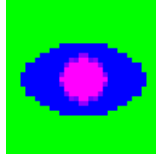


**CUDA VisionWorkbench**  
**Programming and Development Guide**  
v 1.1.0.5  
2/22/2009



### CVWB Code Structure

CUDA VisionWorkbench (CVWB) is implemented in MS Visual Studio 2005 and arranged in a solution, *VisionWorkbench.sln*, which is contained in the main / topmost folder named “*VisionWorkbench*” extracted from the .zip archive. This solution contains two projects, *VisionWorkbench.csproj* and *CudaVisDLL.vcproj*, which are contained in two subdirectories within the solution directory named “*VisionWorkbench*” and “*CudaVisDLL*”.

The code and components for the CVWB user interface is arranged in *VisionWorkbench.csproj*, which is a C# .Net v2.0 project in VS2005 contained within the “*VisionWorkbench*” sub-directory in the solution. This is mainly a high level application layer or front-end to handle user input, manage interactions with the system, allocate and initialize resources, and generate the interactive UI. It is heavily dependent upon the second project, *CudaVisDLL*, to function.

The code for the reusable core processing module used for CVWB is arranged in *CudaVisDLL.vcproj*, which is a standard Win32 DLL project implemented in C++ in VS2005 without MFC or STL and contained within the “*CudaVisDLL*” sub-directory of the solution. The module produced by building this project, *CudaVisDLL.dll*, is a proto imaging library that contains processing functions and some environment and object management utilities. Processing operations within *CudaVisDLL* are implemented both in CUDA and in n-threaded C++ (n presently defaults to the number of processors enumerated on the system, but can be set by the user from the CVWB Processing Menu).

*CudaVisDLL.dll* is highly independent and can be used without the *CUDA VisionWorkbench* front end application. Functions in *CudaVisDLL.dll* may be called by any application—*CUDA VisionWorkbench* is just an example with some specific purposes. The DLL has an external interface that exposes both the host multithreaded C++ processing functions and CUDA / GPU based processing functions.

The CUDA functions in the DLL are internally composed of a C++ wrapper that calls CUDA functions. Implementation of each CUDA function is fully contained in a separate .cu file named after the function. The host CPU functions in the DLL are standard C++ implementations with explicit worker threading. Future versions of the DLL may add explicit SIMD/SSE intrinsics and optimizations to the host CPU processing versions.

### CVWB Visual Studio Solution Configuration

*VisionWorkbench.sln* has been configured so that rebuilding the solution satisfies 3 somewhat-distinct purposes:

- 1) Creating and positioning requisite files (.exe , .dll , manifest , etc) composing a finished stand-alone application in its entirety (not counting system prerequisites). After a solution Rebuild All, these files are completely contained within the appropriate directory for the rebuilt configuration (release or debug version):
  - a. For Windows XP, the stand alone binaries may be found in the following folders:
    - ..\My Documents\Visual Studio 2005\Projects\VisionWorkbench\bin\Win32\Release
    - ..\My Documents\Visual Studio 2005\Projects\VisionWorkbench\bin\Win32\Debug
    - ..\My Documents\Visual Studio 2005\Projects\VisionWorkbench\bin\x64\Release
    - ..\My Documents\Visual Studio 2005\Projects\VisionWorkbench\bin\x64\Debug

- b. For Windows Vista, , the stand alone binaries may be found in the following folders:
    - ..\Documents\Visual Studio 2005\Projects\VisionWorkbench\bin\Win32\Release
    - ..\Documents\Visual Studio 2005\Projects\VisionWorkbench\bin\Win32\Debug
    - ..\Documents\Visual Studio 2005\Projects\VisionWorkbench\bin\x64\Release
    - ..\Documents\Visual Studio 2005\Projects\VisionWorkbench\bin\x64\Debug
- 2) Allowing seamless debug, setting of break points and single stepping from within the front end application (C#) code down into / through the DLL (C++) code and back.
- 3) Creating and positioning all files needed for a Click-Once deployment installer package, when executing Publish from the front-end VisionWorkbench application project.
- a. For Windows XP, the folder for this is
    - ..\My Documents\Visual Studio 2005\Projects\VisionWorkbench\Publish
  - b. For Windows Vista
    - ..\Documents\Visual Studio 2005\Projects\VisionWorkbench\Publish

## Key CVWB Code Files

### Front End

VisionWorkbench.sln	This loads everything in VS2005.
MainForm.cs	The main windows form code file, including primary event handlers
MainFuncs.cs	Additional functions closely linked to the MainForm
CoreImage.cs	Class with all data and methods for core and GUI processing
ImageData.cs	Class for to hold/maintain underlying image core data arrays
ImageDimensions.cs	Thin class for basic image dimensions
ImageManager.cs	Class for loading/unloading image data from/to files or functions to core data
ImageTestPattern.cs	Class for generating test patterns.
ThumbData.cs	Class for Thumb core data and GUI data
BufferedGraphics.cs	Class for doing pipelined drawing to graphics backbuffer with flip to screen
Util.cs	Generic utility functions and static constants for front-end
DLLImports.cs	Static class importing unmanaged DLL functions
KernelDialog.cs	Custom Dialog for Correlation and Convolution Matrix Configuration
HistoDialog.cs	Custom Dialog for reviewing Histogram results
TextEntryDialog.cs	Custom Dialog for build text strings (such as data file path tags)
Help.html	HTML file for detailed CVWB information
ProgrammingRef.html	HTML file for information about programming/development with CVWB
RevisionLog.html	HTML file for CVWB Revision history information

### Core Computation

CudaVisDLL.dll	Compled/Linked binary DLL file
CudaVisDLL.h	Header exporting DLL functions, forward-declaring internal functions, etc.
CudaGeneral.h	Header for CUDA specific global and forward declarations and constants
Utilities.h	Global constants used with Utilities
CudaVisDLL.cpp	Main DLL object
HostProcessing.cpp	C++ host processing function implementations
CudaWrappers.cpp	C++ Wrappers to underlying CUDA based GPU functions
CudaGeneral.cu	CUDA functions for managing GPU enumeration & buffer allocation
Utilities.cpp	Utilities for trace logging, error handling and logging, and timing.
*****.cu	CUDA Image processing files (Convo1655.cu, Convo1677.cu, etc)
cudaart.dll	Interface DLL for NVIDIA CUDA runtime library

### Visual Studio Integration

Cuda21.rules	rules file for use in Visual Studio with CUDA .cu code files
usertype.dat	file listing CUDA data types and functions for Visual Studio syntax highlighting

## CVWB Programming Notes

### Image Buffers

For convenience and optimization, CVWB automatically allocates 5 standard image buffers both on the host and on the GPU (in GMEM). The buffers are all the same size in pixel terms, equal to the height and width of the current image, and having the same bit depth (the current processing bit depth). Whenever a change is made by the operator to image dimensions or bit depth, changes in the underlying buffers are handled automatically. Note that actual allocated memory on the GPU is increased as necessary based upon the image width to pad for 128 bit alignment, but this is managed in the DLL transparent to the application level.

The host buffers, which are presently allocated in managed memory in the .Net GUI application, are dedicated-use buffers for purpose of demonstration: Input, Output, Scratch, Multiply, Subtract. Input and Output are self explanatory. Scratch is an intermediate buffer used in Series ops. Multiply and Subtract are buffers that, for sake of demonstration, are filled once at program initialization with data representing 2D gain and offset compensation matrices for a camera (1 value of gain and offset for each pixel). These 5 host buffers can be used for purposes other than the ones chosen here, but these are what they are used for and called for purposes of demonstration.

The GPU buffers, which are allocated in GPU GMEM, are both flexible-use and dedicated-use buffers for purpose of demonstration. They are in an indexed array simply numbered 0-4. For convenience, [1] and [2] are reserved to persistently hold the 2D Multiply image data and 2D Subtract image data from the host, respectively. This facilitates and simulates a common situation in machine vision 2D image gain and offset compensation need to be repeatedly applied with each new frame acquired... To keep latency to a minimum, these are thus kept in GMEM for fast reuse. (This takes a lot of memory, but as long as we have this to spare, we use it). The other buffers on the GPU ([0], [3] and [4]) are flexible-use buffers. Most commonly, buffer [0] is used as an input buffer, buffer [3] is used as an output buffer and buffer[4] is used as a scratch buffer for intermediate results. But this isn't always the case for series ops on the GPU.

When you call an operation for the GPU, you need to tell the DLL where the data is coming from (~what kind of pointer is being sent, a host pointer or a GPU pointer). iBufLoc is an array of indices used for this. The first position of this array, iBufLoc [0], must be assigned a value corresponding to the origin of first image pointer argument of the GPU function being called. The second position, iBufLoc [1] must be assigned a value corresponding to the origin of the second image pointer argument of the function call. Most functions only have 2 images involved, but for functions with more image args, the 3d position, iBufLoc [2], must be assigned a value corresponding to the origin of the third image pointer argument of the function call. And so on...

If the source of an image is in GPU GMEM, specify a number in the range of 0,1,2,3,4 for the corresponding iBufLoc array member. 0-4 inclusive are the indices of the GPU buffer pointer array.

If the source of the image is a host buffer, specify -1 for the corresponding iBufLoc array member. -1 tells the function that the origin is in host memory, that the pointer argument is a host memory pointer, and that a transfer from host to GPU (or GPU to host) must be undertaken.

For example, 2D image multiplication is a ternary operation with an input image, a 2D multiplier image and an output image. To setup iBufLoc to take the input from the host, use a multiplier image stored in GPU buffer 1 and return the result to GPU buffer 3, make assignments as follows prior to the call and then pass a pointer to iBufLoc.

```
// 2D Image Mult
iBufLoc[0] = -1;
iBufLoc[1] = 1;
iBufLoc[2] = 3;
```

```
// the 0th argument image buffer pointer is on the host (a cudaMemcpy will be used, Host pointer to GPU buffer index 0)
// the 1st argument image buffer is on the GPU with index 1: pre-allocated and assigned, so NO cudaMemcpy
// the 2nd argument image buffer is on the GPU with index 3: pre-allocated and assigned, so NO cudaMemcpy needed
```

## Notes on Intended Interactive Main Display and Thumbnail GUI Behavior

### Thumbnail Views

- 1) The large **red** cursor rectangle in the Main Display corresponds exactly, always to the large **red** cursor rectangle (i.e., the perimeter of the thumbnail) in the thumbnail views. This will either be a 64 x 64 pixel block for low mag or a 32 x 32 pixel block for high mag (see details on **Thumbnail Magnification** below).
- 2) The **red** coordinates in the thumbnail views correspond exactly, always to the position of the **red** reticle, which is fixed at the relative center of the whole thumbnail view, i.e. at the center of the large **red** cursor rectangle.
- 3) The small **green** cursor rectangle in the Main Display corresponds exactly, always to the small **green** cursor rectangle in the thumbnail views. This will always be an 8x8 pixel block, regardless of mag.
- 4) The **green** coordinates in the thumbnail views correspond exactly, always to the position of the **green** reticle, which is fixed at the relative center of the small **green** cursor rectangle.
- 5) The position of the **green** reticle and small **green** cursor rectangle default to the center of the thumbnail view (same position as the **red** reticle and **red** cursor rectangle).
- 6) You can set the relative position of the **green** reticle and small **green** cursor rectangle by double clicking anywhere in either of the thumbnail view client areas.
- 7) The underlying image data grey values for the 8x8 pixel region corresponding to the small **green** cursor rectangle in both the input and output thumbnail views are shown in the formatted **green** textbox matrices positioned directly below the thumbnail views.
- 8) Attempting to set the current location in the main image closer than  $\frac{1}{2}$  the **red** cursor rectangle width isn't allowed (to prevent reading outside of the underlying image). The **red** cursor rectangle will clamp at the nearest allowable position if you try this.
- 9) Attempting to set the **green** reticle and small **green** cursor rectangle locations in the thumbnail views closer than  $\frac{1}{2}$  the **green** cursor rectangle width isn't allowed (to prevent reading outside of the underlying image). The **green** cursor rectangle will snap to the nearest allowable position if you try this.

### Thumbnail Magnification

- 10) The magnification (ratio of Display Pixels to underlying input/output image data pixels) of the thumbnail views can be changed using the radio buttons above the thumbnail views.
- 11) The available low and high magnification options will be 4:1 and 8:1 if the Windows system DPI is set to 96 DPI and will be 5:1 and 10:1 respectively if the Windows system DPI is set to 120 DPI.
- 12) The underlying input/output image data depicted in the thumbnails will always be a 64 x 64 pixel block for low mag or a 32 x 32 pixel block for high mag.
- 13) Changing magnification does not normally change the center location of thumbnails, large **red** cursor rectangles or **red** reticle, unless you are at high mag and the Current Location is within 32 image pixels of the perimeter or corner of the image and then change to low mag. In this exceptional circumstance, the current location will be bumped by just enough to keep the large **red** cursor rectangle from going outside the bounds of the image.
- 14) If the small **green** cursor rectangles and **green** reticle are centered in the thumbnail view, changing magnification also generally does not change the center location of the small **green** cursor rectangles or **green** reticle, unless you are at high mag with current location up against the extreme perimeter or corner of the image and then change to low mag. In this exceptional circumstance, again, the current location will be bumped by just enough to keep the large **red** cursor rectangle from going outside the bounds of the image.
- 15) The relative position of the **green** reticle and small **green** cursor rectangle within the thumbnail view, along with the corresponding coordinates displayed, are reset to the center of the thumbnail view (same position as the **red** reticle and **red** cursor rectangle) when you change magnification.
- 16) You can force a reset of the relative position of the **green** reticle and small **green** cursor rectangle back to the center of the thumbnail view by right-clicking on either thumbnail or Main Display clients and selecting **Reset Small Cursor**

### Interactive **AutoScroll** mode

- 17) Clicking in the **Main Display** client area engages **AutoScroll** mode. This effectively slews the thumbnail views to a *Current Location* in the underlying input and output image data corresponding to the *mouse position* in the Main Display client. This also engages updating of the coordinate displays and reticles (crosshairs) in the thumbnail views and updating of the cursor rectangles in the thumbnail views and the Main Display.
- 18) Clicking again in the Main Display client disengages **AutoScroll** mode.
- 19) Note that, due to limitations in mouse resolution and repetition rate of mouse events, when in **AutoScroll** mode it is likely that the Current Location, large **red** cursor rectangle and small **green** cursor rectangle in the Main Display will lag a bit and also *not quite* update to the extreme limit possible if the mouse is moved quickly from inside the Main Display client area outward past the client edge. The cursor locations (and associated thumbnail views and coordinate displays) will get “parked” a little short of the extreme possible limits. In this case, the locations and views can be moved the small additional distance using the Nudge keys (see detail on **Nudge Functionality** below).

### **Nudge Functionality**

- 20) Pressing one of the arrow keys (or 1,2,3,4,6,7,8,9 keys on the number pad) will cause a “nudge” movement of exactly 1 pixel in the underlying input and output image data and will force an update of thumbnail views and Main Display, along with all of the overlays.
- 21) Note that a nudge displacement of the Current Location of one pixel in the underlying data may not move anything in the Main Display at all because magnification of the main display is usually less than 1 (screen pixels for the main display are ~ 540x540 for screens at 96 DPI whereas most underlying images are 1k x 1k or more).
- 22) But since the thumbnail views are setup to have magnification > 1, (i.e. either 4:1 & 8:1 at 96 DPI / 5:1 & 10:1 at 120 DPI), a nudge of 1 underlying pixel will always move the image presented in the thumbnail view and also increment the coordinates displayed in the thumbnail view.

## Notes on “Smart Sleep” functionality within CudaVisDLL

CUDA functions run on the CPU asynchronously with respect to the host process thread that owns the GPU context and launches the CUDA function. A variety of mechanisms in the CUDA Runtime and Driver API’s, such as explicit “Async” calls and a Streaming API afford the ability to take advantage of this and also manage issues arising from it.

In the CUDA Runtime API, subsequent CUDA calls made from the host process thread that launched a CUDA function asynchronously will await completion of the earlier CUDA function call before commencing and thus constitute a barrier synchronization means. An explicit barrier synchronization function with no other purpose is provided also, `cudaThreadSynchronize()`. Placed after an asynchronous CUDA kernel launch, this function will block advancement of the host process thread that called the CUDA function until the kernel completes. Unfortunately, for CUDA 2.1 and earlier versions of CUDA, it does so using a spin-lock mechanism that saturates CPU core utilization for the CPU core running the host thread in question.

For functions that take more than a few milliseconds to complete, a workaround has been implemented in CudaVisDLL that effectively accomplish the asynchronous GPU function execution and host thread barrier synchronization without excessive CPU utilization from spin-locking. Depending upon the CUDA function this is used with, overall CPU utilization (for a quad core system) may be reduced to as low as 2%, even during a sequence of high-frequency repetitive calls to short but compute-intensive CUDA functions. This accomplished by a combination of:

- A) Calibration timing measurements made at library initiation by the `MeasureGPUFuncTimeFactors` function (see `CudaGeneral.h` and `CudaGeneral.cu` files),
- B) Some simple runtime computations and use of the `CreatWaitableTimer`, `SetWaitableTimer` and `WaitForSingleObject` functions within the `SmartSleep` function (see `Utilities.h` and `Utilities.cpp` files) and as limited by the `MINSLEEP` and `SLEEPFACTOR` constants (see `CudaGeneral.h` file),
- C) Runtime computations immediately following the asynchronous CUDA function call in question, and
- D) The `cudaEventCreate`, `cudaEventRecord` and `CudaEventSynchronize` functions.

An example of the usage of `SmartSleep` is given below:

```
// declarations
cudaError_t cuErr1, cuErr2;
cudaEvent_t cuStartEvent, cuStopEvent;
cudaEventCreate(&cuStartEvent);
cudaEventCreate(&cuStopEvent);

...
// run the main convo kernel
cudaEventRecord(cuStartEvent, 0);
Convolve1633Compute <<<gridSz, blockSz, szShMemLength>>> (usSource, usDest, szGbPixPitch,
    iImageWidth, iImageHeight, uiBlockImageByteWidth, iRadius, fScale);
cudaEventRecord(cuStopEvent, 0);
SmartSleep(dFuncDelayMS[CONVO33] * iImageWidth * iImageHeight);
cuErr1 = cudaGetLastError();
cuErr2 = cudaEventSynchronize(cuStopEvent);

// get the timing and check errors
cudaEventElapsedTime(&fCuElapsedTime, cuStartEvent, cuStopEvent);
cudaEventDestroy(cuStartEvent);
cudaEventDestroy(cuStopEvent);
dTimes[2] = fCuElapsedTime/1000.0f;
```

As mentioned above, this approach isn’t effective for asynchronous computations that take less than about 2 milliseconds to complete. Nonetheless, the `SmartSleep` function used as demonstrated in CudaVisDLL is effective for achieving CPU offload for reasonably intensive CUDA computation functions of greater than a couple of milliseconds in durations, leaving the host CPU and OS available and responsive while the GPU is asynchronously doing teraflops computing.

## Configuring Visual Studio 2005 for use with CVWB and CUDA

**Note:** The following steps 1 - 4 affect **all** projects being manipulated on a Visual Studio installation

### 1) Add the Library, Include and Bin paths used for the CUDA Toolkit and SDK

- a) Start Visual Studio 2005
- b) Select “Tools->Options” from the center of the main menu bar at the top of the Visual Studio window. The *Options* dialog box will appear.
- c) Expand the “Projects and Solutions” node from the tree view at the left and select “VC++ Directories”.
- d) On the top right of the dialog, select “Library Files” from the “Show directories for:” list.
- e) Click the “Add” icon to create a new line in the folder list, and enter:     **\$(CUDA\_LIB\_PATH)**
- f) Left-Click the down arrow several times to move this path to the bottom of the list to ensure the default library paths are searched first.
- g) At the top right of the dialog, select “Include Files” from the “Show directories for:” list.
- h) Click the “Add” icon to create a new line in the folder list, and enter:     **\$(CUDA\_INC\_PATH)**
- i) Left-click the down arrow several times to move this path to the bottom of the list to insure the default library paths are searched first.
- j) Then click the “Add” icon to create a new line in the folder list, and enter:  
                                  **\$(NVSDKCUDA\_ROOT)\common\inc**
- k) Left-click the down arrow several times to move this path to the bottom of the list to insure the default library paths are searched first.
- l) At the top right of the dialog, select “Executable Files” from the “Show directories for:” list.
- m) Click the “Add” icon to create a new line in the folder list, and enter:     **\$(CUDA\_BIN\_PATH)**
- n) Left-Click the down arrow several times to move this path to the bottom of the list to insure the default library paths are searched first.

### 2) Add CUDA Toolkit and SDK lib files to your project

- a) Right-Click on the project name in the Solution Explorer, and select Properties from the popup menu
- b) At the project Property Pages dialog, select Linker-> Input->Additional Dependencies
- c) If not already present, add: “cudart.lib cuda.lib” (for each configuration such as Debug and Release).
- d) To resolve linker conflicts, depending upon the project type, you may need to make an entry under “Ignore Specific Library” in the same dialog box. For example, you may need to ignore MSVCRT.lib for the Release version of a DLL project or LIBCMT.lib for the Debug version of a DLL project.

### 3) Set up syntax highlighting for CUDA .cu files

- a) This will allow different keywords to be highlighted in color just like Visual Studio does by default for .c, .h, .cpp, etc. files with built in support.

- b) Check to see if you have a file called “usertype.dat” in your Visual Studio install directory within the \Common7\IDE folder (**default path: C:\Program Files\Microsoft Visual Studio 8\Common7\IDE for 32 bit windows, C:\Program Files (x86)\Microsoft Visual Studio 8\Common7\IDE for 64 bit windows**).
- i) If you do not have a “usertype.dat” file in this directory copy this file from the Cuda VisionWorkbench solution directory to the \Common 7\IDE folder in the Visual Studio install path.
- ii) If you already have a “usertype.dat” file open the “usertype.dat” file from the above file and copy the contents and paste them into your pre-existing “usertype.dat” file in the Visual Studio \Common7\IDE directory, then save and close the file.
- c) Run Visual Studio
- d) Select “Tools-> Options” from the center of the main menu bar at the top of the Visual Studio window. The **Options** dialog box will appear.
- e) Expand the “Text Editor” node from the tree view at the left and select “File Extension”.
- f) At the top left of the dialog in the box labeled “Extension”, enter “.cu”.
- g) Select “Microsoft Visual C++” in the drop down menu to the immediate right, and then click the “Add” button further to the right and the “OK” button at the bottom right of the dialog.
- h) Close and restart Visual Studio.

#### 4) Turn on line number display

- a) Open the **Options** dialog again (Tools->Options from the main menu).
- b) Select “Text Editor” from the left hand tree view and select “All Languages”.
- c) Check “Line numbers” to turn on line numbering.

**Note:** A useful trick to remember is to use **CTRL-g** at any time in a code file to jump to a particular line number.

#### 5) Setup Custom Build Rules files in Visual Studio to manage custom build options for CUDA

CUDA files (.cu files) are compiled in Visual Studio 2005 by invocation of the nvcc.exe compiler using a “Custom Build Rules” file, which is an xml file that tells Visual Studio to invoke the nvcc compiler, passes along desired switches and arguments to nvcc.exe, and also incorporates formatting information to present these rules for viewing/modification in convenient dialog boxes within VS 2005.

Custom Build Rules in a rules file can be applied to all of the files in a VS 2005 project, a group of files in the project, or on a file-by-file basis within a project. Once the rules file is installed in the project, settings exposed by the rules file can be customized for each project configuration (release, debug, emu, etc) of each .cu file, using convenient dialog boxes generated by VS 2005 and based upon the information in the xml rules file.

To configure custom build rules file in your project, do the following:

- a) **Obtain the rules file and store it in a standardized location**
  - (i) This document assumes the developer is using the **Custom Build Rules** file “Cuda21.rules”. This file is provided in the Cuda VisionWorkbench solution directory.
  - (ii) Copy this rules file it to the following directory for availability by all of your CUDA projects:

**C:\Program Files\NVIDIA Corporation\NVIDIA CUDA SDK\common\scripts**

**Note:** This directory should have been created by the default installation of the CUDA SDK.

- ii) The main options exposed in this particular Custom Build Rules file (Cuda.rules) correspond to the nvcc.exe options described in the NVCC reference manual “nvcc\_2.x.pdf” that comes with the CUDA Toolkit. Refer to that document for a detailed description of nvcc function and options.

**b) Install the rules file in the Project**

- i) Right-Click on the **Project** name for the project containing .cu files in the **Solution Explorer** pane, and select “Custom Build Rules” from the pop-up menu.
- ii) This will cause the **Visual C++ Custom Build Rules Files** dialog to pop up. Click on the “Find Existing” button:
- iii) Then browse to and double click on the “Cuda.rules” file in the **Find Existing Rule File** dialog
- iv) If you see a dialog box asking whether you want to add the directory containing the rules file to the Visual Studio rules file search path, click the **"Yes"** button. This tells Visual Studio to always include this path when searching for rules files and will make this rules file accessible to all of your VS projects:
- v) Returning to the **Visual C++ Custom Build Rules Files** dialog box, now check the box to the left of the “Cuda Build Rule v2.1.x” listing in the “Available Rule Files” pane, and then click the “OK” button.

**c) To customize CUDA/NVCC settings applicable to ALL .cu files in your project**

- i) Set the configuration (Release, Debug, etc) you wish the options to apply to using the drop down combo box menu in the toolbars at the top of the Visual Studio IDE window:
- ii) Right-Click on the VS 2005 project name in the **Solution Explorer** and select “Properties” from the popup menu.
- iii) At the “Project Name” **Property Pages** dialog, select the “CUDA Build Rule 2.1.x” item and find the option you wish to customize. Adjust the setting as desired and then click the “OK” button. Settings changed in this manner will apply to **all** .cu files for the currently selected configuration (Release, Debug, etc).
- iv) For example, a common setting for all cu files in the Release configuration would be the preprocessor constant **WIN32**, which is entered as a string in the “Preprocessor” field.
- v) As another example, a common setting for all cu files in the Debug configuration would be the preprocessor defined constants **WIN32** and **\_DEBUG**:

**d) To customize settings for ONE .cu file or a SubGroup of .cu files in your project**

- i) Set the configuration (Release, Debug, etc) you wish the options to apply to using the drop down combo box menu in the toolbars at the top of the Visual Studio IDE window:
- ii) For each configuration (i.e. Release, Debug, Emu) select a .cu file (or multiple .cu files using Ctrl-Left-click for multiple files) in your **Source Files** list, then Right-click on the file(s) chosen and select “Properties”.
- iii) At the **Property Pages** dialog for the file, select the “General” item and click the drop-down arrow next to the “Tool” entry. Then find and select “CUDA Build Rule v2.1.x” on the drop down list and then click the “OK” button to exit the dialog and accept the default custom build rule options (which are specified within the rules file).
- iv) Select the desired custom build rule item to set under the “CUDA Build Rule v2.1.x” node, choose the sub-option in the pane to the right, and set the option as desired. This setting will apply to the selected files for the currently selected configuration.
- v) As an example, consider a situation where you want to override the default **arch** setting of **sm\_10** for a couple of .cu files in your project that are optimized to take advantage of features only present in later GPU’s, (e.g. SMEM Atomics).
  - (1) To do this, set the configuration (Release, Debug, etc) and select the file(s) in the source file list, Right-Click on the file(s) selected, chose the “Properties” item from the popup menu.

- (2) Then choose the “General” item under the “CUDA Build Rule v 2.1.x” node, click the drop-down arrow to the right of the “GPU Architecture” field, and select the sm\_xx option desired. This setting will then be set on all selected files for the currently selected configuration (Release, Debug, etc).