Project	CUDAfy
Title	CUDAfy User Manual
Reference	
Client Reference	
Author(s)	Nicholas Kopp, Hybrid DSP Systems
Date	November 08, 2012

Revision History

Date	Changes Made	Issue	Initials
March 17, 2011	Support to Cudafy V0.3. First public release.	0.3	Nko
March 29, 2011	Update for .NET Reflector Wrapper	1.0	Nko
May 22, 2011	Remove CudafyReflectorWrapper, .NET Reflector Add-in; add CudafyTranslator (based on ILSpy)	1.1	Nko
February 14, 2012	Updated for V1.8	1.8	Nko
August 26, 2012	Updated for V1.10. Add information on context switching.	1.10	Nko
November 8, 2012	Add section on strongly typed launches; update for CUDA 5.0.	1.12	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 one library (DLL) called Cudafy.NET.dll, the Cudafy command line tool and a Cudafy Module Viewer GUI. From a high level it offers the following:

- Cudafy .NET Library
    - Cudafy Translator (Convert .NET code to CUDA C)
    - Cudafy Library (CUDA support for .NET)
    - Cudafy Host (Host GPU wrapper)
    - Cudafy Math (FFT + BLAS)
- Cudafy by Example demo projects
- Cudafy Examples demo projects
- Cudafy Module Viewer
- Cudafy Command Line Tool

The Translator converts .NET code into CUDA code. It is based on ILSpy – a very useful decompilation tool from SharpDevelop. Its use as part of your daily .NET development is recommended. http://wiki.sharpdevelop.net/ilspy.ashx

As a developer you will also require the NVIDIA CUDA Toolkit. As of V1.12 Cudafy supports version 5.0. You can obtain this from: http://developer.nvidia.com/cuda-downloads

Drivers supporting CUDA 5.0 are required.

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

There are two main components to the Cudafy SDK:

- Translation from .NET to CUDA C and compiling using NVIDIA compiler (this results in a Cudafy module xml file)
- Loading Cudafy modules and communicating with GPU from host

It is not necessary for the target machine to perform the first step above.

1. Add reference to **Cudafy.NET.dll** from your .NET project
2. Add the **Cudafy**, **Cudafy.Host** and **Cudafy.Translator** 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. In your host code before using the GPU functions call **Cudafy.Translator.Cudafy**( ). This returns a Cudafy Module instance.

6. Load the module into a **GPGPU** instance. The **GPGPU** type allows you to interact seamlessly with the GPU from your .NET code.

PREREQUISITES

SUPPORTED OPERATING SYSTEMS + SOFTWARE CONFIGURATIONS

Windows XP SP3

Windows Vista

Windows 7

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

Cudafy is built against Microsoft .NET Framework 4.0.

CUDA SDK for V5.0.

NVIDIA Drivers supporting CUDA 5.0 or later. (NVCUDA.dll).

Cudafy FFT library requires CUFFT 5.0

Cudafy BLAS library requires CUBLAS 5.0

Cudafy RAND library requires CURAND 5.0

Cudafy SPARSE Matrix library requires CUSPARSE 5.0

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 some versions of CUDA.

DEVELOPMENT REQUIREMENTS

The following is required when developing with Cudafy or if the application will perform translation of .NET code to CUDA C and compilation:

- NVIDIA CUDA Toolkit 5.0.
- Microsoft VC++ Compiler (used by NVIDIA CUDA Compiler)

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.

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 1.2 the focus is on supporting Fermi and therefore we recommend using it where possible. A good value card would be a GTX 460 or GTX 560.

SOFTWARE

Ideally you will have:

Windows 7 64-bit

Visual Studio 2010 Professional

CUDA 5.0 SDK

INSTALLATION

CUDAFY SDK

The Cudafy SDK is available as a zip file from www.hybriddsp.com. Unzip to a convenient location. Note that some releases of the SDK will not contain the example projects. These can be obtained from http://cudafy.codeplex.com.

Name	Date modified	Type	Size
bin	14/02/2012 20:44	File folder	
CudafyByExample	14/02/2012 20:49	File folder	
CudafyExamples	14/02/2012 20:52	File folder	
CUDA.NET.Readme.txt	25/05/2011 20:31	Text Document	6 KB
CUDAfy API Documentation	14/02/2012 20:56	Internet Shortcut	1 KB
CUDAfy-License.txt	14/09/2011 22:31	Text Document	25 KB
ILSpy-license.txt	26/04/2011 23:53	Text Document	2 KB
README.txt	14/02/2012 11:23	Text Document	11 KB

The contents of the bin directory is summarized below:

File	**Description**
cudafycl.exe	Standalone translator.
Cudafy.NET.dll	Key library.
Cudafy.NET.xml	Code insight information.
CudafyModuleViewer.exe	A viewing tool for Cudafy modules.

The root directory also contains:

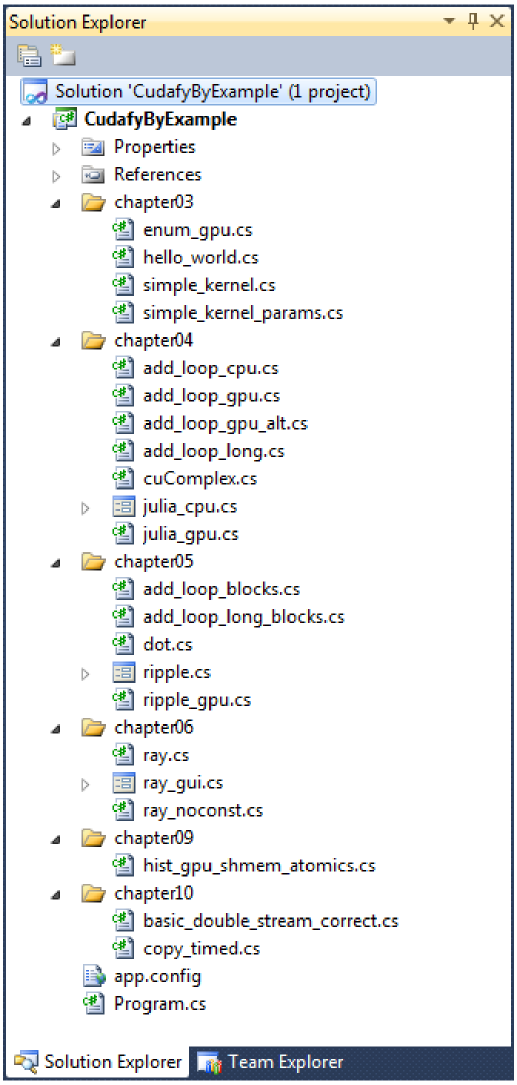
Directory	**Description**
CudafyByExample	This is a Visual Studio 2010 solution containing a project that demonstrates many of the features of Cudafy. The examples are based on the book *CUDA BY EXAMPLE* (Sanders and Kandrot). A copy of this book is highly recommended. Note: Not all releases of SDK contain this.

CudafyExamples	Another Visual Studio 2010 solution demonstrating Cudafy features not covered in CudafyByExample. These include dummy functions, complex numbers and multi dimensional arrays. Note: Not all releases of SDK contain this.
Cudafy-License.txt	License agreement.
README.txt	Release notes.
Short cut to the on-line documentation	Html documentation and this user guide.
CUDA.NET.Readme.txt	Release notes for CUDA.NET
ILSpy-license.txt	License for ILSpy

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 reference to Cudafy.NET.dll may be broken if the relative path is different. This dll is in the bin directory so re-add it 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).

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.

Press F5 or the little green arrow to run the application.

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;
using Cudafy.Translator;
namespace CudafyByExample
{

    public class simple_kernel
    {
        public static void Execute()
        {
            CudafyModule km = CudafyTranslator.Cudafy();

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

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

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

You will see that we include three namespaces:

- Cudafy
- Cudafy.Host
- Cudafy.Translator

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 translator 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 named **CudafyByExample.exe**. When we run this the call to **CudafyTranslator.Cudafy** creates a Cudafy module. 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 translation is a Cudafy module instance. When calling the empty overload of Cudafy an xml file named **simple_kernel.cdfy** is also created (cached) and will be used next time the exact same application is run (ie checksum stored in xml matches that of the declaring assembly).

Okay, on with show. The **CudafyHost** class is static and contains a method called **GetDevice**. 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.

There are three ways of launching: Standard, Strongly Typed and Dynamic. The normal way is commented out and is described next: 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.

```
gpu.Launch().kernel();           // Dynamic Launch OR
gpu.Launch(1,1, “kernel”);       // Standard Launch OR
gpu.Launch(1,1, Action(kernel)); // Strongly Typed Launch
```

The dynamic way uses the Microsoft .NET 4.0 Dynamic Language Runtime to do the same way but in a cleaner style. Since we want only one thread there are zero arguments to Launch method. There are no arguments to kernel so that is also empty. The strongly typed launch has the performance benefits of the standard launch plus the safety of strong typed parameters.

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().add(2, 7, dev_c);
// or standard launch gpu.Launch(1, 1, "add", 2, 7, dev_c);
```

```
// or strongly typed gpu.Launch(1, 1, (Action<int,int,int[]>)(add),2,7,dev_c);
// Note that if the kernel method uses the GThread parameter then this must be
// added e.g. (Action<GThread,int,int,int[]>). However the argument does not
// need to be passed.
```

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.

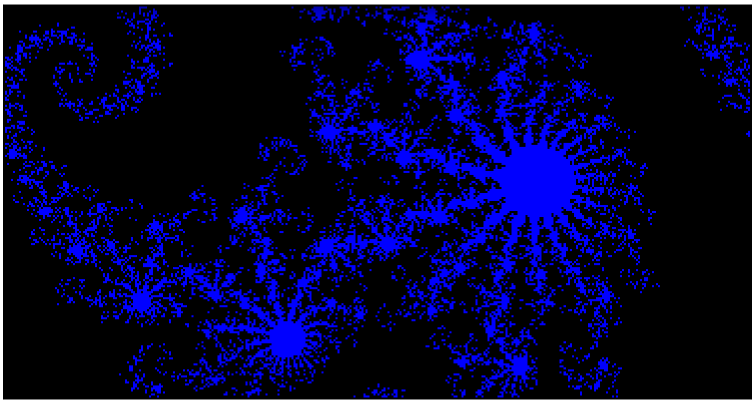
In our GPU function we need an additional GThread property. We are now interested in **blockIdx** and its **x** property and **gridDim** and its **x** property.

```
[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.

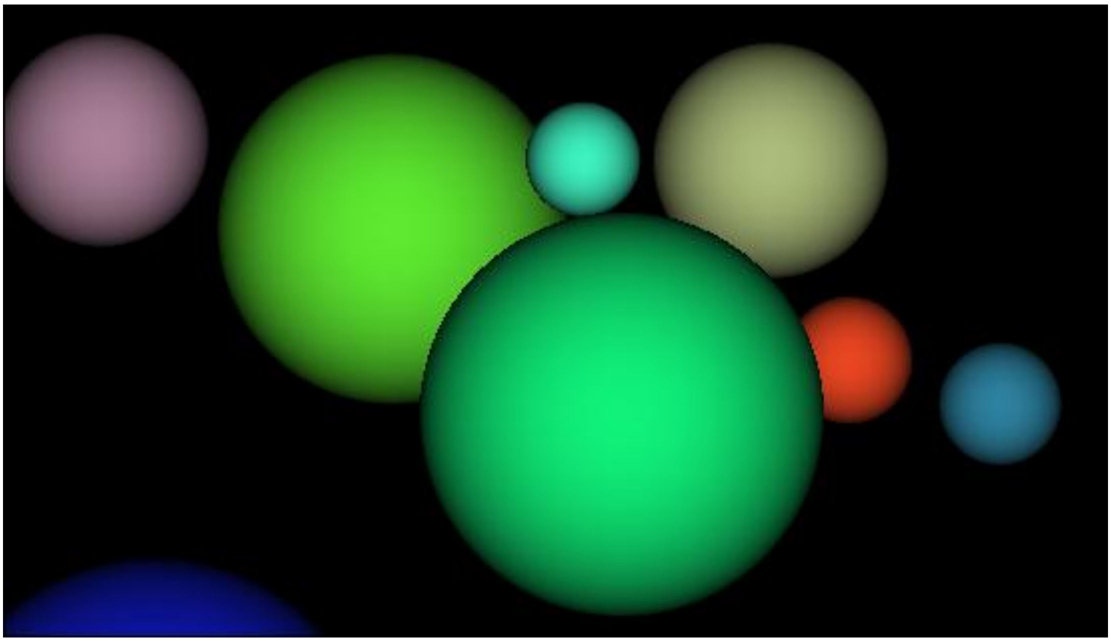
```
            // synchronize threads in this block
            thread.SyncThreads();
```

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 translator 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**. They get the postfix **Async** and take an additional parameter that is the stream id. If you are using the dynamic launcher then a launch will be implicitly asynchronous if a stream id is specified. 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.Launch(N / 256, 256, 1).kernel(dev_a0, dev_b0, dev_c0);
                gpu.Launch(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.

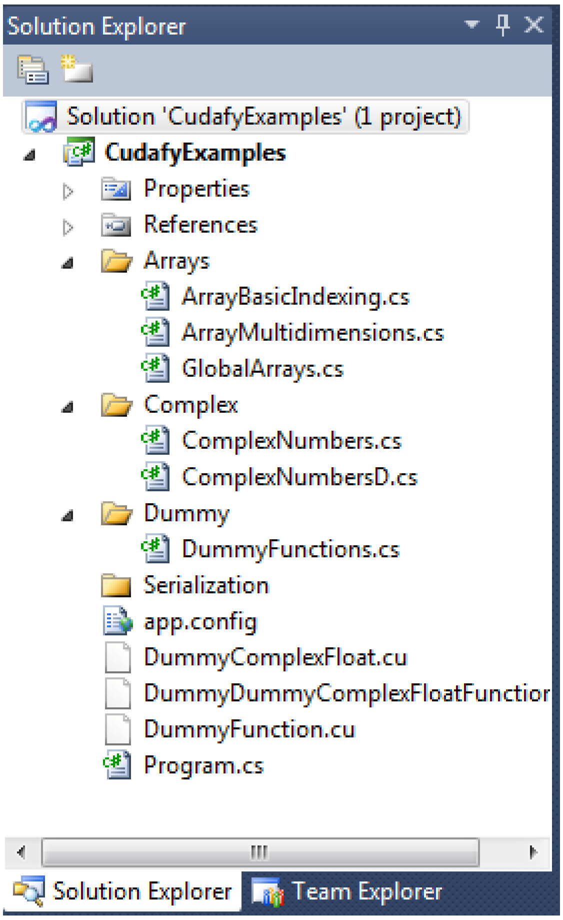
```
gpu.HostFree(host_aPtr);
gpu.HostFree(host_bPtr);
gpu.HostFree(host_cPtr);
gpu.DestroyStream(1);
gpu.DestroyStream(2);
```

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 translator. 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
       __device__ 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]);
}
```

USING THE CUDAFY TRANSLATOR

If all GPU code is also in the same class as the host code doing this is trivial:

```
    public class ArrayBasicIndexing
```

```
{
    public const int N = 1 * 1024;

    public static void Execute()
    {
        CudafyModule km = CudafyTranslator.Cudafy();

        GPGPU gpu = CudafyHost.GetDevice(eGPUType.Cuda);
        gpu.LoadModule(km);
          ...
    }

    [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;
        }
    }
```

Here we make use of a method that is aware of the class that is calling it. Therefore calling the Cudafy method will first check if there is a cached Cudfy module with a matching checksum (see next section), else it will translate the class **ArrayBasicIndexing** finding the method **add**. Default settings are applied for target platform and GPU architecture (current platform – x86 or x64 – and GPU architecture 1.3).

For more complex configurations the following will suffice:

```
/// <summary>
/// This type is used by GlobalArrays and must be selected for Cudafying.
/// </summary>
[Cudafy]
public struct ComplexFloat
{
    public ComplexFloat(float r, float i)
    {
        Real = r;
        Imag = i;
    }
    public float Real;
    public float Imag;
    public ComplexFloat Add(ComplexFloat c)
    {
        return new ComplexFloat(Real + c.Real, Imag + c.Imag);
    }
}

/// <summary>
/// Is dependent on ComplexFloat type.
/// </summary>
public class GlobalArrays
{
    public const int XSIZE = 4;
    public const int YSIZE = 8;
    public const int ZSIZE = 16;
```

```
        public static void Execute()
        {
            CudafyModule km = CudafyTranslator.Cudafy(typeof(ComplexFloat),
typeof(GlobalArrays));

            GPGPU gpu = CudafyHost.GetDevice(eGPUType.Cuda);
            gpu.LoadModule(km);
```

Here we have explicitly provided the types we want to cudafy. Further overloads of the **Cudafy** method allow us to specify the architecture and platform. Note that it is also possible to compile the module via the **Compile** method of the **CudafyModule**.

IMPROVING PERFORMANCE BY CACHING CUDAFY MODULES

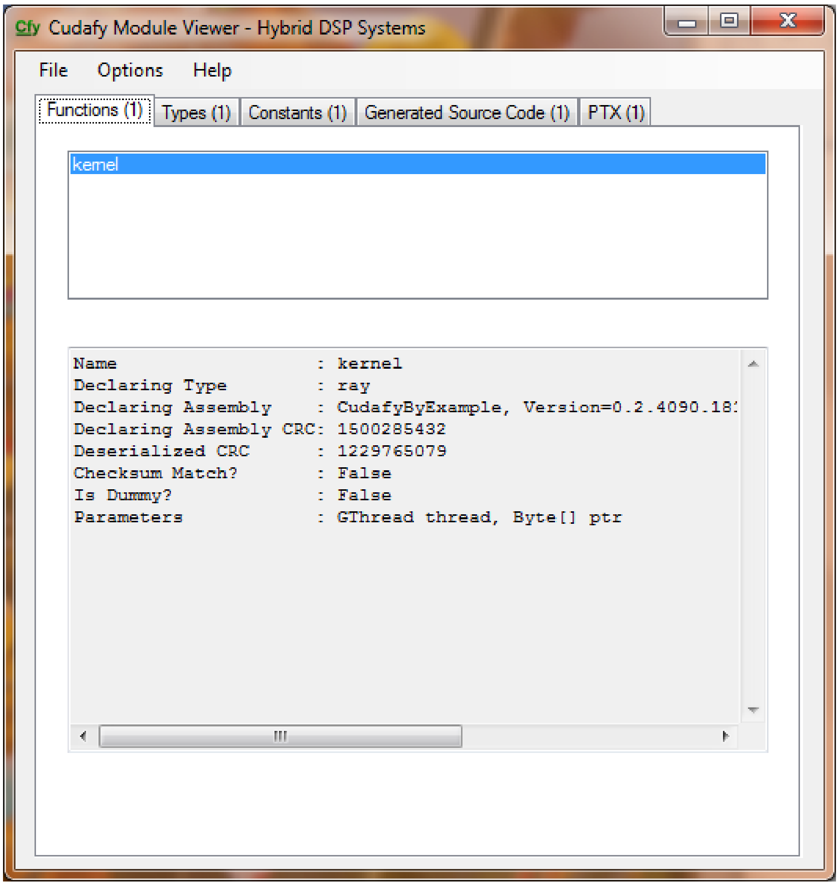
In the interests of performance it may not be desirable to always call **Cudafy** if the GPU code has not changed. We can cache the Cudafy modules by using serialization. The following code illustrates this:

```
    public class ArrayBasicIndexing
    {
            CudafyModule km = CudafyModule.TryDeserialize();
            if (km == null || !km.TryVerifyChecksums())
            {
                km = CudafyTranslator.Cudafy();
                km.Serialize();
            }
```

The **TryDeserialize** method will attempt to find a *.cdfy file in the current directory with the same file name as the calling type (**ArrayBasicIndexing**). If this is not found or fails then **null** is returned and we should try making a new module as shown in previous section. If it is not **null** then we want to check whether the cached module refers to the same version of the .NET code it was created from. To do this call **TryVerifyChecksums**. If this returns **false** then it means the cached module was out of date and it is advisable to cudafy a new one. We call the Serialize method on the CudafyModule to store this to a file with the same name as the calling class (**ArrayBasicIndexing**). Overloaded methods of **TryDeserialize** and **Serialize** allow the specifying of explict file names.

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**.



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

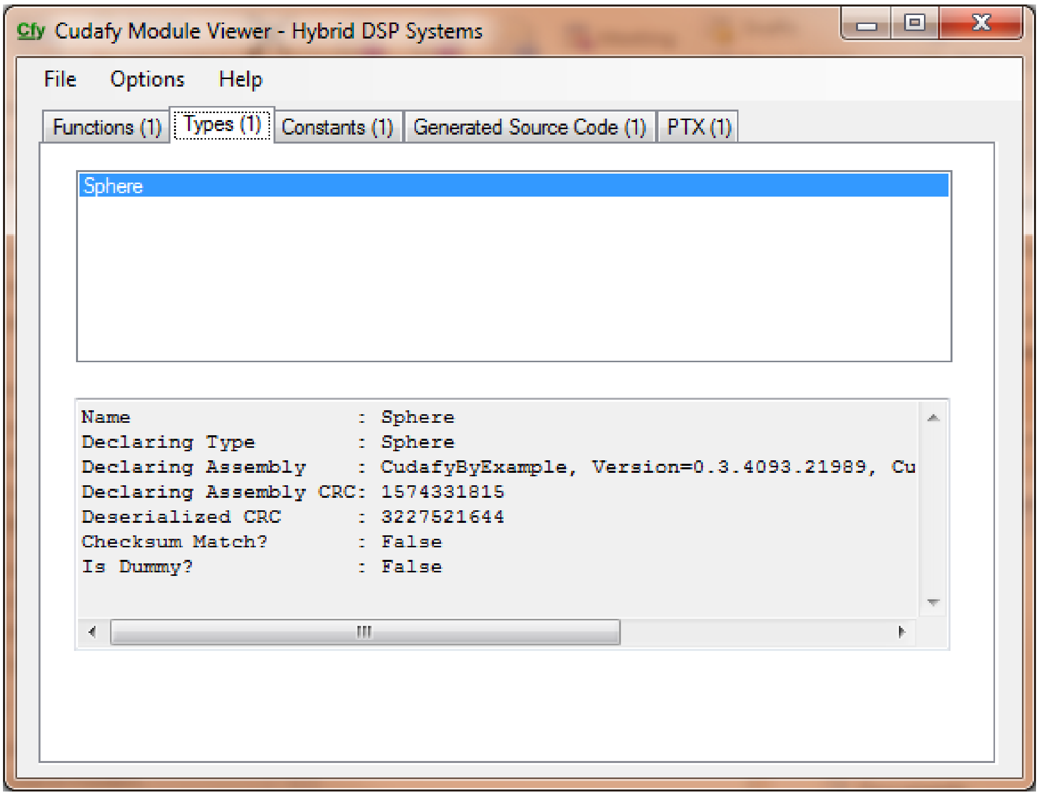
FUNCTIONS

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.

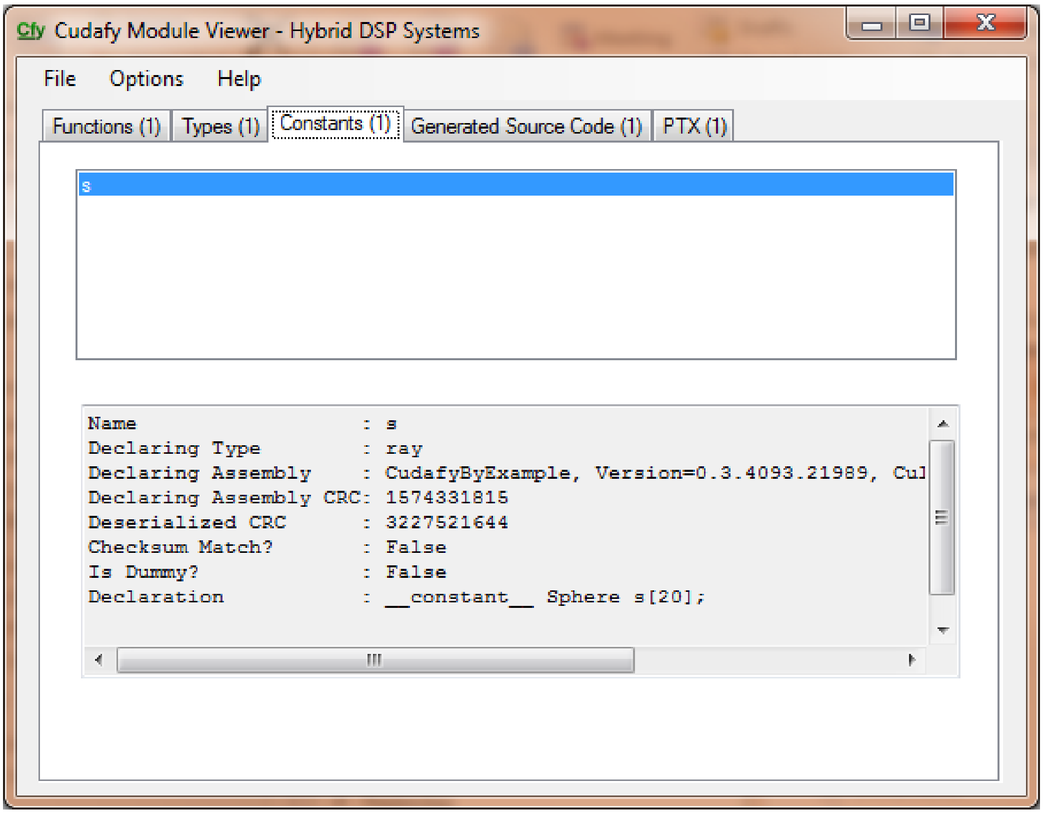


Property	**Description**
Name	The name of the .NET struct from which the GPU struct was translated.
Declaring Type	The type (class) in which the struct is found (if nested) else as Name.
Declaring Assembly	The assembly (DLL) in which the type (class) is found.
Declaring Assembly CRC	The CRC of the current 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

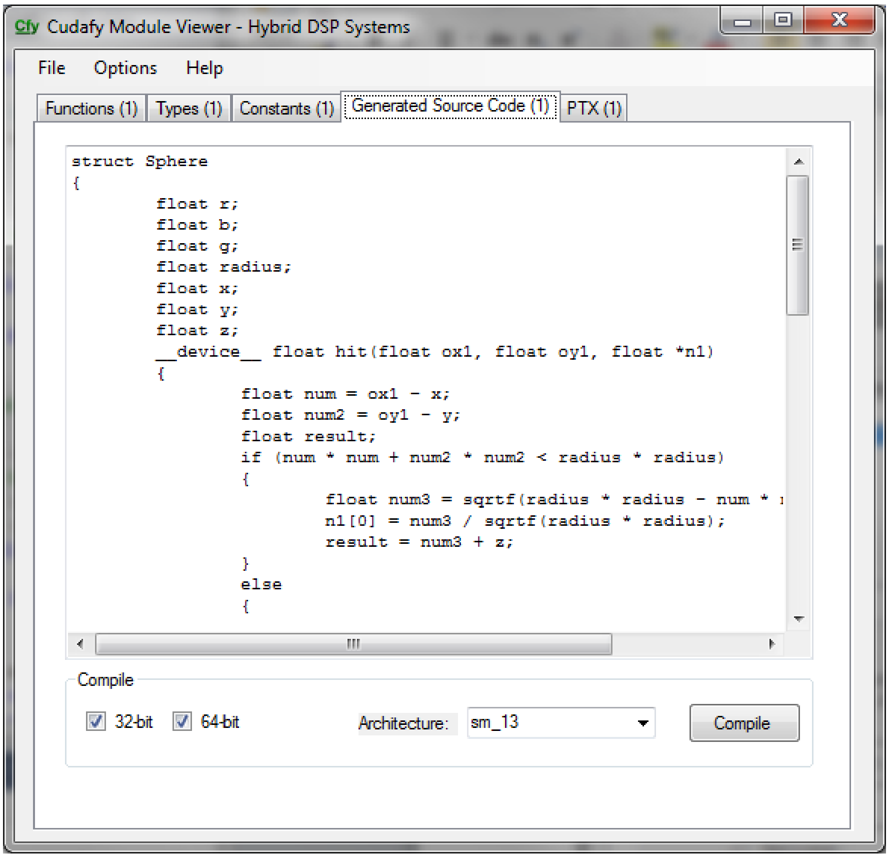
This tab shows a list of variables that are allocated in GPU constant memory. Do not mistake this for normal .NET constants.



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.

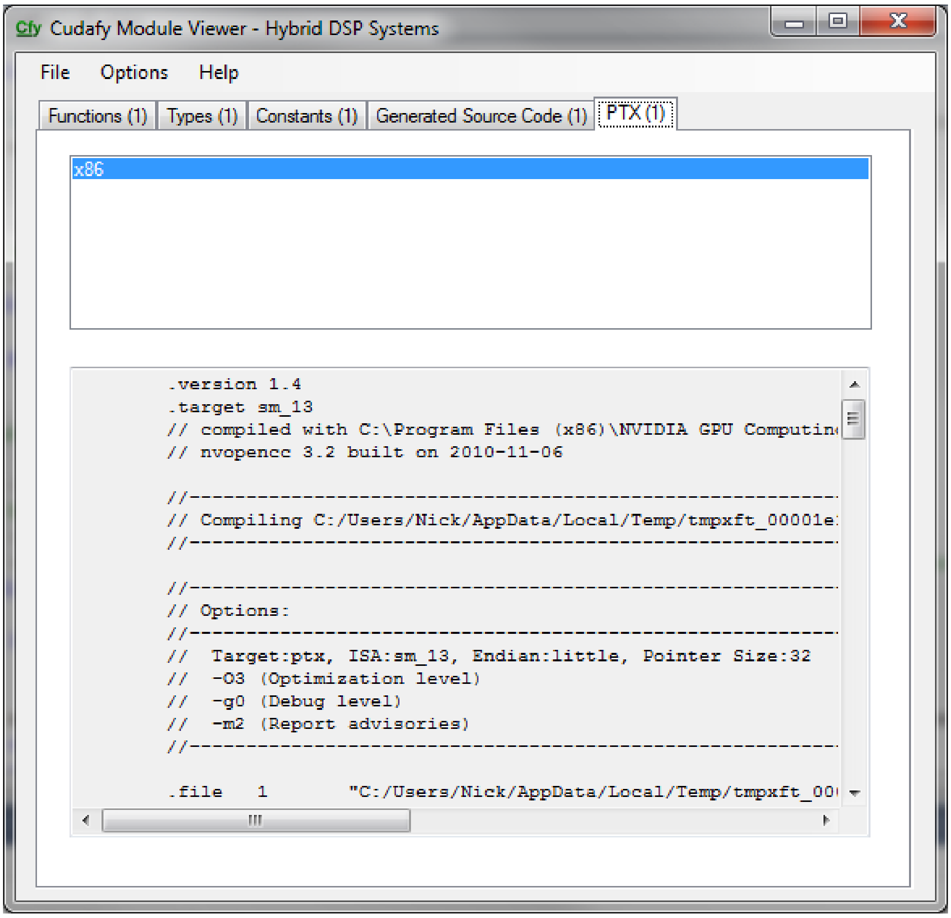
Deserialized CRC	The CRC of the assembly that was actually translated.
Checksum Match?	True if Declaring Assembly CRC and Deserialized CRC are the same, else false. This is simply a warning that there may now be differences between the .NET code and the CUDA module code.
Is Dummy?	True if this function 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



Cudafy 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



The compiled code is in the CUDA PTX format for one or more platform types. This is shown as read only.

CUDAFY COMMAND LINE TOOL

As an alternative to cudafying within the application code, you can elect to use the standalone cudafy command line tool. There are two modes of operation:

- Generate Cudafy Module
- Embed Cudafy Module

In generate mode all types and members within the specified .NET assembly marked with the Cudafy attribute are cudafied and a *.cdfy module is generated.

Embedding the Cudafy module involves cudafying all types and members within the specified .NET assembly marked with the Cudafy attribute and then embedding the resultant Cudafy module in the .NET assembly. The advantage of this is that it is no longer necessary to distribute a separate *.cdfy file.

The usage is as follows:

cudafycl.exe assemblyname.dll [-arch=sm_11|sm_12|sm_13|sm_20] [-cdfy]

The assembly to be cudafied is specified as the first argument.

The optional -arch specifies the minimum device architecture.

The optional –cdfy put the tool in generate mode and the module is not embedded in the assembly. The name of the module is as per the assembly name with a *.cdfy extension.

ADVANCED FEATURES

CONTEXTS – MULTI GPU SYSTEMS

Explicit control over context switching has been added as of version 1.10. The following example demonstrates how to copy data between two GPUs. When two GPUs are used from a single thread some steps must be taken to ensure the correct context is valid at the correct time. Upon creating a GPGPU object through the use of **GetDevice** or **CreateDevice**, a new context is created. From that point on in your code this remains the current context. If you have two GPUs then the context from the second created becomes current. If you attempt an operation in the same thread that addresses the first GPU you will receive an invalid context exception. Instead you must call **SetCurrentContext** on the relevant GPU before using it. You can check if the context is current for a given GPU by checking its **IsCurrentContext** property.

```
[Test]
public void Test_SingleThreadGPUtoGPU()
{
    Random r = new Random();
    for (int i = 0; i < _uintBufferIn0.Length; i++)
        _uintBufferIn0[i] = (uint)r.Next(Int32.MaxValue);

    _gpu0.SetCurrentContext();
    _gpuuintBufferIn0 = _gpu0.CopyToDevice(_uintBufferIn0);
    _gpu1.SetCurrentContext();
    _gpuuintBufferIn1 = _gpu1.CopyToDevice(_uintBufferIn1);
```

```
            _gpu0.SetCurrentContext();
            long loops = 500;
            Stopwatch sw = Stopwatch.StartNew();

            for (int i = 0; i < loops; i++)
                _gpu0.CopyDeviceToDevice(_gpuuintBufferIn0, 0, _gpu1, _gpuuintBufferIn1, 0,
 _uintBufferIn0.Length);
            sw.Stop();

            float mbps = (float)((long)_uintBufferIn0.Length * sizeof(int) * loops) / (floa
t)(sw.ElapsedMilliseconds * 1000);
            Console.WriteLine(mbps);
            _gpu1.SetCurrentContext();
            _gpu1.CopyFromDevice(_gpuuintBufferIn1, _uintBufferOut1);

            Assert.IsTrue(Compare(_uintBufferIn0, _uintBufferOut1));
            ClearOutputsAndGPU(0);
            ClearOutputsAndGPU(1);
        }
```

STRONGLY TYPED LAUNCHES

The safest way to launch kernels is by doing strongly typed launches. An example demonstrates:

```
        public void Test_add_strongly_typed()
        {
            int a = 1;
            int b = 2;
            int c;
            int[] dev_c = _gpu.Allocate<int>();
            _gpu.Launch(1, 1, (Action<GThread,int, int, int[]>)(add), a, b, dev_c);
            _gpu.CopyFromDevice(dev_c, out c);
            Assert.AreEqual(a + b, c);
            _gpu.Free(dev_c);
        }

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