## Static Analyzer

In Analyze mode, you can compile shaders and kernels for a variety of AMD GPUs and APUs, independent from the GPU/APU that is physically installed on your system, and generate AMD ISA, intermediate language and performance statistics for each target platform. CodeXL Analyzer supports the following inputs:

* OpenCL kernels
* DirectX shaders
* OpenGL and Vulkan programs

**[Switching to Analyze mode](#_Switching_to_Analyze)**

[**Creating a new project for Analysis**](#_Creating_a_new)

[**Working with the new CodeXL Analyzer Explorer Tree**](#_Working_with_the)

[**Working with Programs**](#_Working_with_Programs)

[**Working with Folders**](#_Working_with_Folders)

[**Build Options- Defining OpenCL and DirectX build options**](#_Build_Options-_Defining)

[**Output Tab**](#_Output_Tab)

[**Kernel Statistics Tab**](#_Kernels_Statistics_Tab)

[**Shader Statistics Tab**](#_Shaders_Statistics_Tab)

**[Viewing compilation output: IL and ISA](#_Viewing_compilation_output:_1)**

[**Export binaries**](#_Export_binaries)

[**Remove items from Project**](#_Remove_from_Project)

**[Static Analyze Toolbar – for OpenCL source files](#_Static_Analyze_Toolbar)**

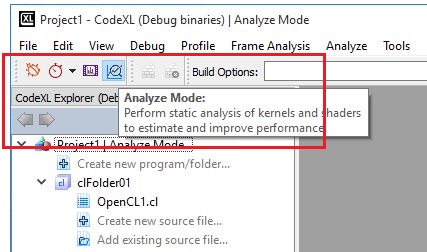
**[Static Analyze Toolbar – for DirectX source files](#_CodeXLAnalyzer_Command_Line)**

**[CodeXLAnalyzer Command Line Interface](#_CodeXLAnalyzer_Command_Line_1)**

### Switching to Analyze mode

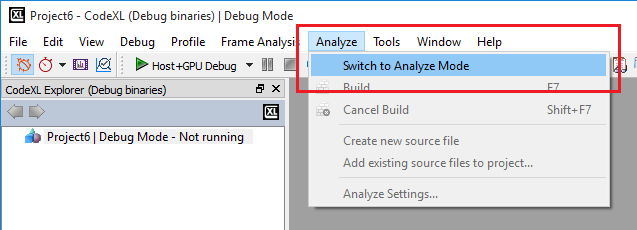
Option 1- Analyze mode button:

Click on the Analyze Mode button in the CodeXL Mode toolbar:



Option 2- Main menu:

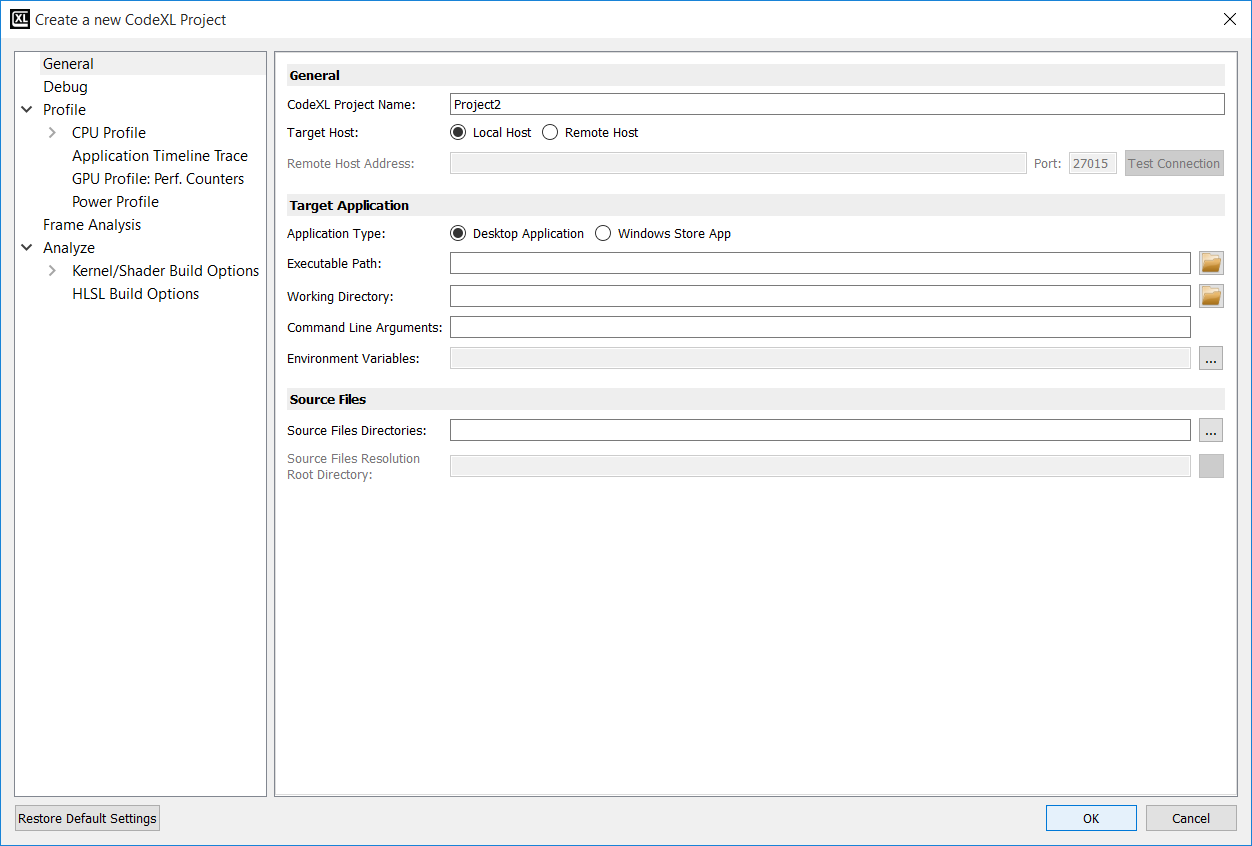
Open the Analyze menu from menu bar and select the ‘Switch to Analyze Mode’ command:



After you switch to Analyze mode, you can also create a new project, open a previously saved project, or load the Teapot or Matrix Multiply samples.

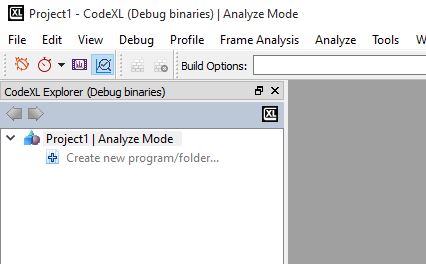
### Creating a new project for Analysis

Click on the “File->Create Project”, or use the Ctrl+N shortcut. The following CodeXL Project Settings dialog will appear:



Rename the project, and click on the OK.

After the new project has been created, the CodeXL Analyzer Explorer Tree should appear in the left pane:



### Working with the new CodeXL Analyzer Explorer Tree

If you are familiar with the former versions of the Analyzer, you probably noticed that the tree has a different structure than the one used in previous versions. Let’s examine the structure of the new CodeXL Analyzer Explorer:

1. **Programs and Folders:** before describing how to technically create Programs and Folders, let’s first discuss what those objects are, and why they can be useful.
2. **Programs (OpenGL, Vulkan):**

As of version 2.0, CodeXL can compile and link together multiple source files for OpenGL and Vulkan. This is especially important when different shaders have mutual impact on one another’s ISA and performance statistics. To provide that type of support, CodeXL Analyzer introduced the concept of a Program. There are two types of Programs in CodeXL 2.0:

* Rendering Programs
* Compute Programs

A Rendering Program represents a graphics pipeline, and can have a single shader attached to each of its stages:

* Vertex
* Tessellation Control
* Tessellation Evaluation
* Geometry
* Fragment

A Compute Program represents a compute pipeline, and can have a single compute shader attached to its single stage.

When you build a program that has multiple shaders attached to it, all shaders are being compiled and linked together. This way, you get more accurate ISA and performance statistics than those generated using previous versions of CodeXL.

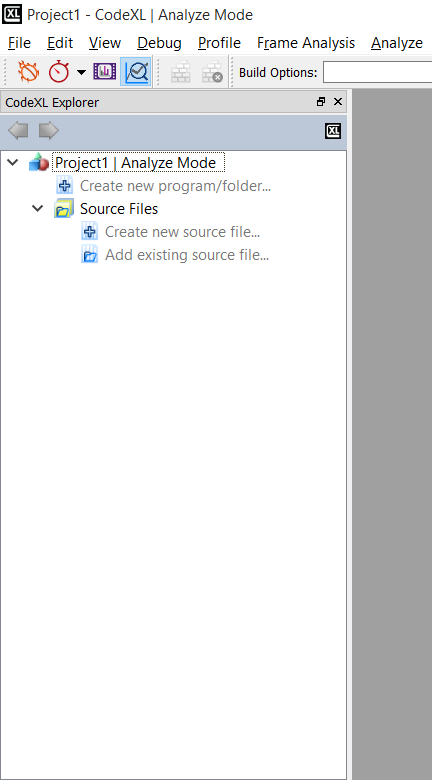
1. **Folders (OpenCL, DirectX):**

Folders are logical containers of source files. When you build a folder that has multiple source files attached to it, the source files are simply being built one after the other. Unlike programs, there is no kind of interdependency between the source files in a given folder: when a folder is being built, each source file is being compiled independently. Folders can be used to organize the project, by serving as a logical separator. They can also be used to ease the process of comparing build results, since now the build results are being maintained per-folder: you can create two different Folders, each containing the same source files, but have a different configuration (for example, create two DirectX Folders, each with a different shader model). After building the two Folders, you can toggle between the performance statistics of the two Folders to see the differences.

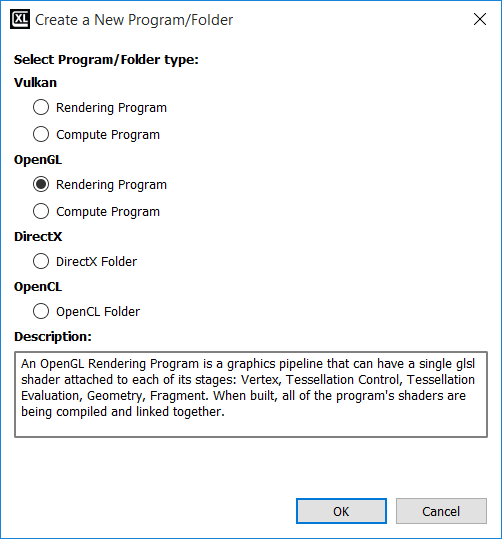
You may ask yourself why CodeXL does not support the concept of DirectX Programs, just like it does for OpenGL and Vulkan. This is a good point. Supporting DirectX Programs is at a high priority in the Analyzer’s roadmap, and we will do our best to add that feature in the upcoming versions of the product.

**Creating a new Program or Folder**

To create a new Program or a Folder, double-click on the “Create new program/folder” item in CodeXL Analyzer Explorer Tree:

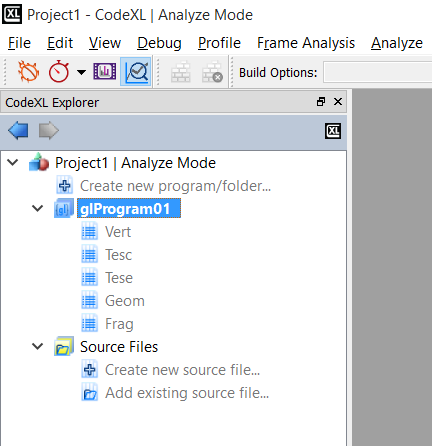


Then, the following dialog would pop-up:



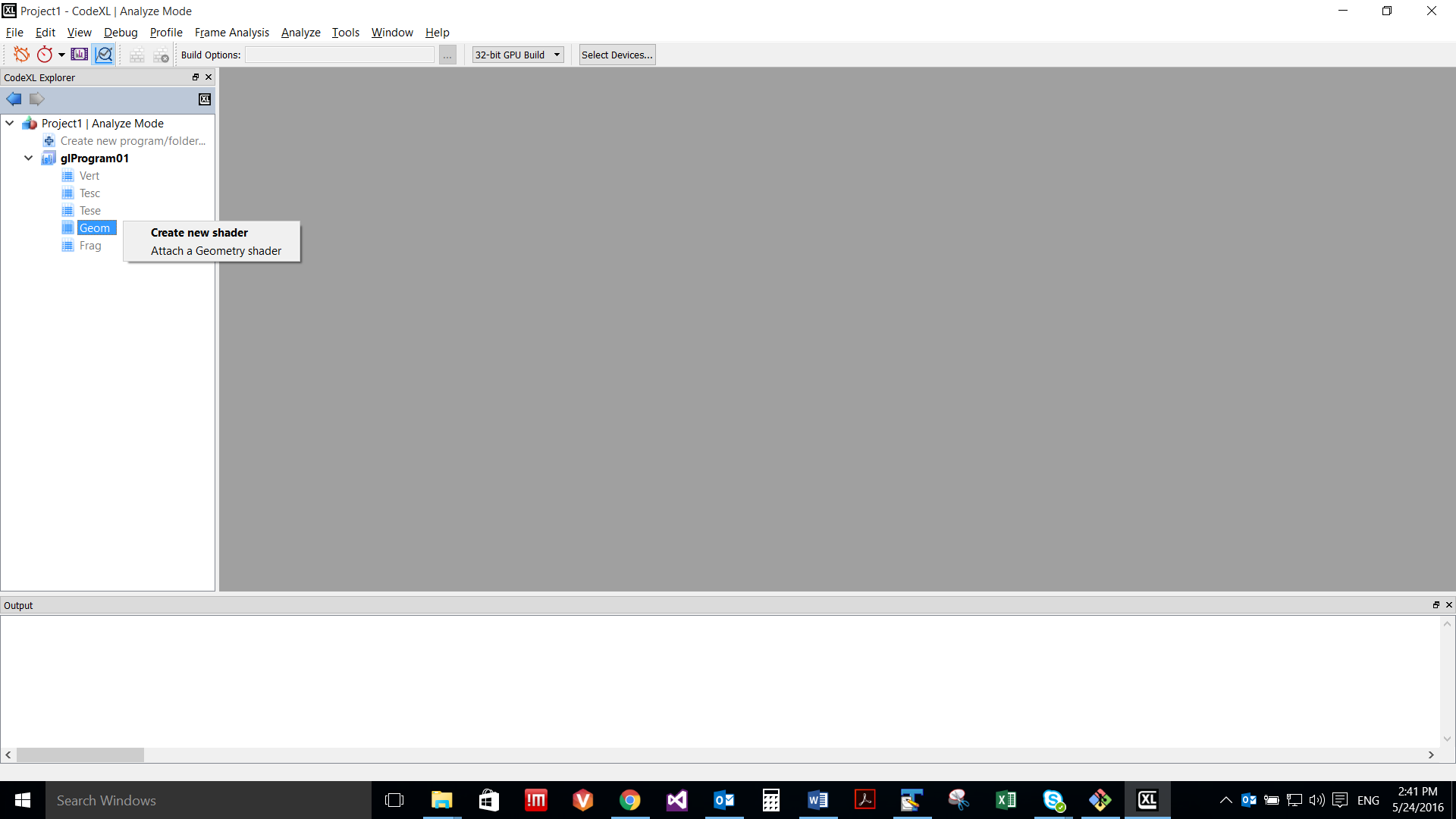
Select the Program/Folder type of choice, and click OK.

Then, the empty Program/Folder would appear in the Explorer Tree. For Example, if you choose an OpenGL Rendering Program, you will see an empty OpenGL Rendering Program created:



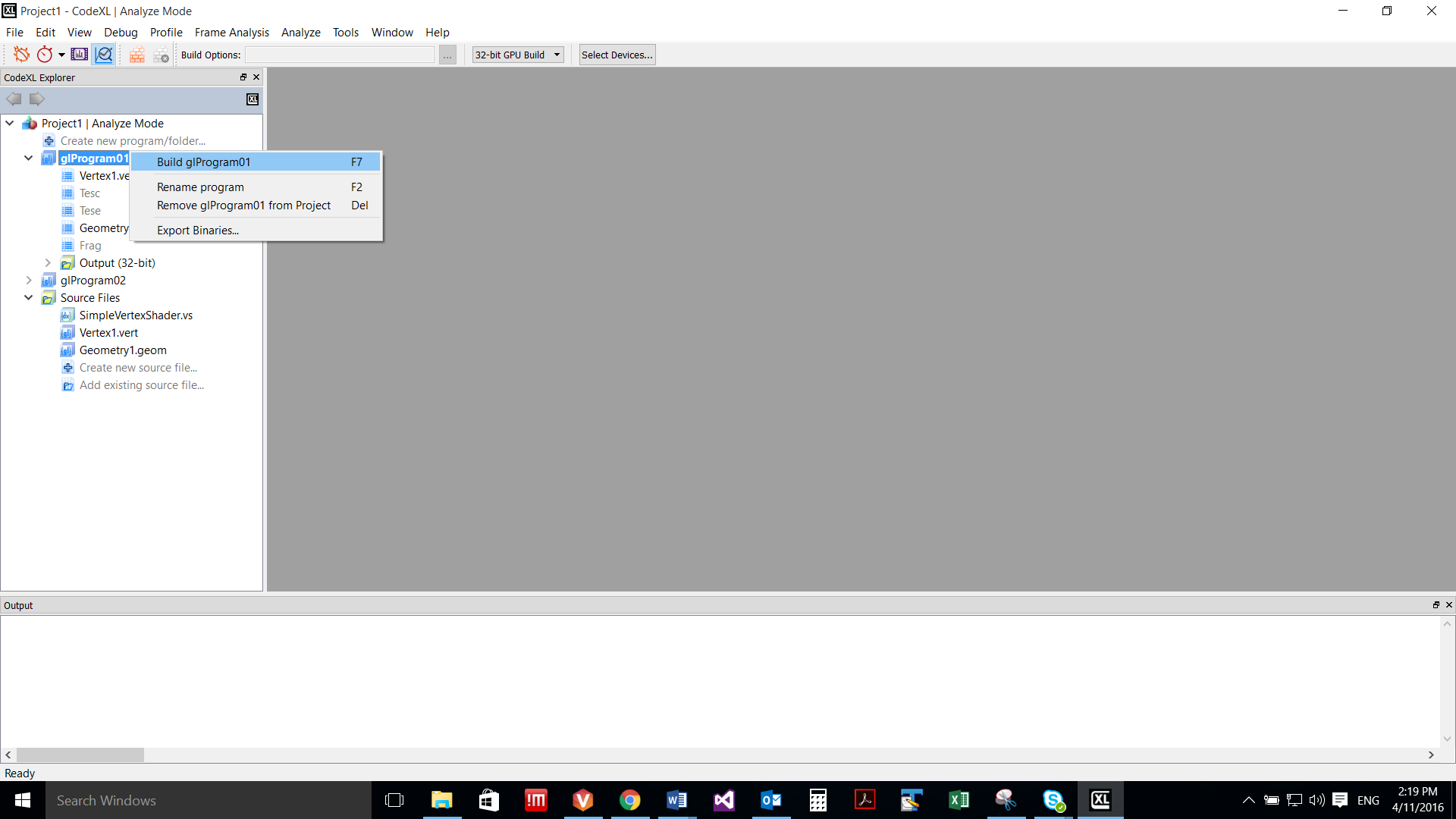
### Working with Programs

After creating a new program, you will see that it contains an empty placeholder for every pipeline stage. Right-click on any stage to add an existing shader or create a new one:

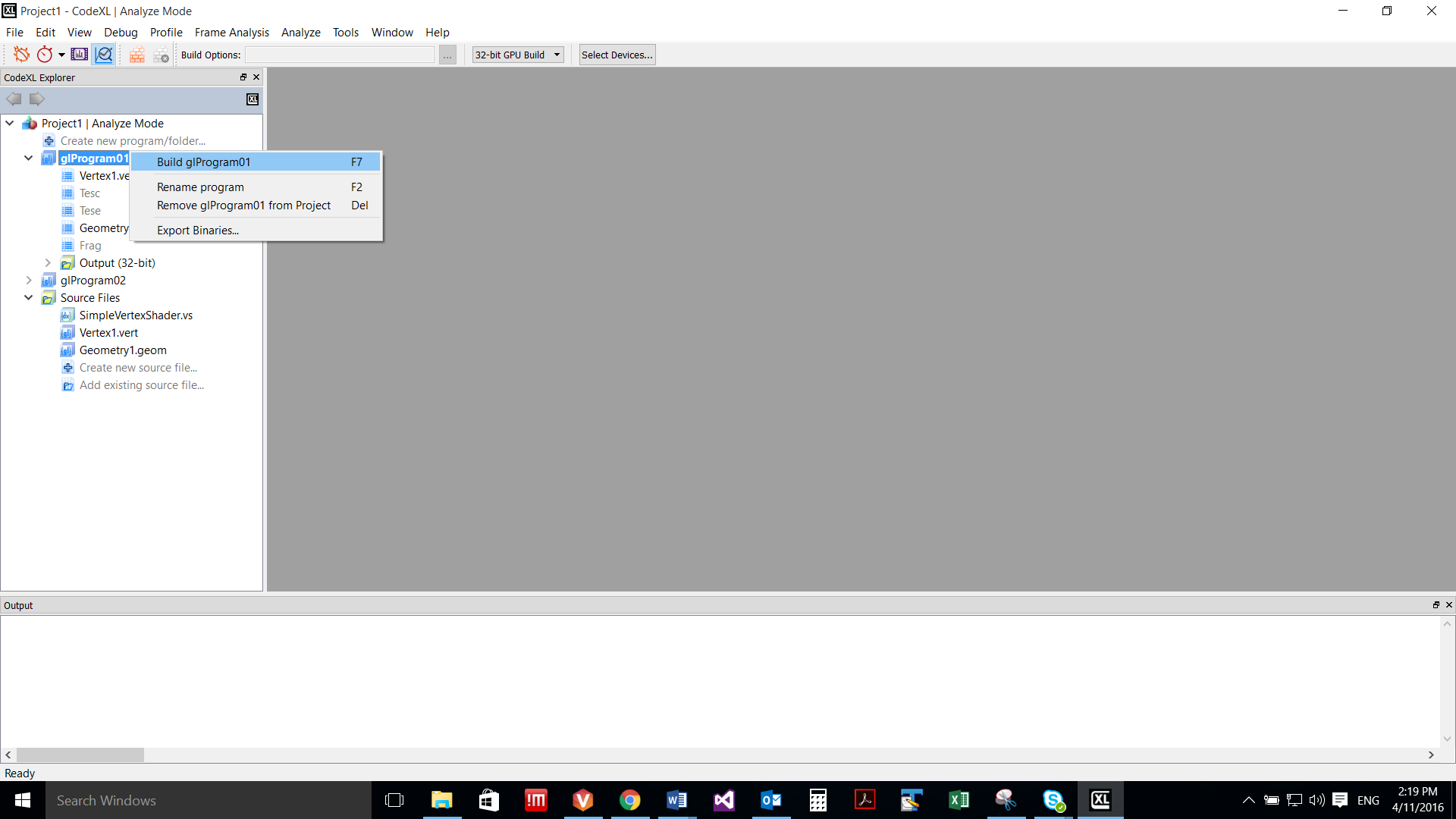


Note: You can also double-click on a stage to create a new shader and automatically attach it to that Program’s stage.

To build the program, right-click on it and select the Build option, or use the F7 shortcut:



You can also select the Program and manually click on the Build button in the Analyzer toolbar:



### Working with Folders

After creating a new OpenCL or DirectX Folder, an empty Folder would be listed in the Explorer Tree:



To create a new source file, and automatically add it to the Folder, double-click on the “Create new source file item…” item of the folder:

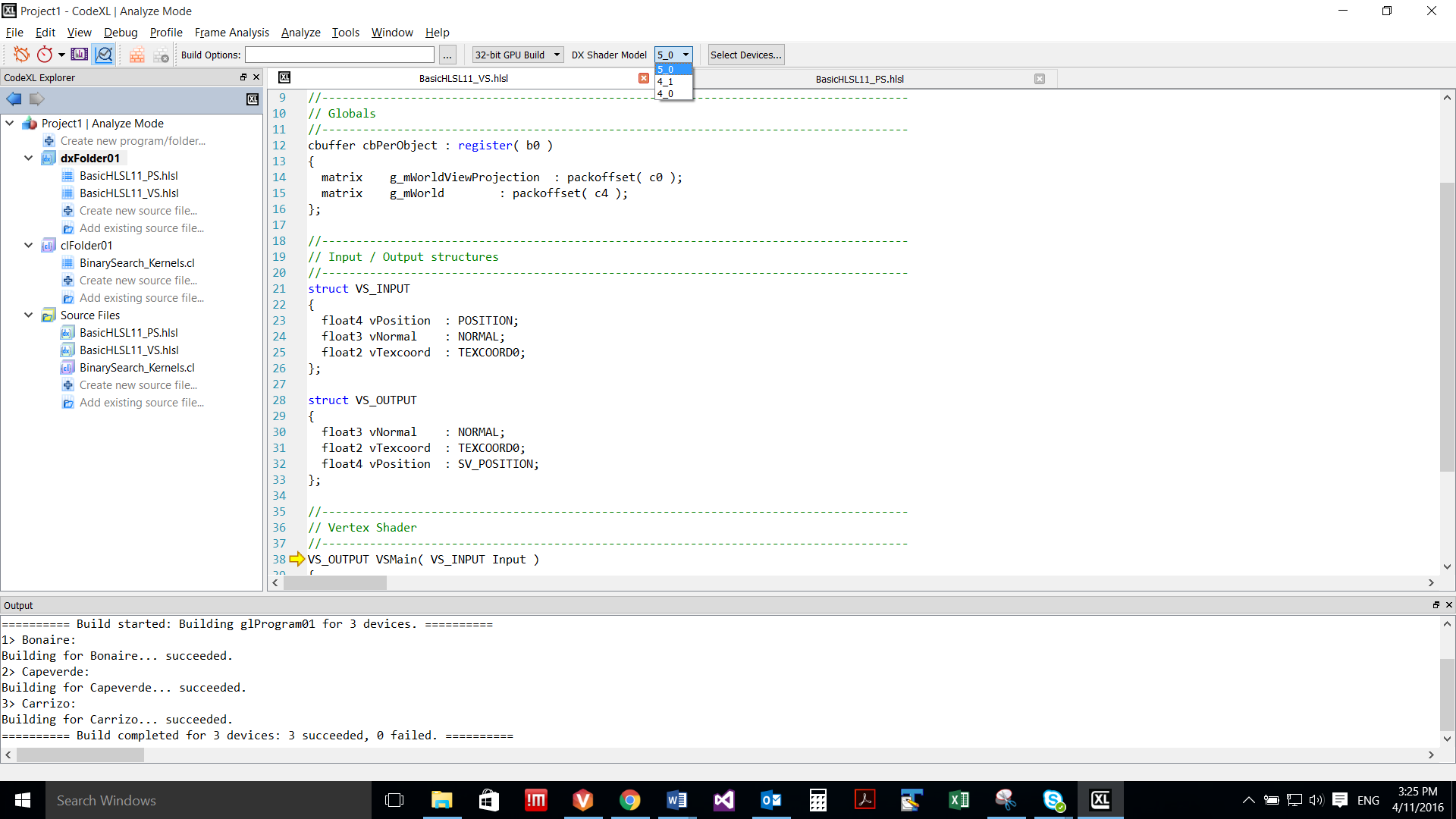


To add an existing source file, and automatically add it to the Folder, double-click on the “Add existing source file item…” item of the folder:



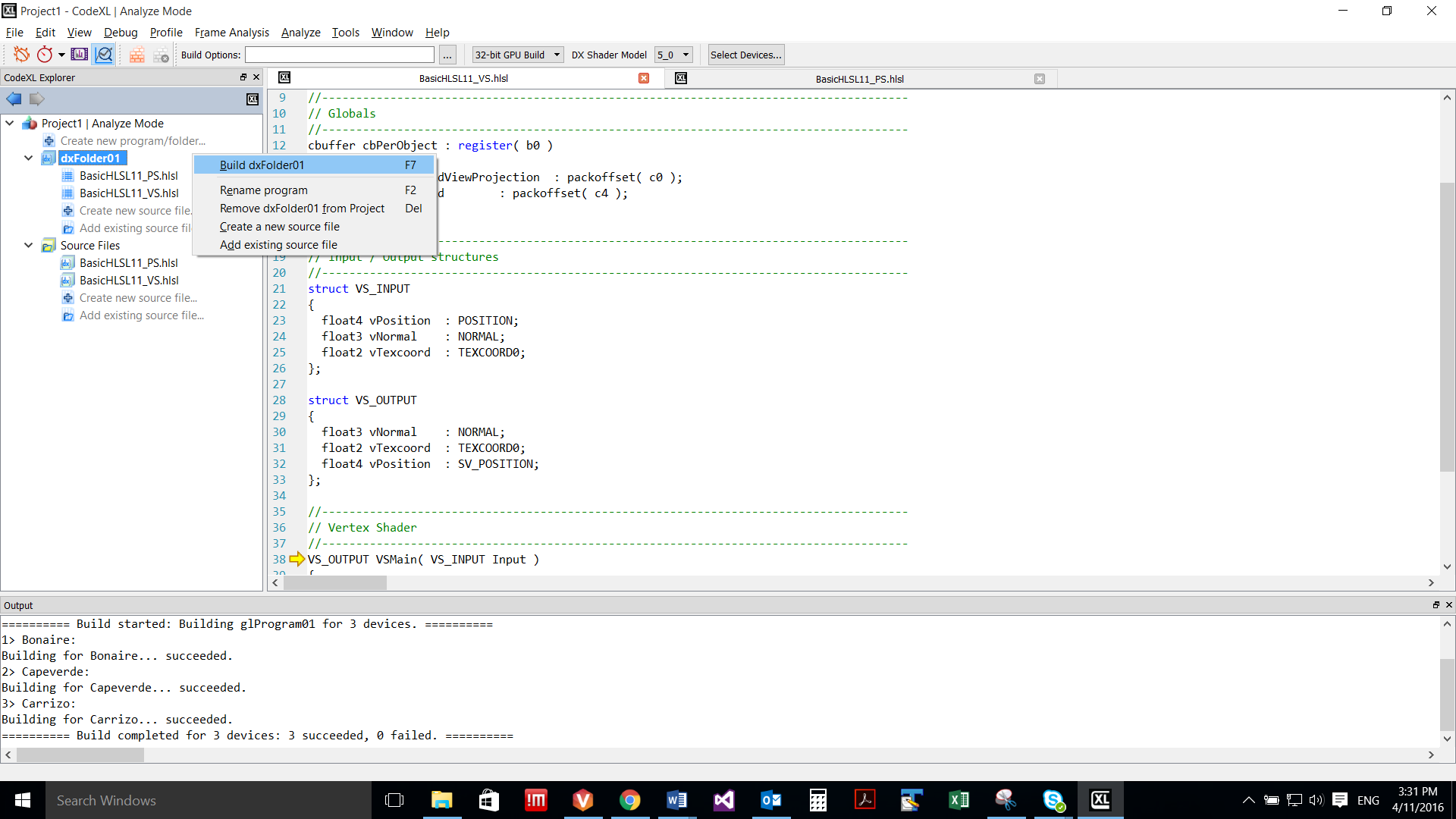
To configure the build properties of a source file under a specific Folder, click on that source file and use the Analyzer toolbar’s Type and Entry point drop-down lists. The first sets the type of the shader and the latter specifies the specific target shader (among the shaders in the source file). This configuration is Folder-specific. That is, the same source file can be set with different properties under different Folders. CodeXL will remember those configurations for you.

To configure the build properties of the Folder, click on the Folder and adjust the enabled items in the Analyzer toolbar. For CodeXL 2.0, this is only relevant to the DX Shader Model property of DX Folders:

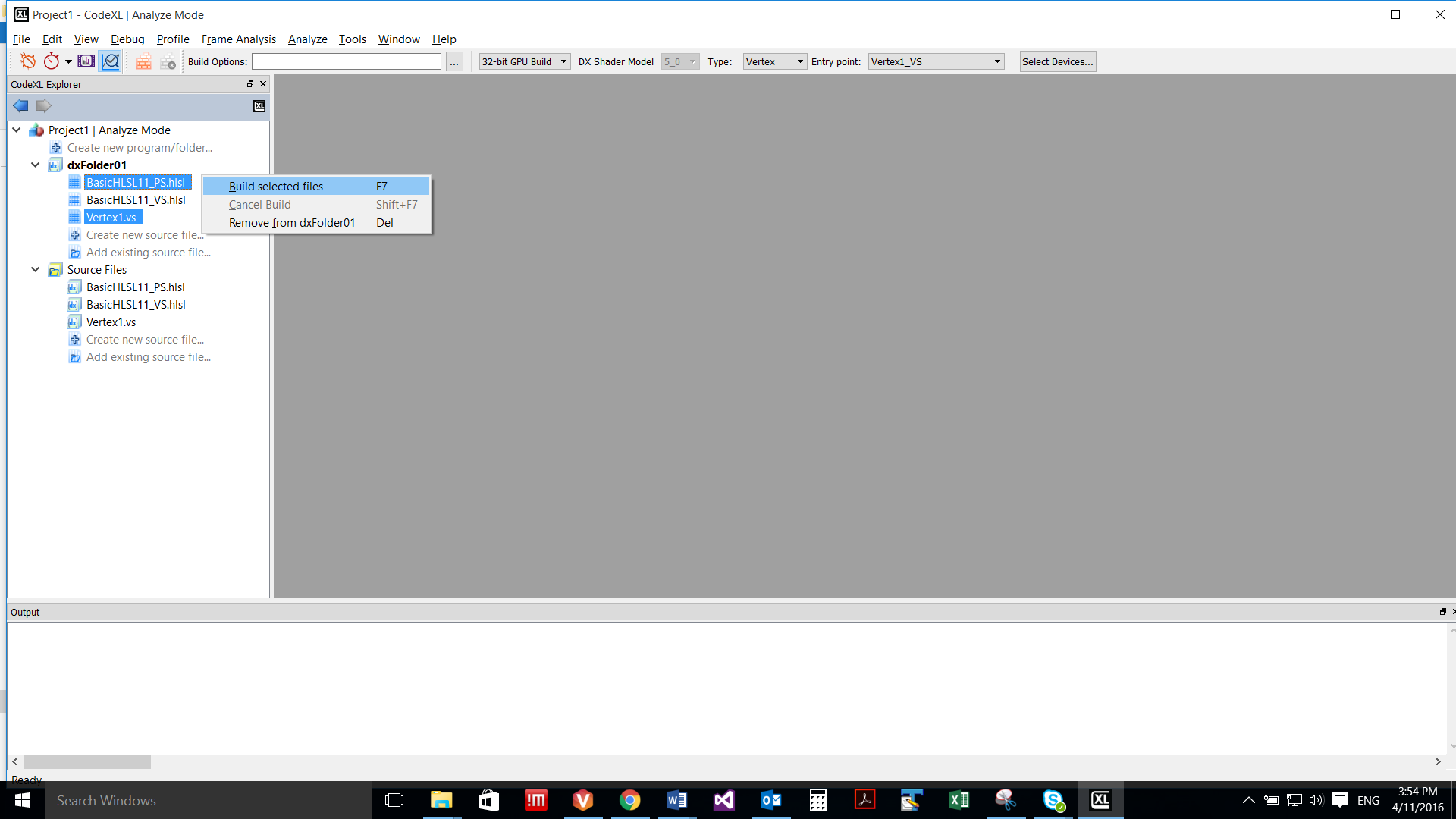


Once set, the DX Shader Model value will hold for all the shaders in the selected Folder. For example, if you choose 5\_0 as the DX Shader Model, any D3D vertex shader in that Folder will be compiled using shader model vs\_5\_0.

To build the whole Folder, right-click on it and select the Build item:

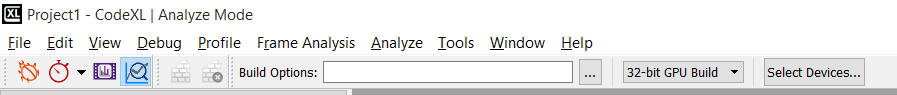


Unlike the case with Programs, Folders are more flexible as they allow you to build selected source files, without being required to build the whole Folder. To build selected source files, click on the selected source files under the program, while holding the Ctrl key. Then, right-click on one of the selected files and select the build option:

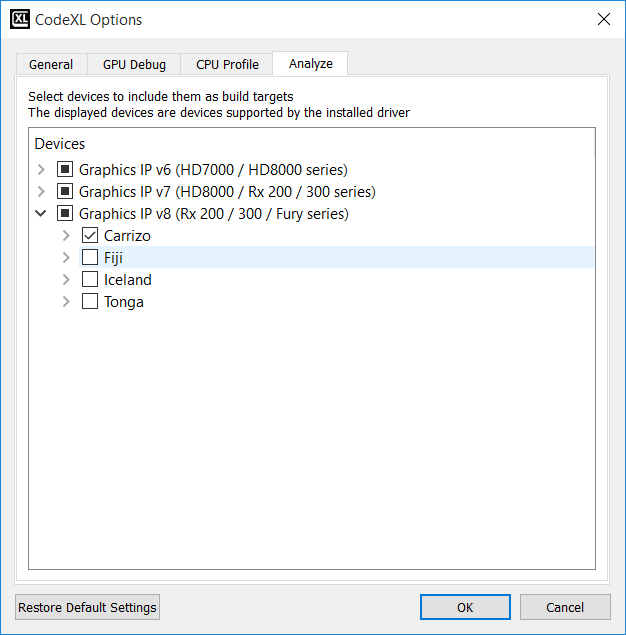


### Selecting target devices

CodeXL Analyzer can target a variety of devices, independent of the device that is physically installed on your system. To select the target devices, for which the build would be performed, first click on the Select Devices button in the Analyzer toolbar:

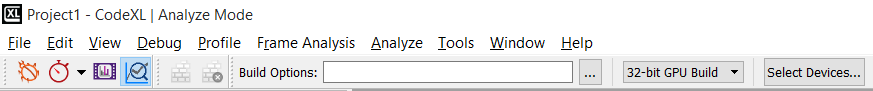


Then, the CodeXL Options dialog would pop-pup with its Analyze tab activated. The devices are grouped by generations. You can use the check boxes to select and remove devices:



### Build Options- Defining OpenCL and DirectX build options

In the Static Analyze toolbar, you can define specific OpenCL or HLSL build options:



The Build Options box is a place to set compiler build flags such as –x clc++ or –o3. Any compiler build flag can be placed in this box.

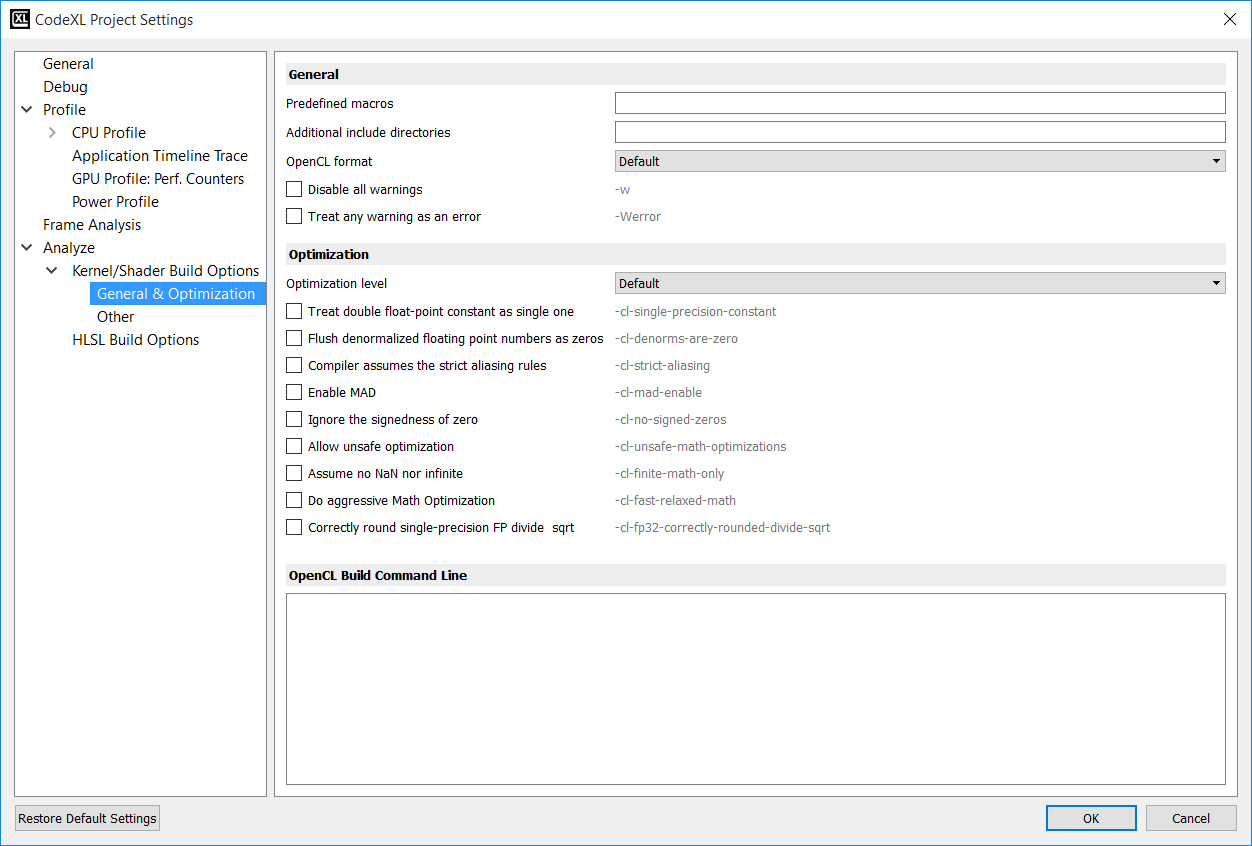
You can set the build options by typing the options directly in the designated text box or by using the OpenCL/HLSL Build Options dialog.

OpenCL Build Options Dialog

This dialog will help you choose the correct OpenCL build options for you and hopefully will prevent making spelling mistakes while typing the options manually.

To open the dialog, press The  button. The dialog will be opened. You can switch between the ”General & Optimization” tab and the ”Other” tab to view all the available options. Once you choose an option, the option text is displayed in the ”OpenCL Build Command Line” text box that appears below. This string will also appear in the menu bar after you click the OK button.

While typing a command in the “OpenCL Build Command Line” text box, you will notice that the relevant controls are being updated accordingly (for example, if you will type “-w”, you will be able to see that the “Disable all warnings” check box becomes checked).



Usage Example: build options

For building the **tpAdvectFieldScalar.cl** kernel from CodeXL’s AMDTTeaPot sample project, enter the following options:

-D GRID\_NUM\_CELLS\_X=64 -D GRID\_NUM\_CELLS\_Y=64 -D GRID\_NUM\_CELLS\_Z=64 -D GRID\_INV\_SPACING=1.000000f -D GRID\_SPACING=1.000000f -D GRID\_SHIFT\_X=6 -D GRID\_SHIFT\_Y=6 -D GRID\_SHIFT\_Z=6 -D GRID\_STRIDE\_Y=64 -D GRID\_STRIDE\_SHIFT\_Y=6 -D GRID\_STRIDE\_Z=4096 -D GRID\_STRIDE\_SHIFT\_Z=12 -I *path\_to\_example\_src*

On windows, *path\_to\_example\_src* should be:

**C:\Program Files\CodeXL\Examples\Teapot\res**

On Linux, *path\_to\_example\_src* should be:

**/opt//CodeXL/bin/examples/Teapot/AMDTTeaPotLib/AMDTTeaPotLib/**

Adding the option ‘-h’ will dump the list of OpenCL compiler available options in the output tab. For additional details, ‘Compile Build Options’ Appendix.

**Build Options**

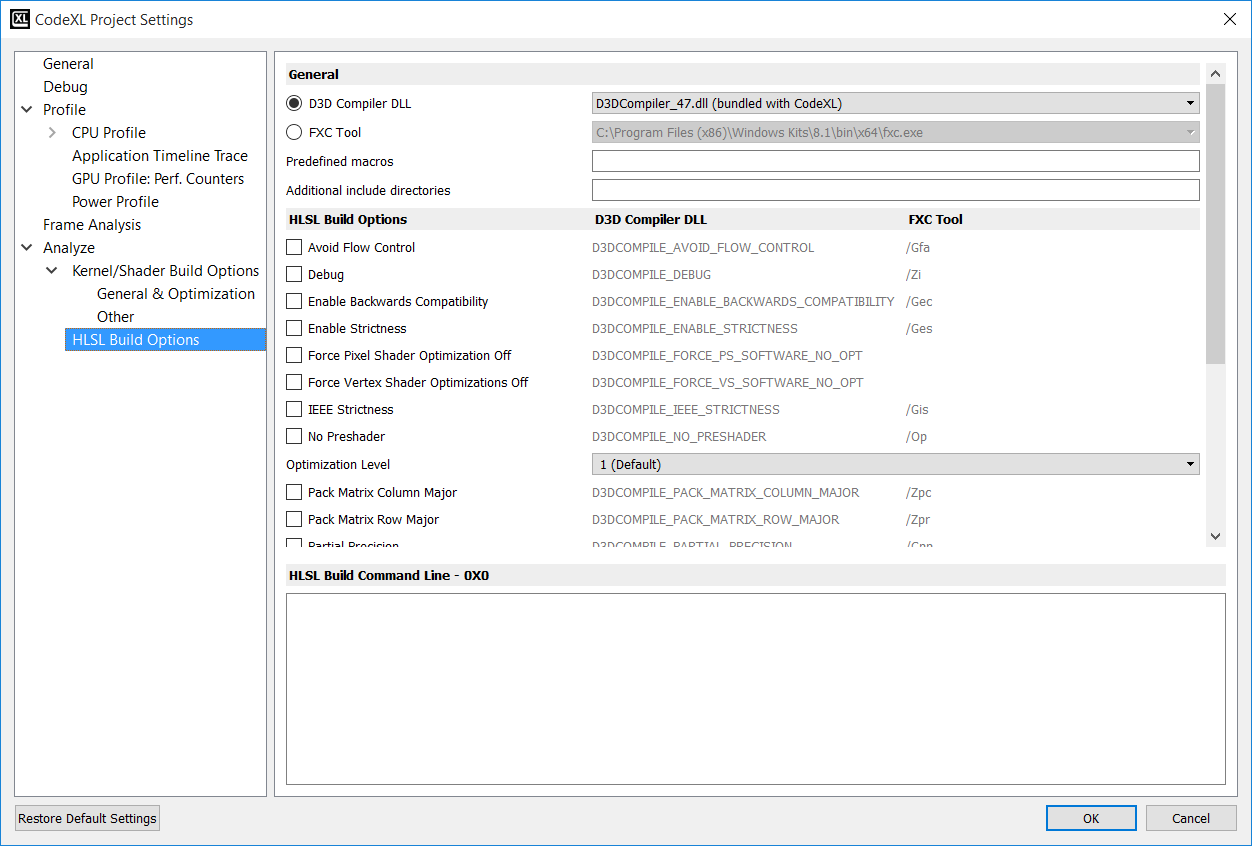
|  |  |  |
| --- | --- | --- |
| **General Options** | | |
| -D | Predefined macros | Predefine macros should be separated by ';'. If the Predefined macro needs to include a space, enclose the macro within parentheses. |
| -I | Additional include directories. | Additional include directories should be separated by ';'. If the directory path includes a space, enclose the path within parentheses. |
| -x clc,-x clc++ | OpenCL format |  |
| -w | Disable all warnings | Inhibit all warning messages. |
| -Werror | Treat any warning as an error | Make all warnings into errors. |
| **Optimization Options** | | |
| -O0,-O1,-O2,-O3,-O4,-O5 | Optimization level |  |
| -cl-single-precision-constant | Treat double float-point constant as single one | Treat double precision floating-point constant as single precision constant |
| -cl-denorms-are-zero | Flush denormalized floating point numbers as zeros | This option controls how single precision and double precision denormalized numbers are handled. If specified as a build option, the single precision denormalized numbers may be flushed to zero and if the optional extension for double precision is supported, double precision denormalized numbers may also be flushed to zero. This is intended to be a performance hint and the OpenCL compiler can choose not to flush denorms to zero if the device supports single precision (or double precision) denormalized numbers. This option is ignored for single precision numbers if the device does not support single precision denormalized numbers i.e. if CL\_FP\_DENORM bit is not set in CL\_DEVICE\_SINGLE\_FP\_CONFIG. This option is ignored for double precision numbers if the device does not support double precision or if it does support double precision but CL\_FP\_DENORM bit is not set in CL\_DEVICE\_DOUBLE\_FP\_CONFIG. This flag only applies to scalar and vector single precision floating-point variables and to computations on these floating-point variables inside a program. It does not apply to reading from or writing to image objects. |
| -cl-strict-aliasing | Compiler assumes the strict aliasing rules | This option allows the compiler to assume the strictest aliasing rules. |
| -cl-mad-enable | Enable MAD | Allow a \* b + c to be replaced by a mad. The mad computes a \* b + c with reduced accuracy. For example, some OpenCL devices implement mad as truncate the result of a \* b before adding it to c. |
| -cl-no-signed-zeros | Ignore the signedness of zero | Allow optimizations for floating-point arithmetic that ignore the signedness of zero. IEEE 754 arithmetic specifies the behavior of distinct +0.0 and -0.0 values, which then prohibits the simplification of expressions such as x+0.0 or 0.0\*x (even with -cl-finite-math-only). This option implies that the sign of a zero result isn't significant. |
| -cl-unsafe-math-optimizations | Allow unsafe optimization | Allow optimizations for floating-point arithmetic that (a) assume that arguments and results are valid, (b) may violate IEEE 754 standard and (c) may violate the OpenCL numerical compliance requirements as defined in section 7.4 for single-precision floating-point, section 9.3.9 for double-precision floating-point, and edge case behavior in section 7.5. This option includes the -cl-no-signed-zeros and -cl-mad-enable options. |
| -cl-finite-math-only | Assume no NaN nor infinite | Allow optimizations for floating-point arithmetic that assume that arguments and results are not NaNs or ±?. This option may violate the OpenCL numerical compliance requirements defined in in section 7.4 for single-precision floating-point, section 9.3.9 for double-precision floating-point, and edge case behavior in section 7.5. |
| -cl-fast-relaxed-math | Do aggressive Math Optimization | Sets the optimization options -cl-finite-math-only and -cl-unsafe-math-optimizations. This allows optimizations for floating-point arithmetic that may violate the IEEE 754 standard and the OpenCL numerical compliance requirements defined in the specification in section 7.4 for single-precision floating-point, section 9.3.9 for double-precision floating-point, and edge case behavior in section 7.5. This option causes the preprocessor macro \_\_FAST\_RELAXED\_MATH\_\_ to be defined in the OpenCL program. |
| -cl-fp32-correctly-rounded-divide-sqrt | Correctly round single-precision FP divide & sqrt | The -cl-fp32-correctly-rounded-divide-sqrt build option to clBuildProgram or clCompileProgram allows an application to specify that single precision floating-point divide (x/y and 1/x) and sqrt used in the program source are correctly rounded. If this build option is not specified, the minimum numerical accuracy of single precision floating-point divide and sqrt are as defined in section 7.4 of the OpenCL specification.\nThis build option can only be specified if the CL\_FP\_CORRECTLY\_ROUNDED\_DIVIDE\_SQRT is set in CL\_DEVICE\_SINGLE\_FP\_CONFIG (as defined in in the table of allowed values for param\_name for clGetDeviceInfo) for devices that the program is being build. clBuildProgram or clCompileProgram will fail to compile the program for a device if the -cl-fp32-correctly-rounded-divide-sqrt option is specified and CL\_FP\_CORRECTLY\_ROUNDED\_DIVIDE\_SQRT is not set for the device. |
| **Other Options** | | |
| -cl-std | CL version supported | Determine the OpenCL C language version to use. A value for this option must be provided. Valid values are:\nCL1.1 - Support all OpenCL C programs that use the OpenCL C language features defined in section 6 of the OpenCL 1.1 specification.\nCL1.2 – Support all OpenCL C programs that use the OpenCL C language features defined in section 6 of the OpenCL 1.2 specification. |
| -cl-kernel-arg-info | Kernel argument info | This option allows the compiler to store information about the arguments of a kernel(s) in the program executable. The argument information stored includes the argument name, its type, the address and access qualifiers used. Refer to the description of clGetKernelArgInfo for information about how to query this information. |
| -create-library | Create library | Create a library of compiled binaries specified in input\_programs argument to clLinkProgram. |
| -enable-link-options | Enable link options | Allows the linker to modify the library behavior based on one or more link options (described in Program Linking Options, below) when this library is linked with a program executable. This option must be specified with the –create-library option. |
| -g | Produce debugging information | This is an experimental feature that lets you use the GNU project debugger, GDB, to debug kernels on x86 CPUs running Linux, or cygwin/minGW under Windows. For more details, see Chapter 3, “Debugging OpenCL.” This option does not affect the default optimization of the OpenCL code. |
| -fper-pointer-uav -fno-per-pointer-uav | Specify that UAV per pointer should be used (HD5XXX and HD6XXX series GPU's only) |  |
| -fbin-bif30 -fno-bin-bif30 | Allow OpenCL binary to be BIF3.0 format |  |
| -fbin-encrypt -fno-bin-encrypt | Generate an encrypted OpenCL binary (not by default) |  |
| -save-temps | Store temporary files in current directory | This option dumps intermediate temporary files, such as IL and ISA code, for each OpenCL kernel. If <prefix> is not given, temporary files are saved in the default temporary directory (the current directory for Linux, C:\\Users\\<user>\\AppData\\Local for Windows). If \\<prefix\\> is given, those temporary files are saved with the given <prefix>. If <prefix> is an absolute path prefix, such as C:\\your\\work\\dir\\mydumpprefix, those temporaries are saved under C:\\your\\work\\dir, with mydumpprefix as prefix to all temporary names. For example, under the default directory |
| -fuse-jit -fno-use-jit | Use JIT for CPU target (disable if debugging is enabled |  |
| -fforce-jit -fno-force-jit | Force use JIT for CPU target (even if debugging is enabled) |  |
| -fdisable-avx -fno-disable-avx | Disable AVX code generation |  |
| -ffma-enable -fno-fma-enable | Enable fma for a\*b+c |  |
| -fuse-native | Replace math function calls with native version |  |

HLSL Build Options Dialog

This dialog will help you choose the correct HLSL build options for you and hopefully will prevent making spelling mistakes while typing the options manually.

To open the dialog, press The  Button. The dialog will be opened. Click the ”HLSL Build Options” node to view the available options.   
Once you choose an option, the option text is displayed in the ”HLSL Build Command Line” text box that appears below.   
This build option string will also appear in the toolbar’s build options box after you click the OK button.

As an alternative to selecting options through the radio buttons, it is possible to type a command in the “HLSL Build Command Line” text box. Build options types in the text box will automatically be translated to update of the relevant controls accordingly. For example, typing “D3DCOMPILE\_DEBUG” in the lower text box automatically updates the “Debug” check box to be checked.



**Build Options**

The compilation of DirectX shaders can be executed either by directly referencing the D3D compiler DLL or by going through Microsoft’s FXC tool.

The CodeXL installation includes a copy of the Microsoft DirectX compiler DLL: d3dcompiler\_47.dll. You may specify a different path if you want CodeXL to use a different d3dcompiler module. If you select the FXC compiler tool, you must specify a path to the location of FXC.exe.

To select the path of the compiler module, click the ‘Browse…” option from the combo-box. When selecting Browse, a dialog box will open for selecting the compiler file.

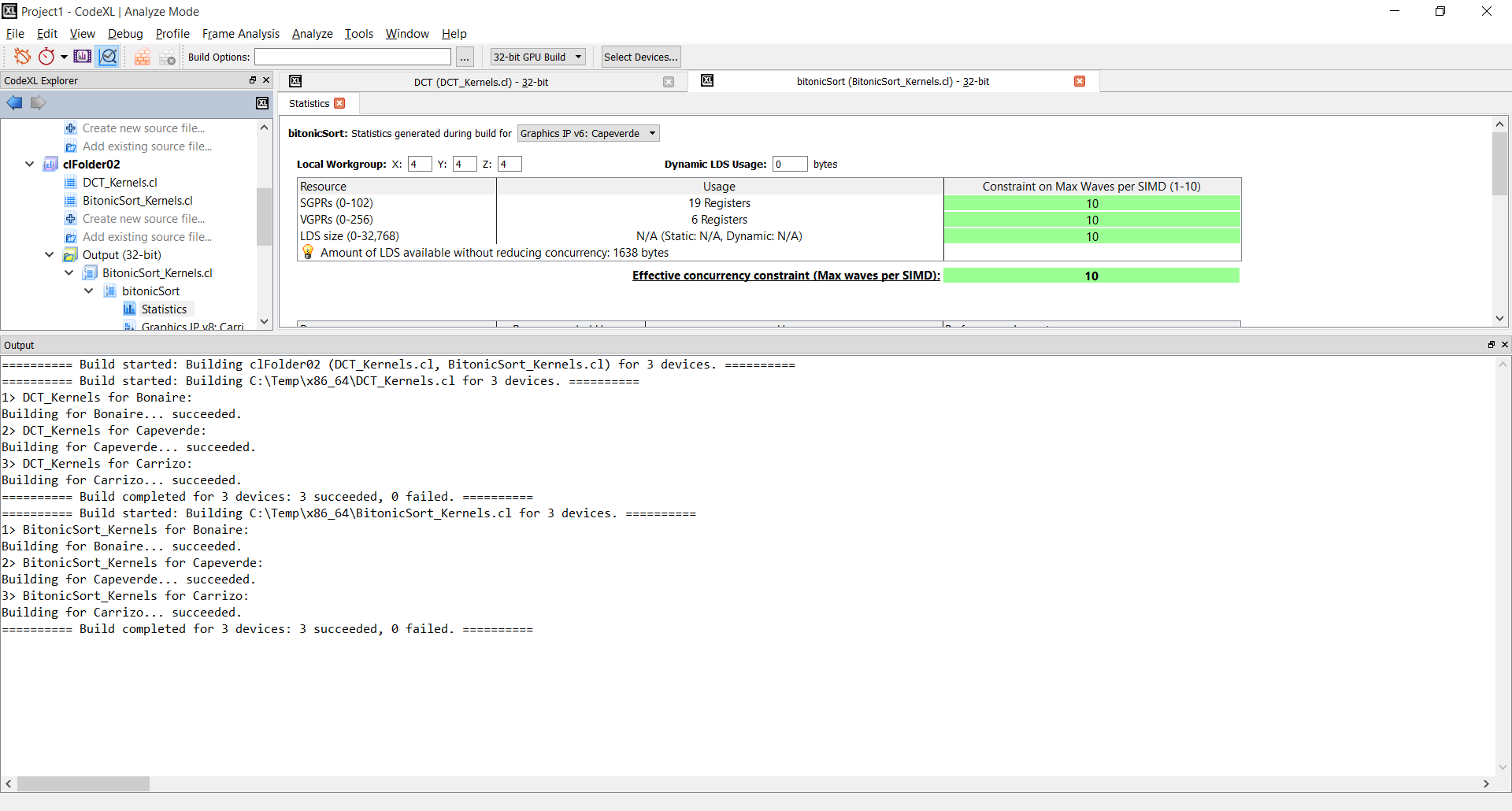
* For D3D compiler – any file called d3compiler\_\*.dll can be selected.
* For FXC compiler – only files named FXC.exe can be selected.

|  |  |  |  |
| --- | --- | --- | --- |
| **D3D compile command** | **FXC Compile command** | **Build Option** |  |
| -D | | Predefined macros | Predefine macros should be separated by ';'. If the Predefined macro needs to include a space, enclose the macro within parentheses. The macro won’t appear in the “HLSL Build Command Line” |
| -I | | Additional include directories | Additional include directories should be separated by ';'. If the directory path includes a space, enclose the path within parentheses. the include won’t appear in the “HLSL Build Command Line” |
| D3DCOMPILE\_AVOID\_FLOW\_CONTROL | /Gfa | Avoid Flow Control | Directs the compiler to not use flow-control constructs where possible. |
| D3DCOMPILE\_DEBUG | /Zi | Debug | Directs the compiler to insert debug file/line/type/symbol information into the output code. |
| D3DCOMPILE\_ENABLE\_BACKWARDS\_COMPATIBILITY | /Gec | Enable Backwards Compatibility | Directs the compiler to enable older shaders to compile to 5\_0 targets. |
| D3DCOMPILE\_ENABLE\_STRICTNESS | /Ges | Enable Strictness | Forces strict compile, which might not allow for legacy syntax.  By default, the compiler disables strictness on deprecated syntax. |
| D3DCOMPILE\_FORCE\_PS\_SOFTWARE\_NO\_OPT |  | Force Pixel Shader Optimization Off | Directs the compiler to compile a pixel shader for the next highest shader profile. This constant also turns debugging on and optimizations off. |
| D3DCOMPILE\_FORCE\_VS\_SOFTWARE\_NO\_OPT |  | Force Vertex Shader Optimizations Off | Directs the compiler to compile a vertex shader for the next highest shader profile. This constant turns debugging on and optimizations off." |
| D3DCOMPILE\_IEEE\_STRICTNESS | /Gis | IEEE Strictness | Forces the IEEE strict compile |
| D3DCOMPILE\_NO\_PRESHADER | /Op | No Preshader | Directs the compiler to disable Preshaders. If you set this constant, the compiler does not pull out static expression for evaluation. |
| \* D3DCOMPILE\_SKIP\_OPTIMIZATION \* D3DCOMPILE\_OPTIMIZATION\_LEVEL0 \* .. (no flag for default optimization) \* D3DCOMPILE\_OPTIMIZATION\_LEVEL2 \* D3DCOMPILE\_OPTIMIZATION\_LEVEL3 | \* /Od \* /O0 \* .. (no flag for default) \* /O1 \* /O2 \* /O3 | Optimization Level: \* Skip Optimization \* Level 0 - Lowest optimization \* Level 1 – Default Optimization \* Level 2 \* Level 3 - Highest optimization | Directs the compiler to use the specified level of optimization: \* Skip - skip optimization steps during code generation. We recommend that you set this constant for debug purposes only. \* Lowest level - If you set this constant, the compiler might produce slower code but produces the code more quickly.   Set this constant when you develop the shader iteratively. \* Second lowest level - Second highest level. \* Highest level - If you set this constant, the compiler produces the best possible code but might take significantly longer to do so.   Set this constant for final builds of an application when performance is the most important factor. |
| D3DCOMPILE\_PACK\_MATRIX\_COLUMN\_MAJOR | /Zpc | Pack Matrix Column Major | Directs the compiler to pack matrices in column-major order on input and output from the shader. This type of packing is generally more efficient because a series of dot-products can then perform vector-matrix multiplication |
| D3DCOMPILE\_PACK\_MATRIX\_ROW\_MAJOR | /Zpr | Pack Matrix Row Major | Directs the compiler to pack matrices in row-major order on input and output from the shader. |
| D3DCOMPILE\_PARTIAL\_PRECISION | /Gpp | Partial Precision | Directs the compiler to perform all computations with partial precision. If you set this constant, the compiled code might run faster on some hardware. |
| D3DCOMPILE\_PREFER\_FLOW\_CONTROL | /Gfp | Prefer Flow Control | Directs the compiler to use flow-control constructs where possible. |
| D3DCOMPILE\_RESOURCES\_MAY\_ALIAS | /res\_may\_alias | Resources May Alias | Directs the compiler to assume that unordered access views (UAVs) and shader resource views (SRVs) may alias for cs\_5\_0.  Note: This is a new compiler symbol (supported as of D3dcompiler\_47.dll). |
| D3DCOMPILE\_SKIP\_VALIDATION | /Vd | Skip Validation | Directs the compiler not to validate the generated code against known capabilities and constraints. We recommend that you use this constant only with shaders that have been successfully compiled in the past. DirectX always validates shaders before it sets them to a device. |
| D3DCOMPILE\_WARNINGS\_ARE\_ERRORS | /WX | Warnings Are Errors | Directs the compiler to treat all warnings as errors when it compiles the shader code. We recommend that you use this constant for new shader code, so that you can resolve all warnings and lower the number of hard-to-find code defects |
|  | /Lx | Output hexadecimal literals |  |
|  | /Ni | Numbering of instructions in assembly listings |  |
|  | /No | Output instruction byte offset in assembly listings |  |
|  | /Qstrip\_debug | Strip debug data from 4.0 + shader bytecode |  |
|  | /Qstrip\_priv | Strip private data from 4.0 + shader bytecode |  |
|  | /Qstrip\_reflect | Strip reflection data from 4.0 + shader bytecode |  |

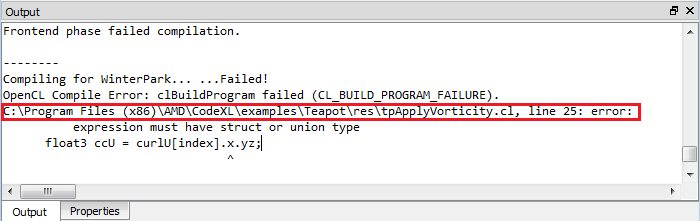
* Note: some of the flags are only relevant to the FXC tool.

### Output Tab

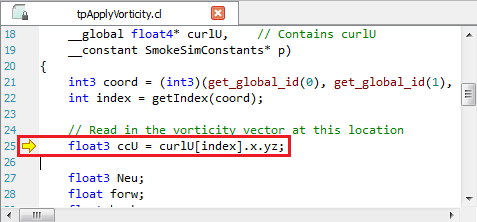
The compiler output appears in the Output tab. The example below shows successful builds (no warnings or errors) for 4 devices.



If there were errors, the output will display the error and the line where the error occurred:

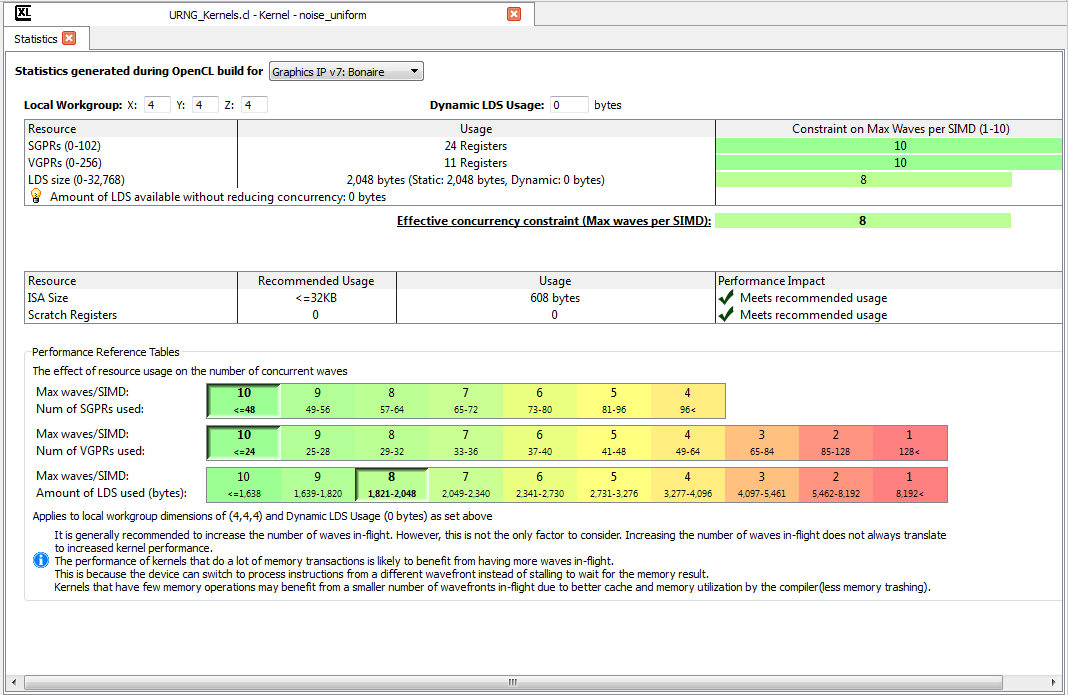


Double clicking on an error navigates the user to the Source Code view, displaying the kernel source code:



### Kernel Statistics Tab

The statistics tab gives detailed statistics for the selected kernel for each target device.   
To open the statistics tab, expand the desired kernel in the project tree, and double-click the statistics node:



You can see in this statistics output that Northern Islands devices do not use scalar registers (SGPRs and MaxSGPRs are N/A). Northern Islands VGPRs are quad-sized so the 20 VGPRs actually represent 80 float values. You can also see that Southern Islands devices use 44 scalar GPRs that are shared across the wavefront and 49 VGPRs that are used per thread equaling 49 float values.

This view puts the emphasis on giving the programmer the wave constraints based on the SGPRs, VGPRs and LDS size.

In the upper section there is a table that shows the current constraints based on the kernels information for the current selected device.

In the lower section, there is a reference table to help the programmer see the effect of the resources usage on the number of concurrent waves.

The LDS is constructed from the static & dynamic values. The dynamic value can be defined by the user to and its possible effect on the constraint can be immediately viewed. LDS is also affected by local workgroups size, so the values can be changed in order to see the impact on the performance of the kernel.

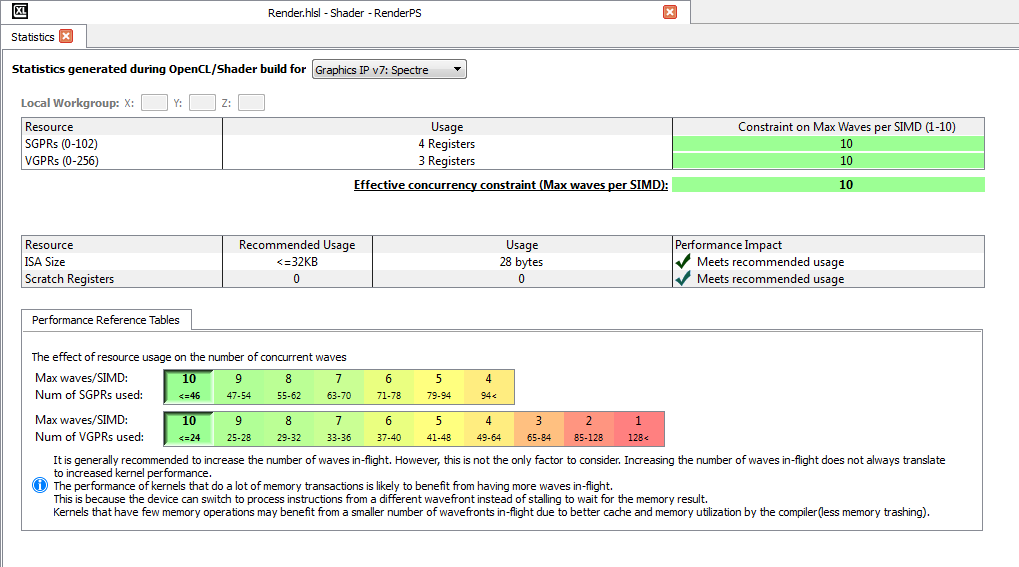
The middle table shows if the “ISA size” and “Scratch Registers” might have impact on the performance and what are the recommended values.

### Shader Statistics Tab

The statistics tab gives detailed statistics about the selected shader for each target device.   
To open the statistics tab expand the desired shader in the project tree, and double-click the statistics node:

There is no support for shader statistics for v5 and earlier generation devices.

Statistics page for GCN devices (v6 and later generations):



The displayed information is explained in the following table:

|  |  |
| --- | --- |
| **Column Name** | **Explanation** |
| SGPRs | The number of scalar General Purpose Registers allocated by the shader |
| VGPRs | The number of vector General Purpose Registers allocated by the kernel. ReqdWorkGroupX - Required workgroup X size specified for kernel. N/A without optional \_\_attribute\_\_((reqd\_work\_group\_size(X, Y, Z))) |
| ISA size | Compiled code size |
| Scratch Registers | The number of scratch registers used by the kernel. If this value is bigger than 0, the shader may be incurring a performance penalty |
| MaxSGPRs | The maximum number of scalar General Purpose Registers per kernel supported by the device |
| MaxVGPRs | The maximum number of vector General Purpose Registers per kernel supported by the device |

This view puts the emphasis on giving the programmer the wave constraints based on the SGPRs and VGPRs.

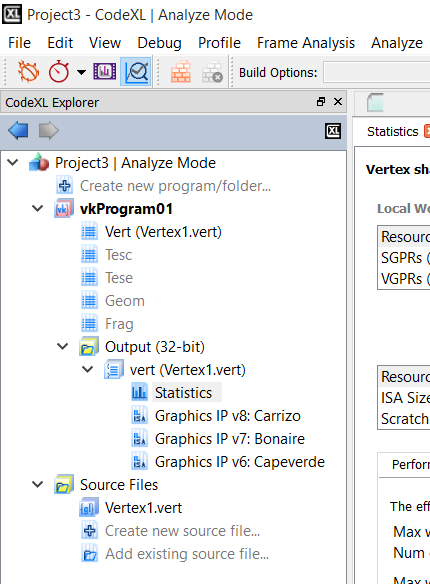
In the upper section there is a table that shows the current constraints based on the shaders information for the current selected device.

In the lower section there is a reference table to help the programmer see the effect of the resources usage on the number of concurrent waves.

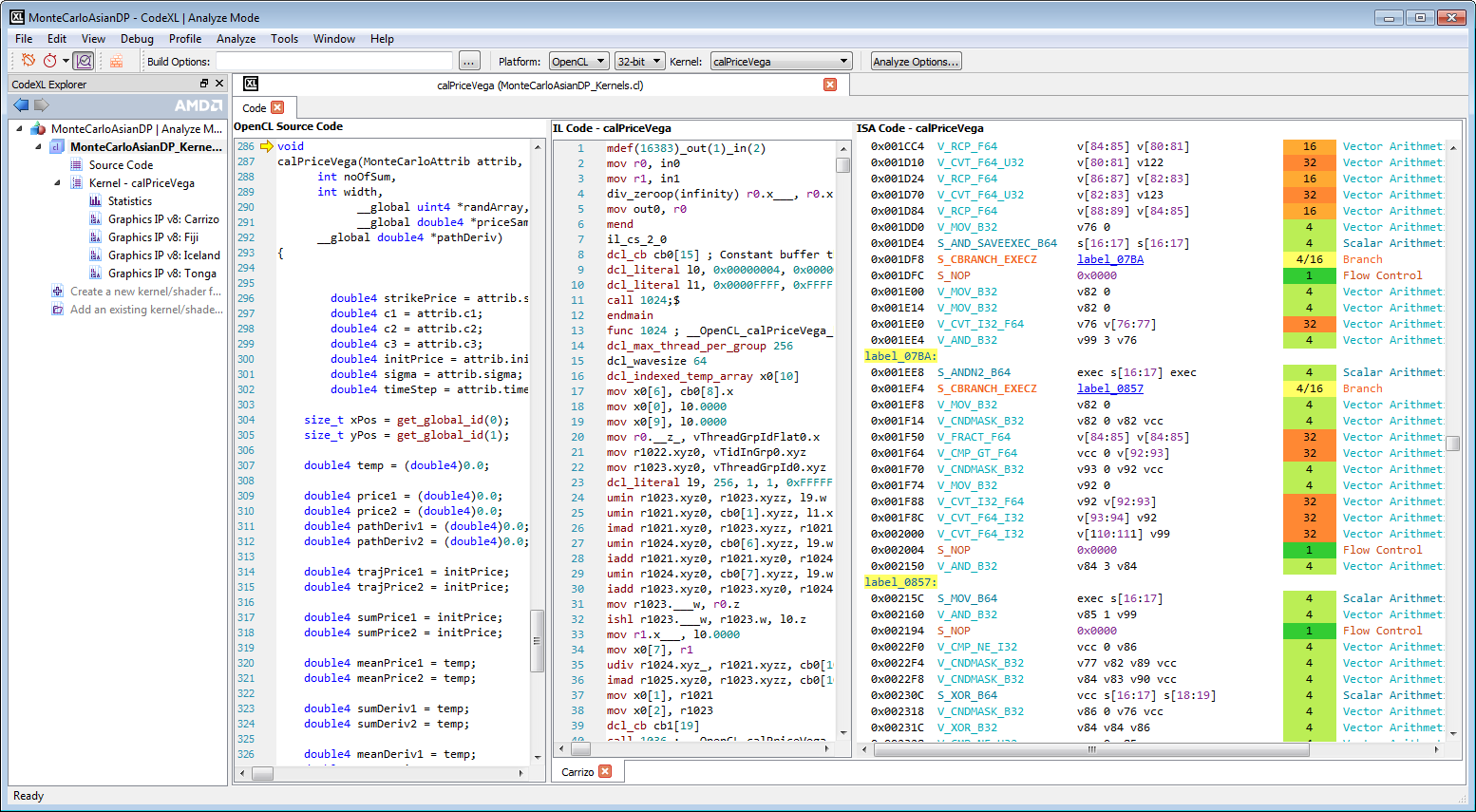
The middle table shows if the “ISA size” and “Scratch Registers” might have impact on the performance and what are the recommended values.

### Viewing compilation output: IL and ISA

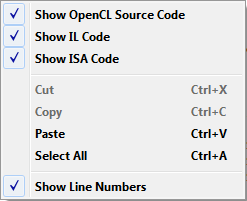
The performance statistics tab will be opened automatically when the build process is over. To view the compilation output, double click the node of the desired ASIC in the explorer tree, under the Program/Folder and configuration (32-bit or 64-bit):

s

This will open a tab containing the source code, the IL and the ISA. The program source code and the IL code will be presented as standard text documents. The ISA will be presented in the “Enhanced ISA View” for GCN devices, and as a standard text document for pre-GCN devices.



The context menu also enables showing/hiding line numbers for each source code/IL/ISA tab.



### Navigating through ISA code with the Enhanced ISA View

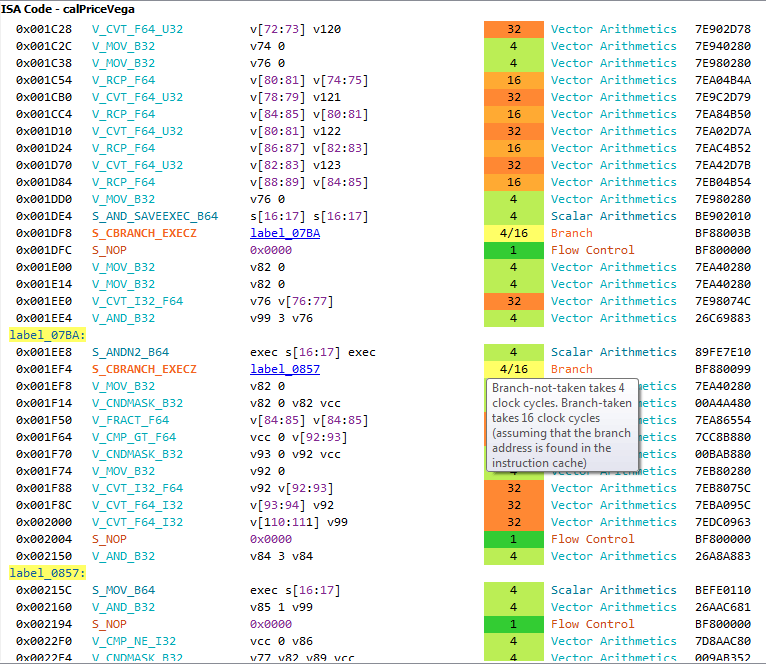
Using this view, you can inspect the ISA code of GCN devices and see the estimation for instruction cost in clock cycle. The view contains 5 columns:

* **Address**: the instruction’s offset within the program (in bytes)
* **Opcode**: the operation to be performed
* **Operands**: the data for the operation
* **Cycles**: the number of clock cycles which are required by a Compute Unit in order to process the instruction for a 64-thread Wavefront, while neglecting the system load and any other runtime-related factor.
* **Instruction** Type: the category of instructions to which the instruction belongs
* **Hex**: binary representation of the instruction, in hexadecimal format

Notes:

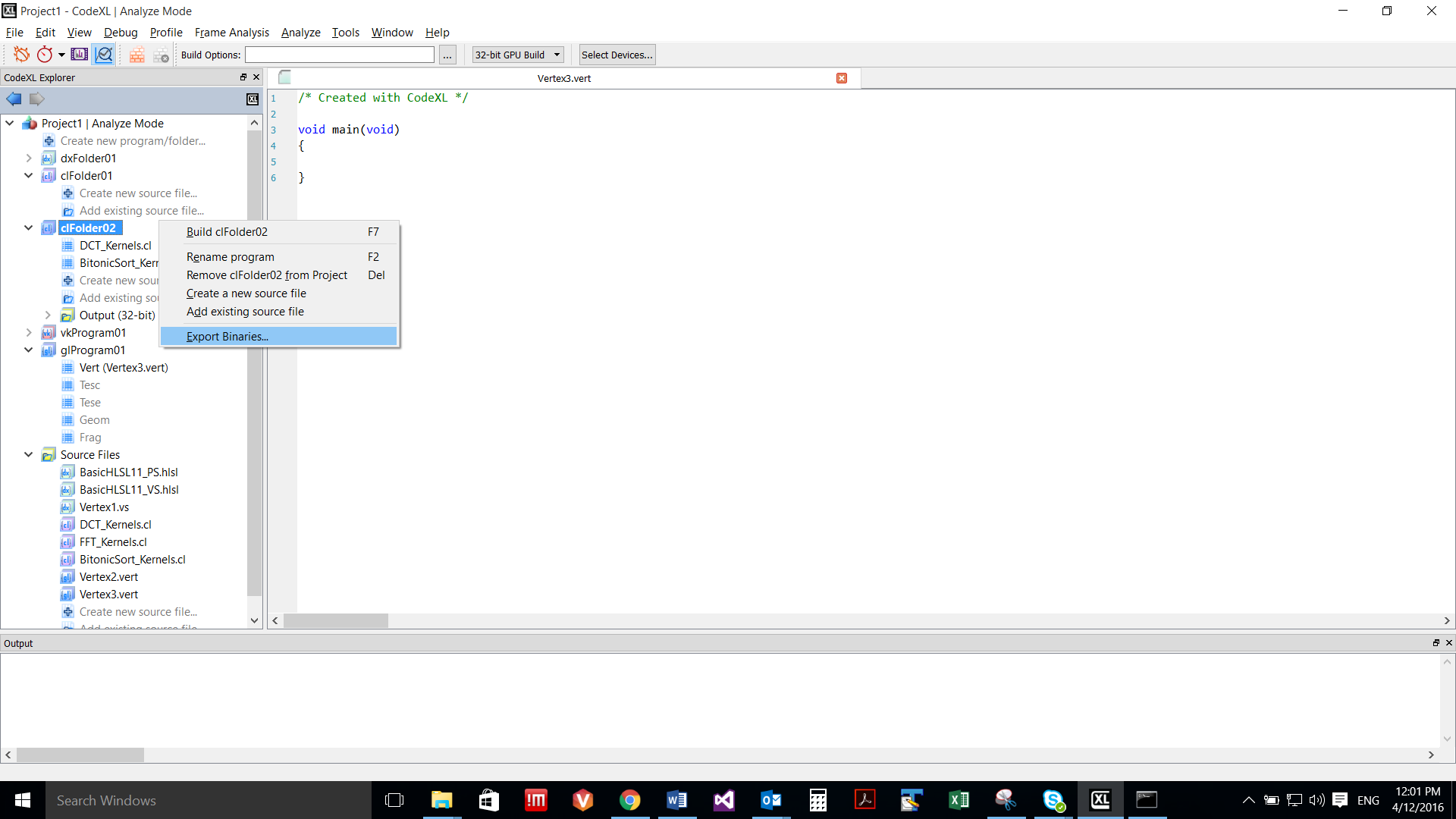
1. Note that code labels which appear in the Operands column are clickable. By clicking on a label link, you can navigate to the label’s spot in the code.

2. Note that this view is only available for GCN devices. For pre-GCN devices, the plain textual ISA view will be displayed.

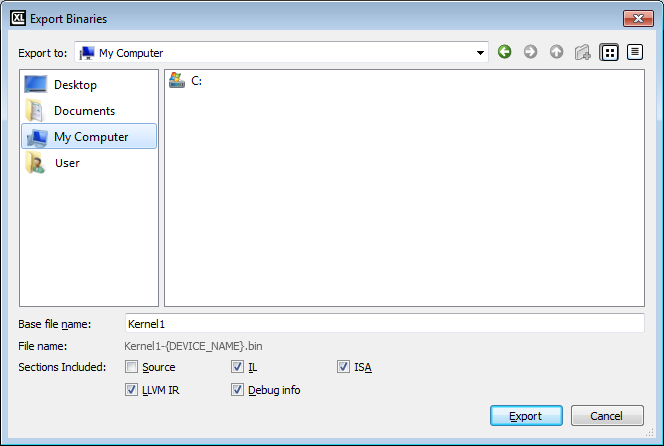


### Export binaries

You can export the binaries of the last build by right clicking on the folder or file in the explorer tree and selecting the “Export binaries…” option.



When you select the “Export Binaries…” option, an “Export Binaries” dialog will open where you can select the destination folder as well as which parts of the binaries will be included.



You can change the default base name by entering desired name into “Base file name” text field. All binary files will be created with the chosen base name followed by device name.

All the devices that were built will be exported.

\* Notice: this option in only supported in OpenCL mode.

### Remove items from Project

Programs, Folders and source files can be removed from the project. To remove an item from the project, right-click on it and select the Remove option.

### Static Analyze Toolbar – for OpenCL source files



|  |  |
| --- | --- |
|  | Build command |
|  | Kernel Build options |
|  | Opens the “Build Options” dialog |
|  | The bitness of the compilation process (for example, choose OpenCL, 64-bit to compile the files in using the 64-bit OpenCL compiler). |
|  | Displays the last selected kernel for the selected file. Jumps to the selected kernel in the active source if the source code is open. |

### Static Analyze Toolbar – for DirectX shaders



|  |  |
| --- | --- |
|  | Build only command |
|  | Shader Build options |
|  | Opens the “Build Options” dialog |
|  | The bitness of the compilation process (for example, choose OpenCL, 64-bit to compile the files in using the 64-bit OpenCL compiler). |
|  | The selected shader model. This parameter is set once at the Folder level, and holds for all the shaders in that Folder. |
|  | The type of the selected shader (Vertex, Pixel, Hull, etc.) |
|  | The target shader for the compilation process. |

### CodeXLAnalyzer Command Line Interface

CodeXLAnalyzer (formerly CodeXLKernelAnalyzer) enables compiling and generating performance statistics for OpenCL kernels, DirectX Shaders and OpenGL shaders. The compilation and statistics generation processes can be targeted at a variety of AMD GPUs and APUs, regardless to the actual GPU/APU type that is installed on your system. The application capabilities were expanded. OpenCL Kernels as well as DirectX shaders can now be compiled.

CodeXLAnalyzer can be executed from a shell window using the CodeXLAnalyzer console application which is located in the CodeXL installation directory.

Note: On Windows, CodeXLAnalyzer is available in both 32-bit and 64-bit versions. To execute the compilation and statistics generation in 64-bit, invoke CodeXLAnalyzer-x64.exe. On Linux, only a 64-bit version is available.

**[Using CodeXLAnalyzer Command Line Interface to compile OpenCL Kernels](#_Using_CodeXLAnalyzer_Command_3)**

[**Using CodeXLAnalyzer Command Line Interface to compile DirectX shaders**](#_Using_CodeXLAnalyzer_Command_1)

**[Using CodeXLAnalyzer Command Line Interface to compile OpenGL and Vulkan programs](#_Using_CodeXLAnalyzer_Command_2)**

**[Generating and interpreting CodeXLAnalyzer CLI’s live register analysis report](#_Generating_and_Interpreting)**

#### Using CodeXLAnalyzer Command Line Interface to compile OpenCL Kernels

OpenCL is the default language for CodeXLAnalyzer, so in order to compile OpenCL kernels specifying the input source code language is optional.

CodeXLAnalyzer uses the actual AMD OpenCL Driver installed on the computer, i.e the Catalyst driver to perform offline compilation.

If no GPU is present, the OpenCL driver installed with APP SDK can be used.

Details of available commands:

|  |  |  |
| --- | --- | --- |
| -h | View available options | |
| CodeXLAnalyzer.exe  -h | |
|  | | |
| -s | Specify the source language for the compilation. “cl” is the default which means that for OpenCL kernel compilation there is no need to specify the –s switch | |
| CodeXLAnalyzer.exe  -s cl -l | |
|  | | |
| -l [ --list-asics ] | List known ASIC targets. | |
| CodeXLAnalyzer.exe  -l | |
|  | | |
| --verbose | View supported ASICS with detailed marketing names | |
| CodeXLAnalyzer.exe  -l --verbose | |
|  | | |
| --version | Print version string. | |
| CodeXLAnalyzer.exe  --version | |
|  | | |
| -a [ --analysis ] arg | Path to output analysis file. Requires --kernel. | |
|  | CodeXLAnalyzer.exe foo.cl --kernel myKernel --analysis foo.csv | |
|  | | |
| -c [ --asic ] arg | Which ASIC to target. Repeatable. | |
| CodeXLAnalyzer.exe foo.cl --kernel myKernel -isa foo.isa --asic Bonaire | |
|  | | |
| --list-kernels | List the kernels functions available in the specify cl file | |
| CodeXLAnalyzer foo.cl --list-kernels | |
|  | | |
| --isa arg | Path to output ISA disassembly file(s). This command requires compilation switches identifying the required kernel --kernel | |
| CodeXLAnalyzer.exe foo.cl --kernel Foo --isa c:\files\Foo  Detailed explanation:  --kernel FooKernel: compile and get ISA for “Foo” kernel  --isa c:\files\Foo: specify the designated output location and prefix. This will creates files such as c:\files\Foo-Bonaire.amdisa, c:\files\Foo-Hawaii.amdisa, etc. | |
|  | | |
| --il arg | Path to output IL file(s). Requires --kernel. | |
| CodeXLAnalyzer foo.cl --kernel myKernel --il foo.il --asic Tahity | |
|  | | |
| --debugil arg | Path to output Debug IL file(s). | |
| CodeXLAnalyzer foo.cl --kernel myKernel --debugil foo.debugil | |
|  | | |
| --metadata arg | Path to output Metadata file(s). Requires --kernel. | |
| CodeXLAnalyzer foo.cl --kernel myKernel -- metadata foo.metadata | |
|  | | |
| -b [ --binary ] arg | Path to binary output file(s). | |
| CodeXLAnalyzer foo.cl --kernel myKernel –b foo.bin | |
|  | | |
| --suppress arg | Section to omit from binary output. Repeatable. | |
|  | CodeXLAnalyzer foo.cl --kernel myKernel –suppress .source –b foo.bin | |
|  | | |
| -k [ --kernel ] arg | Kernel to analyze or make IL or ISA. | |
| CodeXLAnalyzer foo.cl --kernel myKernel –b foo.bin | |
|  | | |
| --OpenCLoption arg | OpenCL compiler options. Repeatable. | |
| CodeXLAnalyzer foo.cl --kernel myKernel --isa foo.isa --OpenCLoption -cl-enable-mad --OpenCLoption -w | |
|  | | |
| -D [ --define ] arg | Define symbol or symbol=value. Repeatable. | |
| CodeXLAnalyzer foo.cl --kernel myKernel --isa foo.isa –D myDefine | |
|  | | |
| --csv-separator arg | | Override to default separator for analysis items. |
| CodeXLAnalyzer foo.cl --kernel myKernel --analysis foo.csv –csv-separator # |
|  | | |
| --livereg arg | | Path to the live register analysis output file (note that “--isa arg" must be used in conjunction with the --livereg switch for live register analysis to be performed, since the live register analysis engine works by analyzing the ISA disassembly).  Note: this is a beta feature of CodeXLAnalyzer CLI. You can find more info about it in the “Generating and Interpreting CodeXLAnalyzer CLI’s Live Register Analysis Report” |
| CodeXLAnalyzer –s cl –c Bonaire --kernel myKernel --isa foo.isa --livereg fooLiveRegFile.txt --il fooIl.il myClFile.cl |

Usage examples:

Create binary files output/foo-ASIC.bin for foo.cl.

CodeXLAnalyzer foo.cl --bin outdir/foo

List the kernels available in foo.cl.

CodeXLAnalyzer foo.cl --list-kernels

Generate ISA and performance statistics for all ASICs for kernel myKernel which is defined in foo.cl

CodeXLAnalyzer foo.cl --kernel myKernel --analysis foo.csv

List the ASICs that the runtime supports.

CodeXLAnalyzer --list-asics

Generate ISA and IL code for Cypress (ASIC) for kernel myKernel which is defined in source file foo.cl

CodeXLAnalyzer foo.cl --kernel myKernel --il foo --isa foo --asic Cypress

Generate ISA code and live register analysis report for Iceland (ASIC), for kernel myKernel which is defined in source file foo.cl

CodeXLAnalyzer foo.cl --kernel myKernel –livereg livereg.txt --isa foo --asic Iceland

#### Using CodeXLAnalyzer Command Line Interface to compile DirectX shaders

CodeXLAnalyzer command line tool supports offline compilation and statistics generation for DirectX shaders. Naturally, it is supported on Windows only.

The Analyzer works in 2 stages:

1. By default, CodeXLAnalyzer compiles the shader using the D3D Compiler. CodeXL ships with a default compiler (d3dcompiler\_47.dll). Unless the --DXLocation command line switch is specified, CodeXL will use the default compiler. The compilation can also go through Microsoft’s FXC tool instead of directly through the D3D compiler. To use FXC, you need to specify the –FXC command line switch with the location of FXC.exe.
2. CodeXLAnalyzer compiles the D3D ASM code generated by the D3D Compiler into AMD ISA, and generates statistics.

Regardless to the chosen compilation chain, all build errors and warnings will be printed in the command line window.

Details of available commands:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Details of available commands: | | | | |
| -h | View available options | | | |
| CodeXLAnalyzer.exe  -h | | | |
|  | | | | |
| -s | Specify the source language for the compilation. For all DX options, you need to specify the source since OpenCL is the default language. Available options: DXAsm, HLSL, CL (default) | | | |
| CodeXLAnalyzer.exe  -s hlsl | | | |
|  | | | | |
| -s hlsl –l | View supported ASICS for DirectX (also- this is default ASICs list for compilation) | | | |
| CodeXLAnalyzer.exe  -s hlsl –l | | | |
|  | | | | |
| -s hlsl -l --verbose | View supported ASICS with detailed marketing names | | | |
| CodeXLAnalyzer.exe  -s hlsl –l – verbose | | | |
|  | | | | |
| --isa arg | Path to output ISA disassembly file(s). This command requires compilation switches such as –f, -p. | | | |
| CodeXLAnalyzer.exe -s hlsl -f VsMain -p vs\_5\_0 -p vs\_5\_0 c:/files/myShader.fx --isa c:/files/myShader.isa  Detailed explanation:  –s HLSL: run HSLS compilation (vs CL)  –f VsMain: compile and get ISA for “VsMain” function  –p vs\_5\_0: the target in D3DCompile: ps\_5\_0, vs\_5\_0, gs\_5\_0  c:/files/myShader.fx: the file to compile  –isa c:/files/myShader.isa: specify the designated output file. | | | |
|  | | | | |
| -c [ --asic ] arg | By default, compilation will be done to all default available devices (run –s hlsl –l to view the list). This option enables you to choose which ASIC to target. This option is repeatable. | | | |
| CodeXLAnalyzer.exe -s hlsl -f VsMain -p vs\_5\_0 c:/files/myShader.fx --isa c:/files/myShader.isa -c Hawaii -c Kaldini | | | |
|  | | | | |
| -a [ --analysis ] | Path to output analysis file. Requires compilation. | | | |
| CodeXLAnalyzer.exe -s hlsl -f VsMain c:/files/myShader.fx -p vs\_5\_0 -a c:/files/myShader.csv | | | |
|  | | | | |
| -D | To compile using defines use –D, repeatable. | | | |
| CodeXLAnalyzer.exe -s hlsl -f VsMain c:/files/myShader.fx --isa c:/files/myShader.isa -p vs\_5\_0 –D MyDefine | | | |
|  | | | | |
| -- DXFlags | Compile using DXFlags | | | |
| CodeXLAnalyzer.exe -s hlsl -f VsMain c:/files/myShader.fx --isa c:/files/myShader.isa -p vs\_5\_0 --DXFlagx 1 | | | |
|  | | | | |
| --version | View the Catalyst driver version installed | | | |
| CodeXLAnalyzer\_d.exe -s hlsl –version | | | |
|  | | | | |
| --DXLocation | | | Compile with a specific D3Dcompiler DLL. Note that the path need to be in quotes if it contains spaces. | |
| CodeXLAnalyzer.exe -s hlsl -f VsMain -p vs\_5\_0 c:/files/myShader.fx --isa c:/files/myShader.isa -c Hawaii --DXLocation "C:\Program Files (x86)\Windows Kits\8.1\bin\x86\d3dcompiler\_47.dll" | |
|  | | | | |
| -s DXAsm | | Compile from a blob. In this case, the application assumes D3DCompilation is done so it skips this and does the AMD DX Compilation only. | | |
| CodeXLAnalyzer.exe -f  VsMain -s DXAsm -p vs\_5\_0 c:/files/myShader.obj  --isa c:\temp\dxTest.isa | | |
|  | | | | |
| --FXC | | Compile using FXC. You need to provide the full FXC path and arguments, and need to use /Fo switch. Also, the output of the FXC file should be the input file for KA. Use with the DXAsm switch. | | |
| CodeXLAnalyzer.exe -s DXAsm -f  VsMain -p vs\_5\_0 c:/files/myShader.fx --isa c:\files\myShader.isa -c tahiti --FXC "\"C:\Program Files (x86)\Windows Kits\8.1\bin\x86\fxc.exe\" /E VsMain /T vs\_5\_0  /Fo c:/files/myShader.obj c:/files/myShader.fx" | | |
|  | | | | |
| -s DXAsmT | | Compile from a blob represented as text (DX ASM as Text) . In this case, the application assumes D3DCompilation is done so it skips this and does the AMD DX Compilation only. | | |
| CodeXLAnalyzer.exe -s DXAsmT -f  VsMain -p vs\_5\_0 c:/files/myShaderAsText.asm --isa c:\files\myShader.isa -c tahiti | | |
|  | | | | |
| --DumpMSIntermediate arg | | | | When using this option, the MS Compilation output will be saved in the designated file |
| CodeXLAnalyzer.exe -s hlsl -f ps\_main c:\temp\Pixel.psh  --isa c:\temp\dxOutput.isa -c Tahiti -c Bonaire -p ps\_5\_0  --DumpMSIntermediate c:\temp\DumpMSIntermediate.txt |
|  | | | | |
| --livereg arg | | Path to the live register analysis output file (note that “--isa arg" must be used in conjunction with the --livereg switch for live register analysis to be performed, since the live register analysis engine works by analyzing the ISA disassembly).  Note: this is a beta feature of CodeXLAnalyzer CLI. You can find more info about it in the “Generating and Interpreting CodeXLAnalyzer CLI’s Live Register Analysis Report” | | |
| CodeXLAnalyzer.exe -s hlsl -c Fiji -f VSMain -p vs\_5\_0 --isa c:\temp\.txt --livereg c:\temp\lreg.txt c:\temp\dx\BasicHLSL11\_VS.hlsl | | |

Usage examples:

1. Suppose that you would like to compile and generate the ISA code of a DirectX pixel shader (C:\Users\shaders\Render.hlsl), using a the default D3D compiler that ships with CodeXL, and would like the output files which contain the ISA code to be located at c:\temp\ and be named myISA-<device-name>.isa:

CodeXLAnalyzer.exe -s hlsl -p ps\_5\_0 -f RenderPS --isa c:\temp\myISA.isa C:\Users\shaders\Render.hlsl

1. Suppose that you would like to compile and generate the ISA code of a DirectX pixel shader (C:\Users\shaders\Render.hlsl), using a specific D3D compiler (C:\Program Files (x86)\Windows Kits\8.1\bin\x86\d3dcompiler\_47.dll), and would like the output files which contain the ISA code to be located at c:\temp\ and be named myISA-<device-name>.isa:

CodeXLAnalyzer.exe -s hlsl -p ps\_5\_0 -f RenderPS --DXLocation "C:\Program Files (x86)\Windows Kits\8.1\bin\x86\d3dcompiler\_47.dll" --isa c:\temp\myISA.isa C:\Users\shaders\Render.hlsl

1. Suppose that you would like to compile and generate the ISA code of a DirectX pixel shader (C:\Users\shaders\Render.hlsl) only for “Iceland”, using a specific D3D compiler (C:\Program Files (x86)\Windows Kits\8.1\bin\x86\d3dcompiler\_47.dll), and would like the output files which contain the ISA code to be located at c:\temp\ and be named myISA-<device-name>.isa:

CodeXLAnalyzer.exe -s hlsl -p ps\_5\_0 -f RenderPS –c Iceland --DXLocation "C:\Program Files (x86)\Windows Kits\8.1\bin\x86\d3dcompiler\_47.dll" --isa c:\temp\myISA.isa C:\Users\shaders\Render.hlsl

#### Using CodeXLAnalyzer Command Line Interface to compile OpenGL and Vulkan programs

CodeXLAnalyzer.exe command line tool supports compilation and statistics generation OpenGL and Vulkan programs.

Details of available commands:

|  |  |  |
| --- | --- | --- |
| Details of available commands: | | |
| -h | | View available options |
| CodeXLAnalyzer.exe  -h |
|  | | |
| -s | | Specify the source platform for the compilation. |
| CodeXLAnalyzer.exe  -s opengl  CodeXLAnalyzer.exe  -s vulkan |
|  | | |
| -s <platform> -l | | View supported ASICS for DirectX (also- this is default ASICs list for compilation) |
| CodeXLAnalyzer.exe  -s opengl –l |
|  | | |
| --isa arg | | Path to output ISA disassembly file(s). |
| --vert <arg> | | Full path to vertex shader’s location |
| --tesc <arg> | | Full path to tessellation control shader’s location |
| --tese <arg> | | Full path to tessellation evaluation shader’s location |
| --geom <arg> | | Full path to geometry shader’s location |
| --frag <arg> | | Full path to fragment shader’s location |
|  | | |
| -c [ --asic ] arg | | By default, compilation will be done to all default available devices This option is repeatable. |
|
|  | | |
| -a [ --analysis ] | | Path to the performance statistics output file. Requires compilation. |
|  | |  |
| --version | View the Catalyst driver version installed | |
| CodeXLAnalyzer\_d.exe -s opengl –version | |

Usage examples:

1. To build an OpenGL program with a vertex shader and a fragment shader attached and generate ISA

CodeXLAnalyzer.exe -s opengl --vert c:\shaders\glVertex.vert --geom c:\shaders\glGeom.geom --isa c:\output\myISA.txt

1. To build an Vulkan program with a vertex shader and a geometry shader attached, and generate ISA and performance statistics

CodeXLAnalyzer.exe -s vulkan --vert c:\shaders\vkVertex.glsl --geom c:\shaders\vkGeom.geom --isa c:\output\myISA.isa –a c:\output\myStats.txt

#### Generating and Interpreting CodeXLAnalyzer CLI’s Live Register Analysis Report

Using CodeXLAnalyzer CLI’s live register analysis report, you can better understand the register usage of your HLSL shaders and OpenCL kernels throughout their execution. Live register analysis is a beta feature of CodeXLAnalyzer CLI, and it currently only fully supports HLSL shaders and partially supports OpenCL kernels.

Generating a live register analysis report for your kernel or shader:

As mentioned in the “Details of available commands” section above, in order to generate a live register analysis report, you need to make sure that your invocation command includes the following command line switches:

1. -- isa <arg> which instructs CodeXLAnalyzer to generate ISA disassembly for your kernel/shader
2. --livereg <arg> which instructs CodeXLAnalyzer to perform a live register analysis of the generated ISA disassembly

Usage examples:

CodeXLAnalyzer.exe –s cl –c Fiji --kernel DCT --isa c:\output\.isa --livereg c:\output\livereg.txt DCT\_Kernels.cl

Let’s break down the above command to understand its structure:

1. “-s cl” instructs CodeXLAnalyzer to work in OpenCL mode
2. “-c Fiji” sets Fiji as the target ASIC
3. “--kernel DCT” sets DCT as the target kernel (this is the kernel to be analyzed; it is defined in DCT\_Kernels.cl, which is the last argument in the above command)
4. “--isa c:\output\.isa” instructs CodeXLAnalyzer to generate an ISA disassembly file and save it in c:\output with a “.isa” file extension. The output file name is generated automatically.
5. “--livereg “c:\output\livereg.txt” instructs CodeXLAnalyzer to perform live register analysis, save the report in c:\output, and use “livereg.txt” as the report file name’s suffix and extension.

After running the above command, we see the following output files in c:\output (our destination folder):

Fiji\_DCT.isa

Fiji\_DCT\_livereg.txt

The live register analysis report file is Fiji\_DCT\_livereg.txt.

For HLSL, the usage is similar:

CodeXLAnalyzer.exe -s hlsl -c Fiji -f VSMain -p vs\_5\_0 --isa c:\temp\.txt --livereg c:\temp\lreg.txt c:\temp\dx\BasicHLSL11\_VS.hlsl

1. “-s hlsl” instructs CodeXLAnalyzer to work in HLSL mode
2. “-c Fiji” sets Fiji as the target ASIC
3. “-f VSMain” sets VSMain as the target shader
4. “--isa c:\output\.isa” instructs CodeXLAnalyzer to generate an ISA disassembly file and save it in c:\output with a “.isa” file extension. The output file name is generated automatically.
5. “--livereg “c:\output\livereg.txt” instructs CodeXLAnalyzer to perform live register analysis, save the report in c:\output, and use “livereg.txt” as the report file name’s suffix and extension.

Report structure:

If you open up the live register analysis report file, you will see that it is a plain textual file. Each line in the file gives a snapshot of the register usage when the PC is at that specific ISA line. Each line in the report is of the following format:

<line number> | <number of live registers> | <list of registers + access type> | <ISA instruction>

Where:

1. <line number> is the number of the current ISA disassembly line
2. <number of live registers> is the number of live registers when the PC is at that ISA line
3. <list of registers + access type> is a list of n columns. Each column (except for the first one) refers to a register:
   1. ‘^’ indicates a register is written to
   2. ‘v’ indicates a register is read
   3. ‘x’ is used for a register which is written and read
   4. ‘:’ is used for register where the contents must be preserved across this instruction (live register)
   5. A blank means that the register is not used
4. <ISA instruction> is the ISA disassembly of the relevant instruction

At the end of the report, you will find a summary in the following format:

Maximum # VGPR used <Max VGPR used>, # VGPR allocated: <Number of VGPR allocated>

Where:

1. <Max VGPR used> is the number of VGPRs actually used throughout the code
2. <Number of VGPR allocated> is the number of VGPRs that were allocated

Two things to remember when inspecting the live register analysis report are:

1. If the number of live registers is lower than the number of allocated registers, it indicates that the SC could reduce VGPRs without spilling by introducing moves.
2. If registers have a very long liveness range without read/write access, those registers could be likely spilled at low cost.

Here is a sample live register analysis report:

1 | 9 | ::::::: :: | label\_basic\_block\_1: s\_swappc\_b64 s[2:3], s[2:3]

2 | 9 | ::::::: :: | s\_andn2\_b32 s0, s9, 0x3fff0000

3 | 9 | ::::::: :: | s\_mov\_b32 s1, s0

4 | 9 | ::::::: :: | s\_mov\_b32 s2, s10

5 | 9 | ::::::: :: | s\_mov\_b32 s3, s11

6 | 9 | ::::::: :: | s\_mov\_b32 s0, s8

7 | 9 | ::::::: :: | s\_buffer\_load\_dwordx8 s[4:11], s[0:3], 0x00

8 | 9 | ::::::: :: | s\_buffer\_load\_dwordx8 s[12:19], s[0:3], 0x20

9 | 9 | ::::::: :: | s\_waitcnt lgkmcnt(0)

10 | 10 | ^ :::v::: :: | v\_mul\_f32 v0, s7, v7

11 | 11 | :^ :::v::: :: | v\_mul\_f32 v1, s11, v7

12 | 12 | ::^ :::v::: :: | v\_mul\_f32 v2, s15, v7

13 | 13 | :::^:::v::: :: | v\_mul\_f32 v3, s19, v7

14 | 12 | x:::::v ::: :: | v\_mac\_f32 v0, s6, v6

15 | 12 | :x::::v ::: :: | v\_mac\_f32 v1, s10, v6

16 | 12 | ::x:::v ::: :: | v\_mac\_f32 v2, s14, v6

17 | 12 | :::x::v ::: :: | v\_mac\_f32 v3, s18, v6

18 | 11 | x::::v ::: :: | v\_mac\_f32 v0, s5, v5

19 | 11 | :x:::v ::: :: | v\_mac\_f32 v1, s9, v5

20 | 11 | ::x::v ::: :: | v\_mac\_f32 v2, s13, v5

21 | 11 | :::x:v ::: :: | v\_mac\_f32 v3, s17, v5

22 | 10 | x:::v ::: :: | v\_mac\_f32 v0, s4, v4

23 | 10 | :x::v ::: :: | v\_mac\_f32 v1, s8, v4

24 | 10 | ::x:v ::: :: | v\_mac\_f32 v2, s12, v4

25 | 10 | :::xv ::: :: | v\_mac\_f32 v3, s16, v4

26 | 9 | vvvv ::: :: | exp pos0, v0, v1, v2, v3

27 | 5 | ::: :: | s\_buffer\_load\_dwordx4 s[4:7], s[0:3], 0x40

28 | 5 | ::: :: | s\_buffer\_load\_dwordx4 s[8:11], s[0:3], 0x50

29 | 5 | ::: :: | s\_buffer\_load\_dwordx4 s[0:3], s[0:3], 0x60

30 | 5 | ::: :: | s\_waitcnt expcnt(0)

31 | 6 | ^ ::v :: | v\_mul\_f32 v0, s6, v10

32 | 7 | :^ ::v :: | v\_mul\_f32 v1, s10, v10

33 | 8 | ::^ ::v :: | v\_mul\_f32 v2, s2, v10

34 | 7 | x:: :v :: | v\_mac\_f32 v0, s5, v9

35 | 7 | :x: :v :: | v\_mac\_f32 v1, s9, v9

36 | 7 | ::x :v :: | v\_mac\_f32 v2, s1, v9

37 | 6 | x:: v :: | v\_mac\_f32 v0, s4, v8

38 | 6 | :x: v :: | v\_mac\_f32 v1, s8, v8

39 | 6 | ::x v :: | v\_mac\_f32 v2, s0, v8

40 | 6 | :::^ :: | v\_mov\_b32 v3, 1.0

41 | 7 | ::::^ :: | v\_mov\_b32 v4, 0

42 | 7 | vvvv: :: | exp param0, v0, v1, v2, v3

43 | 4 | vv vv | exp param1, v12, v13, v4, v3

44 | 0 | | s\_endpgm

Maximum # VGPR used 13, # VGPR allocated: 14