TidePowerd GPU.NET

Introducing "Native" GPU support for .NET

Introducing GPU.NET

- Issues with current GPU development tools
- Goals of GPU.NET
  - How GPU.NET solves problems of other tools
- How does GPU.NET work?
- A brief look at the implications of GPU.NET
- Writing code with GPU.NET

Issues with current tools

- In general, code is vendor- and OS-specific
- Even OpenCL is vendor-specific in practice
- Programming model has a steep learning curve
- Not targeted towards ‘enterprise’ development
  - Developers stuck writing lots of ‘integration’ code

Goals of GPU.NET

- "GPU computing made easy"
  - Should be intuitive for developers without GPU experience
- Reduce or eliminate boilerplate code
- Add hardware acceleration to "write once, run everywhere"
- Performance at least on par with current tools

Overview of GPU.NET

- Visual Studio integration
- Device methods library
- Assembly rewriting
- Runtime
  - Vendor-specific plugins

Source Code	C#	F#	VB.NET	
Compile to .NET Assembly	.NET Assembly / CIL Bytecode			
Process assembly with GPU.NET	GPU.NET “AssemblyProcessor” + Runtime			
Vendor-Specific Runtime Plugins	.NET / CPU (Fallback)	nVidia GPU / PTX Modules	AMD GPU / Stream IL	Other Acceleration Hardware

Overview of GPU.NET

- Write kernels like any other method in .NET-based languages like C#, F#, or VB.NET
  - Kernel methods stay ‘in-sync’ with rest of code
  - IntelliSense support
- Easy to GPU-accelerate existing .NET codebases
- Ability to use the built-in unit testing features of VS to ensure correctness of kernel code

Visual Studio Integration

- GPU.NET tab in project properties page
  - Provides options to fine-tune how code will run
  - Intuitive location for developers familiar with VS
- “Squiggles” denote invalid kernel code
  - Allows developers to fix errors before compile-time
- Integrates with error list

Device Methods

- Provides .NET-based implementations of common device intrinsics
  - E.g., PopulationCount()
- Used for .NET-based ‘fallback’ methods
- Allows new hardware intrinsics to be emulated on older hardware (in most cases)

Assembly Rewriting

- Performs final validation of the assembly
- Injects reference to the GPU.NET Runtime
- Rewriting process ensures that metadata stays valid – important for GPU-accelerated libraries

Runtime

- "Deep" integration with .NET
  - Allows selected types to be used directly in kernels
  - E.g., System.Drawing.Bitmap
- Dynamically detects hardware via plugins
  - Kernels transparently scheduled on multiple GPUs
  - Manages device resources
- Fallback to .NET-based methods if no GPU

Vendor Plugins

- Interfaces with device drivers
- Performs final stage of JIT compilation
- Applications can be sped up after deployment by updating plugins
  - Without needing to be recompiled

Implications of GPU.NET

- JIT compilation means DLR support in future
  - Write kernels in IronPython, Lisp, SmallTalk, etc.
- Generate/execute kernels via .NET Reflection
- GPU-accelerated LINQ-to-objects queries
- GPU.NET's architecture designed for maximum speed on *all* platforms

tidepowerd

Implications of GPU.NET

- A single binary covers nearly all end-users
  - 32/64-bit versions of Windows, Mac OS X, & Linux
- Controlling the entire 'stack' allows us to make unique performance optimizations
- Simplifies development of GPU-accelerated plugins for existing applications
  - E.g., MS SQL Server, MS Dynamics, Adobe Photoshop

Conclusion

- Developers can now write kernel code in a language they are familiar with
- VS integration reduces GPU learning curve
- Eliminates time spent writing boilerplate code
- Easy to accelerate existing .NET codebases

Availability

- Public beta will be available on tidepowerd.com by end of October 2010
- RTM of v1.0 expected around the end of 2010