User Guide

AMD GPU Performance API

Contents

1. Introduction 2
2. Usage 2
   - 2.1. Dynamically Loaded Library on Windows 2
   - 2.2. Shared-Object Library on Linux 2
   - 2.3. Registering a Logging Callback 3
   - 2.4. Initializing GPUPerfAPI 3
   - 2.5. Obtaining Available Counters 3
   - 2.6. Retrieving Information about the Counters 4
   - 2.7. Enabling Counters 4
   - 2.8. Disabling Counters 4
   - 2.9. Multi-Pass Profiling 5
   - 2.10. Sampling Counters 5
   - 2.11. Counter Results 6
   - 2.12. Result Buffering 7
   - 2.13. Closing GPUPerfAPI 7
   - 2.14. Deploying GPUPerfAPI 8
3. Example Code 8
   - 3.1. Startup 8
   - 3.2. Render Loop 8
   - 3.3. On Exit 9
4. Counter Groups 10
5. Counter Descriptions 13
6. API Functions 21
7. Utility Function 50

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 DirectX11, OpenGL, OpenGLES, OpenCL, and HSA on GCN-based Radeon™ graphics cards and APUs
- Supports Microsoft Windows as a dynamically loaded library.
  - DirectX11, OpenGL, OpenGLES and OpenCL only
- Supports Linux as a shared-object library:
  - Targeting Ubuntu (14.04 and later) and RHEL (7 and later), distributions
  - OpenCL, OpenGL, OpenGLES and HSA only
- Provides derived counters based on raw HW performance counters.
- Manages memory automatically – no allocations required.
- Requires Radeon Software Crimson Edition 16.2.1 or later (Driver Packaging Version 16.15 or later).

2. Usage

2.1. Dynamically Loaded Library on Windows

To use the GPUPerfAPI library on Windows,

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.

2.2. Shared-Object Library on Linux

To use the GPUPerfAPI shared library on Linux,

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 libGPUPerfAPI.so for your chosen API.
5. For each function in GPUPerfAPI, call dlsym(…).
6. Use the functions to profile your application.

2.3. Registering a Logging Callback

An entrypoint is available for registering an optional callback function which GPUPerfAPI will use to report back additional information about errors, messages, and/or API usage. In order to use this feature, you must define a static function with the following signature in your application:

```
void MyLoggingFunction( GPA_Logging_Type messageType, const char*
message );
```

The function may be registered using the following GPUPerfAPI entrypoint:

```
GPA_Status GPA_RegisterLoggingCallback( GPA_Logging_Type loggingType,
GPA_LoggingCallbackPtrType pCallbackFuncPtr );
```

You will only receive callbacks for message types that you choose to receive, and the message type is passed into your logging function so that you may handle them differently if desired (perhaps errors are output to cerr or display an assert, while messages and trace information is output to your normal log file). The messages passed into your logging function will not have a newline at the end, allowing for more flexible handling of the message.

2.4. Initializing 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. For HSA, GPA_Initialize must be called prior to the first call to hsa_init().

```
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* pContext );
```

The supplied context must either point to a DirectX device, be the handle to the OpenGL rendering context, the OpenCL command queue handle, or the HSA queue. The return value indicates whether or not the current hardware is supported by GPUPerfAPI. See the API Functions section 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* pCount );
```

To retrieve the name of a counter, call:

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

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

```
GPA_Status GPA_GetCounterIndex( const char* pCounter,
                                gpa_uint32* pIndex );
```

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** ppDescription );
```

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* pDataType );
```

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* pCounter );
```

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* pCounter );
```

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* pNumPasses );
```

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* pSessionID );

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
```

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* pReadyResult,
                               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* pReadyResult,
                              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* pResult );

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

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

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

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:

```
GPA_Status GPA_Destroy();
```

2.14. Deploying GPUPerfAPI

To deploy an application that uses GPUPerfAPI, simply make sure that the necessary GPUPerfAPI library is available and can be loaded using the normal library search mechanism for the host operating system (i.e. in the **PATH** on Windows and **LD_LIBRARY_PATH** on Linux).

When deploying the DirectX11 version on Windows, you will also need to deploy GPUPerfAPIDXGetAMDDeviceInfo.dll or GPUPerfAPIDXGetAMDDeviceInfo-x64.dll, if you need to support systems with multiple AMD GPUs. This library is used by GPA to determine which GPU is being used for rendering at runtime. For single-GPU systems, this library is not required.

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, the handle to the GL context should be passed into the `OpenContext` function; for OpenCL, the command queue handle should be supplied.

```
GPA_Initialize();
D3D11CreateDeviceAndSwapChain( . . . &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 GPA 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. Below is a collective list of counters from all the supported hardware generations. Some of the counters may not be available depending on the hardware being profiled.

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[3]	CSBusy CSTime DepthStencilTestBusy DSBusy DSTime GPUBusy GPUTime GSBusy GSTime HSBusy HSTime PrimitiveAssemblyBusy PSBusy PSTime TessellatorBusy TexUnitBusy VSBusy VSTime
VertexShader[3]	VSSALUBusy VSSALUInstCount VSVALUBusy VSVALUInstCount VSVerticesIn
HullShader[3]	HSPatches HSSALUBusy HSSALUInstCount HSVALUBusy HSVALUInstCount

GeometryShader[3]	GSPrimsIn GSSALUBusy GSSALUInstCount GSVALUBusy GSVALUInstCount GSVerticesOut
PrimitiveAssembly[3]	ClippedPrims CulledPrims PAStalledOnRasterizer PrimitivesIn
DomainShader[3]	DSSALUBusy DSSALUInstCount DSVALUBusy DSVALUInstCount DSVerticesIn
PixelShader[3]	PSExportStalls PSPixelsOut PSSALUBusy PSSALUInstCount PSVALUBusy PSVALUInstCount
TextureUnit[3]	TexAveAnisotropy TexTriFilteringPct TexVolFilteringPct
General[1]	FlatVMemInsts GDSInsts SALUBusy SALUInsts SFetchInsts VALUBusy VALUInsts VALUUtilization VFetchInsts VWriteInsts Wavefronts
ComputeShader[3]	CSALUStalledByLDS CSCacheHit CSFetchInsts CSFetchSize CSFlatLDSInsts CSFlatVMemInsts CSGDSInsts CSLDSBankConflict CSLDSInsts CSMemUnitBusy CSMemUnitStalled

	CSSALUBusy CSSALUInsts CSThreadGroups CSThreads CSVALUBusy CSVALUInsts CSVALUUtilization CSVFetchInsts CSVWriteInsts CSWavefronts CSWriteSize CSWriteUnitStalled
DepthAndStencil[3]	HiZQuadsCulled HiZTilesAccepted PostZQuads PostZSamplesFailingS PostZSamplesFailingZ PostZSamplesPassing PreZQuadsCulled PreZSamplesFailingS PreZSamplesFailingZ PreZSamplesPassing PreZTilesDetailCulled ZUnitStalled
ColorBuffer[3]	CBMemRead CBMemWritten CBSlowPixelPct
GlobalMemory[1]	FetchSize CacheHit MemUnitBusy MemUnitStalled WriteSize WriteUnitStalled
LocalMemory[1]	FlatLDSInsts LDSBankConflict LDSInsts
D3D11[2]	CInvocations CPrimitives CSInvocations D3DGPUTime DSInvocations GSInvocation GSPrimitives HSInvocations IAPrimitives IAVertices

	Occlusion OcclusionPredicate OverflowPred OverflowPred_S0 OverflowPred_S1 OverflowPred_S2 OverflowPred_S3 PrimsStorageNeed PrimsStorageNeed_S0 PrimsStorageNeed_S1 PrimsStorageNeed_S2 PrimsStorageNeed_S3 PrimsWritten PrimsWritten_S0 PrimsWritten_S1 PrimsWritten_S2 PrimsWritten_S3 PSInvocations VSInvocations

[1] Exposed only by the OpenCL and HSA versions of the GPU Performance API
[2] Exposed only by the DirectX11 version of the GPU Performance API
[3] Exposed only by the DirectX11, OpenGL and OpenGLES versions 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 which APIs contain the counters and the hardware version on which they are available.

OpenCL and HSA Counter Descriptions

Counter	Description
CacheHit	The percentage of fetches from the video memory that hit the data cache. Value range: 0% (no hit) to 100% (optimal).
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.
FlatLDSInsts[1]	The average number of FLAT instructions that read from or write to LDS executed per work item (affected by flow

	control).
FlatVMemInsts[1]	The average number of FLAT instructions that read from or write to the video memory executed per work item (affected by flow control). Includes FLAT instructions that read from or write to scratch.
GDSInsts	The average number of GDS read or GDS write instructions executed per work-item (affected by flow control).
LDSBankConflict	The percentage of GPUTime LDS is stalled by bank conflicts.
LDSInsts	The average number of LDS read or LDS write instructions executed per work item (affected by flow control). On 2nd Generation GCN-based hardware, this value excludes FLAT instructions that read from or write to LDS.
MemUnitBusy	The percentage of GPUTime the memory unit is active. The result includes the stall time (MemUnitStalled). This is measured with all extra fetches and writes and any cache or memory effects taken into account. Value range: 0% to 100% (fetch-bound).
MemUnitStalled	The percentage of GPUTime the memory unit is stalled. Try reducing the number or size of fetches and writes if possible. Value range: 0% (optimal) to 100% (bad).
SALUBusy	The percentage of GPUTime scalar ALU instructions are processed. Value range: 0% (bad) to 100% (optimal).
SALUInsts	The average number of scalar ALU instructions executed per work-item (affected by flow control).
SFetchInsts	The average number of scalar fetch instructions from the video memory executed per work-item (affected by flow control).
VALUBusy	The percentage of GPUTime vector ALU instructions are processed. Value range: 0% (bad) to 100% (optimal).
VALUInsts	The average number of vector ALU instructions executed per work-item (affected by flow control).
VALUUtilization	The percentage of active vector ALU threads in a wave. A lower number can mean either more thread divergence in a wave or that the work-group size is not a multiple of 64. Value range: 0% (bad), 100% (ideal - no thread divergence).
VFetchInsts	The average number of vector fetch instructions from the video memory executed per work-item (affected by flow control). On 2nd Generation GCN-based hardware, this value excludes FLAT instructions that fetch from video memory.
VWriteInsts	The average number of vector write instructions to the video memory executed per work-item (affected by flow

	control). On 2nd Generation GCN-based hardware, this value excludes FLAT instructions that write to video memory.
Wavefronts	Total wavefronts.
WriteSize	The total kilobytes written to the video memory. This is measured with all extra fetches and any cache or memory effects taken into account.
WriteUnitStalled	The percentage of GPUTime Write unit is stalled.

[1] Only available on 2nd generation Graphics Core Next based AMD Radeon™ Graphics Cards or newer

OpenGL and DirectX Counter Descriptions

Counter	Description
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.
CInvocations	Number of primitives that were sent to the rasterizer.
ClippedPrims	The number of primitives that required one or more clipping operations due to intersecting the view volume or user clip planes.
CPrimitives	Number of primitives that were rendered.
CSALUStalledByLDS	The percentage of GPUTime ALU units are stalled by the LDS input queue being full or the output queue being not ready. If there are LDS bank conflicts, reduce them. Otherwise, try reducing the number of LDS accesses if possible. Value range: 0% (optimal) to 100% (bad).
CSBusy	The percentage of time the ShaderUnit has compute shader work to do.
CSCacheHit	The percentage of fetches from the global memory that hit the texture cache.
CSFetchInsts	Average number of fetch instructions executed in the CS per execution. Affected by the flow control.
CSFetchSize	The total kilobytes fetched from the video memory. This is measured with all extra fetches and any cache or memory effects taken into account.
CSFlatLDSInsts[1]	The average number of FLAT instructions that read from or write to LDS executed per work item (affected by flow control).
CSFlatVMemInsts[1]	The average number of FLAT instructions that read from or write to the video memory executed per work item (affected by flow control). Includes FLAT instructions that read from or write to scratch.
CSGDSInsts	The average number of instructions to/from the GDS executed per work-item (affected by flow control).

CSInvocations	Number of times a compute shader was invoked.
CSLDSBankConflict	The percentage of GPUTime the LDS is stalled by bank conflicts.
CSLDSInsts	The average number of LDS read/write instructions executed per work-item (affected by flow control).
CSMemUnitBusy	The percentage of GPUTime the memory unit is active. The result includes the stall time (MemUnitStalled). This is measured with all extra fetches and writes and any cache or memory effects taken into account. Value range: 0% to 100% (fetch-bound).
CSMemUnitStalled	The percentage of GPUTime the memory unit is stalled. Try reducing the number or size of fetches and writes if possible. Value range: 0% (optimal) to 100% (bad).
CSSALUBusy	The percentage of GPUTime scalar ALU instructions are processed. Value range: 0% (bad) to 100% (optimal).
CSSALUInsts	The average number of scalar ALU instructions executed per work-item (affected by flow control).
CSThreadGroups	Total number of thread groups.
CSThreads	The number of CS threads processed by the hardware.
CSTime	Time compute shaders are busy in milliseconds.
CSVALUBusy	The percentage of GPUTime vector ALU instructions are processed. Value range: 0% (bad) to 100% (optimal).
CSVALUInsts	The average number of vector ALU instructions executed per work-item (affected by flow control).
CSVALUUtilization	The percentage of active vector ALU threads in a wave. A lower number can mean either more thread divergence in a wave or that the work-group size is not a multiple of 64. Value range: 0% (bad), 100% (ideal - no thread divergence).
CSVFetchInsts	The average number of vector fetch instructions from the video memory executed per work-item (affected by flow control).
CSVWriteInsts	The average number of vector write instructions to the video memory executed per work-item (affected by flow control).
CSWavefronts	The total number of wavefronts used for the CS.
CSWriteSize	The total kilobytes written to the video memory. This is measured with all extra fetches and any cache or memory effects taken into account.
CSWriteUnitStalled	The percentage of GPUTime the Write unit is stalled. Value range: 0% to 100% (bad).
CulledPrims	The number of culled primitives. Typical reasons include scissor, the primitive having zero area, and back or front face culling.
D3DGPUTime	Time spent in GPU
DepthStencilTestBusy	Percentage of GPUTime spent performing depth and

	stencil tests.
DSBusy	The percentage of time the ShaderUnit has domain shader work to do.
DSInvocations	Number of times a domain shader was invoked.
DSSALUBusy[1]	The percentage of GPUTime scalar ALU instructions are being processed by the DS.
DSSALUInstCount[1]	Average number of scalar ALU instructions executed in the DS. Affected by flow control.
DSTime	Time domain shaders are busy in milliseconds.
DSVALUBusy[1]	The percentage of GPUTime vector ALU instructions are being processed by the DS.
DSVALUInstCount[1]	Average number of vector ALU instructions executed in the DS. Affected by flow control.
DSVerticesIn	The number of vertices processed by the DS.
GPUBusy	The 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.
GSBusy	The percentage of time the ShaderUnit has geometry shader work to do.
GSInvocations	Number of times a geometry shader was invoked.
GSPrimitives	Number of primitives output by a geometry shader.
GSPrimsIn	The number of primitives passed into the GS.
GSSALUBusy[1]	The percentage of GPUTime scalar ALU instructions are being processed by the GS.
GSSALUInstCount[1]	Average number of scalar ALU instructions executed in the GS. Affected by flow control.
GSTime	Time geometry shaders are busy in milliseconds.
GSVALUBusy[1]	The percentage of GPUTime vector ALU instructions are being processed by the GS.
GSVALUInstCount[1]	Average number of vector ALU instructions executed in the GS. Affected by flow control.
GSVerticesOut	The number of vertices output by the GS.
HiZQuadsCulled	Percentage of quads that did not have to continue on in the pipeline after HiZ. They may be written directly to the depth buffer, or culled completely. Consistently low values here may suggest that the Z-range is not being fully utilized.
HiZTilesAccepted	Percentage of tiles accepted by HiZ and will be rendered to the depth or color buffers.
HSBusy	The percentage of time the ShaderUnit has hull shader work to do.
HSInvocations	Number of times a hull shader was invoked.
HSPatches	The number of patches processed by the HS.
HSSALUBusy[1]	The percentage of GPUTime scalar ALU instructions are

	being processed by the HS.
HSSALUInstCount[1]	Average number of scalar ALU instructions executed in the HS. Affected by flow control.
HSTime	Time hull shaders are busy in milliseconds.
HSVALUBusy[1]	The percentage of GPUTime vector ALU instructions are being processed by the HS.
HSVALUInstCount[1]	Average number of vector ALU instructions executed in the HS. Affected by flow control.
IAPrimitives	Number of primitives read by the input assembler.
IAVertices	Number of vertices read by input assembler.
Occlusion	Get the number of samples that passed the depth and stencil tests.
OcclusionPredicate	Did any samples pass the depth and stencil tests?
OverflowPred	Determines if any of the streaming output buffers overflowed.
OverflowPred_S0	Determines if the stream 0 buffer overflowed.
OverflowPred_S1	Determines if the stream 1 buffer overflowed.
OverflowPred_S2	Determines if the stream 2 buffer overflowed.
OverflowPred_S3	Determines if the stream 3 buffer overflowed.
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.
PostZQuads	Percentage of quads for which the pixel shader will run and may be PostZ tested.
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.
PreZQuadsCulled	Percentage of quads rejected based on the detailZ and earlyZ tests.
PreZSamplesFailingS	Number of samples tested for Z before shading and failed stencil test.
PreZSamplesFailingZ	Number of samples tested for Z before shading and failed Z test.
PreZSamplesPassing	Number of samples tested for Z before shading and passed.
PreZTilesDetailCulled	Percentage of tiles rejected because the associated prim had no contributing area.
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.
PrimsStorageNeed	Primitives not written to the SO buffers due to limited space.
PrimsStorageNeed_S0	Primitives not written to stream 0 due to limited space.
PrimsStorageNeed_S1	Primitives not written to stream 1 due to limited space.
PrimsStorageNeed_S2	Primitives not written to stream 2 due to limited space.
PrimsStorageNeed_S3	Primitives not written to stream 3 due to limited space.
PrimsWritten	Number of primitives written to the stream-output buffers.
PrimsWritten_S0	Number of primitives written to the stream 0 buffer.
PrimsWritten_S1	Number of primitives written to the stream 1 buffer.
PrimsWritten_S2	Number of primitives written to the stream 2 buffer.
PrimsWritten_S3	Number of primitives written to the stream 3 buffer.
PSBusy[7]	The percentage of time the ShaderUnit has pixel shader work to do.
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.
PSInvocations	Number of times a pixel shader was invoked.
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.
PSSALUBusy[1]	The percentage of GPUTime scalar ALU instructions are being processed by the PS.
PSSALUInstCount[1]	Average number of scalar ALU instructions executed in the PS. Affected by flow control.
PSTime	Time pixel shaders are busy in milliseconds.
PSVALUBusy[1]	The percentage of GPUTime vector ALU instructions are being processed by the PS.
PSVALUInstCount[1]	Average number of vector ALU instructions executed in the PS. Affected by flow control.
TessellatorBusy	The percentage of time the tessellation engine is busy.
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.
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.

VSBusy[7]	The percentage of time the ShaderUnit has vertex shader work to do.
VSInvocations	Number of times a vertex shader was invoked.
VSSALUBusy[1]	The percentage of GPUTime scalar ALU instructions are being processed by the VS.
VSSALUInstCount[1]	Average number of scalar ALU instructions executed in the VS. Affected by flow control.
VSTime	Time vertex shaders are busy in milliseconds.
VSVSALUBusy[1]	The percentage of GPUTime vector ALU instructions are being processed by the VS.
VSVALUInstCount[1]	Average number of vector ALU instructions executed in the VS. Affected by 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.

[1] Available on 2nd generation Graphics Core Next based AMD Radeon™ Graphics Cards or newer

6. API Functions

Begin Sampling Pass

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

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

`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`: Sample could not be started due to internal error.

`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 Profile Session with the Currently Enabled Set of Counters

Syntax `GPA_Status GPA_BeginSession( gpa_uint32* pSessionID )`

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 *`pSessionID`* 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 supplied as the `pSessionID` 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 the Counters in the Currently Active Context

Syntax `GPA_Status GPA_CloseContext()`

Description Counters must be reopened with `GPA_OpenContext` before using 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 `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 `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.

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`: Counters cannot be disabled if a session is active.

`GPA_STATUS_OK`: On success.

Disable a Specific Counter

Syntax `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 be enabled explicitly.

Parameters

index	The index of the counter to disable. 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`: Counters 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 Specific Counter Using the Counter Name (Case Insensitive)

Syntax `GPA_Status GPA_DisableCounterStr( const char* pCounter )`

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

Parameters

`pCounter`	The name of the counter to disable.

Returns

`GPA_STATUS_ERROR_NULL_POINTER`: A null pointer was supplied as the `pCounter` parameter.

`GPA_STATUS_ERROR_CANNOT_CHANGE_COUNTERS_WHEN_SAMPLING`: Counters 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 index does identify an available counter, but the counter was not previously enabled, so it cannot be disabled.

`GPA_STATUS_OK`: On success.

Enable All Counters

Syntax `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`: Counters cannot be disabled if a session is active.

`GPA_STATUS_OK`: On success.

Enable a Specific Counter

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

`GPA_STATUS_ERROR_CANNOT_CHANGE_COUNTERS_WHEN_SAMPLING`: Counters cannot be enabled if a session is active.

`GPA_STATUS_ERROR_ALREADY_ENABLED`: The specified counter is already enabled.

`GPA_STATUS_OK`: On success.

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

Syntax

```
GPA_Status GPA_EnableCounterStr( const char* pCounter )
```

Description

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

Parameters

`pCounter` The name of the counter to enable.

Returns

`GPA_STATUS_ERROR_NULL_POINTER`: A null pointer was supplied as the `pCounter` parameter.

`GPA_STATUS_ERROR_CANNOT_CHANGE_COUNTERS_WHEN_SAMPLING`: Counters 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 `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 `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 trying 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 Profiling Session

Syntax `GPA_Status GPA_EndSession()`

Description Ends the profiling 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_SAMPLE_NOT_STARTED`: `GPA_BeginSample` must be called before trying to end a sample.

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

`GPA_STATUS_OK`: On success.

Get the Counter Data Type of the Specified Counter

Syntax

```
GPA_Status GPA_GetCounterDataType(
                                    gpa_uint32 index,
                                    GPA_Type* pCounterDataType )
```

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.
`pCounterDataType`	The value that holds the data type 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`: A null pointer was supplied as the `pCounterDataType` parameter.

`GPA_STATUS_OK`: On success.

Get Description of the Specified Counter

Syntax

```
GPA_Status GPA_GetCounterDescription(
                                        gpa_uint32 index,
                                        GPA_Type* pDescription )
```

Description Retrieves a description of the counter at the supplied index.

Parameters

`index`	The index of the counter. Must lie between 0 and (`GPA_GetNumCounters` result - 1), inclusive.
`pDescription`	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`: A null pointer was supplied as the `pDescription` parameter.

`GPA_STATUS_OK`: On success.

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

Syntax

```
GPA_Status GPA_GetCounterIndex(
                                    const char* pCounter,
                                    gpa_uint32* pIndex )
```

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

Parameters

`pCounter` The name of the counter to get the index for.

`pIndex` Holds the index of the requested counter 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`: A null pointer was supplied as the `pCounter` or `pIndex` parameter.

`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

```
GPA_Status GPA_GetCounterName(
                                    gpa_uint32 index,
                                    const char** ppName )
```

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.

`ppName` 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_NULL_POINTER`: A null pointer was supplied as the `ppName` parameter.

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

`GPA_STATUS_OK`: On success.

Get the Usage of a Specific Counter

Syntax

```
GPA_Status GPA_GetCounterUsageType(
                                    gpa_uint32 index,
                                    GPA_Usage_Type* pCounterUsageType )
```

Description Retrieves the usage type (milliseconds, percentage, etc) of the counter at the supplied index.

Parameters

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

`pCounterUsageType` The value that holds the usage 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`: A null pointer was supplied as the `pCounterUsageType` parameter.

`GPA_STATUS_OK`: On success.

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

Syntax

```
GPA_Status GPA_GetDataTypeAsStr(
                                    GPA_Type counterDataType,
                                    const char** ppTypeStr )
```

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.
`ppTypeStr`	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`: A null pointer was supplied as the `ppTypeStr` parameter.

`GPA_STATUS_OK`: On success.

Get the Device Description

Syntax

```
GPA_Status GPA_GetDeviceDesc( const char** ppDesc )
```

Description Gets the device description associated with the current context.

Parameters

`ppDesc`	Address of the variable that is set to the device description 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`: A null pointer was supplied as the `ppDesc` parameter.

`GPA_STATUS_ERROR_NOT_FOUND`: The device description is unavailable.

`GPA_STATUS_OK`: On success.

Get the Device Identifier

Syntax `GPA_Status GPA_GetDeviceID( gpa_uint32* pDeviceID )`

Description Gets the device identifier associated with the current context.

Parameters

`pDeviceID`	Address of the variable that is set to the device identifier 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`: A null pointer was supplied as the `pDeviceID` parameter.

`GPA_STATUS_ERROR_NOT_FOUND`: The device identifier is unavailable.

`GPA_STATUS_OK`: On success.

Get the Number of Enabled Counters

Syntax `GPA_Status GPA_GetEnabledCount( gpa_uint32* pCount )`

Description Retrieves the number of enabled counters.

Parameters

`pCount`	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`: A null pointer was supplied as the `pCount` parameter.

`GPA_STATUS_OK`: On success.

Get the Index for an Enabled Counter

Syntax

```
GPA_Status GPA_GetEnabledIndex(
                                    gpa_uint32 enabledNumber,
                                    gpa_uint32* pEnabledCounterIndex )
```

Description For example, if `GPA_GetEnabledCount` returns 3, then call this function with `enabledNumber` equal to 0 to get the counter index of the first enabled counter. The returned counter index can then be used to look up the counter name, data type, usage, etc.

Parameters

`enabledNumber`	The number of the enabled counter for which to get the counter index. Must lie between 0 and (`GPA_GetEnabledCount` result - 1), inclusive.
`pEnabledCounterIndex`	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`: A null pointer was supplied as the `pEnabledCounterIndex` parameter.

`GPA_STATUS_ERROR_INDEX_OUT_OF_RANGE`: The supplied `enabledNumber` is outside the range of enabled counters.

`GPA_STATUS_OK`: On success.

Get the Number of Counters Available

Syntax

```
GPA_Status GPA_GetNumCounters( gpa_uint32* pCount )
```

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

`pCount`	Address of the variable that is set to the number of 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`: A null pointer was supplied as the `pCount` parameter.

`GPA_STATUS_OK`: On success.

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

Syntax

```
GPA_Status GPA_GetPassCount( gpa_uint32* pNumPasses )
```

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

`pNumPasses` Holds the number of required passes 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`: A null pointer was supplied as the `pNumPasses` parameter.

`GPA_STATUS_OK`: On success.

Get the Number of Samples a Specified Session Contains

Syntax

```
GPA_Status GPA_GetSampleCount(
                                gpa_uint32 sessionID,
                                gpa_uint32* pSamples )
```

Description This is useful if samples are conditionally created and a count is not maintained by the application.

Parameters

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

`pSamples` 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`: A null pointer was supplied as the `pSamples` parameter.

`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

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

Description

This function blocks further processing until the result is available. Use `GPA_IsSampleReady` to test for result availability without blocking.

Parameters

`sessionID`	The session identifier with the sample for which to retrieve the result.
`sampleID`	The sample identifier for which to get the result.
`counterIndex`	The counter index for which to get the result.
`pResult`	Holds 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_INDEX_OUT_OF_RANGE`: The supplied `counterIndex` does not identify an available counter.

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

`GPA_STATUS_ERROR_NULL_POINTER`: A null pointer was supplied as the `pResult` parameter.

`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

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

Description

This function blocks further processing until the result is available. Use `GPA_IsSampleReady` to test for result availability without blocking.

Parameters

`sessionID`	The session identifier with the sample for which to retrieve the result.
`sampleID`	The sample identifier for which to get the result.
`counterIndex`	The counter index for which to get the result.
`pResult`	Holds 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_INDEX_OUT_OF_RANGE`: The supplied `counterIndex` does not identify an available counter.

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

`GPA_STATUS_ERROR_NULL_POINTER`: A null pointer was supplied as the `pResult` parameter.

`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

```
GPA_Status GPA_GetSampleUInt32(
                                    gpa_uint32 sessionID,
                                    gpa_uint32 sampleID,
                                    gpa_uint32 counterIndex,
                                    gpa_uint32* pResult )
```

Description This function blocks further processing until the result is available. Use `GPA_IsSampleReady` to test for result availability without blocking.

Parameters

`sessionID`	The session identifier with the sample for which to retrieve the result.
`sampleID`	The sample identifier for which to get the result.
`counterIndex`	The counter index for which to get the result.
`pResult`	Holds 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_INDEX_OUT_OF_RANGE`: The supplied `counterIndex` does not identify an available counter.

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

`GPA_STATUS_ERROR_NULL_POINTER`: A null pointer was supplied as the `pResult` parameter.

`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

```
GPA_Status GPA_GetSampleUInt64(
                                    gpa_uint32 sessionID,
                                    gpa_uint32 sampleID,
                                    gpa_uint32 counterIndex,
                                    gpa_uint64* pResult )
```

Description

This function blocks further processing until the result is available. Use `GPA_IsSampleReady` to test for result availability without blocking.

Parameters

`sessionID`	The session identifier with the sample for which to retrieve the result.
`sampleID`	The sample identifier for which to get the result.
`counterIndex`	The counter index for which to get the result.
`pResult`	Holds 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_INDEX_OUT_OF_RANGE`: The supplied `counterIndex` does not identify an available counter.

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

`GPA_STATUS_ERROR_NULL_POINTER`: A null pointer was supplied as the `pResult` parameter.

`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.

Gets a String Version of the Status Value

Syntax

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

Description This is useful for converting 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 Usage Type

Syntax

```
GPA_Status GPA_GetUsageTypeAsStr(
                                    GPA_Usage_Type counterUsageType,
                                    const char** ppTypeStr )
```

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.

`ppTypeStr` 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`: A null pointer was supplied as the `ppTypeStr` parameter.

`GPA_STATUS_OK`: On success.

Checks if a Counter is Enabled

Syntax `GPA_Status GPA_IsCounterEnabled( gpa_uint32 counterIndex )`

Description Indicates if the specified counter is enabled.

Parameters

`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 `GPA_Status GPA_Initialize()`

Description For DirectX 11, in order to access the counters, UAC may also need to be disabled and / or your application must be set to run with administrator privileges.

Returns

`GPA_STATUS_FAILED`: If an internal error occurred. UAC or lack of administrator privileges may be the cause.

`GPA_STATUS_OK`: On success.

Determines if an Individual Sample Result is Available

Syntax

```
GPA_Status GPA_IsSampleReady(
                                    bool* pReadyResult,
                                    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

`pReadyResult`	The value that contains the result of the ready sample. True if ready.
`sessionID`	The session containing the sample.
`sampleID`	The sample identifier 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 `pReadyResult` 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, but there shouldn't be.

`GPA_STATUS_OK`: On success.

Determines if All Samples Within a Session are Available

Syntax

```
GPA_Status GPA_IsSessionReady(
                                bool* pReadyResult,
                                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 instead.

Parameters

`pReadyResult` The value that indicates if the session is 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 `pReadyResult` 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

```
GPA_Status GPA_OpenContext( void* pContext )
```

Description

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

Parameters

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

Returns

`GPA_STATUS_ERROR_NULL_POINTER`: The supplied `pContext` 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 or driver is not supported by GPU Performance API. This may also be returned if `GPA_Initialize()` was not called before the supplied context was created.

`GPA_STATUS_OK`: On success.

Register Optional Callback for Additional Information

Syntax

```
GPA_Status GPA_RegisterLoggingCallback(
                    GPA_Logging_Type loggingType,
                    GPA_LoggingCallbackPtrType pCallbackFuncPtr )
```

Description

Registers an optional callback function that will be used to output additional information about errors, messages, and API usage (trace). Only one callback function can be registered, so the callback implementation should be able to handle the different types of messages. A parameter to the callback function will indicate the message type being received. Messages will not contain a newline character at the end of the message.

Parameters

`loggingType` Identifies the type of messages for which to receive callbacks.

`pCallbackFuncPtr` Pointer to the callback function.

Returns

`GPA_STATUS_ERROR_NULL_POINTER`: The supplied `pCallbackFuncPtr` parameter is `NULL` and `loggingType` is not `GPA_LOGGING_NONE`.

`GPA_STATUS_OK`: On success. Also, if you register to receive messages, a message will be output to indicate that the `"Logging callback registered successfully."`

Select a Previously-opened Context as the Active Context

Syntax

```
GPA_Status GPA_SelectContext( void* pContext )
```

Description

The selected context must have previously been opened with a call to `GPA_OpenContext()`.

Parameters

`pContext` The context to select. The same value that was passed to an earlier `GPA_OpenContext()` call.

Returns

`GPA_STATUS_ERROR_NULL_POINTER`: The supplied `pContext` parameter is `NULL`.

`GPA_STATUS_ERROR_COUNTERS_NOT_OPEN`: The supplied `pContext` parameter does not represent a previously-opened context.

`GPA_STATUS_OK`: On success.

7. Utility Function

The following is an example of how to read the data back from the completed session and 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

For GPU Developer Tools:
URL: **http://www.gpuopen.com**
Forum: **http://devgurus.amd.com**