CUDAfy

Project	CUDAfy
Title	CUDAfy User Manual
Reference	
Client Reference	
Author(s)	Nicholas Kopp, Hybrid DSP Systems
Date	March 17, 2011

Revision History

Date	Changes Made	Issue	Initials
March 17, 2011	Support to CUDAfy V0.3. First public release.	0.3	Nko

Hybrid DSP Systems

Email: info @ hybriddsp . com

Web: www.hybriddsp.com

INTRODUCTION

CUDAfy is a set of libraries and tools that permit general purpose programming of NVIDIA CUDA Graphics Processing Units (GPUs) from the Microsoft .NET framework. Its aim is to be the leading set of tools for this task; combining flexibility, performance and ease of use.

The CUDAfy SDK comprises the following:

- CUDAfy .NET Reflector Add-in
- CUDAfy Library
- CUDAfy Host Library
- CUDAfy Math Library (Optional)
- CUDAfy by Example demo projects
- CUDAfy Examples demo projects

The .NET Reflector Add-in converts .NET code into CUDA code. To use it you will need Red Gate's.NET Reflector which is available from www.red-gate.com

As a developer you will also require the NVIDIA CUDA Toolkit. The latest version that CUDAfy supports is version 3.2. You can obtain this from: http://developer.nvidia.com/object/cuda_3_2_downloads.html

It is highly recommended that the user first learns the basics of CUDA. The NVIDIA website is a good starting point as is the book *CUDA by Example* by Sanders and Kandrot.

GENERAL CUDAFY PROCESS

The general process for working with CUDAfy is to:

1. Add references to Cudafy.dll and Cudafy.Host.dll from your .NET project
2. Add the Cudafy and Cudafy.Host namespaces to source files (**using** in C#)
3. Add a parameter of **GThread** type to GPU functions and use it to access thread, block and grid information as well as specialist synchronization and local shared memory features.
4. Place a **Cudafy** attribute on the functions.
5. Fire up .NET Reflector and add the CUDAfy add-in. This consists of a language and a GUI element. The language is CUDA and the GUI allows the sub-selection of types to convert.
6. Click the Cudafy button to generate a Cudafy Module. This is an xml file that can be loaded into a **GPGPU** instance. The **GPGPU** type allows you to interact seamlessly with the GPU from your .NET code.

CUDAFY VERSIONS

As of writing the following versions of the CUDAfy SDK are planned:

- CUDAfy Express
- CUDAfy Standard
- CUDAfy Professional

CUDAFY EXPRESS

This is a free version that supports non-commercial; non-redistributable 32-bit application development (can run on 32-bit or 64-bit Windows). It includes the Cudafy and Cudafy.Host libraries, Cudafy Module Viewer and Cudafy .NET Reflector Add-in. Support is provided via the forum at www.hybriddsp.com.

CUDAFY STANDARD

The Standard version is a low cost SDK, licensed per developer seat, that permits the creation of commercial, 32-bit and 64-bit applications that can be freely redistributed. Furthermore it includes an easy to use Fourier Transform and Basic Linear Algebra library (Cudafy.Math is a .NET wrapper for CUBLAS and CUFFT).

CUDAFY PROFESSIONAL

This version adds an extended math and signal processing library as well as source code and a one year support contract. Support can be extended for a reduced price.

Standard and Professional include all minor version upgrades and major version upgrades at a reduced price.

PREREQUISITES

SUPPORTED OPERATING SYSTEMS

Windows XP SP3

Windows Vista

Windows 7

Both 32-bit / 64-bit OS versions are supported.

SUPPORTED .NET VERSIONS

CUDAfy is built against Microsoft .NET Framework 3.5.

SUPPORTED HARDWARE

All NVIDIA CUDA capable GPUs with compute capability 1.1 or higher are supported. Note that some language features may not available in earlier versions of CUDA.

DEVELOPMENT REQUIREMENTS

The following is required when developing with CUDAfy:

- NVIDIA CUDA Toolkit 3.2
- Red Gate's .NET Reflector 7.0 Standard or higher

These are not included in the CUDAfy download must be downloaded separately.

RECOMMENDED TOOLS

Although not necessary the use of Visual Studio 2010 Professional is recommended. For 32-bit applications Visual Studio Express can be used. NVIDIA Parallel NSight may be installed however its use from .NET applications is not currently implemented.

Although Cudafy simplifies the use of CUDA a basic understanding of CUDA is essential especially in terms of architecture (threads, blocks, grids, synchronization). There are various websites with useful information and the book *CUDA BY EXAMPLE* (Sanders and Kandrot) is highly recommended (many of the CUDAfy examples included in the SDK are direct .NET versions of the code in this book).

RECOMMENDED SET-UP

PC SPECIFICATION

To make use of the built in emulation that CUDAfy offers, you will ideally be using a recent multi-core AMD or Intel processor. Emulation of blocks containing thousands of threads is very inefficient for CPUs due to the massive thread management overhead.

GRAPHICS CARD

The introduction by NVIDIA of the Fermi architecture (compute capability 2.x) brings a significant advancement in terms of programming features and performance. Fermi allows better performance with less tuning of GPU code. Although CUDAfy supports compute capability from 1.1 the focus is on supporting Fermi and therefore we recommend using it where possible. A good value card would be a GT460 or GT560.

SOFTWARE

Ideally you will have:

Windows 7 64-bit

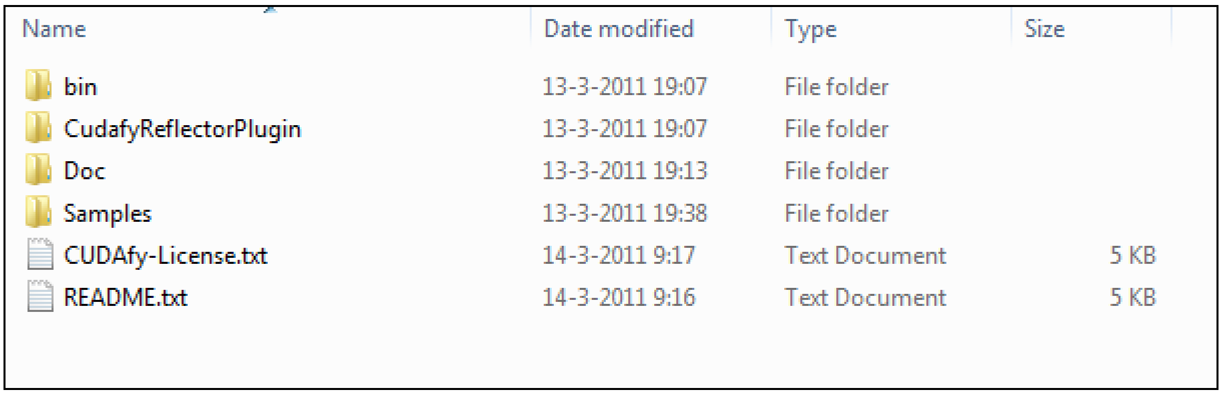
Visual Studio 2010 Professional

Red Gate .NET Reflector VSPro

INSTALLATION

CUDAFY SDK

The CUDAfy SDK is available as a zip file download from www.hybriddsp.com. Unzip to a convenient location.



Name	Date modified	Type	Size
bin	13-3-2011 19:07	File folder	
CudafyReflectorPlugin	13-3-2011 19:07	File folder	
Doc	13-3-2011 19:13	File folder	
Samples	13-3-2011 19:38	File folder	
CUDAfy-License.txt	14-3-2011 9:17	Text Document	5 KB
README.txt	14-3-2011 9:16	Text Document	5 KB

The contents of the zip file include bin, CudafyReflectorPlugin, Doc, Samples directories and two text files: a license and a readme. The bin directory contains the libraries (DLLs) that you link to from your .NET application. These are listed below:

File	Description
CUDA.NET.dll	This is a .NET CUDA wrapper created by GASS. Typically you will not need to interact with this.
CUDA.NET.Readme.txt	Release notes for CUDA.NET
CUDA.NET.XML	Code insight information.
Cudafy.dll	Key library containing all language features required for creating GPU code in .NET. This includes main constructs such as the **Cudafy** attribute and the **GThread** class.
Cudafy.xml	Code insight information.
Cudafy.Host.dll	This library is used for loading in Cudafy modules and interacting with the GPU.

Cudafy.Host.xml	Code insight information.
CudafyModuleViewer.exe	A viewing tool for Cudafy modules.

The CudafyReflectorPlugin directory contains:

File	**Description**
CudafyReflectorPlugin.dll	Contains the .NET Reflector Cudafy GUI and language add-ins.

The Doc directory contains:

File	**Description**
CUDAfyVx.x.HtmlDocumentation.zip	Html documentation created from xml comments in source code. Unzip the contents and browse to the index.html file.

The Samples directory contains two sub-directories:

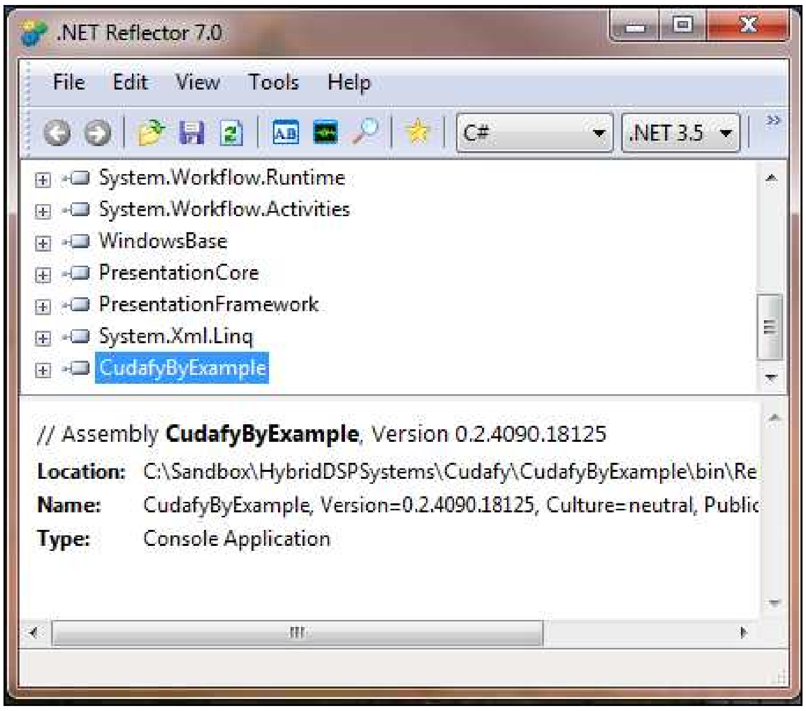
Directory	**Description**
CudafyByExample	This is a Visual Studio 2010 solution containing a project that demonstrates many of the features of Cudafy. The examples are based on the book *CUDA BY EXAMPLE* (Sanders and Kandrot). A copy of this book is highly recommended.
CudafyExamples	Another Visual Studio 2010 solution demonstrating Cudafy features not covered in CudafyByExample. These include dummy functions, complex numbers and multi dimensional arrays.

The root directory contains two text files:

Directory	Description
CUDAfy-License.txt	License agreement.
README.txt	Release notes.

INSTALLATION – CUDAFY .NET REFLECTOR ADD-IN

Follow the instructions from Red Gate for installing and starting .NET Reflector.



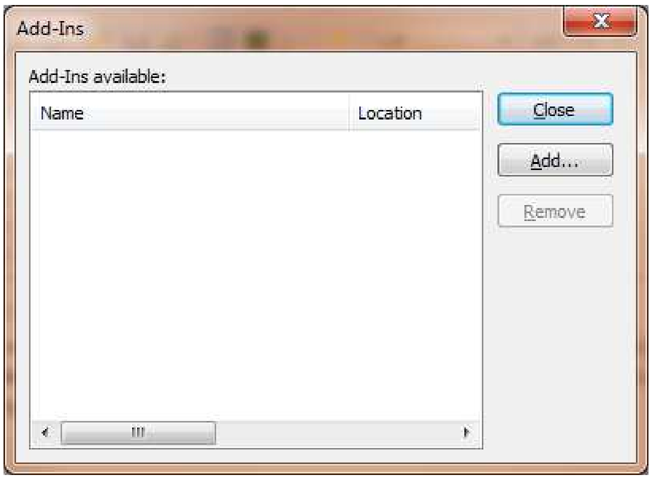
// Assembly **CudafyByExample**, Version 0.2.4090.18125

Location: C:\Sandbox\HybridDSPSystems\Cudafy\CudafyByExample\bin\Re

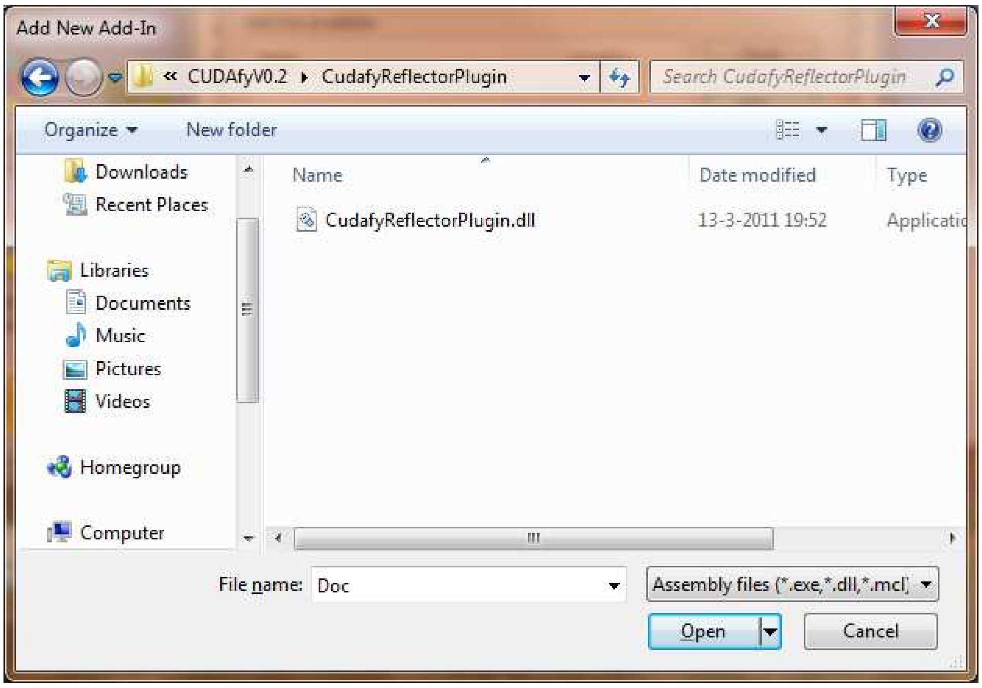
Name: CudafyByExample, Version=0.2.4090.18125, Culture=neutral, Public

Type: Console Application

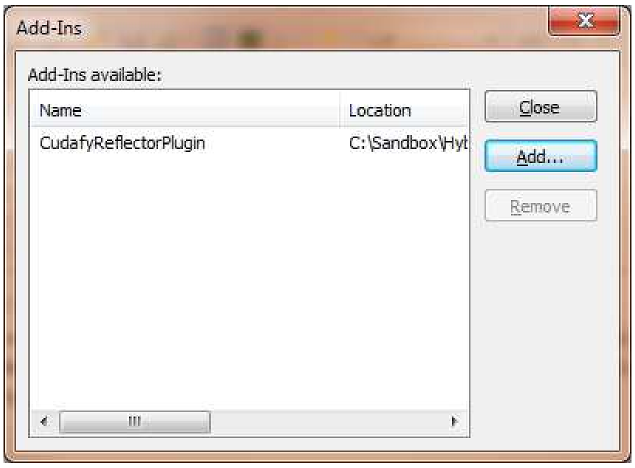
To manage add-ins click on **Tools : Add-Ins…**



Now click **Add...** An open-file dialog will appear. Navigate to the CudafyReflectorPlugin directory in the folder where you unzipped the CUDAfy SDK.

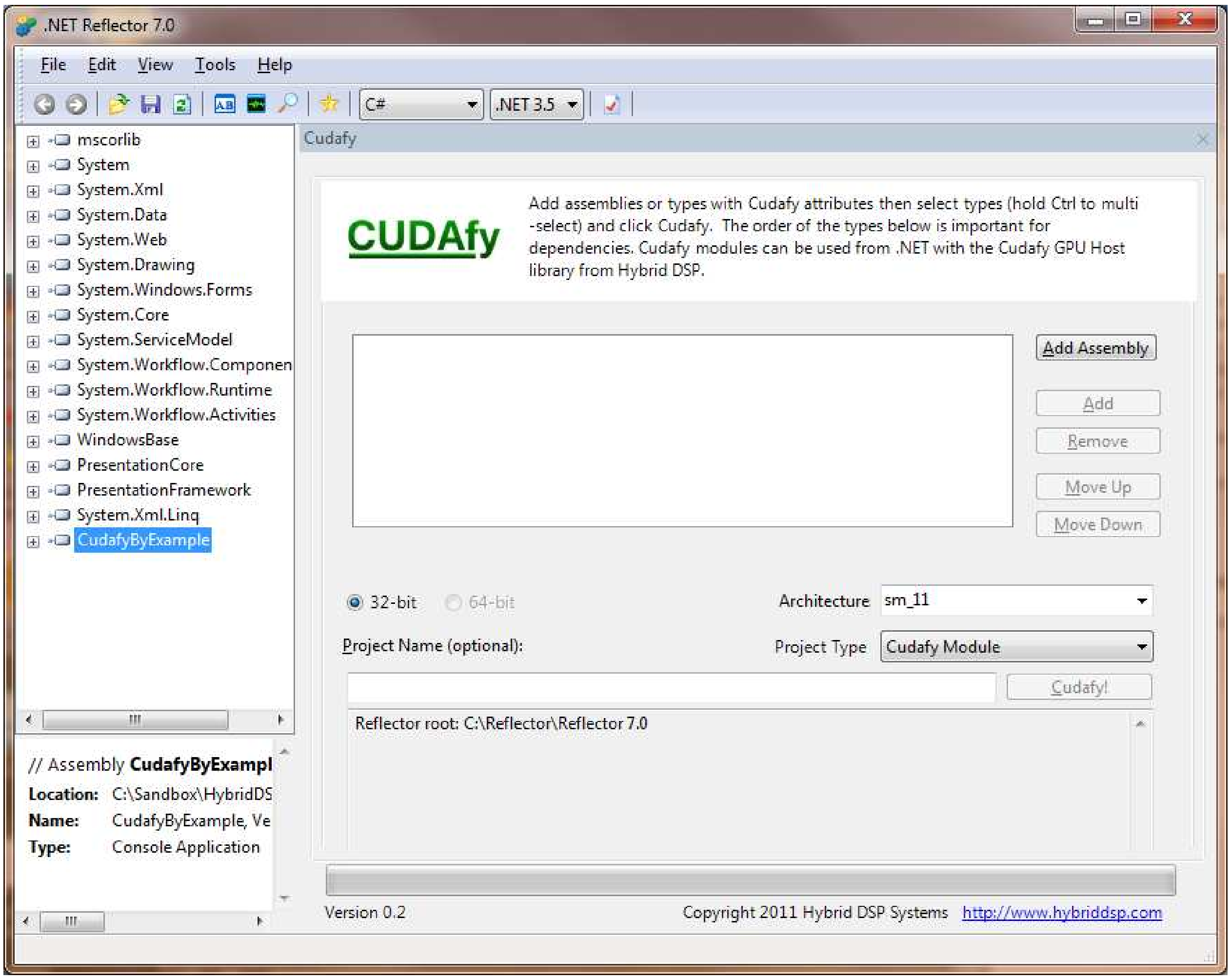


Select CudafyReflectorPlugin.dll and then click **Open**.



Click **Close** then go to the **Tools** menu again where you will find at the bottom a new menu item named **Cudafy**.

You will now see something like:



Add assemblies or types with Cudafy attributes then select types (hold Ctrl to multi -select) and click Cudafy. The order of the types below is important for dependencies. Cudafy modules can be used from .NET with the Cudafy GPU Host library from Hybrid DSP.

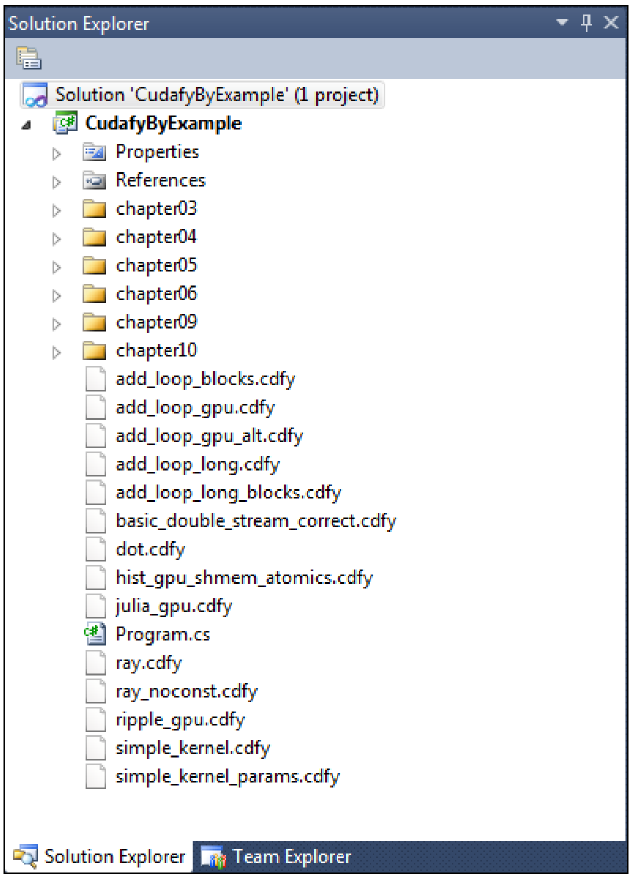
Reflector root: C:\Reflector\Reflector 7.0

// Assembly **CudafyByExampl**
Location: C:\Sandbox\HybridDS
Name: CudafyByExample, Ve
Type: Console Application

CUDAFY BY EXAMPLE

The quickest way to get up and running with CUDAfy is to take a look at the example projects. These can be found in the Samples sub-directory of the SDK. You may wish to make a copy of these directories before you begin building and modifying them – if so bear in mind that if you open the copies then the references to Cudafy.dll and Cudafy.Host.dll may be broken if the relative path is different. These two dlls are in the bin directory so re-add them if necessary.

Navigate to CudafyByExample. If you have Visual Studio 2010 installed you can simply click the solution file (*.sln). You will soon see something like this:



The folders chapter03 through chapter10 refer to the chapters of the book *CUDA BY EXAMPLE* (Sanders and Kandrot). The collection of *.cdfy files are pre-*Cudafied* .NET code modules. They were made using the CUDAfy .NET Reflector Add-in targeting compute capability 1.1 (except for the atomics example with needs 1.2 or higher).

Open the file Program.cs. Since this is a Console application this is the code that will run when you run it. The static **CudafyModes** class is a helper for storing our code generation and target settings so all examples can access them. Basically we set the code generation to CUDA C and the target to a CUDA GPU. You can also set to **Emulator** but it's more fun at this stage not to since the more complex examples will be painfully slow. The majority of the samples have an **Execute** method and our Main method simply calls each sequentially.

You are probably itching to press F5 and run but first check that the target architecture is x86 if you are running CUDAfy Express. Go to **Configuration Manager** and select **x86**. Failure to select this correctly will result in a **BadImageFormatException** and general bad news.

The various examples are described below:

HELLO_WORLD

This is only included to keep things in line with *CUDA BY EXAMPLE*. Hopefully no explanation is needed!

SIMPLE_KERNEL

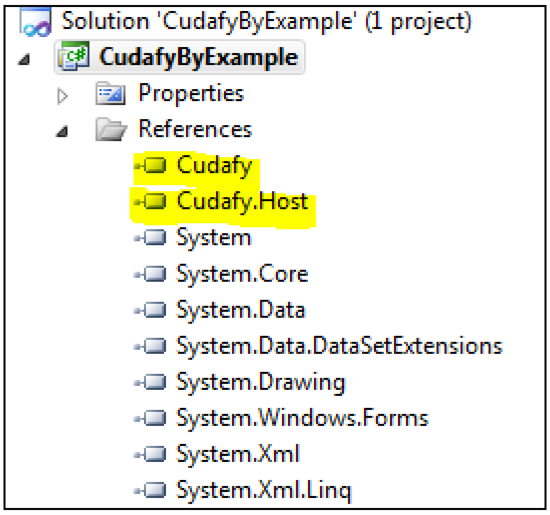
Now we are going to run a very simple function on the GPU. Functions running on a GPU are often referred to as *kernels*.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using Cudafy;
using Cudafy.Host;

namespace CudafyByExample
{

    public class simple_kernel
    {
        public static void Execute()
        {
            CudafyModule km = CudafyModule.Deserialize(typeof(simple_kernel).Name);
            if (!km.TryVerifyChecksums())
                Console.WriteLine(CudafyModes.csCRCWARNING);

            GPGPU gpu = CudafyHost.GetGPGPU(CudafyModes.Target);
            gpu.LoadModule(km);
            gpu.Launch(1, 1, "kernel");

            Console.WriteLine("Hello, World!");
        }

        [Cudafy]
        public static void kernel()
        {
        }
    }
}
```

You will see that we include two namespaces:

- Cudafy
- Cudafy.Host

If you click expand the references folder in the solution explorer you will see that they correspond to the two main Cudafy dlls of the same name as the namespaces.


Solution 'CudafyByExample' (1 project)
CudafyByExample
Properties
References
Cudafy
Cudafy.Host
System
System.Core
System.Data
System.Data.DataSetExtensions
System.Drawing
System.Windows.Forms
System.Xml
System.Xml.Linq

Now follows some key points of using Cudafy. The function we wish to run on the GPU is named **kernel**. We put an attribute on there name **Cudafy**. This tells the .NET Reflector Add-in that we wish to *cudafy* this method. A GPU method that is callable from a host application must return **void**. We will return to this later but briefly when CudafyByExample is compiled an executable is produced name **CudafyByExample.exe**. We open this in .NET Reflector and then select the types we wish to translate (cudafy). In this case we selected **simple_kernel**. This type contains only one item marked for cudafying – the method **kernel**. This method does nothing useful but importantly it still does this nothing useful business on the GPU. The output of the CUDAfy .NET Reflector Add-in is a file named **simple_kernel.cdfy**.

We load this into a **CudafyModule** object using this line:

```
CudafyModule km = CudafyModule.Deserialize(typeof(simple_kernel).Name);
```

Using **typeof** is a way of avoiding hardcoding the file name.

The next line does something interesting and can be a useful warning. We check whether the file **simple_kernel.cdfy** was created from the same version of .NET assembly as the currently available one. Since there is a good chance you are using the default *.cdfy files supplied the method **TryVerifyChecksums** will fail. There is a good reason for this. Imagine changing our **kernel** method to do something else such as add numbers together like the next example does. If we do not revisit .NET Reflector and again cudafy the **simple_kernel** type in the **CudafyByExample.exe** assembly we'd have a mismatch between our *.cdfy module and the .NET code. You do not really want this, or rather you want to know when this is the case so you at least know to look out for version issues.

Okay, on with show. The **CudafyHost** class is static and contains a method called **GetGPGPU**. We have stored the target type in our **Main** method in **Program.cs**. Hopefully it is set to **Cuda**, but there is nothing wrong with choosing **Emulator**. Either way you will get back a **GPGPU** object. This is your interface with the GPU in your computer. The **CudafyModule** we deserialized in the first line is loaded and then we can **Launch** our function. **Launch** is a dramatic sounding GPU term for starting a function on the GPU.

We will go into details of what the first two arguments are later but basically it means we are launching 1 x 1 = 1 thread. Later we'll be launching rather more threads in parallel. The third argument is the name of the function to run. Our module only has one but it could have many so it is required that you provide this. The name is "kernel" to match the name of the **kernel** method.

SIMPLE_KERNEL_PARAMS

This is a slightly more useful example in that it actually does some processing on the GPU though a CPU or even perhaps a calculator or doing the math in your head may be faster. Here we pass some arguments into our GPU function:

```
[Cudafy]
public static void add(int a, int b, int[] c)
{
    c[0] = a + b;
}
```

Since we cannot return any value from a GPU function our result is passed out via parameter **c**. Currently there is a limitation and the **Out** keyword is not supported so we use a vector instead. We need to actually allocate memory on the GPU for this even though it will contain only one **Int32** value.

```
int[] dev_c = gpu.Allocate<int>(); // cudaMalloc one Int32
```

If you take a look at the array **dev_c** in the debugger you'll see that it has length zero. You cannot and should not try to use variables that are on the GPU in your CPU side code. They act merely as pointers.

We launch the function with:

```
gpu.Launch(1, 1, "add", 2, 7, dev_c);
```

Put the arguments in the same order as the parameters of the **add** method. Finally we need to copy the result back to the CPU:

```
int c;
gpu.CopyFromDevice(dev_c, out c);
```

With any luck you should end up with the correct answer.

ENUM_GPU

GPUs can list their properties and these can be useful for your application. Access the properties for all CUDA GPUs via:

```
foreach (GPGPUProperties prop in CudafyHost.GetDeviceProperties(CudafyModes.Target,
false))
```

The first parameter is the GPU type and the second is whether to get advanced properties or not. Advanced properties require that the **cudart** DLL is available in addition to the standard **nvidia** dll.

ADD_LOOP_CPU

This sample demonstrates how we might add two vectors of length **N** on the CPU.

ADD_LOOP_GPU

And now how to do the same on the GPU. We allocate three arrays on the CPU and the GPU. As a short cut we can use an overloaded version of **Allocate** that takes a CPU array as argument and then allocates the equivalent memory on the GPU. You could get the same effect by passing the length in elements.

You will see that the Launch call passes the value **N** as first argument. We are going to launch **N** threads, so that means we will add each element of the arrays in a separate thread.

How does each **add** thread know what element to operate on? This is done by adding a **GThread** parameter to the GPU function. You do not need to specify an instance of this when launching as this will occur automatically. Within **GThread** there are several properties. For now we are interested in **blockIdx** and its **x** property.

```
[Cudafy]
public static void add(GThread thread, int[] a, int[] b, int[] c)
{
    int tid = thread.blockIdx.x;
    if (tid < N)
        c[tid] = a[tid] + b[tid];
}
```

Variable **tid** will work out to be a number between 0 and N - 1 inclusive for our N threads. Now each **add** knows who he is. The rest of the code should explain itself though the last three lines are important, especially for .NET developers not used to cleaning up their garbage:

```
// free the memory allocated on the GPU
gpu.Free(dev_a);
gpu.Free(dev_b);
gpu.Free(dev_c);
```

Here we explicitly release the memory we allocated on the GPU. The CUDAfy host (**GPGPU**) would also do this when it goes out of scope but since memory on a GPU is limited in comparison to that of the host and does not automatically cleanup it is good practice to do this.

ADD_LOOP_GPU_ALT

Basically the same as the previous sample but avoids the additional calls to **Allocate** by using overloads of **CopyToDevice**:

```
// copy the arrays 'a' and 'b' to the GPU
int[] dev_a = gpu.CopyToDevice(a);
int[] dev_b = gpu.CopyToDevice(b);
```

Since we do not specify a destination for our CPU arrays **a** and **b**, CUDAfy automatically creates them and returns the pointers **dev_a** and **dev_b**.

ADD_LOOP_LONG

Here we are adding two much longer vectors. Instead of adding each element in a separate thread, each thread will be responsible for adding N / 128 elements. The first argument in **Launch** is 128 which is the total number of threads.

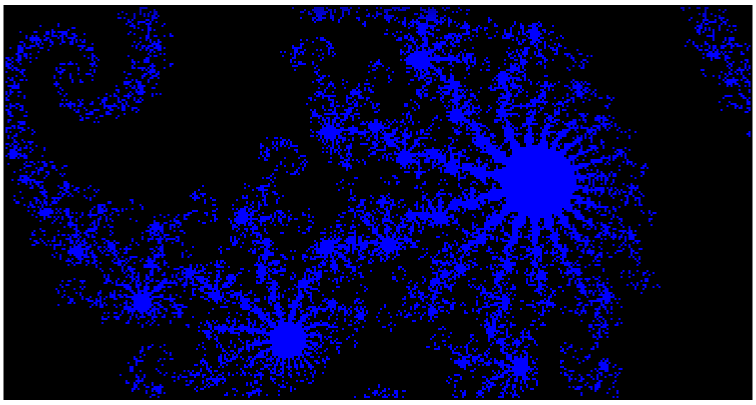
In our GPU function we need an additional GThread property. We are now interested in **blockIdx** and its **x** property and **gridDim** and its **x** property.

```
[Cudafy]
public static void add(GThread thread, int[] a, int[] b, int[] c)
{
    int tid = thread.blockIdx.x;
    while (tid < N)
    {
        c[tid] = a[tid] + b[tid];
        tid += thread.gridDim.x;
    }
}
```

Variable **tid** is incremented by the number of blocks in the grid (128) which is given by **gridDim.x**.

JULIA_CPU AND JULIA_GPU

These are graphical demos for CPU and GPU. On the GPU it makes use of 2D blocks of threads. Of note is the calling of a GPU function from another GPU function. Only GPU functions that can be launched must return **void**, others may return values.



ADD_LOOP_BLOCKS AND ADD_LOOP_LONG_BLOCKS

In CUDA you have grids, blocks and threads. Grids contain 1 or more blocks and blocks contain one or more threads. The earlier examples for adding vectors made us of grids and blocks. Now we use blocks and threads to obtain the same result. In more complex examples a combination is used.

DOT

This example introduces the concept of shared memory. This is memory shared between threads of the same block. There are good performance reasons for this and you are referred to the CUDA literature for background reading. To use shared memory from CUDAfy you call the **AllocateShared** method of **GThread**.

```
float[] cache = thread.AllocateShared<float>("cache", threadsPerBlock);
```

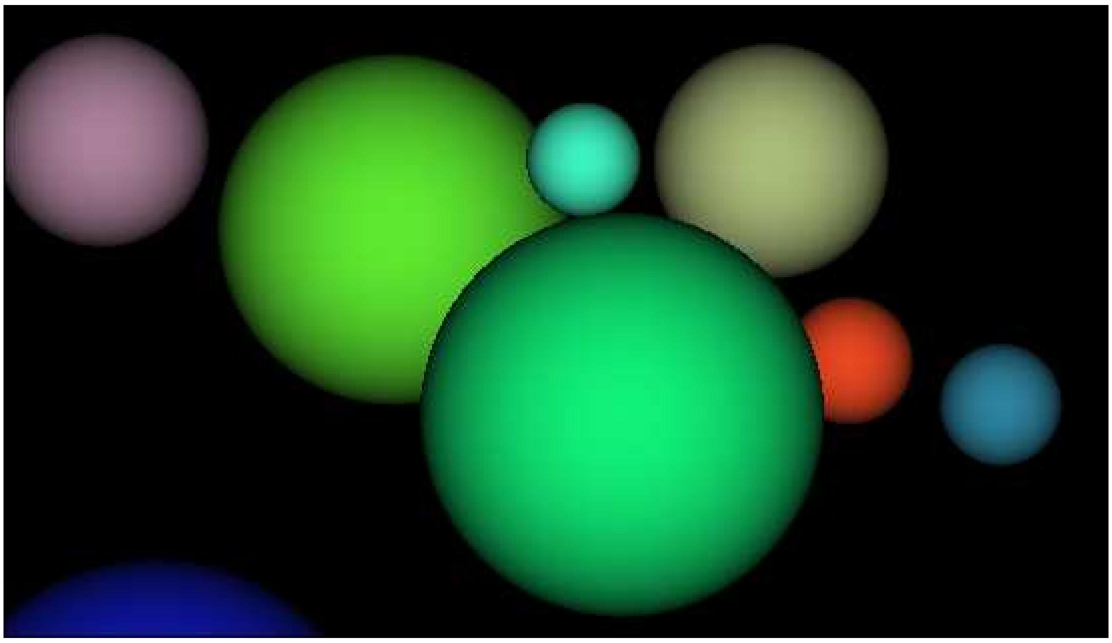
The parameters are an id and the number of elements. We get back an array of the type specified between the angle brackets. Another new concept is that of a barrier for the threads of a single block. This is necessary for synchronizing all the threads at a certain point.

```
            // synchronize threads in this block
            thread.SyncThreads();
```

RIPPLE

Another graphics demo that makes use of 2D blocks and a 2D grid. You will also see the use of a **GMath** class. **GMath** is a CUDAfy class that contains some specific versions of .NET **Math** methods. The reason is that some **Math** methods such as **Sqrt** only provide overloads for **double** and not **float**. When the CUDAfy .NET Reflector add-in translates to CUDA it would therefore add an unnecessary cast if **Math** was used, hence the use of **GMath**.

RAY AND RAY_NOCONST



These are two almost identical samples that illustrate a simple ray tracing implementation. They provide some insight into some other CUDA features exposed via CUDAfy, namely performance timing, constant memory and cudafying of structs. One example uses constant memory, the other does not. The difference you get in timing will vary depending on whether .NET and the GPU are ‘warmed up’, your GPU and the target compute capability used when creating the cudafy module. With the new Fermi cards there is not a significant difference.

The **Sphere** struct is declared as:

```
    [Cudafy]
    public struct Sphere
    {
        public float r;
        public float b;
        public float g;
        public float radius;
        public float x;
        public float y;
        public float z;

        public float hit(float ox1, float oy1, ref float n1)
        {
            float dx = ox1 - x;
            float dy = oy1 - y;
            if (dx * dx + dy * dy < radius * radius)
            {
                float dz = GMath.Sqrt(radius * radius - dx * dx - dy * dy);
                n1 = dz / GMath.Sqrt(radius * radius);
                return dz + z;
            }
            return -2e10f;
        }
    }
```

Placing the **Cudafy** attribute on classes does not work, only structs are supported. Operator overloading is also not currently supported. Be aware that all types on the GPU whether in a struct or copied between CPU and GPU or in a launch command, must be **blittable**. This means that they have to be in a standard number format – e.g. **byte**, **int**, **float**, **double**.

Constant memory is a special kind of memory on the GPU that can be written only by the host CPU and is read only for the GPU. It can in many circumstances be faster than the global memory of the GPU, however its size is rather small (typically 64K). In the sample with constant memory we have an array of **Sphere**s here:

```
        public const int SPHERES = 20;

        [Cudafy]
        public static Sphere[] s = new Sphere[SPHERES];
```

Note you should not put a **Cudafy** attribute on **SPHERES**. .NET Constants (const) are automatically placed into cudafied code. We copy the Spheres we created on the host to the GPU's constant memory with a special method, where **temp_s** is an array of **SPHERES Sphere**s:

```
       Sphere[] temp_s = new Sphere[SPHERES];
       ...
       ...
       gpu.CopyToConstantMemory(temp_s, s);
```

Finally we should look at the timer functionality. Timing GPU code is vital to ensure that the effort that goes into fine tuning is paying off. We start and stop a timer with:

```
       gpu.StartTimer();
       ...
       ...
       float elapsedTime = gpu.StopTimer();
```

HIST_GPU_SHMEM_ATOMICS

This is an example of a simple GPU algorithm that really shines. It makes use of shared memory and atomic operations. Atomic operations are an optimized way of performing some basic commands such as addition in a thread safe manner. They are accessible from .NET by using the **Cudafy.Atomics** namespace and will then appear as extension methods of **GThread**. Note that a GPU with compute capability of 1.2 or higher is needed.

```
        [Cudafy]
        public void histo_kernel(GThread thread, byte[] buffer, long size, 
uint[] histo)
        {

            // clear out the accumulation buffer called temp
            // since we are launched with 256 threads, it is easy
            // to clear that memory with one write per thread
            uint[] temp = thread.AllocateShared<uint>("temp", 256);
            temp[thread.threadIdx.x] = 0;
            thread.SyncThreads();

            // calculate the starting index and the offset to the next
            // block that each thread will be processing
            int i = thread.threadIdx.x + thread.blockIdx.x * thread.blockDim.x;
            int stride = thread.blockDim.x * thread.gridDim.x;
            while (i < size)
            {
                thread.atomicAdd(ref temp[buffer[i]], 1 );
                i += stride;
            }
            // sync the data from the above writes to shared memory
            // then add the shared memory values to the values from
            // the other thread blocks using global memory
            // atomic adds
            // same as before, since we have 256 threads, updating the
            // global histogram is just one write per thread!
            thread.SyncThreads();
            thread.atomicAdd(ref(histo[thread.threadIdx.x]),temp[thread.threadIdx.x]);
}
```

BASIC_DOUBLE_STREAM_CORRECT

GPUs can perform multiple functions in parallel. To do this we use stream ids. Stream id zero is the default and what has been implicitly used up until now. Commands with the same stream id are queued sequentially. Stream zero will synchronize any stream id so when doing parallel operations we want to avoid its use. Of course to do all this we need to make sure our commands are asynchronous. There are asynchronous versions of **CopyToDevice**, **Launch** and **CopyFromDevice**. The get the postfix **Async** and take an additional parameter that is the stream id. To make sure all the asynchronous commands are completed we use the **SynchronizeStream** method.

```
            // now loop over full data, in bite-sized chunks
            for (int i = 0; i < FULL_DATA_SIZE; i += N * 2)
            {
                gpu.CopyToDeviceAsync(host_aPtr, i, dev_a0, N, 1);
                gpu.CopyToDeviceAsync(host_bPtr, i, dev_b0, N, 2);
                gpu.CopyToDeviceAsync(host_aPtr, i + N, dev_a1, N, 1);
                gpu.CopyToDeviceAsync(host_bPtr, i + N, dev_b1, N, 2);
                gpu.LaunchAsync(N / 256, 256, 1, "kernel", dev_a0, dev_b0, dev_c0);
                gpu.LaunchAsync(N / 256, 256, 2, "kernel", dev_a1, dev_b1, dev_c1);
                gpu.CopyFromDeviceAsync(dev_c0, host_cPtr, i, N, 1);
                gpu.CopyFromDeviceAsync(dev_c1, host_cPtr, i + N, N, 2);
            }
            gpu.SynchronizeStream(1);
            gpu.SynchronizeStream(2);
```

Another difference here is that the data on the host needs to be allocated as pinned memory. This is a specially aligned data that offers higher performance and is a prerequisite for asynchronous transfers. We can allocate this memory on the host with **HostAllocate**. Instead of getting an array back we get an **IntPtr**. This is not as much fun as working with arrays so fortunately from CUDA 4.0 and the accompanying CUDAfy release this is no longer needed. For earlier versions you can either copy host arrays to and from pinned memory with **GPGPU.CopyOnHost()** or set values using the **IntPtr** extension method **Set**.

Remember to free the **IntPtrs** on the host and destroy the streams.

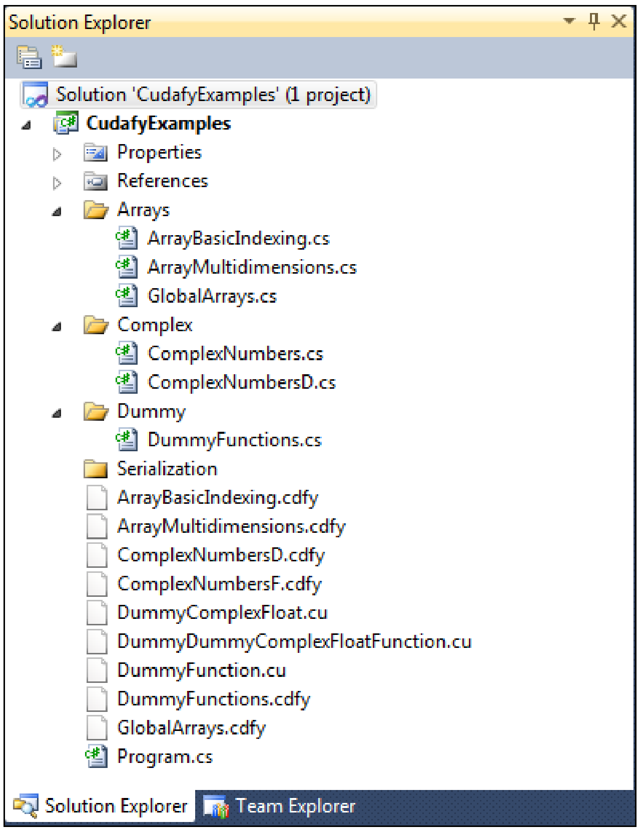
```
            gpu.HostFree(host_aPtr);
            gpu.HostFree(host_bPtr);
            gpu.HostFree(host_cPtr);
            gpu.DestroyStream(1);
            gpu.DestroyStream(2);
```

COPY_TIMED

This sample compares the read and write performance of normal CPU to GPU transfers with that of pinned memory to GPU transfers. Allocation of pinned memory was covered in the previous example. You should see a significant difference.

CUDAFY EXAMPLES

The second example project is called **CudafyExamples.sln**.



Samples cover arrays, complex numbers and dummy functions.

ARRAY BASIC INDEXING

Only a sub-set of the standard .NET functionality is supported for GPU side code. With future releases of CUDAfy and of NVIDIA's CUDA Toolkit this will be expanded. Strings are for example not supported as are many of default classes and methods we are used to in .NET (Array, DateTime, etc). However some are and these include the Length, GetLength and Rank members of arrays. You can freely use these in GPU code.

ARRAY MULTI DIMENSIONS

Typically we work with large arrays on GPUs. The reason for this is that small amounts of data are not very efficient for processing on GPU and can be far better handled on the CPU. CUDAfy supports one-, two- and three-dimensional arrays in global, constant and shared memory.

Jagged arrays are not supported. Use the notation [,] for 2D and [,,] for 3D.

GLOBAL ARRAYS

This collection of samples shows how to work with 1D, 2D and 3D arrays of values (Int32) and structs (ComplexFloat).

COMPLEX NUMBERS

Complex numbers are used very frequently in many disciplines. CUDA has a complex number type built in (float and double varieties) and CUDAfy supports this via **ComplexF** and **ComplexD**. These are in the Cudafy.Types namespace of Cudafy.dll. The real part is name **x** and the imaginary part **y**. A number of operations are provided:

- Abs
- Add
- Conj
- Divide
- Multiply
- Subtract

Bear in mind that due to the nature of floating point values the results you get with .NET and those with the GPU will not be exactly the same.

DUMMY FUNCTIONS

Say you already have some CUDA C code and you want to use it from .NET, then dummies are the answer. The attribute **CudafyDummy** used in the same manner as the **Cudafy** attribute makes this possible. Items marked with **CudafyDummy** are handled differently by the .NET Reflector Add-in. Instead of converting to CUDA C the add-in expects there to be a ***.cu** file with the same name as the function or struct and that it also contains a function or struct with that name.

```
[CudafyDummy]
public struct DummyComplexFloat
{
    public DummyComplexFloat(float r, float i)
    {
        Real = r;
        Imag = i;
    }
    public float Real;
```

```
        public float Imag;
        public DummyComplexFloat Add(DummyComplexFloat c)
        {
            return new DummyComplexFloat(Real + c.Real, Imag + c.Imag);
        }
    }
```

A file named DummyComplexFloat.cu must exist and contain code such as this:

```
struct DummyComplexFloat
{
       public: float Real;
       public: float Imag;

       // Methods
       DummyComplexFloat(float  r, float  i)
       {
              Real = r;
              Imag = i;
       }


       __device__ DummyComplexFloat  Add(DummyComplexFloat  c)
       {
              return DummyComplexFloat((Real + c.Real), (Imag + c.Imag));
       }
};

        [CudafyDummy]
        public static void DummyDummyComplexFloatFunction(DummyComplexFloat[] result)
        {
            for (int i = 0; i < XSIZE; i++)
            {
                result[i] = result[i].Add(result[i]);
            }
        }
```

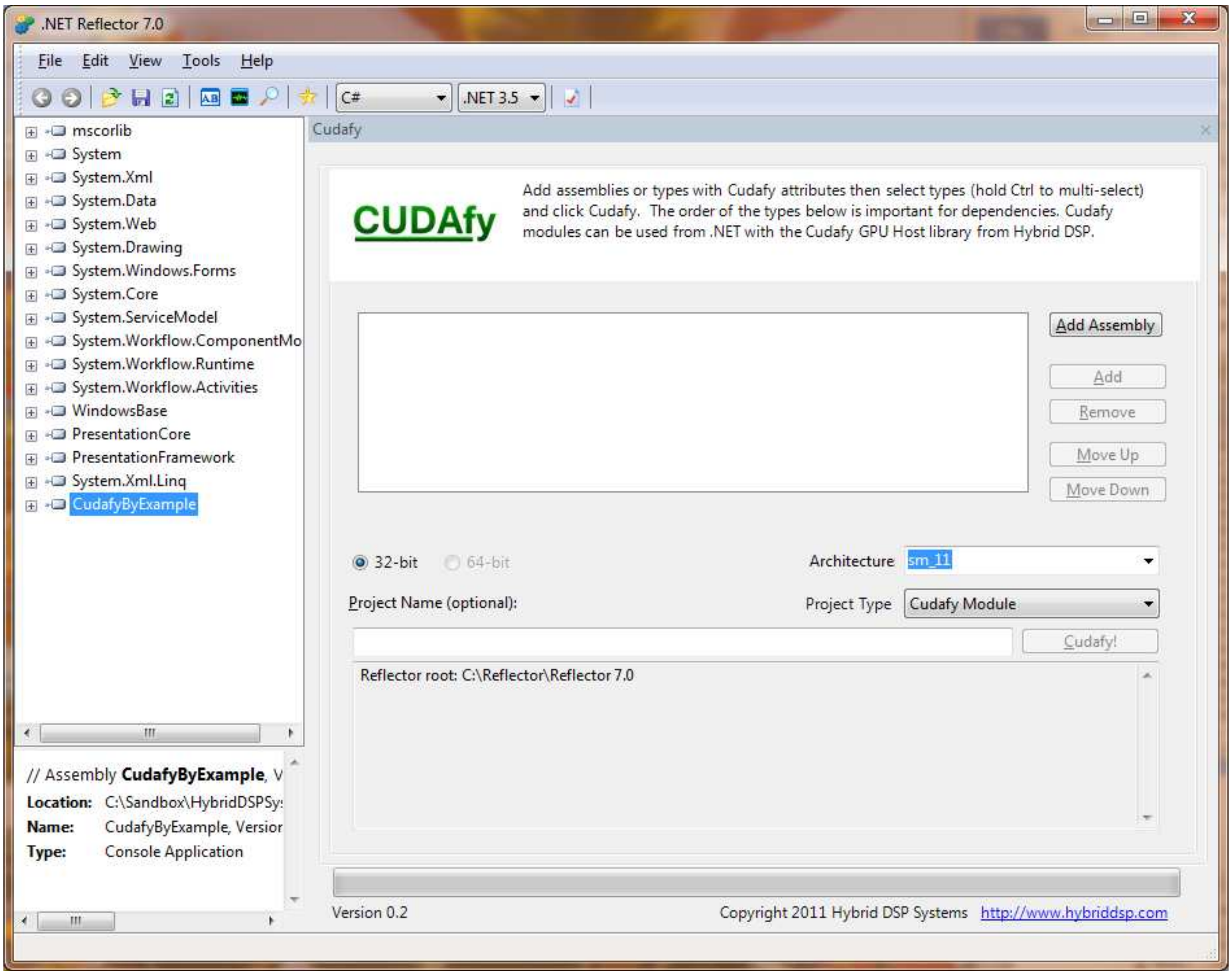
A file name DummyDummyComplexFloatFunction.cu must exist and contain code such as this:

```
extern "C" __global__ void  DummyDummyComplexFloatFunction(DummyComplexFloat  *result)
{
    int  x = blockIdx.x;
    result[x] = result[x].Add(result[x]);
}
```

CUDAFYING WITH .NET REFLECTOR

Either open Red Gate's .NET Reflector directly or by right clicking on the assembly you wish to cudafy and selecting Browse with .NET Reflector… (For the latter option Reflector must be integrated with Windows Explorer. Do this via **Tools : Visual Studio and Windows Explorer Integration…**)

If you have not already done so then see the earlier section on how to install the CUDAfy add-in. Two things have happened: you have a new target language named **CUDA** (see the left hand of the two drop down menus at the top) and via **Tools : Cudafy** you can display the GUI component.

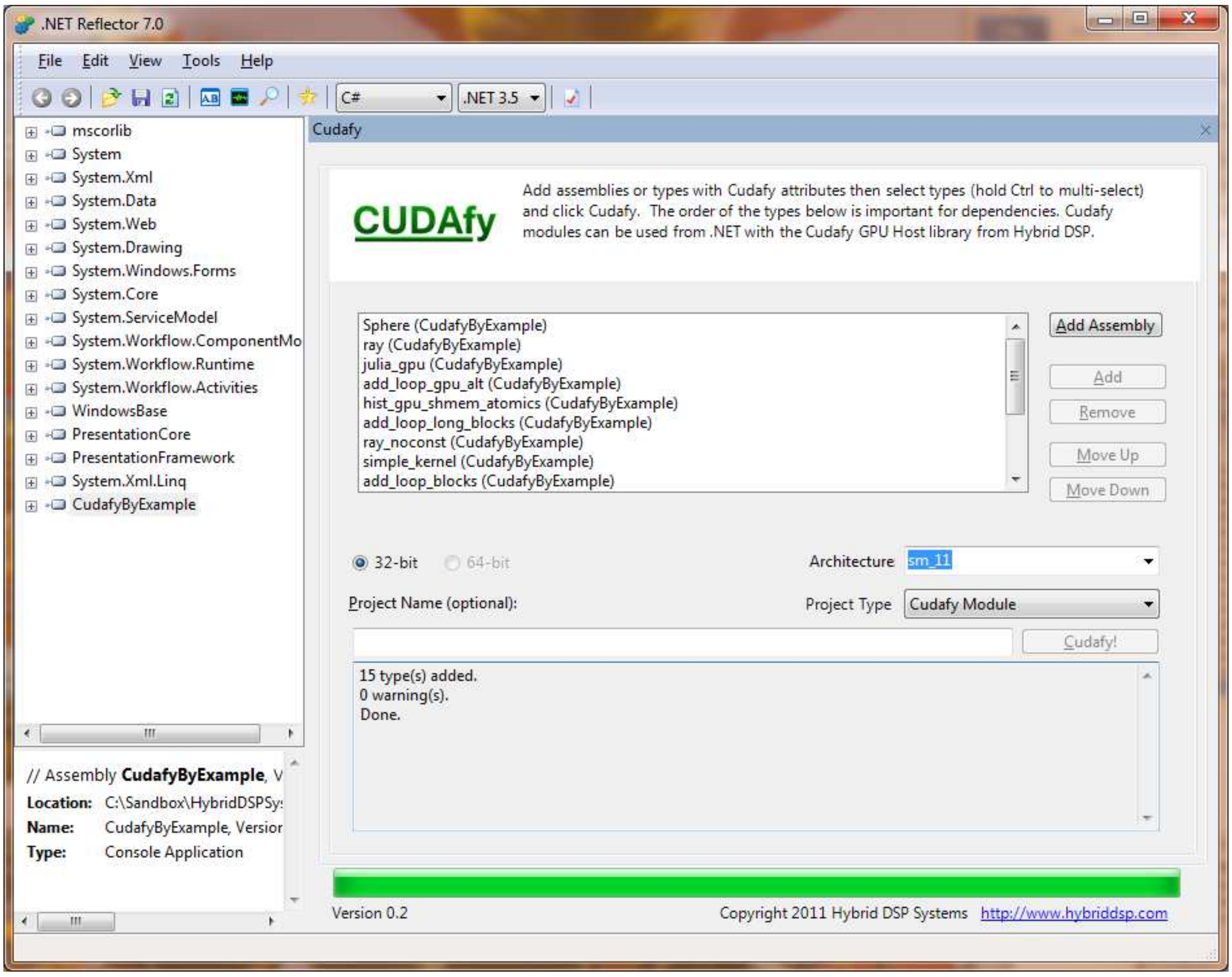


Add assemblies or types with Cudafy attributes then select types (hold Ctrl to multi-select) and click Cudafy. The order of the types below is important for dependencies. Cudafy modules can be used from .NET with the Cudafy GPU Host library from Hybrid DSP.

Reflector root: C:\Reflector\Reflector 7.0

// Assembly **CudafyByExample**, V
Location: C:\Sandbox\HybridDSPSy
Name: CudafyByExample, Versior
Type: Console Application

Highlight your assembly on the left hand side. In this case the assembly is named CudafyByExample. You now have two choices. You can add the complete assembly via the **Add Assembly** button on the left, or you can navigate down to individual types and click **Add**. Let's do the former.



// Assembly **CudafyByExample**, V
Location: C:\Sandbox\HybridDSPSy
Name: CudafyByExample, Version
Type: Console Application

Add assemblies or types with Cudafy attributes then select types (hold Ctrl to multi-select) and click Cudafy. The order of the types below is important for dependencies. Cudafy modules can be used from .NET with the Cudafy GPU Host library from Hybrid DSP.

Sphere (CudafyByExample)
ray (CudafyByExample)
julia_gpu (CudafyByExample)
add_loop_gpu_alt (CudafyByExample)
hist_gpu_shmem_atomics (CudafyByExample)
add_loop_long_blocks (CudafyByExample)
ray_noconst (CudafyByExample)
simple_kernel (CudafyByExample)
add_loop_blocks (CudafyByExample)

[✓ 32-bit] 64-bit
Architecture: sm_11
Project Name (optional):
Project Type: Cudafy Module

```
15 type(s) added.
0 warning(s).
Done.
```

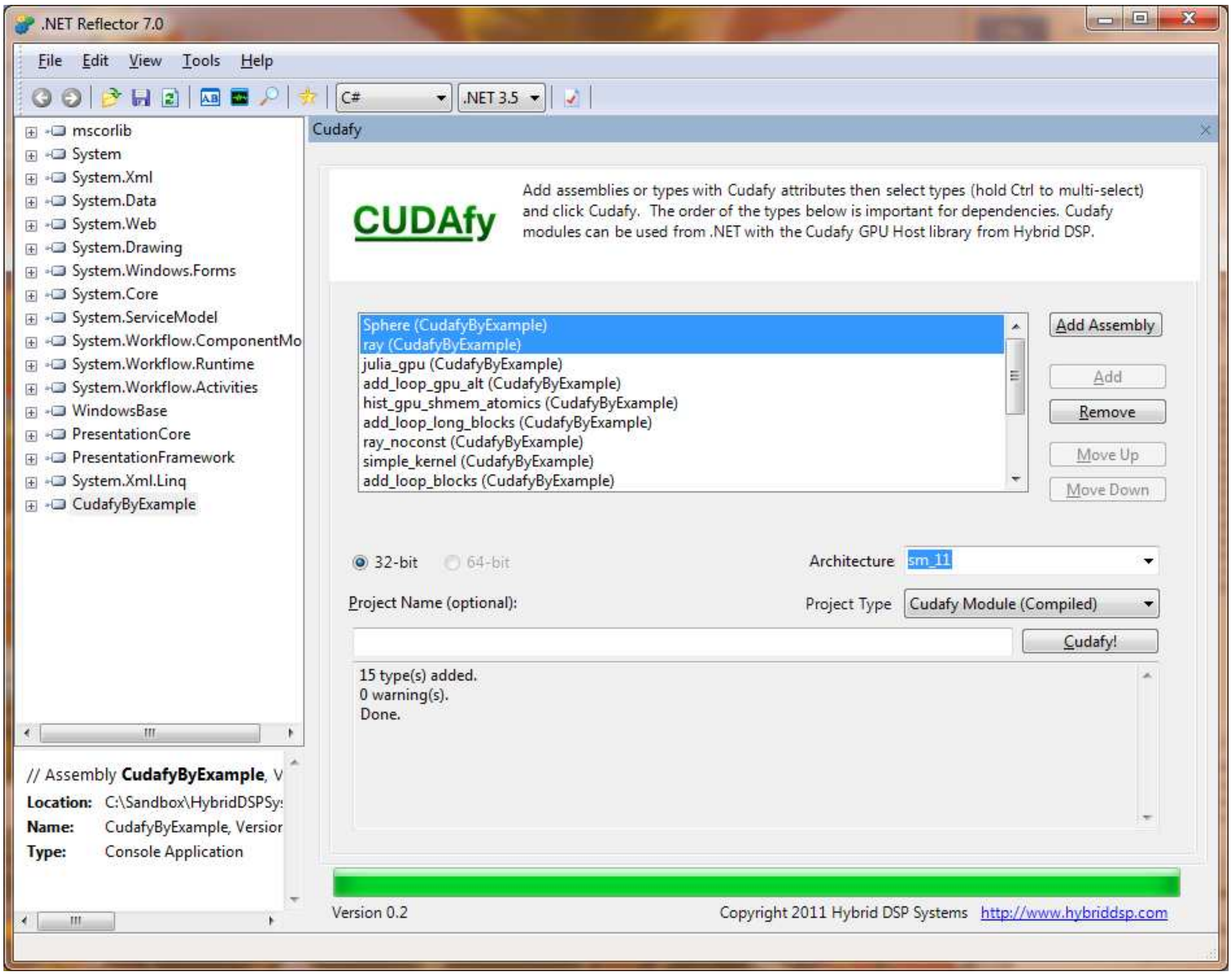
Version 0.2 Copyright 2011 Hybrid DSP Systems http://www.hybriddsp.com

We’re only interested in the cudafying our ray tracing example, so select **ray (CudafyByExample)** from the big list box. Now recall that the ray example needs a **Sphere** struct. We’re going to need to select him too. Hold the Ctrl key and left click on **Sphere (CudafyByExample)**.

Now the order of these two types is very important. The **ray** type is dependent on the **Sphere** type therefore **Sphere** must appear higher in the list than **ray**. Here they are already in the correct sequence. If they were not then select the type you wish to move and then use the **Move Up** and **Move Down** buttons. Selected types can be removed through the use of the **Remove** button or pressing the delete key on your keyboard.

The next stage is to choose a minimum GPU architecture. Fermi is sm_20, earlier NVIDIA GPUs are covered by sm_11, sm_12 and sm_13. Generally it is best for performance to target the highest architecture you expect your app to run on, however then you reduce your potential user base.

There are three project types: Cudafy Module, Cudafy Module (Compiled) and Cuda C. The recommended setting is Cudafy Module (Compiled) as this allows direct use of the module in your host application.



// Assembly **CudafyByExample**, V
Location: C:\Sandbox\HybridDSPSy:
Name: CudafyByExample, Version
Type: Console Application

Add assemblies or types with Cudafy attributes then select types (hold Ctrl to multi-select) and click Cudafy. The order of the types below is important for dependencies. Cudafy modules can be used from .NET with the Cudafy GPU Host library from Hybrid DSP.

Sphere (CudafyByExample)
ray (CudafyByExample)
julia_gpu (CudafyByExample)
add_loop_gpu_alt (CudafyByExample)
hist_gpu_shmem_atomics (CudafyByExample)
add_loop_long_blocks (CudafyByExample)
ray_noconst (CudafyByExample)
simple_kernel (CudafyByExample)
add_loop_blocks (CudafyByExample)

[✓ 32-bit] 64-bit
Architecture: sm_11
Project Type: Cudafy Module (Compiled)

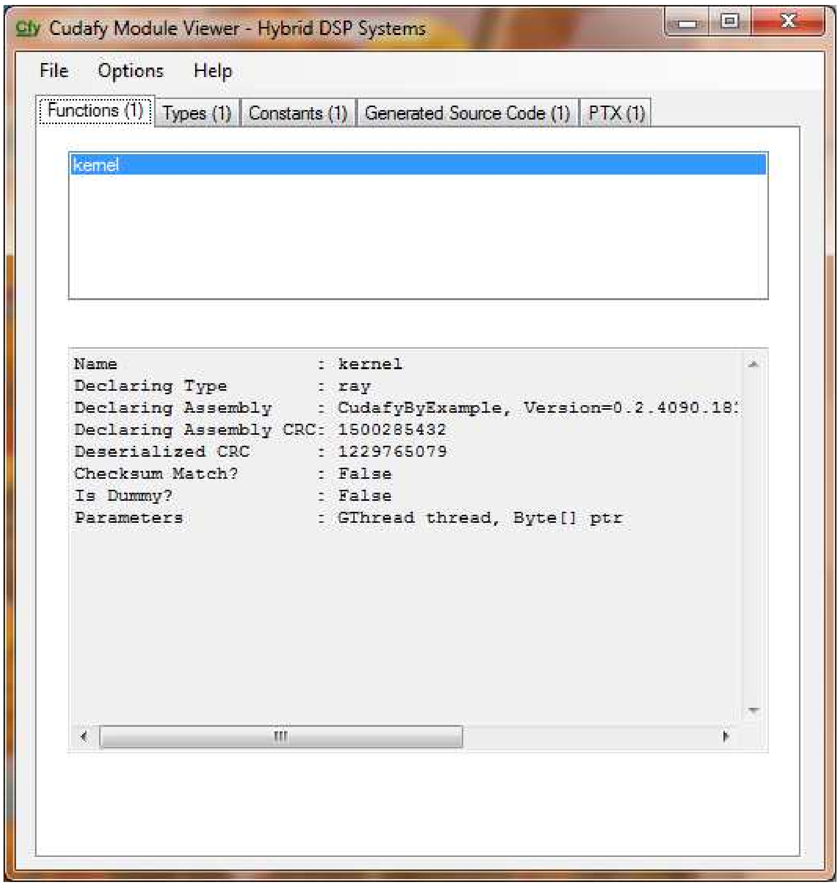
```
15 type(s) added.
0 warning(s).
Done.
```

Cudafy Modules contain as minimum reflection information. With this information the original .NET types can be found and instaniated and the expected parameter types for specific GPU functions are recorded. In addition generated CUDA C source code can be present as can compiled CUDA code (PTX). Cudafy Modules can be serialized and deserialized to / from xml. The choice of Cudafy Module as project type is a module with CUDA C code. The compiled variant also calls the NVIDIA CUDA C compiler (**nvcc.exe**) to create the PTX code that can be loaded into the GPU module. The final option – Cuda C –creates a ***.cu** source code file.

It is important to study the output if any from the **nvcc** compiler. Here you may see errors and warnings that the Cudafy .NET Reflector add-in did not catch, such as unknown types or unsupported .NET features.

CUDAFY MODULE VIEWER

Present in the bin directory of the SDK is a tool for examining ***.cdfy** files. It is a graphical interface called the **Cudafy Module Viewer**.


Cudafy Module Viewer - Hybrid DSP Systems
File
Options
Help
Functions (1)
Types (1)
Constants (1)
Generated Source Code (1)
PTX (1)
kernel

```
Name                   : kernel
Declaring Type         : ray
Declaring Assembly     : CudafyByExample, Version=0.2.4090.18:
Declaring Assembly CRC: 1500285432
Deserialized CRC       : 1229765079
Checksum Match?        : False
Is Dummy?              : False
Parameters             : GThread thread, Byte[] ptr
```

Start the application by double clicking the exe file. For convenience you may also choose to set in Windows Explorer that ***.cdfy** files should always be opened with **Cudafy Module Viewer** as default. Double click a ***.cdfy** file and when Windows asks you which program to use to open the file, choose **Select a program from a list of installed programs**, then choose **Browse…** and navigate to **Cudafy Module Viewer**.

The screen shots in this chapter are based on opening **ray.cdfy** which is located in the **CudafyByExample** project. There are five tabs:

- Functions
- Types
- Constants
- Generated Source Code

- PTX

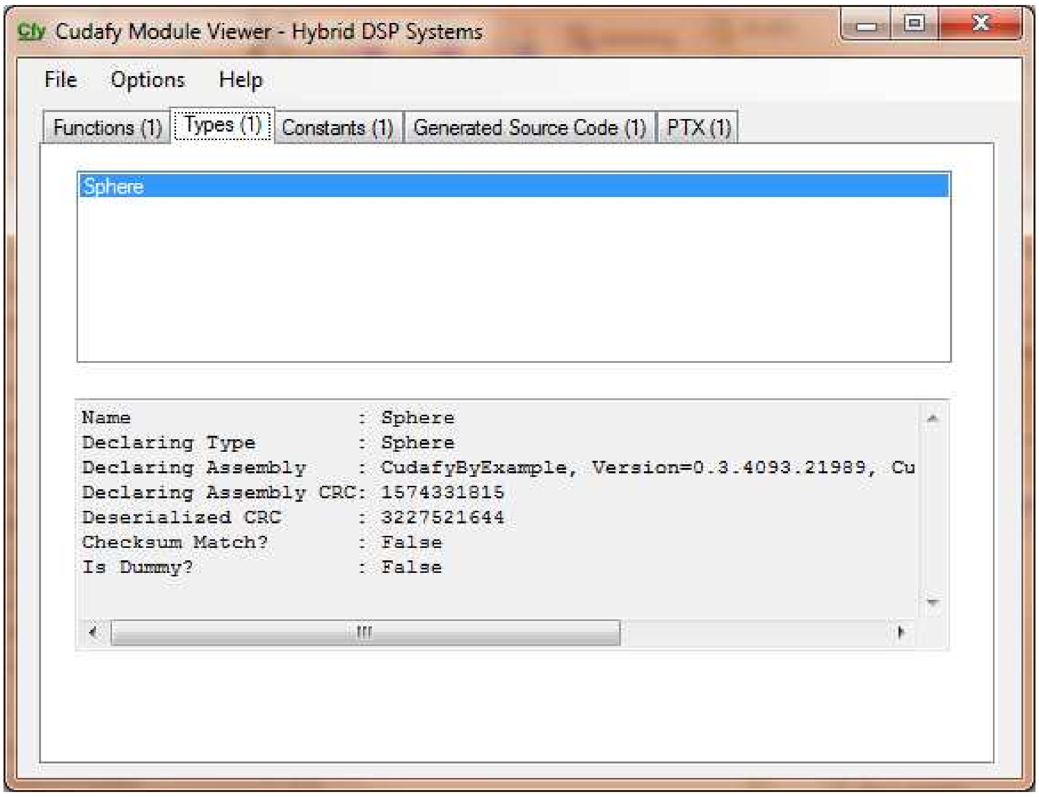
FUNCTIONS

A list of all GPU functions is shown in the top list box of this tab. Below are the details relating to the selected function.

Property	**Description**
Name	The name of the .NET method from which the GPU function was translated.
Declaring Type	The type (class) in which the method is found.
Declaring Assembly	The assembly (DLL) in which the type (class) is found.
Declaring Assembly CRC	The CRC of the current version of the assembly.
Deserialized CRC	The CRC of the assembly that was actually translated.
Checksum Match?	True if Declaring Assembly CRC and Deserialized CRC are the same, else false. This is simply a warning that there may now be differences between the .NET code and the CUDA module code.
Is Dummy?	True if this function is a dummy function, else false. Dummy functions are not actually translated by CUDAfy. Instead they correspond to an existing CUDA C file.
Parameters	A list of the parameters for the .NET method.

TYPES

The Types tab shows a list of all structs in the Cudafy module.


Cudafy Module Viewer - Hybrid DSP Systems
File
Options
Help
Functions (1)
Types (1)
Constants (1)
Generated Source Code (1)
PTX (1)
Sphere

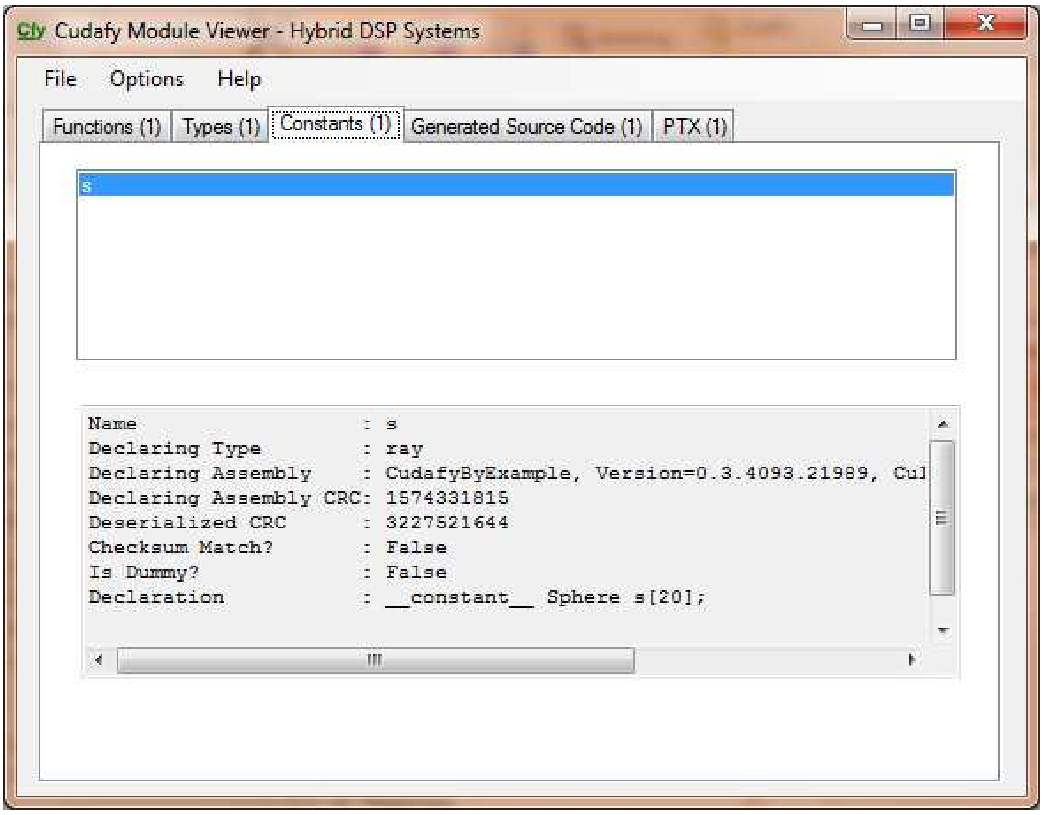
```
Name                   : Sphere
Declaring Type         : Sphere
Declaring Assembly     : CudafyByExample, Version=0.3.4093.21989, Cu
Declaring Assembly CRC: 1574331815
Deserialized CRC       : 3227521644
Checksum Match?        : False
Is Dummy?              : False
```

Property	**Description**
Name	The name of the .NET struct from which the GPU struct was translated.
Declaring Type	The type (class) in which the struct is found (if nested) else as Name.
Declaring Assembly	The assembly (DLL) in which the type (class) is found.
Declaring Assembly CRC	The CRC of the current version of the assembly.
Deserialized CRC	The CRC of the assembly that was actually translated.
Checksum Match?	True if Declaring Assembly CRC and Deserialized CRC are the same, else false. This is simply a warning that there may now be differences between the .NET code and the CUDA module code.

Is Dummy?	True if this struct is a dummy struct, else false. Dummy structs are not actually translated by CUDAfy. Instead they correspond to an existing CUDA C file.

CONSTANTS

This tab shows a list of variables that are allocated in GPU constant memory. Do not mistake this for normal .NET constants.

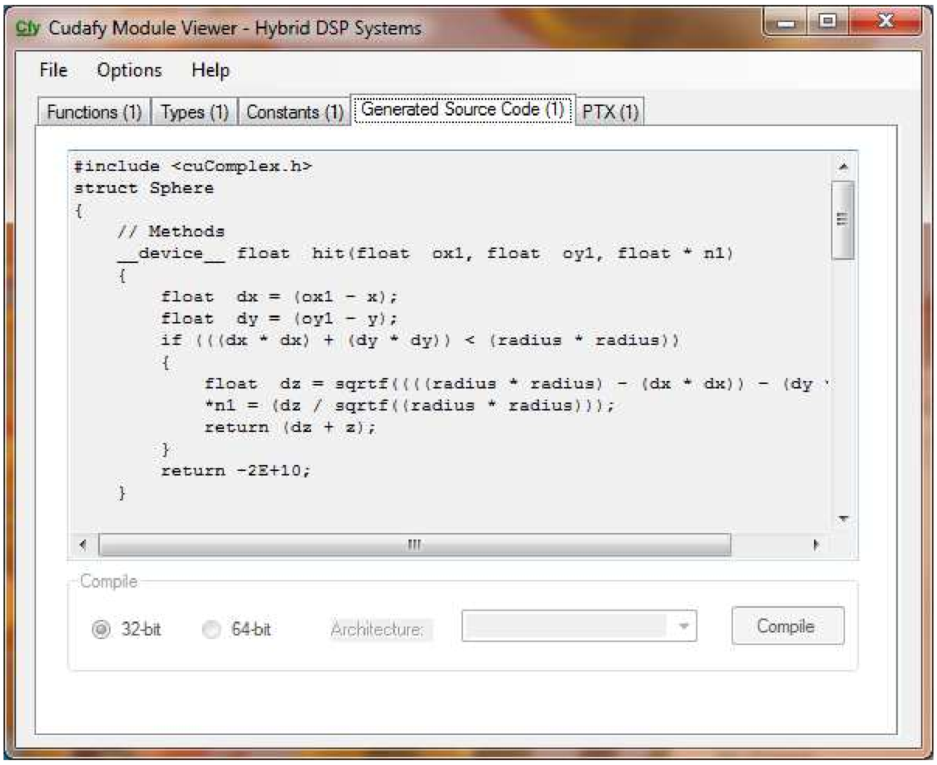

Cudafy Module Viewer - Hybrid DSP Systems
File
Options
Help
Functions (1)
Types (1)
Constants (1)
Generated Source Code (1)
PTX (1)
s

```
Name                    : s
Declaring Type          : ray
Declaring Assembly      : CudafyByExample, Version=0.3.4093.21989, Cul
Declaring Assembly CRC: 1574331815
Deserialized CRC        : 3227521644
Checksum Match?         : False
Is Dummy?               : False
Declaration             : __constant__ Sphere s[20];
```

Property	**Description**
Name	The name of the .NET constant from which the GPU constant was translated.
Declaring Type	The type (class) in which the constant is found.
Declaring Assembly	The assembly (DLL) in which the type (class) is found.

Declaring Assembly CRC	The CRC of the current version of the assembly.
Deserialized CRC	The CRC of the assembly that was actually translated.
Checksum Match?	True if Declaring Assembly CRC and Deserialized CRC are the same, else false. This is simply a warning that there may now be differences between the .NET code and the CUDA module code.
Is Dummy?	True if this function is a dummy function, else false. Dummy functions are not actually translated by CUDAfy. Instead they correspond to an existing CUDA C file.
Declaration	Shows how the constant looks in CUDA C.

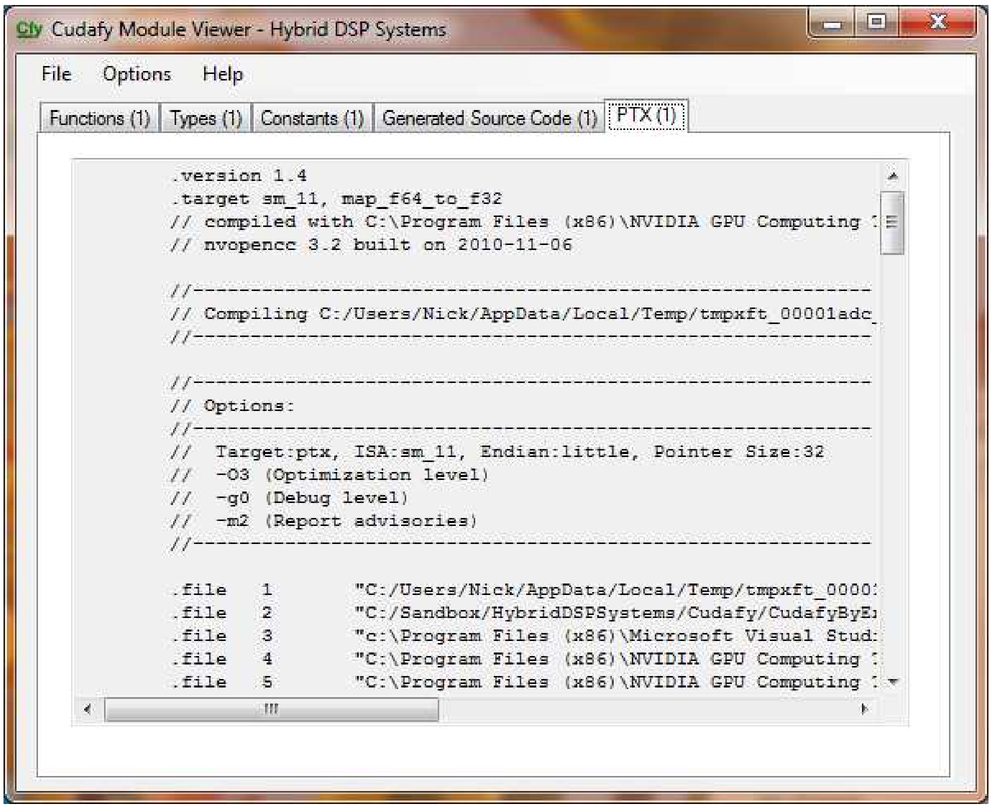
GENERATED SOURCE CODE



```
#include <cuComplex.h>
struct Sphere
{
    // Methods
    __device__ float  hit(float  ox1, float  oy1, float * n1)
    {
        float  dx = (ox1 - x);
        float  dy = (oy1 - y);
        if (((dx * dx) + (dy * dy)) < (radius * radius))
        {
            float  dz = sqrtf((((radius * radius) - (dx * dx)) - (dy *
            *n1 = (dz / sqrtf((radius * radius)));
            return (dz + z);
        }
        return -2E+10;
    }
```

Cuadfy Modules also contain the source code that was generated when the .NET assembly was cudafied. You can optionally edit and recompile this code by going to **Options : Enable Editing** and then selecting **Architecture** and pushing **Compile**.

PTX


Cudafy Module Viewer - Hybrid DSP Systems
File
Options
Help
Functions (1)
Types (1)
Constants (1)
Generated Source Code (1)
PTX (1)

```
	.version 1.4
	.target sm_11, map_f64_to_f32
	// compiled with C:\Program Files (x86)\NVIDIA GPU Computing 
	// nvopencc 3.2 built on 2010-11-06

	//-----------------------------------------------------------
	// Compiling C:/Users/Nick/AppData/Local/Temp/tmpxft_00001adc_
	//-----------------------------------------------------------

	//-----------------------------------------------------------
	// Options:
	//-----------------------------------------------------------
	//  Target:ptx, ISA:sm_11, Endian:little, Pointer Size:32
	//  -O3	(Optimization level)
	//  -g0	(Debug level)
	//  -m2	(Report advisories)
	//-----------------------------------------------------------

	.file	1	"C:/Users/Nick/AppData/Local/Temp/tmpxft_0000
	.file	2	"C:/Sandbox/HybridDSPSystems/Cudafy/CudafyByE
	.file	3	"c:\Program Files (x86)\Microsoft Visual Stud
	.file	4	"C:\Program Files (x86)\NVIDIA GPU Computing 
	.file	5	"C:\Program Files (x86)\NVIDIA GPU Computing 
```

The compiled code is in the CUDA PTX format. This is shown as read only.