( gpa_uint32 index, const char** name );
```

To retrieve the index for a given counter name, call:

```
GPA_Status GPA_GetCounterIndex( const char* counter, gpa_uint32* index );
```

2.6 Retrieving Information About the Counters

To retrieve a description about a given counter, call:

```
GPA_Status GPA_GetCounterDescription( gpa_uint32 index, const char** description );
```

To retrieve the data type of the counter (gpa_float32, gpa_float64, gpa_uint32, gpa_uint64), call:

```
GPA_Status GPA_GetCounterDataType( gpa_uint32 index, GPA_Type* dataType );
```

To retrieve the usage type of the counter (percentage, byte, milliseconds, ratio, items, etc.), call:

```
GPA_Status GPA_GetCounterUsageType( gpa_uint32 index, GPA_Usage_Type usageType );
```

2.7 Enabling Counters

By default, all counters are disabled and must be explicitly enabled. To enable a counter given its index, call:

```
GPA_Status GPA_EnableCounter( gpa_uint32 index );
```

To enable a counter given its name, call:

```
GPA_Status GPA_EnableCounterStr( const char* counter );
```

To enable all available counters, call:

```
GPA_Status GPA_EnableAllCounters();
```

2.8 Disabling Counters

Disabling counters can reduce data collection time. To disable a counter given its index, call:

```
GPA_Status GPA_DisableCounter( gpa_uint32 index );
```

To disable a counter given its name, call:

```
GPA_Status GPA_DisableCounterStr( const char* counter );
```

To disable all enabled counters, call:

```
GPA_Status GPA_DisableAllCounters();
```

2.9 Multi-Pass Profiling

The set of counters that can be sampled concurrently is dependent on the hardware and the API. Not all counters can be collected at once (in a single pass). A *pass* is defined as a set of operations to be profiled. To query the number of passes required to collect the current set of enabled counters, call:

```
GPA_Status GPA_GetPassCount( gpa_uint32* numPasses );
```

If multiple passes are required, the set of operations executed in the first pass must be repeated for each additional pass. If it is impossible or impractical to repeat the operations to be profiled,

select a counter set requiring only a single pass. For sets requiring more than one pass, results are available only after all passes are complete.

2.10 Sampling Counters

A profile with a given set of counters is called a *Session*. The counter selection cannot change within a session. GPUPerfAPI generates a unique ID for each session, which later is used to query the results of the session. Sessions are identified by begin/end blocks:

```
GPA_Status GPA_BeginSession( gpa_uint32* sessionID );

GPA_Status GPA_EndSession();
```

More than one *pass* may be required, depending on the set of enabled counters. A single session must contain all the passes needed to complete the counter collection. Each pass is also identified by begin/end blocks:

```
GPA_Status GPA_BeginPass();

GPA_Status GPA_EndPass();
```

Each pass, and each session, can contain one or more *samples*. Each sample is a data point for which a set of counter results is returned. All enabled counters are collected within begin/end blocks:

```
GPA_Status GPA_BeginSample( gpa_uint32 sampleID );

GPA_Status GPA_EndSample();
```

Each sample must have a unique identifier within the pass so that the results of the individual sample can be retrieved. If multiple passes are required, use the same identifier for the first sample of each pass; each additional sample must use its unique identifier, thus relating the same sample from each pass.

The following example collects a set of counters for two data points:

```
BeginSession
    BeginPass
        BeginSample( 1 )
            <Operations for data point 1>
        EndSample
        BeginSample( 2 )
            <Operations for data point 2>
        EndSample
    EndPass
EndSession
```

If multiple passes are required:

```
BeginSession
    BeginPass
        BeginSample( 1 )
            <Operations for data point 1>
        EndSample
        BeginSample( 2 )
            <Operations for data point 2>
        EndSample
    EndPass
    BeginPass
        BeginSample( 1 )
            <Identical operations for data point 1>
        EndSample
        BeginSample( 2 )
            <Identical operations for data point 2>
        EndSample
    EndPass
EndSession
```

NOTE: The GPUPerfAPI uses the OpenGL `GL_EXT_timer_query` / `GL_ARB_timer_query` extensions to access the GPUTime counter. These extensions ensure that only one `GL_TIME_ELAPSED` query can be active at any time. A query cannot be generated when other query types are active. For this reason, GPUPerfAPI automatically starts and stops existing queries, as needed, to ensure that the GPUTime measurements are accurate. However, active queries may return invalid results if calls to BeginSample / EndSample are between the `glBeginQuery` and `glEndQuery` API calls.

2.11 Counter Results

Results for a session can be retrieved after `EndSession` has been called and before the counters are closed. The unique sessionID provided by GPUPerfAPI can be used to query if the session is available, without stalling the pipeline to wait for the results:

```
GPA_Status GPA_IsSessionReady( bool* readyResult, gpa_uint32 sessionID );
```

Similarly, the sampleID that was provided at each `BeginSample` call can be used to check if individual sample results are available without stalling the pipeline:

```
GPA_Status GPA_IsSampleReady( bool* readyResult, gpa_uint32 sessionID, gpa_uint32
sampleID );
```

Once the results are available, the following calls can be used to retrieve the results. These are blocking calls, so if you are continuously collecting data, it is important to call these as few times as possible to avoid stalls and overhead.

```
GPA_Status GPA_GetSampleUInt32( gpa_uint32 sessionID, gpa_uint32 sampleID,
gpa_uint32 counterID, gpa_uint32* result );

GPA_Status GPA_GetSampleUInt64( gpa_uint32 sessionID, gpa_uint32 sampleID,
gpa_uint32 counterID, gpa_uint64* result );

GPA_Status GPA_GetSampleFloat32( gpa_uint32 sessionID, gpa_uint32 sampleID,
gpa_uint32 counterID, gpa_float32* result );

GPA_Status GPA_GetSampleFloat64( gpa_uint32 sessionID, gpa_uint32 sampleID,
gpa_uint32 counterID, gpa_float64* result );
```

2.12 Result Buffering

The GPUPerfAPI buffers an API-dependent number of sessions (at least four). When more sessions are sampled, the oldest session results are replaced by new ones. Usually, this is not an issue, because the availability of results is checked regularly by your application. Ensure that your application checks the results more frequently than the number of buffered session. This prevents previous sessions from becoming unavailable. If a session is unavailable, `GPA_STATUS_ERROR_SESSION_NOT_FOUND` is returned.

2.13 Closing GPUPerfAPI

To stop the currently selected context from using the counters, call:

```
GPA_Status GPA_CloseContext();
```

After your application has released all rendering contexts or devices, GPUPerfAPI must disable the counters so that performance of other applications is not affected. To do so, call:

```
GPU_Status_GPA_Destroy();
```

3 Example Code

This sample shows the code for:

- Initializing the counters.
- Sampling all the counters for two draw calls every frame.
- Writing out the results to a file when they become available.
- Shutting down the counters.

3.1 Startup

Open the counter system on the current Direct3D device, and enable all available counters. If using OpenGL, pass the handle to the GL context into the `OpenContext` function; for OpenCL, the command queue handle should be supplied.

```
GPA_Initialize();
D3D10CreateDeviceAndSwapChain( . . . &g_pd3dDevice );
GPA_OpenContext( g_pd3dDevice );
GPA_EnableAllCounters();
```

3.2 Render Loop

At the start of the application's rendering loop, begin a new session, and begin the GPUPerfAPI pass loop to ensure that all the counters are queried. Sample one or more API calls before ending the pass loop and ending the session. After the session results are available, save the data to disk for later analysis.

```
static gpa_uint32 currentWaitSessionID = 1;

gpa_uint32 sessionID;
GPA_BeginSession( &sessionID );

gpa_uint32 numRequiredPasses;
GPA_GetPassCount( &numRequiredPasses );

for ( gpa_uint32 i = 0; i < numRequiredPasses; i++ )

{

    GPA_BeginPass();

    GPA_BeginSample( 0 );
        <API function call>
    GPA_EndSample();

    GPA_BeginSample( 1 );
        <API function call>
    GPA_EndSample();

    GPA_EndPass();

    GPA_EndSession();

}

bool readyResult = false;
if ( sessionID != currentWaitSessionID )
{
    GPA_Status sessionStatus;
    sessionStatus = GPA_IsSessionReady( &readyResult,
                                                currentWaitSessionID );

    while ( sessionStatus == GPA_STATUS_ERROR_SESSION_NOT_FOUND )
    {
        // skipping a session which got overwritten
        currentWaitSessionID++;
        sessionStatus = GPA_IsSessionReady( &readyResult,
                                            currentWaitSessionID );
    }
}

if ( readyResult )
{
    WriteSession( currentWaitSessionID,
                  "c:\\PublicCounterResults.csv" );
    currentWaitSessionID++;
}
```

3.3 On Exit

Ensure that the counter system is closed before the application exits.

```
GPA_CloseContext();
g_pd3dDevice->Release();
GPA_Destroy();
```

4 Counter Groups

The counters exposed through GPU Performance API are organized into groups to help provide clarity and organization to all the available data.

It is recommended you initially profile with counters from the Timing group to determine whether the profiled calls are worth optimizing (based on GPUTime value), and which parts of the pipeline

are performing the most work. Note that because the GPU is highly parallelized, various parts of the pipeline can be active at the same time; thus, the “Busy” counters probably will sum over 100 percent. After identifying one or more stages to investigate further, enable the corresponding counter groups for more information on the stage and whether or not potential optimizations exist.

Group	Counters
Timing	DepthStencilTestBusy GPUTime GPUBusy InterpBusy PrimitiveAssemblyBusy ShaderBusy ShaderBusyCS ShaderBusyDS ShaderBusyGS ShaderBusyHS ShaderBusyPS ShaderBusyVS TessellatorBusy TexUnitBusy
VertexShader	VertexMemFetched VertexMemFetchedCost VSALUBusy VSALUEfficiency VSALUInstCount VSALUTexRatio VSTexBusy VSTexInstCount VSVerticesIn
HullShader[1]	HSALUBusy HSALUEfficiency HSALUInstCount HSALUTexRatio HSTexBusy HSTexInstCount HSPatches
PixelShader	PSALUBusy PSALUEfficiency PSALUInstCount PSALUTexRatio PSExportStalls PSPixelsIn PSPixelsOut PSTexBusy PSTexInstCount

Group	Counters
GeometryShader	GSALUBusy GSALUEfficiency GSALUInstCount GSALUTexRatio GSExportPct GSPrimsIn GSTexBusy GSTexInstCount GSVerticesOut
PrimitiveAssembly	ClippedPrims CulledPrims PAPixelsPerTriangle PAStalledOnRasterizer PrimitivesIn
DomainShader[1]	DSALUBusy DSALUEfficiency DSALUInstCount DSALUTexRatio DSTexBusy DSTexInstCount DSVerticesIn
ComputeShader[1]	CSALUBusy CSALUFetchRatio CSALUInsts CSALUPacking CSALUStalledByLDS CSCacheHit CSCompletePath CSFastPath CSFetchInsts CSLDSBankConflict CSLDSFetchInsts CSLDSWriteInsts CSPathUtilization CSTexBusy CSThreads CSWavefronts CSWriteInsts

Group	Counters
TextureUnit	TexAveAnisotropy TexCacheStalled TexCostOfFiltering TexelFetchCount TexMemBytesRead TexMissRate TexTriFilteringPct TexVolFilteringPct
TextureFormat	Pct64SlowTexels Pct128SlowTexels PctCompressedTexels PctDepthTexels PctInterlacedTexels PctTex1D PctTex1Darray PctTex2D PctTex2Darray PctTex2DMSAA PctTex2DMSAAArray PctTex3D PctTexCube PctTexCubeArray PctUncompressedTexels PctVertex64SlowTexels PctVertex128SlowTexels PctVertexTexels
General[3]	ALUBusy ALUFetchRatio ALUInsts ALUPacking FetchInsts LDSFetchInsts LDSWriteInsts Wavefronts WriteInsts

Group	Counters
DepthAndStencil	HiZReject HiZTrivialAccept PostZSamplesFailingS PostZSamplesFailingZ PostZSamplesPassing PreZSamplesFailingS PreZSamplesFailingZ PreZSamplesPassing ZUnitStalled
ColorBuffer[2]	CBMemRead CBMemWritten CBSlowPixelPct
GlobalMemory[3]	CompletePath FastPath FetchSize FetchUnitBusy FetchUnitStalled CacheHit PathUtilization WriteUnitStalled
LocalMemory[3]	ALUStalledByLDS LDSBankConflict

1. Available only for ATI Radeon™ HD 5000 series graphic cards.
2. Available only on ATI Radeon™ HD 4000 and 5000 series graphics cards.
3. Exposed only by the OpenCL version of the GPU Performance API.

5 Counter Descriptions

The GPU Performance API supports many hardware counters and attempts to maintain the same set of counters across all supported graphics APIs and all supported hardware generations. In some cases, this is not possible because either features are not available in certain APIs or the hardware evolves through the generations. The following table lists all the supported counters, along with a brief description that can be queried through the API. To clearly define the set of counters, they have been separated into sections based on the which APIs contain the counters and the hardware version on which they are available.

Counter	Description
Common to DX and GL on All Supported Graphics Cards	
ClippedPrims	The number of primitives that required one or more clipping operations due to intersecting the view volume or user clip planes.
CulledPrims	The number of culled primitives. Typical reasons include scissor, the primitive having zero area, and back or front face culling.
DepthStencilTestBusy	Percentage of GPUTime spent performing depth and stencil tests.
GPUBusy	Percentage of time GPU was busy
GPUTime	Time, in milliseconds, this API call took to execute on the GPU. Does not include time that draw calls are processed in parallel.
GSALUBusy	The percentage of GPUTime ALU instructions are processed by the GS.
GSALUEfficiency	ALU vector packing efficiency. Values below 70 percent indicate that ALU dependency chains may prevent full use of the processor.
GSALUInstCount	Average number of ALU instructions executed in GS. Affected by the flow control.
GSALUTexRatio	The ratio of ALU to texture instructions in the GS. This can be tuned appropriately to match the target hardware.
GSExportPct	The percentage of GS work that is related to exporting primitives.
GSPrimsIn	The number of primitives passed into the GS.
GSTexBusy	The percentage of GPUTime texture instructions are processed by the GS.
GSTexInstCount	Average number of texture instructions executed in GS. Affected by the flow control.
GSVerticesOut	The number of vertices output by the GS.
HiZReject	Percentage of tiles that are rejected by HiZ.
HiZTrivialAccept	Percentage of tiles that can be accepted by HiZ without doing per-pixel Z tests.
PAStalledOnRasterizer	Percentage of GPUTime that primitive assembly waits for rasterization to be ready to accept data. This roughly indicates the percentage of time the pipeline is bottlenecked by pixel operations.
Pct128SlowTexels	Percentage of texture fetches from a 128-bit texture (slow path). There also are 128-bit formats that take a fast path; they are included in PctUncompressedTexels.
PctCompressedTexels	Percentage of texture fetches from compressed textures.
PctDepthTexels	Percentage of texture fetches from depth textures.
PctInterlacedTexels	Percentage of texture fetches from interlaced textures.
PctTex1D	Percentage of texture fetches from a 1D texture.
PctTex1DArray	Percentage of texture fetches from a 1D texture array.
PctTex2D	Percentage of texture fetches from a 2D texture.
PctTex2DArray	Percentage of texture fetches from a 2D texture array.
PctTex2DMSAA	Percentage of texture fetches from a 2D MSAA texture.
PctTex2DMSAAArray	Percentage of texture fetches from a 2D MSAA texture array.
PctTex3D	Percentage of texture fetches from a 3D texture.
PctTexCube	Percentage of texture fetches from a cube map.
PctUncompressedTexels	Percentage of texture fetches from uncompressed textures. Does not include depth or interlaced textures.
PostZSamplesFailingS	Number of samples tested for Z after shading and failed stencil test.
PostZSamplesFailingZ	Number of samples tested for Z after shading and failed Z test.
PostZSamplesPassing	Number of samples tested for Z after shading and passed.
PreZSamplesFailingS	Number of samples tested for Z before shading and failed stencil test.

Counter	Description
PreZSamplesFailingZ	Number of samples tested for Z before shading and failed Z test.
PreZSamplesPassing	Number of samples tested for Z before shading and passed.
PrimitiveAssemblyBusy	Percentage of GPUTime that primitive assembly (clipping and culling) is busy. High values may be caused by having many small primitives; mid to low values may indicate pixel shader or output buffer bottleneck.
PrimitivesIn	The number of primitives received by the hardware.
PSALUBusy	The percentage of GPUTime ALU instructions are processed by the PS.
PSALUEfficiency	ALU vector packing efficiency. Values below 70 percent indicate that ALU dependency chains may prevent full use of the processor.
PSALUInstCount	Average number of ALU instructions executed in PS. Affected by the flow control.
PSALUTexRatio	The ratio of ALU to texture instructions in the PS. This can be tuned appropriately to match the target hardware.
PSExportStalls	Percentage of GPUTime that PS output is stalled. Should be zero for PS or further upstream limited cases; if not zero, indicates a bottleneck in late z testing or in the color buffer.
PSPixelsIn	The number of pixels processed by the PS. Does not count pixels culled out by early z or stencil tests.
PSPixelsOut	The number of pixels exported from shader to color buffers. Does not include killed or alpha-tested pixels. If there are multiple render targets, each receives one export, so this is 2 for 1 pixel written to two RTs.
PSTexBusy	The percentage of GPUTime texture instructions are processed by the PS.
PSTexInstCount	Average number of texture instructions executed in the PS. Affected by the flow control.
ShaderBusy	The percentage of GPUTime that the shader unit is busy.
ShaderBusyGS	The percentage of work done by shader units for GS.
ShaderBusyPS	The percentage of work done by shader units for PS.
ShaderBusyVS	The percentage of work done by shader units for VS.
TexAveAnisotropy	The average degree (between 1 and 16) of anisotropy applied. The anisotropic filtering algorithm only applies samples where they are required (there are no extra anisotropic samples if the view vector is perpendicular to the surface), so this can be much lower than the requested anisotropy.
TexCacheStalled	Percentage of GPUTime the texture cache is stalled. Try reducing the number of textures or reducing the number of bits per pixel (use compressed textures), if possible.
TexCostOfFiltering	The effective cost of all texture filtering. Percentage indicating the cost relative to all bilinear filtering. Should always be greater than, or equal to, 100 percent. Significantly higher values indicate heavy usage of trilinear or anisotropic filtering.
TexelFetchCount	The total number of texels fetched. This includes all shader types, and any extra fetches caused by trilinear filtering, anisotropic filtering, color formats, and volume textures.
TexMemBytesRead	Texture memory read in bytes.
TexMissRate	Texture cache miss rate (bytes/texel). A normal value for mipmapped textures on typical scenes is approximately (texture_bpp / 4). For 1:1 mapping, it is texture_bpp.
TexTriFilteringPct	Percentage of pixels that received trilinear filtering. Note that not all pixels for which trilinear filtering is enabled receive it (for example, if the texture is magnified).
TexUnitBusy	Percentage of GPUTime the texture unit is active. This is measured with all extra fetches and any cache or memory effects taken into account.
TexVolFilteringPct	Percentage of pixels that received volume filtering.
VertexMemFetched	Number of bytes read from memory due to vertex cache miss.
VSALUBusy	The percentage of GPUTime ALU instructions are processed by the VS.
VSALUEfficiency	ALU vector packing efficiency. Values below 70 percent indicate that ALU dependency chains may prevent full use of the processor.
VSALUInstCount	Average number of ALU instructions executed in the VS. Affected by the flow control.

Counter	Description
VSALUTexRatio	The ratio of ALU to texture instructions in the VS. This can be tuned appropriately to match the target hardware.
VSTexBusy	The percentage of GPUTime texture instructions are processed by the VS.
VSTexInstCount	Average number of texture instructions executed in VS. Affected by the flow control.
VSVerticesIn	The number of vertices processed by the VS.
ZUnitStalled	Percentage of GPUTime the depth buffer spends waiting for the color buffer to be ready to accept data. High figures here indicate a bottleneck in color buffer operations.
Common to DX and GL on 2000, 3000, and 4000 Series Graphics Cards	
InterpBusy	Percentage of GPUTime that the interpolator is busy.
Common to DX and GL on 4000 and 5000 Series Graphics Cards	
CBMemRead	Number of bytes read from the color buffer.
CBMemWritten	Number of bytes written to the color buffer.
CBSlowPixelPct	Percentage of pixels written to the color buffer using a half-rate or quarter-rate format.
Pct64SlowTexels	Percentage of texture fetches from a 64-bit texture (slow path). There are also 64-bit formats that take a fast path; they are included in PctUncompressedTexels.
PctTexCubeArray	Percentage of texture fetches from a cube map array.
PctVertex64SlowTexels	Percentage of texture fetches from a 64-bit vertex texture (slow path). There are also 64-bit formats that take a fast path; they are included in PctVertexTexels.
PctVertex128SlowTexels	Percentage of texture fetches from a 128-bit vertex texture (slow path). There are also 128-bit formats that take a fast path; they are included in PctVertexTexels.
PctVertexTexels	Percentage of texture fetches from vertex textures.
VertexMemFetchedCost	The percentage of GPUTime that is spent fetching from vertex memory due to cache miss. To reduce this, Improve vertex reuse or use smaller vertex formats.
Common to DX and GL on 5000 Series Graphics Cards	
PAPixelsPerTriangle	The ratio of rasterized pixels to the number of triangles after culling. This does not account for triangles generated due to clipping.
DSALUBusy	The percentage of GPUTime ALU instructions are processed by the DS.
DSALUEfficiency	ALU vector packing efficiency. Values below 70 percent indicate that ALU dependency chains may prevent full use of the processor.
DSALUInstCount	Average number of ALU instructions executed in the DS. Affected by flow control.
DSALUTexRatio	The ratio of ALU to texture instructions. This can be tuned to match the target hardware.
DSTexBusy	The percentage of GPUTime texture instructions are processed by the DS.
DSTexInstCount	Average number of texture instructions executed in DS. Affected by the flow control.
DSVerticesIn	The number of vertices processed by the DS.
HSALUBusy	The percentage of GPUTime ALU instructions processed by the HS.
HSALUEfficiency	ALU vector packing efficiency. Values below 70 percent indicate that ALU dependency chains may prevent full use of the processor.
HSALUInstCount	Average number of ALU instructions executed in the HS. Affected by the flow control.
HSALUTexRatio	The ratio of ALU to texture instructions. This can be tuned to match the target hardware.
HSTexBusy	The percentage of GPUTime texture instructions are processed by the HS.
HSTexInstCount	Average number of texture instructions executed in HS. Affected by the flow control.
HSPatches	The number of patches processed by the HS.
ShaderBusyDS	Percentage of work done by shader units for DS.
ShaderBusyHS	Percentage of work done by shader units for HS.

Counter	Description
TessellatorBusy	Percentage of time the tessellation engine is busy.
Specific to DX11 on 5000 Series Graphics Cards	
CSALUStalledByLDS	The percentage of GPUTime ALU units are stalled by the LDS input queue being full or the output queue is not ready. If there are LDS bank conflicts, reduce it.; otherwise, try reducing the number of LDS accesses.
CSALUBusy	The percentage of GPU Time ALU instructions are processed by the CS.
CSALUPacking	ALU vector packing efficiency. Values below 70 percent indicate that ALU dependency chains may prevent full use of the processor.
CSALUInsts	The number of ALU instructions executed in the CS. Affected by the flow control.
CSALUFetchRatio	The ratio of ALU to fetch instructions. This can be tuned to match the target hardware.
CSCompletePath	The total bytes read and written through the CompletePath. This includes extra bytes needed for addressing, atomics, etc. This number indicates a big performance impact (higher number equals lower performance). Reduce it by removing atomics and non 32-bit types, or move them into a second shader.
CSCacheHit	The percentage of fetches from the global memory that hit the L1 cache.
CSLDSBankConflict	The percentage of GPUTime the LDS is stalled by bank conflicts.
CSLDSBankConflictAccess	The percentage of LDS accesses that caused a bank conflict.
CSFastPath	The total bytes written through the FastPath (no atomics, 32-bit type only). This includes extra bytes needed for addressing.
CSFetchInsts	The average number of fetch instructions executed in the CS per execution (affected by flow control).
CSLDSFetchInsts	The average number of fetch instructions from the local memory executed per thread (affected by flow control).
CSLDSWriteInsts	The average number of write instructions to the local memory executed per thread (affected by flow control).
CSPathUtilization	The percentage of bytes read and written through the FastPath or CompletePath compared to the total number of bytes transferred over the bus. To increase the path utilization, remove atomics and non 32-bit types.
CSTexBusy	The percentage of GPUTime texture instructions are processed by the CS.
CSThreads	The number of CS threads processed by the hardware.
CSWavefronts	The total number of wavefronts used for the CS.
CSWriteInsts	The average number of write instructions executed in CS per execution (affected by flow control).
ShaderBusyCS	Percentage work done by shader units for CS.
Specific to OpenCL on 4000 and 5000 Series Graphics Cards	
ALUBusy	The percentage of GPUTime ALU instructions are processed.
ALUFetchRatio	The ratio of ALU to fetch instructions. If the number of fetch instructions is zero, then one is used instead.
ALUInsts	The average number of ALU instructions executed per work-item (affected by flow control).
ALUPacking	The ALU vector packing efficiency (in percentage). This value indicates how well the Shader Compiler packs the scalar or vector ALU in your kernel to the 5-way VLIW instructions. Values below 70 percent indicate that ALU dependency chains may be preventing full use of the processor.
FetchInsts	The averaged fetch instructions (from global memory) executed per work-item (affected by flow control).
FetchSize	The total kilobytes fetched from the video memory. This is measured with all extra fetches and any cache or memory effects taken into account.
FetchUnitBusy	The percentage of time the Fetch unit is active relative to GPUTime. This is measured with all extra fetches and any cache or memory effects taken into account.

Counter	Description
FetchUnitStalled	The percentage of time the Fetch unit is stalled relative to GPUTime. If possible, try to reduce the number of fetches or reducing the amount per fetch.
CacheHit	The percentage of fetches from the video memory that hit the data cache. Value range is 0% (no hit) to 100% (optimal).
Wavefronts	Total wavefronts.
WriteInsts	The average number of write instructions (to global memory) executed per work-item (affected by flow control).
WriteUnitStalled	The percentage of time the Write unit is stalled relative to GPUTime.
Specific to OpenCL on 5000 Series Graphics Cards	
ALUStalledByLDS	The percentage of GPUTime the ALU units are stalled because the LDS input queue is full or the output queue is not ready. If there are LDS bank conflicts, reduce it; otherwise, try reducing the number of LDS accesses if possible.
CompletePath	The total kilobytes written to the global memory through the CompletePath, which supports atomics and sub-32 bit types (byte, short). This number includes bytes for load, store, and atomics operations on the buffer. This number can indicate a large impact on performance (higher number equals lower performance). If possible, remove the usage of this Path by moving atomics to the local memory or partition the kernel.
FastPath	The total kilobytes written to the global memory through the FastPath. This is an optimized path in the hardware that only supports basic operations: no atomics or sub-32 bit types.
LDSBankConflict	The percentage of GPUTime the LDS is stalled by bank conflicts.
LDSFetchInsts	The average number of fetch instructions from the local memory executed per work-item (affected by flow control).
LDSWriteInsts	The average number of write instructions to the local memory executed per work-item (affected by flow control).
PathUtilization	The percentage of bytes written through the FastPath or CompletePath compared to the total number of bytes transferred over the bus. To increase the path utilization, use FastPath.

6 API Functions

Begin Sampling Pass

Syntax **GPALIB_DECL GPA_Status GPA_BeginPass ( )**

Description It is expected that a sequence of repeatable operations exist between `BeginPass` and `EndPass` calls. If this is not the case, activate only counters that execute in a single pass. The number of required passes can be determined by enabling a set of counters, then calling `GPA_GetPassCount`. Loop the operations inside the `BeginPass`/`EndPass` calls over `GPA_GetPassCount` result number of times.

Returns `GPA_STATUS_ERROR_COUNTERS_NOT_OPEN`: `GPA_OpenContext` must be called before this call to initialize the counters.

`GPA_STATUS_ERROR_SAMPLING_NOT_STARTED`: `GPA_BeginSession` must be called before this call to initialize the profiling session.

`GPA_STATUS_ERROR_PASS_ALREADY_STARTED`: `GPA_EndPass` must be called to finish the previous pass before a new pass can be started.

`GPA_STATUS_OK`: On success.

Begin a Sample Using the Enabled Counters

Syntax

```
GPALIB_DECL GPA_Status GPA_BeginSample ( gpa_uint32 sampleID )
```

Description

Multiple samples can be done inside a BeginSession/EndSession sequence. Each sample computes the values of the counters between BeginSample and EndSample. To identify each sample, the user must provide a unique sampleID as a parameter to this function. The number must be unique within the same BeginSession/EndSession sequence. The BeginSample must be followed by a call to EndSample before BeginSample is called again.

Parameters

sampleID Any integer, unique within the BeginSession/EndSession sequence, used to retrieve the sample results.

Returns

GPA_STATUS_ERROR_COUNTERS_NOT_OPEN: GPA_OpenContext must be called before this call to initialize the counters.

GPA_STATUS_ERROR_SAMPLING_NOT_STARTED: GPA_BeginSession must be called before this call to initialize the profiling session.

GPA_STATUS_ERROR_PASS_NOT_STARTED: GPA_BeginPass must be called before this call to mark the start of a profile pass.

GPA_STATUS_ERROR_SAMPLE_ALREADY_STARTED: GPA_EndSample must be called to finish the previous sample before a new sample can be started.

GPA_STATUS_ERROR_FAILED: The sample could not be started due to an internal error.

GPA_STATUS_OK: On success.

Begin Profile Session with the Currently Enabled Set of Counters

Syntax

```
GPALIB_DECL GPA_Status GPA_BeginSession ( gpa_uint32 * sessionID )
```

Description

This must be called to begin the counter sampling process. A unique sessionID is returned, which later is used to retrieve the counter values. Session identifiers are integers and always start from 1 on a newly opened context. The set of enabled counters cannot be changed inside a BeginSession/EndSession sequence.

Parameters

sessionID The value to be set to the session identifier.

Returns

GPA_STATUS_ERROR_COUNTERS_NOT_OPEN: GPA_OpenContext must be called before this call to initialize the counters.

GPA_STATUS_ERROR_NULL_POINTER: A null pointer was suppled as the sessionID parameter. A reference to a gpa_uint32 value is expected.

GPA_STATUS_ERROR_NO_COUNTERS_ENABLED: No counters were enabled for this session.

GPA_STATUS_ERROR_SAMPLING_ALREADY_STARTED: GPA_EndSession must be called in order to finish the previous session before a new session can be started.

GPA_STATUS_OK: On success.

Close Counters in the Currently Active Context

Syntax **GPALIB_DECL GPA_Status GPA_CloseContext ( )**

Description Counters must be reopened with GPA_OpenContext before using the GPUPerfAPI again.

Returns GPA_STATUS_ERROR_COUNTERS_NOT_OPEN: GPA_OpenContext must be called before this call to initialize the counters.

GPA_STATUS_ERROR_SAMPLING_NOT_ENDED: GPA_EndSession must be called in order to finish the previous session before the counters can be closed.

GPA_STATUS_OK: On success.

Undo any Initialization Needed to Access Counters

Syntax **GPALIB_DECL GPA_Status GPA_Destroy ( )**

Description Calling this function after the rendering context or device has been released is important so that counter availability does not impact the performance of other applications.

Returns GPA_STATUS_FAILED: An internal error occurred.

GPA_STATUS_OK: On success.

Disable All Counters

Syntax **GPALIB_DECL GPA_Status GPA_DisableAllCounters ( )**

Description Subsequent sampling sessions do not provide values for any disabled counters. Initially, all counters are disabled and must be enabled explicitly.

Parameters *sessionID* The value to be set to the session identifier.

Returns GPA_STATUS_ERROR_COUNTERS_NOT_OPEN: GPA_OpenContext must be called before this call to initialize the counters.

GPA_STATUS_ERROR_CANNOT_CHANGE_COUNTERS_WHEN_SAMPLING: Counter cannot be disabled if a session is active.

GPA_STATUS_OK: On success.

Disable a Specified Counter

Syntax

```
GPALIB_DECL GPA_Status GPA_DisableCounter ( gpa_uint32 index )
```

Description

Subsequent sampling sessions do not provide values for any disabled counters. Initially, all counters are disabled and must explicitly be enabled.

Parameters

index	The index of the counter to enable. Must lie between 0 and (GPA_GetNumCounters result - 1), inclusive.

Returns

GPA_STATUS_ERROR_INDEX_OUT_OF_RANGE: The supplied index does not identify an available counter.

GPA_STATUS_ERROR_CANNOT_CHANGE_COUNTERS_WHEN_SAMPLING: Counter cannot be disabled if a session is active.

GPA_STATUS_ERROR_NOT_ENABLED: The supplied index does identify an available counter, but the counter was not previously enabled, so it cannot be disabled.

GPA_STATUS_OK: On success.

Disable a Specified Counter Using the Counter Name (Case Insensitive)

Syntax

```
GPALIB_DECL GPA_Status GPA_DisableCounterStr ( const char * counter )
```

Description

Subsequent sampling sessions do not provide values for any disabled counters. Initially, all counters are disabled and must explicitly be enabled.

Parameters

counter	The name of the counter to disable.

Returns

GPA_STATUS_ERROR_NULL_POINTER: A null pointer was suppled as the counter parameter.

GPA_STATUS_ERROR_CANNOT_CHANGE_COUNTERS_WHEN_SAMPLING: Counter cannot be disabled if a session is active.

GPA_STATUS_ERROR_NOT_FOUND: A counter with the specified name could not be found.

GPA_STATUS_ERROR_NOT_ENABLED: The supplied counter identifies an available counter, but the counter was not previously enabled, so it cannot be disabled.

GPA_STATUS_OK: On success.

Enable All Counters

Syntax

```
GPALIB_DECL GPA_Status GPA_EnableAllCounters ( )
```

Description

Subsequent sampling sessions provide values for all counters. Initially, all counters are disabled and must explicitly be enabled by calling a function that enables them.

Returns

GPA_STATUS_ERROR_COUNTERS_NOT_OPEN: GPA_OpenContext must be called before this call to initialize the counters.

GPA_STATUS_ERROR_CANNOT_CHANGE_COUNTERS_WHEN_SAMPLING: Counter cannot be disabled if a session is active.

GPA_STATUS_OK: On success.

Enable a Specified Counter

Syntax

```
GPALIB_DECL GPA_Status GPA_EnableCounter ( gpa_uint32 index )
```

Description

Subsequent sampling sessions provide values for enabled counters. Initially, all counters are disabled and must explicitly be enabled by calling this function.

Parameters

index The index of the counter to enable. Must lie between 0 and (GPA_GetNumCounters result - 1), inclusive.

Returns

GPA_STATUS_ERROR_COUNTERS_NOT_OPEN: GPA_OpenContext must be called before this call to initialize the counters.

GPA_STATUS_ERROR_INDEX_OUT_OF_RANGE: The supplied index does not identify an available counter.

GPA_STATUS_ERROR_CANNOT_CHANGE_COUNTERS_WHEN_SAMPLING: Counter cannot be disabled if a session is active.

GPA_STATUS_ERROR_ALREADY_ENABLED: The specified counter is already enabled.

GPA_STATUS_OK: On success.

Enable a Specified Counter Using the Counter Name (Case Insensitive)

Syntax

```
GPALIB_DECL GPA_Status GPA_EnableCounterStr ( const char * counter )
```

Description

Subsequent sampling sessions provide values for enabled counters. Initially, all counters are disabled and must explicitly be enabled by calling this function.

Parameters

counter The name of the counter to enable.

Returns

GPA_STATUS_ERROR_COUNTERS_NOT_OPEN: GPA_OpenContext must be called before this call to initialize the counters.

GPA_STATUS_ERROR_NULL_POINTER: A null pointer was suppled as the counter parameter.

GPA_STATUS_ERROR_CANNOT_CHANGE_COUNTERS_WHEN_SAMPLING: Counter cannot be disabled if a session is active.

GPA_STATUS_ERROR_NOT_FOUND: A counter with the specified name could not be found.

GPA_STATUS_ERROR_ALREADY_ENABLED: The specified counter is already enabled.

GPA_STATUS_OK: On success.

End Sampling Pass

Syntax **GPALIB_DECL GPA_Status GPA_EndPass ( )**

Description It is expected that a sequence of repeatable operations exist between BeginPass and EndPass calls. If this is not the case, activate only counters that execute in a single pass. The number of required passes can be determined by enabling a set of counters and then calling GPA_GetPassCount. Loop the operations inside the BeginPass/EndPass calls the number of times specified by the GPA_GetPassCount result. This is necessary to capture all counter values because counter combinations sometimes cannot be captured simultaneously.

Returns GPA_STATUS_ERROR_COUNTERS_NOT_OPEN: GPA_OpenContext must be called before this call to initialize the counters.

GPA_STATUS_ERROR_PASS_NOT_STARTED: GPA_BeginPass must be called to start a pass before a pass can be ended.

GPA_STATUS_ERROR_SAMPLE_NOT_ENDED: GPA_EndSample must be called to finish the last sample before the current pass can be ended.

GPA_STATUS_ERROR_VARIABLE_NUMBER_OF_SAMPLES_IN_PASSES: The current pass does not contain the same number of samples as the previous passes. This can only be returned if the set of enabled counters requires multiple passes.

GPA_STATUS_OK: On success.

End Sampling Using the Enabled Counters

Syntax **GPALIB_DECL GPA_Status GPA_EndSample ( )**

Description BeginSample must be followed by a call to EndSample before BeginSample is called again.

Returns GPA_STATUS_ERROR_COUNTERS_NOT_OPEN: GPA_OpenContext must be called before this call to initialize the counters.

GPA_STATUS_ERROR_SAMPLE_NOT_STARTED: GPA_BeginSample must be called before try to end a sample.

GPA_STATUS_ERROR_FAILED: An internal error occurred while trying to end the current sample.

GPA_STATUS_OK: On success.

End Sampling with the Currently Enabled Set of Counters

Syntax

```
GPALIB_DECL GPA_Status GPA_EndSampling ( )
```

Description

Ends the sampling session so that the counter results can be collected.

Returns

GPA_STATUS_ERROR_COUNTERS_NOT_OPEN: GPA_OpenContext must be called before this call to initialize the counters.

GPA_STATUS_ERROR_SAMPLING_NOT_STARTED: A session must be started before it can be ended.

GPA_STATUS_ERROR_PASS_NOT_ENDED: The current pass must be ended before the session can be ended.

GPA_STATUS_ERROR_NOT_ENOUGH_PASSES: The currently selected set of counter requires additional passes. The session cannot be ended until the right number of passes have been completed.

GPA_STATUS_OK: On success.

Get the Counter Data Type of the Specified Counter

Syntax

```
GPALIB_DECL GPA_Status GPA_GetCounterDataType (gpa_uint32 index,
GPA_Type * counterDataType )
```

Description

Retrieves the data type of the counter at the supplied index.

Parameters

index	The index of the counter. Must lie between 0 and (GPA_GetNumCounters result - 1), inclusive.
counterDataType	The value that holds the description upon successful execution.

Returns

GPA_STATUS_ERROR_COUNTERS_NOT_OPEN: GPA_OpenContext must be called before this call to initialize the counters.

GPA_STATUS_ERROR_INDEX_OUT_OF_RANGE: The supplied index does not identify an available counter.

GPA_STATUS_ERROR_NULL_POINTER: The supplied counterDataType parameter is null. A reference to a GPA_Type variable is expected.

GPA_STATUS_OK: On success

Get Description of the Specified Counter

Syntax

```
GPALIB_DECL GPA_Status GPA_GetCounterDescription ( gpa_uint32 index,
const char ** description )
```

Description

Retrieves a description of the counter at the supplied index.

Parameters

index	The index of the counter to query. Must lie between 0 and (GPA_GetNumCounters result - 1), inclusive.
description	The value that holds the description upon successful execution.

Returns

GPA_STATUS_ERROR_COUNTERS_NOT_OPEN: GPA_OpenContext must be called before this call to initialize the counters.

GPA_STATUS_ERROR_INDEX_OUT_OF_RANGE: The supplied index does not identify an available counter.

GPA_STATUS_ERROR_NULL_POINTER: The supplied description parameter is null.

GPA_STATUS_OK: On success.

Get Index of a Counter Given its Name (Case Insensitive)

Syntax

```
GPALIB_DECL GPA_Status GPA_GetCounterIndex ( const char * counter,
gpa_uint32 * index )
```

Description

Retrieves a counter index from the string name. Useful for searching the availability of a specific counter.

Parameters

counter	The name of the counter to get the index for.
index	The index of the requested counter.

Returns

GPA_STATUS_ERROR_COUNTERS_NOT_OPEN: GPA_OpenContext must be called before this call to initialize the counters.

GPA_STATUS_ERROR_NULL_POINTER: The supplied counter or index parameter is null.

GPA_STATUS_ERROR_NOT_FOUND: A counter with the specified name could not be found.

GPA_STATUS_OK: On success.

Get the Name of a Specific Counter

Syntax

```
GPALIB_DECL GPA_Status GPA_GetCounterName ( gpa_uint32 index, const char ** name )
```

Description

Retrieves a counter name from a supplied index. Useful for printing counter results in a readable format.

Parameters

index	The index of the counter name to query. Must lie between 0 and (`GPA_GetNumCounters` result - 1), inclusive.
name	The value that holds the name upon successful execution.

Returns

`GPA_STATUS_ERROR_COUNTERS_NOT_OPEN`: `GPA_OpenContext` must be called before this call to initialize the counters.

`GPA_STATUS_ERROR_INDEX_OUT_OF_RANGE`: The supplied index does not identify an available counter.

`GPA_STATUS_ERROR_NULL_POINTER`: The supplied name parameter is null.

`GPA_STATUS_OK`: On success.

Get the Type of the Specified Counter

Syntax

```
GPALIB_DECL GPA_Status GPA_GetCounterType ( gpa_uint32 index, GPA_CounterType * counterType )
```

Description

Retrieves the counter, whether it is static or dynamic. Static counters are not likely to change between executions of the same API call; dynamic counters may have some variation.

Parameters

index	The index of the counter. Must lie between 0 and (`GPA_GetNumCounters` result - 1), inclusive.
counterType	The value that holds the description upon successful execution.

Returns

`GPA_STATUS_ERROR_COUNTERS_NOT_OPEN`: `GPA_OpenContext` must be called before this call to initialize the counters.

`GPA_STATUS_ERROR_INDEX_OUT_OF_RANGE`: The supplied index does not identify an available counter.

`GPA_STATUS_ERROR_NULL_POINTER`: The supplied counterType parameter is null.

`GPA_STATUS_OK`: On success.

Get the Specified Counter's Usage Type

Syntax

```
GPALIB_DECL GPA_Status GPA_GetCounterUsageType ( gpa_uint32 index,
GPA_Usage_Type * counterUsageType )
```

Description

Retrieves the usage type of the counter at the supplied index.

Parameters

index	The index of the counter. Must lie between 0 and (GPA_GetNumCounters result - 1), inclusive.
counterUsageType	The value that holds the description upon successful execution.

Returns

GPA_STATUS_ERROR_COUNTERS_NOT_OPEN: GPA_OpenContext must be called before this call to initialize the counters.

GPU_STATUS_ERROR_INDEX_OUT_OF_RANGE: The supplied index does not identify an available counter.

GPA_STATUS_ERROR_NULL_POINTER: The supplied counterDataType parameter is null. A reference to a GPA_Type variable is expected.

GPA_STATUS_OK: On success.

Get a String with the Name of the Specified Counter Data Type

Syntax

```
GPALIB_DECL GPA_Status GPA_GetDataTypeAsStr ( GPA_Type counterDataType,
const char ** typeStr )
```

Description

Typically used to display counter types along with their name (for example, counterDataType of GPA_TYPE_UINT64 returns gpa_uint64).

Parameters

counterDataType	The type to get the string for.
typeStr	The value set to contain a reference to the name of the counter data type.

Returns

GPA_STATUS_ERROR_NOT_FOUND: An invalid counterDataType parameter was supplied.

GPA_STATUS_ERROR_NULL_POINTER: The supplied typeStr parameter is null.

GPA_STATUS_OK: On success.

Get the Number of Enabled Counters

Syntax

```
GPALIB_DECL GPA_Status GPA_GetEnabledCount ( gpa_uint32 * count )
```

Description

Retrieves the number of enabled counters.

Parameters

count	Address of the variable that is set to the number of enabled counters if GPA_STATUS_OK is returned. This is not modified if an error is returned.

Returns

GPA_STATUS_ERROR_COUNTERS_NOT_OPEN: GPA_OpenContext must be called before this call to initialize the counters.

GPA_STATUS_ERROR_NULL_POINTER: The supplied count parameter is null.

GPA_STATUS_OK: On success.

Get the Counter Index for an Enabled Counter

Syntax

```
GPALIB_DECL GPA_Status GPA_GetEnabledIndex ( gpa_uint32 enabledNumber,
gpa_uint32 * enabledCounterIndex )
```

Description

For example, if GPA_GetEnabledIndex returns 3, then call this function with enabledNumber equal to 0 to get the counter index of the first enabled counter.

Parameters

enabledNumber The number of the enabled counter for which to get the counter index. Must lie between 0 and (GPA_GetEnabledIndex result - 1), inclusive.

enabledCounterIndex Contains the index of the counter.

Returns

GPA_STATUS_ERROR_COUNTERS_NOT_OPEN: GPA_OpenContext must be called before this call to initialize the counters.

GPA_STATUS_ERROR_NULL_POINTER: The supplied enabledCounterIndex parameter is null.

GPA_STATUS_ERROR_INDEX_OUT_OF_RANGE: The supplied enabledNumber does not identify an enabled counter.

GPA_STATUS_OK: On success.

Get the Number of Counters Available

Syntax

```
GPALIB_DECL GPA_Status GPA_GetNumCounters ( gpa_uint32 * count )
```

Description

Retrieves the number of counters provided by the currently loaded GPUPerfAPI library. Results can vary based on the current context and available hardware.

Parameters

count Holds the count upon successful execution.

Returns

GPA_STATUS_ERROR_COUNTERS_NOT_OPEN: GPA_OpenContext must be called before this call to initialize the counters.

GPA_STATUS_ERROR_NULL_POINTER: The supplied count parameter is null.

GPA_STATUS_OK: On success.

Get the Number of Passes Required for the Currently Enabled Set of Counters

Syntax

```
GPALIB_DECL GPA_Status GPA_GetPassCount ( gpa_uint32 * numPasses )
```

Description

This represents the number of times the same sequence must be repeated to capture the counter data. On each pass a different (compatible) set of counters is measured.

Parameters

numPasses The value of the number of passes.

Returns

GPA_STATUS_ERROR_COUNTERS_NOT_OPEN: GPA_OpenContext must be called before this call to initialize the counters.

GPA_STATUS_ERROR_NULL_POINTER: The supplied numPasses parameter is null.

GPA_STATUS_OK: On success.

Get the Number of Samples a Specified Session Contains

Syntax

```
GPALIB_DECL GPA_Status GPA_GetSampleCount ( gpa_uint32 sessionID,
gpa_uint32 * samples )
```

Description

This is useful if samples are conditionally created and a count is not kept.

Parameters

sessionID	The session for which to get the number of samples for.
samples	The number of samples contained within the session.

Returns

GPA_STATUS_ERROR_COUNTERS_NOT_OPEN: GPA_OpenContext must be called before this call to initialize the counters.

GPA_STATUS_ERROR_NULL_POINTER: The supplied samples parameter is null.

GPA_STATUS_ERROR_SESSION_NOT_FOUND: The supplied sessionID does not identify an available session.

GPA_STATUS_OK: On success.

Get a Sample of Type 32-bit Float

Syntax

```
GPALIB_DECL GPA_Status GPA_GetSampleFloat32 ( gpa_uint32 sessionID,
gpa_uint32 sampleID, gpa_uint32 counterIndex, gpa_float32 * result )
```

Description

This function blocks further processing until the value is available. Use GPA_IsSampleReady for no blocking.

Parameters

sessionID	The session identifier with the sample for which to retrieve the result.
sampleID	The identifier of the sample for which to get the result.
counterIndex	The counter index for which to get the result.
result	The counter result upon successful execution.

Returns

GPA_STATUS_ERROR_COUNTERS_NOT_OPEN: GPA_OpenContext must be called before this call to initialize the counters.

GPA_STATUS_ERROR_SESSION_NOT_FOUND: The supplied sessionID does not identify an available session.

GPA_STATUS_ERROR_NOT_ENABLED: The specified counterIndex does not identify an enabled counter.

GPA_STATUS_ERROR_INDEX_OUT_OF_RANGE: The supplied counterIndex does not identify an available counter.

GPA_STATUS_ERROR_NULL_POINTER: The supplied result parameter is null.

GPA_STATUS_ERROR_COUNTER_NOT_OF_SPECIFIED_TYPE: The supplied counterIndex identifies a counter that is not a gpa_float32.

GPA_STATUS_OK: On success.

Get a Sample of Type 64-bit Float

Syntax

```
GPALIB_DECL GPA_Status GPA_GetSampleFloat64 ( gpa_uint32 sessionID,
gpa_uint32 sampleID, gpa_uint32 counterIndex, gpa_float64 * result )
```

Description

This function blocks further processing until the value is available. Use GPA_IsSampleReady for no blocking.

Parameters

sessionID	The session identifier with the sample for which to retrieve the result.
sampleID	The identifier of the sample for which to get the result.
counterIndex	The counter index for which to get the result.
result	The value to contain the counter result upon successful execution.

Returns

GPA_STATUS_ERROR_COUNTERS_NOT_OPEN: GPA_OpenContext must be called before this call to initialize the counters.

GPA_STATUS_ERROR_SESSION_NOT_FOUND: The supplied sessionID does not identify an available session.

GPA_STATUS_ERROR_NOT_ENABLED: The specified counterIndex does not identify an enabled counter.

GPA_STATUS_ERROR_INDEX_OUT_OF_RANGE: The supplied counterIndex does not identify an available counter.

GPA_STATUS_ERROR_NULL_POINTER: The supplied result parameter is null.

GPA_STATUS_ERROR_COUNTER_NOT_OF_SPECIFIED_TYPE: The supplied counterIndex identifies a counter that is not a gpa_float64.

GPA_STATUS_OK: On success.

Get a Sample of Type 32-bit Unsigned Integer

Syntax

```
GPALIB_DECL GPA_Status GPA_GetSampleUInt32 ( gpa_uint32 sessionID,
gpa_uint32 sampleID, gpa_uint32 counterIndex, gpa_uint32 * result )
```

Description

This function blocks further processing until the value is available. Use GPA_IsSampleReady to not block further processing.

Parameters

sessionID	The session identifier with the sample for which to retrieve the result.
sampleID	The identifier of the sample for which to get the result.
counterIndex	The counter index for which to get the result.
result	The value to contain the counter result upon successful execution.

Returns

GPA_STATUS_ERROR_COUNTERS_NOT_OPEN: GPA_OpenContext must be called before this call to initialize the counters.

GPA_STATUS_ERROR_SESSION_NOT_FOUND: The supplied sessionID does not identify an available session.

GPA_STATUS_ERROR_NOT_ENABLED: The specified counterIndex does not identify an enabled counter.

GPA_STATUS_ERROR_INDEX_OUT_OF_RANGE: The supplied counterIndex does not identify an available counter.

GPA_STATUS_ERROR_NULL_POINTER: The supplied result parameter is null.

GPA_STATUS_ERROR_COUNTER_NOT_OF_SPECIFIED_TYPE: The supplied counterIndex identifies a counter that is not a gpa_uint32.

GPA_STATUS_OK: On success.

Get a Sample of Type 64-bit Unsigned Integer

Syntax

```
GPALIB_DECL GPA_Status GPA_GetSampleUInt64 ( gpa_uint32 sessionID,
gpa_uint32 sampleID, gpa_uint32 counterID, gpa_uint64 * result )
```

Description

This function blocks further processing until the value is available. Use GPA_IsSampleReady to not block.

Parameters

sessionID	The session identifier with the sample for which to retrieve the result.
sampleID	The counter index for which to get the result.
counterID	The identifier of the sample for which to get the result.
result	The value to contain the counter result upon successful execution.

Returns

GPA_STATUS_ERROR_COUNTERS_NOT_OPEN: GPA_OpenContext must be called before this call to initialize the counters.

GPA_STATUS_ERROR_SESSION_NOT_FOUND: The supplied sessionID does not identify an available session.

GPA_STATUS_ERROR_NOT_ENABLED: The specified counterIndex does not identify an enabled counter.

GPA_STATUS_ERROR_INDEX_OUT_OF_RANGE: The supplied counterIndex does not identify an available counter.

GPA_STATUS_ERROR_NULL_POINTER: The supplied result parameter is null.

GPA_STATUS_ERROR_COUNTER_NOT_OF_SPECIFIED_TYPE: The supplied counterIndex identifies a counter that is not a gpa_uint64.

GPA_STATUS_OK: On success.

Get a String Version of the Status Value

Syntax

```
GPALIB_DECL const char* GPA_GetStatusAsStr( GPA_Status status )
```

Description

This converts the status into a string to print in a log file.

Parameters

status	The status for which to get a string value.

Returns

A string version of the status value, or Unknown Error if an unrecognized value is supplied. Does not return NULL.

Get a String with the Name of the Specified Counter Data Type

Syntax

```
GPALIB_DECL GPA_Status GPA_GetTypeAsStr ( GPA_CounterType counterType,
const char ** typeStr )
```

Description

This indicates the frequency that the value could change (different from the data type of the counter).

Parameters

counterType	The counter type to get the string for.
typeStr	The to contain a reference to the name of the counter type.

Returns

GPA_STATUS_ERROR_NOT_FOUND: An invalid counterType parameter was supplied.

GPA_STATUS_ERROR_NULL_POINTER: The supplied typeStr parameter is null.

GPA_STATUS_OK: On success.

Get a String with the Name of the Specified Counter Usage Type

Syntax

```
GPALIB_DECL GPA_Status GPA_GetUsageTypeAsStr (
GPA_CounterType counterUsageType, const char ** )
```

Description

Typically used to display counters along with their usage (for example, counterUsageType of GPA_USAGE_TYPE_PERCENTAGE returns “percentage.”

Parameters

counterUsageType	The usage type for which to get the string.
usageTypeStr	The value set to contain a reference to the name of the counter usage type.

Returns

GPA_STATUS_ERROR_NOT_FOUND: An invalid counterUsageType parameter was supplied.

GPA_STATUS_ERROR_NULL_POINTER: The supplied usageTypeStr parameter is null.

GPA_STATUS_OK: On success.

Check if a Counter is Enabled

Syntax

```
GPALIB_DECL GPA_Status GPA_IsCounterEnabled ( gpa_uint32 counterIndex )
```

Description

Indicates if the specified counter is enabled.

Parameter

counterIndex	The index of the counter. Must lie between 0 and (GPA_GetNumCounters result - 1), inclusive.

Returns

GPA_STATUS_ERROR_INDEX_OUT_OF_RANGE: The supplied counterIndex does not identify an available counter.

GPA_STATUS_ERROR_NOT_FOUND: The counter is not enabled.

GPA_STATUS_OK: On success.

Initialize the GPUPerfAPI for Counter Access

Syntax

```
GPALIB_DECL GPA_Status GPA_Initialize ( )
```

Description

To access counters when using DirectX 10 or 11, the UAC may have to be disabled and your application must be set to run with administrator privileges.

Returns

GPA_STATUS_FAILED: An internal error occurred. UAC or administrator privileges can be the cause.

GPA_STATUS_OK: On success.

Determine if an Individual Sample Result is Available

Syntax

```
GPALIB_DECL GPA_Status GPA_IsSampleReady ( bool * readyResult,
gpa_uint32 sessionID, gpa_uint32 sampleID )
```

Description

After a sampling session, results may be available immediately or take time to become available. This function indicates when a sample can be read. The function does not block further processing, permitting periodic polling. To block further processing until a sample is ready, use a GetSample* function instead. It can be more efficient to determine if the data of an entire session is available by using GPA_IsSessionReady.

Parameters

ReadyResult The value that contains the result of the ready sample. True if ready.

sessionID The session containing the sample to determine availability.

sampleID The sample identifier of the sample for which to query availability.

Returns

GPA_STATUS_ERROR_COUNTERS_NOT_OPEN: GPA_OpenContext must be called before this call to initialize the counters.

GPA_STATUS_ERROR_NULL_POINTER: The supplied readyResult parameter is null.

GPA_STATUS_ERROR_SESSION_NOT_FOUND: The supplied sessionID does not identify an available session.

GPA_STATUS_ERROR_SAMPLE_NOT_FOUND_IN_ALL_PASSES: The requested sampleID is not available in all the passes. There can be a different number of samples in the passes of a multi-pass profile.

GPA_STATUS_OK: On success.

Determine if All Samples within a Session are Available

Syntax

```
GPALIB_DECL GPA_Status GPA_IsSessionReady ( bool * readyResult,
gpa_uint32 sessionID )
```

Description

After a sampling session, results may be available immediately or take time to become available. This function indicates when the results of a session can be read. The function does not block further processing, permitting periodic polling. To block further processing until a sample is ready, use a `GetSample*` function.

Parameters

`ReadyResult` The value that contains the result of the ready session. True if ready.

`sessionID` The session for which to determine availability.

Returns

`GPA_STATUS_ERROR_COUNTERS_NOT_OPEN`: `GPA_OpenContext` must be called before this call to initialize the counters.

`GPA_STATUS_ERROR_NULL_POINTER`: The supplied readyResult parameter is null.

`GPA_STATUS_ERROR_SESSION_NOT_FOUND`: The supplied sessionID does not identify an available session.

`GPA_STATUS_OK`: On success.

Open the Counters in the Specified Context

Syntax

```
GPALIB_DECL GPA_Status GPA_OpenContext ( void * context )
```

Description

Opens the counters in the specified context for reading. Call this function after `GPA_Initialize()` and after the rendering / compute context has been created.

Parameter

context The context for which to open counters. Typically, a device pointer or handle to a rendering context.

Returns

`GPA_STATUS_ERROR_NULL_POINTER`: The supplied context parameter is null.

`GPA_STATUS_ERROR_COUNTERS_ALREADY_OPEN`: The counters are already open and do not need to be opened again.

`GPA_STATUS_ERROR_FAILED`: An internal error occurred while trying to open the counters.

`GPA_STATUS_ERROR_HARDWARE_NOT_SUPPORTED`: The current hardware is not supported by GPU Performance API.

`GPA_STATUS_OK`: On success.

7 Utility Function

The following is an example of how to read data back from the completed session, as well as how to save the data to a comma-separated value file (.csv).

```
#pragma warning( disable : 4996 )

/// Given a sessionID, query the counter values and save them to a file
void WriteSession( gpa_uint32 currentWaitSessionID, const char* filename )
{
   static bool doneHeadings = false;

   gpa_uint32 count;
   GPA_GetEnabledCount( &count );

   FILE* f;

   if ( !doneHeadings )
   {
      const char* name;

      f = fopen( filename, "w" );
      assert( f );

      fprintf( f, "Identifier, " );

      for ( gpa_uint32 counter = 0 ; counter < count ; counter++ )
      {
         gpa_uint32 enabledCounterIndex;
         GPA_GetEnabledIndex( counter, &enabledCounterIndex );
         GPA_GetCounterName( enabledCounterIndex, &name );

         fprintf( f, "%s, ", name );
      }

      fprintf( f, "\n" );

      fclose( f );

      doneHeadings = true;
   }

   f = fopen( filename, "a+" );

   assert( f );

   gpa_uint32 sampleCount;
   GPA_GetSampleCount( currentWaitSessionID, &sampleCount );


   for ( gpa_uint32 sample = 0 ; sample < sampleCount ; sample++ )
   {
      fprintf( f, "session: %d; sample: %d, ", currentWaitSessionID,
               sample );
```

```
      for ( gpa_uint32 counter = 0 ; counter < count ; counter++ )
      {
         gpa_uint32 enabledCounterIndex;
         GPA_GetEnabledIndex( counter, &enabledCounterIndex );
         GPA_Type type;
         GPA_GetCounterDataType( enabledCounterIndex, &type );

         if ( type == GPA_TYPE_UINT32 )
         {
            gpa_uint32 value;
            GPA_GetSampleUInt32( currentWaitSessionID,
                                 sample, enabledCounterIndex, &value );

            fprintf( f, "%u,", value );
         }
         else if ( type == GPA_TYPE_UINT64 )
         {
            gpa_uint64 value;
            GPA_GetSampleUInt64( currentWaitSessionID,
                                 sample, enabledCounterIndex, &value );
            fprintf( f, "%I64u,", value );
         }
         else if ( type == GPA_TYPE_FLOAT32 )
         {
            gpa_float32 value;
            GPA_GetSampleFloat32( currentWaitSessionID,
                                  sample, enabledCounterIndex, &value );
            fprintf( f, "%f,", value );
         }
         else if ( type == GPA_TYPE_FLOAT64 )
         {
            gpa_float64 value;
            GPA_GetSampleFloat64( currentWaitSessionID,
                                  sample, enabledCounterIndex, &value );
            fprintf( f, "%f,", value );
         }
         else
         {
            assert(false);
         }
      }

      fprintf( f, "\n" );
   }

   fclose( f );
}


#pragma warning( default : 4996 )
```

Contact

Advanced Micro Devices, Inc.
One AMD Place
P.O. Box 3453
Sunnyvale, CA, 94088-3453
Phone: +1.408.749.4000

For GPU Developer Tools:
URL: **http://developer.amd.com/GPU**
Questions: **gputools.support@amd.com**