User Guide

AMD GPU Performance API

Contents

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 AMD Radeon™ 2000 series and newer graphics cards and APUs
- Supports OpenCL on AMD Radeon™ 4000 series and newer graphics cards and APUs
- Supports Microsoft Windows as a dynamically loaded library.
- Supports Linux as a shared-object library:
    - Targeting Ubuntu (12.04 and later) and RHEL (7 and later), distributions
    - OpenCL and OpenGL only
- Provides derived counters based on raw HW performance counters.
- Manages memory automatically – no allocations required.
- Requires AMD Catalyst™ driver 14.12 or later.

2. Usage

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

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 libGPUPerfAPICL.so or libGPUPerfAPIGL.so

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 callbackFuncPtr );
```

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.

```
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 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* 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:

```
GPA_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, the handle to the GL context should be passed 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 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	CSBusy CSTime DepthStencilTestBusy DSBusy DSTime GPUTime GPUBusy GSBusy GSTime HSBusy HSTime InterpBusy PrimitiveAssemblyBusy PSBusy PSTime ShaderBusy ShaderBusyCS ShaderBusyDS ShaderBusyGS ShaderBusyHS ShaderBusyPS ShaderBusyVS TessellatorBusy TexUnitBusy VSBusy VSTime
VertexShader	VertexMemFetched VertexMemFetchedCost

	VSALUBusy VSALUEfficiency VSALUInstCount VSALUTexRatio VSSALUBusy VSSALUInstCount VSTexBusy VSTexInstCount VSVALUBusy VSVALUInstCount VSVerticesIn
HullShader[2]	HSALUBusy HSALUEfficiency HSALUInstCount HSALUTexRatio HSTexBusy HSTexInstCount HSPatches HSSALUBusy HSSALUInstCount HSVALUBusy HSVALUInstCount
GeometryShader	GSALUBusy GSALUEfficiency GSALUInstCount GSALUTexRatio GSExportPct GSPrimsIn GSSALUBusy GSSALUInstCount GSTexBusy GSTexInstCount GSVALUBusy GSVALUInstCount GSVerticesOut
PrimitiveAssembly	ClippedPrims CulledPrims PAPixelsPerTriangle PAStalledOnRasterizer PrimitivesIn
DomainShader[2]	DSALUBusy DSALUEfficiency DSALUInstCount DSALUTexRatio DSTexBusy DSTexInstCount

	DSVerticesIn
PixelShader	PSALUBusy PSALUEfficiency PSALUInstCount PSALUTexRatio PSExportStalls PSPixelsIn PSPixelsOut PSSALUBusy PSSALUInstCount PSTexBusy PSTexInstCount PSVALUBusy PSVALUInstCount
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[1]	ALUBusy ALUFetchRatio ALUInsts ALUPacking FetchInsts GDSInsts

	SALUBusy SALUInsts SFetchInsts VALUBusy VALUInsts VALUUtilization VFetchInsts VWriteInsts Wavefronts WriteInsts
ComputeShader[2]	CSALUBusy CSALUFetchRatio CSALUInsts CSALUPacking CSALUStalledByLDS CSCacheHit CSCompletePath CSFastPath CSFetchInsts CSFetchSize CSGDSInsts CSLDSBankConflict CSLDSFetchInsts CSLDSWriteInsts CSMemUnitBusy CSMemUnitStalled CSPathUtilization CSSALUBusy CSSALUInsts CSSFetchInsts CSFetchSize CSGDSInsts CSTexBusy CSThreadGroups CSThreads CSVALUBusy CSVALUInsts CSVALUUtilization CSVFetchInsts CSVWriteInsts CSWavefronts CSWriteInsts CSWriteSize CSWriteUnitStalled
DepthAndStencil	HiZQuadsCulled HiZTilesAccepted

	PostZQuads PostZSamplesFailingS PostZSamplesFailingZ PostZSamplesPassing PreZQuadsCulled PreZSamplesFailingS PreZSamplesFailingZ PreZSamplesPassing PreZTilesDetailCulled ZUnitStalled
ColorBuffer[3]	CBMemRead CBMemWritten CBSlowPixelPct
GlobalMemory[1]	CompletePath FastPath FetchSize FetchUnitBusy FetchUnitStalled CacheHit MemUnitBusy MemUnitStalled PathUtilization WriteSize WriteUnitStalled
LocalMemory[1]	LDSBankConflict LDSFetchInsts LDSInsts LDSWriteInsts
D3D11[4]	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 version of the GPU Performance API
[2] Available only on AMD Radeon™ HD 5000 Series Graphics Cards or newer
[3] Available only on AMD Radeon™ HD 4000 Series Graphics Cards or newer
[4] Exposed only by the DirectX11 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 which APIs contain the counters and the hardware version on which they are available.

OpenCL Counter Descriptions

Counter	**Description**
ALUBusy[1]	The percentage of GPUTime ALU instructions are processed.
ALUFetchRatio[1]	The ratio of ALU to fetch instructions. If the number of fetch instructions is zero, then one will be used instead.
ALUInsts[1]	The average ALU instructions executed per work-item (affected by flow control).
ALUPacking[1]	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 utilization of the processor.
CacheHit	The percentage of fetches from the video memory that hit the data cache. Value range: 0% (no hit) to 100% (optimal).
CompletePath[2]	The total kilobytes written to the global memory through the CompletePath which supports atomics and sub-32 bit

	types (byte, short). This number includes load, store and atomics operations on the buffer. This number may indicate a big performance impact (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[2]	The total kilobytes written to the global memory through the FastPath which only support basic operations: no atomics or sub-32 bit types. This is an optimized path in the hardware.
FetchInsts[1]	The average number of fetch instructions from the 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[1]	The percentage of GPUTime the Fetch unit is active. This is measured with all extra fetches and any cache or memory effects taken into account.
FetchUnitStalled[1]	The percentage of GPUTime the Fetch unit is stalled. Try reducing the number of fetches or reducing the amount per fetch if possible.
GDSInsts[3]	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.
LDSFetchInsts[2]	The average number of fetch instructions from the local memory executed per work-item (affected by flow control).
LDSInsts[3]	The average number of LDS read or LDS write instructions executed per work item (affected by flow control).
LDSWriteInsts[2]	The average number of write instructions to the local memory executed per work-item (affected by flow control).
MemUnitBusy[3]	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[3]	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).
PathUtilization[2]	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 the FastPath.
SALUBusy[3]	The percentage of GPUTime scalar ALU instructions are processed. Value range: 0% (bad) to 100% (optimal).
SALUInsts[3]	The average number of scalar ALU instructions executed per work-item (affected by flow control).
SFetchInsts[3]	The average number of scalar fetch instructions from the video memory executed per work-item (affected by flow control).
VALUBusy[3]	The percentage of GPUTime vector ALU instructions are processed. Value range: 0% (bad) to 100% (optimal).
VALUInsts[3]	The average number of vector ALU instructions executed per work-item (affected by flow control).
VALUUtilization[3]	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[3]	The average number of vector fetch instructions from the video memory executed per work-item (affected by flow control).
VWriteInsts[3]	The average number of vector write instructions to the video memory executed per work-item (affected by flow control).
Wavefronts	Total wavefronts.
WriteInsts[1]	The average number of write instructions to the global memory executed per work-item (affected by flow control).
WriteSize[3]	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 AMD Radeon™ Graphics Cards based on pre-Graphics Core Next
[2] Only available on AMD Radeon™ 5000 and 6000 Series Graphics Cards based on pre-Graphics Core Next
[3] Only available on AMD Radeon™ Graphics Cards based on Graphics Core Next

OpenGL and DirectX Counter Descriptions

Counter	**Description**
CBMemRead[4]	Number of bytes read from the color buffer.
CBMemWritten[4]	Number of bytes written to the color buffer.
CBSlowPixelPct[6]	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.
CSALUBusy[5]	The percentage of GPU Time ALU instructions are processed by the CS.
CSALUFetchRatio[5]	The ratio of ALU to fetch instructions. This can be tuned to match the target hardware.
CSALUInsts[5]	The number of ALU instructions executed in the CS. Affected by the flow control.
CSALUPacking[5]	ALU vector packing efficiency. Values below 70 percent indicate that ALU dependency chains may prevent full use of the processor.
CSALUStalledByLDS[6]	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[7]	The percentage of time the ShaderUnit has compute shader work to do.
CSCacheHit[6]	The percentage of fetches from the global memory that hit the texture cache.
CSCompletePath[5]	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.
CSFastPath[5]	The total bytes written through the FastPath (no atomics, 32-bit type only). This includes extra bytes needed for addressing.
CSFetchInsts[5]	Average number of fetch instructions executed in the CS per execution. Affected by the flow control.
CSFetchSize[7]	The total kilobytes fetched from the video memory. This is measured with all extra fetches and any cache or memory effects taken into account.
CSGDSInsts[7]	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[6]	The percentage of GPUTime the LDS is stalled by bank conflicts.
CSLDSFetchInsts[5]	The average number of LDS fetch instructions executed per work-item (affected by flow control). This counter is a subset of the VALUInsts counter.
CSLDSInsts[7]	The average number of LDS read/write instructions executed per work-item (affected by flow control).

CSLDSWriteInsts[5]	The average number of LDS write instructions executed per work-item (affected by flow control). This counter is a subset of the VALUInsts counter.
CSMemUnitBusy[7]	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[7]	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).
CSPathUtilization[5]	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.
CSSALUBusy[7]	The percentage of GPUTime scalar ALU instructions are processed. Value range: 0% (bad) to 100% (optimal).
CSSALUInsts[7]	The average number of scalar ALU instructions executed per work-item (affected by flow control).
CSSFetchInsts[7]	The average number of scalar fetch instructions from the video memory executed per work-item (affected by flow control).
CSTexBusy[5]	The percentage of GPUTime texture instructions are processed by the CS.
CSThreadGroups[7]	Total number of thread groups.
CSThreads[6]	The number of CS threads processed by the hardware.
CSTime[7]	Time compute shaders are busy in milliseconds.
CSVALUBusy[7]	The percentage of GPUTime vector ALU instructions are processed. Value range: 0% (bad) to 100% (optimal).
CSVALUInsts[7]	The average number of vector ALU instructions executed per work-item (affected by flow control).
CSVALUUtilization[7]	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[7]	The average number of vector fetch instructions from the video memory executed per work-item (affected by flow control).
CSVWriteInsts[7]	The average number of vector write instructions to the video memory executed per work-item (affected by flow control).
CSWavefronts[6]	The total number of wavefronts used for the CS.
CSWriteInsts[5]	The average number of write instructions executed in compute shader per execution. Affected by flow control.
CSWriteSize[7]	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[7]	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.
DSALUBusy[5]	The percentage of GPUTime ALU instructions are processed by the DS.
DSALUEfficiency[5]	ALU vector packing efficiency. Values below 70 percent indicate that ALU dependency chains may be prevent full use of the processor.
DSALUInstCount[5]	Average number of ALU instructions executed in the DS. Affected by flow control.
DSALUTexRatio[5]	The ratio of ALU to texture instructions. This can be tuned to match the target hardware.
DSBusy[7]	The percentage of time the ShaderUnit has domain shader work to do.
DSInvocations	Number of times a domain shader was invoked.
DSSALUBusy[8]	The percentage of GPUTime scalar ALU instructions are being processed by the DS.
DSSALUInstCount[8]	Average number of scalar ALU instructions executed in the DS. Affected by flow control.
DSTexBusy[5]	The percentage of GPUTime texture instructions are processed by the DS.
DSTexInstCount[5]	Average number of texture instructions executed in DS. Affected by the flow control.
DSTime[7]	Time domain shaders are busy in milliseconds.
DSVALUBusy[8]	The percentage of GPUTime vector ALU instructions are being processed by the DS.
DSVALUInstCount[8]	Average number of vector ALU instructions executed in the DS. Affected by flow control.
DSVerticesIn[6]	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.
GSALUBusy[2]	The percentage of GPUTime ALU instructions are processed by the GS.
GSALUEfficiency[2]	ALU vector packing efficiency. Values below 70 percent indicate that ALU dependency chains may prevent full use of the processor.
GSALUInstCount[2]	Average number of ALU instructions executed in GS.

	Affected by the flow control.
GSALUTexRatio[2]	The ratio of ALU to texture instructions in the GS. This can be tuned appropriately to match the target hardware.
GSBusy[7]	The percentage of time the ShaderUnit has geometry shader work to do.
GSExportPct[2]	The percentage of GS work that is related to exporting primitives.
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[8]	The percentage of GPUTime scalar ALU instructions are being processed by the GS.
GSSALUInstCount[8]	Average number of scalar ALU instructions executed in the GS. Affected by flow control.
GSTexBusy[2]	The percentage of GPUTime texture instructions are processed by the GS.
GSTexInstCount[2]	Average number of texture instructions executed in GS. Affected by the flow control.
GSTime[7]	Time geometry shaders are busy in milliseconds.
GSVALUBusy[8]	The percentage of GPUTime vector ALU instructions are being processed by the GS.
GSVALUInstCount[8]	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.
HSALUBusy[5]	The percentage of GPUTime ALU instructions processed by the HS.
HSALUEfficiency[5]	ALU vector packing efficiency. Values below 70 percent indicate that ALU dependency chains may prevent full use of the processor.
HSALUInstCount[5]	Average number of ALU instructions executed in the HS. Affected by the flow control.
HSALUTexRatio[5]	The ratio of ALU to texture instructions. This can be tuned to match the target hardware.
HSBusy[7]	The percentage of time the ShaderUnit has hull shader work to do.
HSInvocations	Number of times a hull shader was invoked.
HSPatches[6]	The number of patches processed by the HS.
HSSALUBusy[8]	The percentage of GPUTime scalar ALU instructions are being processed by the HS.

HSSALUInstCount[8]	Average number of scalar ALU instructions executed in the HS. Affected by flow control.
HSTexBusy[5]	The percentage of GPUTime texture instructions are processed by the HS.
HSTexInstCount[5]	Average number of texture instructions executed in HS. Affected by the flow control.
HSTime[7]	Time hull shaders are busy in milliseconds.
HSVALUBusy[8]	The percentage of GPUTime vector ALU instructions are being processed by the HS.
HSVALUInstCount[8]	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.
InterpBusy[1]	Percentage of GPUTime that the interpolator is busy.
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.
PAPixelsPerTriangle[5]	The ratio of rasterized pixels to the number of triangles after culling. This does not account for triangles generated due to clipping.
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.
Pct64SlowTexels[3]	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.
Pct128SlowTexels[2]	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[2]	Percentage of texture fetches from compressed textures.
PctDepthTexels[2]	Percentage of texture fetches from depth textures.
PctInterlacedTexels[2]	Percentage of texture fetches from interlaced textures.
PctTex1D[2]	Percentage of texture fetches from a 1D texture.
PctTex1DArray[2]	Percentage of texture fetches from a 1D texture array.
PctTex2D[2]	Percentage of texture fetches from a 2D texture.
PctTex2DArray[2]	Percentage of texture fetches from a 2D texture array.
PctTex2DMSAA[2]	Percentage of texture fetches from a 2D MSAA texture.
PctTex2DMSAAArray[2]	Percentage of texture fetches from a 2D MSAA texture

	array.
PctTex3D[2]	Percentage of texture fetches from a 3D texture.
PctTexCube[2]	Percentage of texture fetches from a cube map.
PctTexCubeArray[3]	Percentage of texture fetches from a cube map array.
PctUncompressedTexels[2]	Percentage of texture fetches from uncompressed textures. Does not include depth or interlaced textures.
PctVertex64SlowTexels[3]	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[3]	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[3]	Percentage of texture fetches from vertex textures.
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.
PreZTilesDetailedCulled[4]	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.
PSALUBusy[2]	The percentage of GPUTime ALU instructions are processed by the PS.
PSALUEfficiency[2]	ALU vector packing efficiency. Values below 70 percent indicate that ALU dependency chains may prevent full use of the processor.
PSALUInstCount[2]	Average number of ALU instructions executed in PS. Affected by the flow control.
PSALUTexRatio[2]	The ratio of ALU to texture instructions in the PS. This can be tuned appropriately to match the target hardware.
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.
PSPixelsIn[2]	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.
PSSALUBusy[8]	The percentage of GPUTime scalar ALU instructions are being processed by the PS.
PSSALUInstCount[8]	Average number of scalar ALU instructions executed in the PS. Affected by flow control.
PSTexBusy[2]	The percentage of GPUTime texture instructions are processed by the PS.
PSTexInstCount[2]	Average number of texture instructions executed in the PS. Affected by the flow control.
PSTime[7]	Time pixel shaders are busy in milliseconds.
PSVALUBusy[8]	The percentage of GPUTime vector ALU instructions are being processed by the PS.
PSVALUInstCount[8]	Average number of vector ALU instructions executed in the PS. Affected by flow control.
ShaderBusy[2]	The percentage of GPUTime that the shader unit is busy.
ShaderBusyCS[5]	The percentage of work done by shader units for CS.
ShaderBusyDS[5]	The percentage of work done by shader units for DS.
ShaderBusyGS[2]	The percentage of work done by shader units for GS.
ShaderBusyHS[5]	The percentage of work done by shader units for HS.
ShaderBusyPS[2]	The percentage of work done by shader units for PS.
ShaderBusyVS[2]	The percentage of work done by shader units for VS.
TessellatorBusy[6]	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.
TexCacheStalled[2]	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[2]	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[2]	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[2]	Texture memory read in bytes.
TexMissRate[2]	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[2]	Number of bytes read from memory due to vertex cache miss.
VertexMemFetchedCost[3]	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.
VSALUBusy[2]	The percentage of GPUTime ALU instructions are processed by the VS.
VSALUEfficiency[2]	ALU vector packing efficiency. Values below 70 percent indicate that ALU dependency chains may prevent full use of the processor.
VSALUInstCount[2]	Average number of ALU instructions executed in the VS. Affected by the flow control.
VSALUTexRatio[2]	The ratio of ALU to texture instructions in the VS. This can be tuned appropriately to match the target hardware.
VSBusy[7]	The percentage of time the ShaderUnit has vertex shader work to do.
VSInvocations	Number of times a vertex shader was invoked.

VSSALUBusy[8]	The percentage of GPUTime scalar ALU instructions are being processed by the VS.
VSSALUInstCount[8]	Average number of scalar ALU instructions executed in the VS. Affected by flow control.
VSTexBusy[2]	The percentage of GPUTime texture instructions are processed by the VS.
VSTexInstCount[2]	Average number of texture instructions executed in VS. Affected by the flow control.
VSTime[7]	Time vertex shaders are busy in milliseconds.
VSVSALUBusy[8]	The percentage of GPUTime vector ALU instructions are being processed by the VS.
VSVALUInstCount[8]	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 AMD Radeon™ HD 2000-4000 Series Graphics Cards
[2] Available on AMD Radeon™ Graphics Cards based on pre-Graphics Core Next
[3] Available on AMD Radeon™ HD 4000-6000 Series Graphics Cards based on pre-Graphics Core Next
[4] Available on AMD Radeon™ HD 4000 Series Graphics Cards or newer
[5] Available on AMD Radeon™ HD 5000-6000 Series Graphics Cards based on pre-Graphics Core Next
[6] Available on AMD Radeon™ HD 5000 Series Graphics Cards or newer
[7] Available on AMD Radeon™ Graphics Cards based on Graphics Core Next
[8] 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* 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 supplied 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 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* counter )`
Description	Subsequent sampling sessions do not provide values for any disabled counters. Initially, all counters are disabled and must be enabled explicitly.
Parameters	`counter` The name of the counter to disable.
Returns	`GPA_STATUS_ERROR_NULL_POINTER`: A null pointer was supplied as the `counter` 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* 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_NULL_POINTER`: A null pointer was supplied as the `counter` 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* 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 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 `counterDataType` parameter.

`GPA_STATUS_OK`: On success.

Get Description of the Specified Counter

Syntax

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

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

`GPA_STATUS_OK`: On success.

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

Syntax

```
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`	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 `description` 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** 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_NULL_POINTER`: A null pointer was supplied as the `description` 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* counterUsageType )`	
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.
	`counterUsageType`	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 `counterUsageType` 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** 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`: A null pointer was supplied as the `typeStr` parameter.

`GPA_STATUS_OK`: On success.

Get the Number of Enabled Counters

Syntax

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

`GPA_STATUS_OK`: On success.

Get the Index for an Enabled Counter

Syntax	`GPA_Status GPA_GetEnabledIndex( gpa_uint32 enabledNumber, gpa_uint32* enabledCounterIndex )`
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.
`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`: A null pointer was supplied as the `enabledCounterIndex` 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* 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 number of available counters 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 `count` parameter.

`GPA_STATUS_OK`: On success.

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

Syntax	`GPA_Status GPA_GetNumCounters( 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` 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 `numPasses` 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* samples )`
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. `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`: A null pointer was supplied as the `samples` 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* result )
```

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.
`result`	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 `result` 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* result )
```

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.
`result`	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 `result` 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* result )
```

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.
`result`	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 `result` 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* result )
```

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.
`result`	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 `result` 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** typeStr )
```

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.
`typeStr`	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 `typeStr` 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 10 or 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* 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.
`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 `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, 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* 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 instead.

Parameters

`readyResult`	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 `readyResult` parameter is null.

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

`GPA_STATUS_OK`: On success.

Register Optional Callback for Additional Information

Syntax	`GPA_Status GPA_RegisterLoggingCallback( GPA_Logging_Type loggingType, GPA_LoggingCallbackPtrType callbackFuncPtr )`	
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.
	`callbackFuncPtr`	Pointer to the callback function.
Returns	`GPA_STATUS_ERROR_NULL_POINTER`: The supplied `callbackFuncPtr` 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."`	

Open the Counters in the Specified Context

Syntax	`GPA_Status GPA_OpenContext( void* context )`
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	`context` 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 `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 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.

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://developer.amd.com/tools-and-sdks** **Forum: http://devgurus.amd.com**