Extended Abstract: F# OpenCL Type Provider*

Kirill Smirenko
St.Petersburg State University
St.Petersburg, Russia
k.smirenko@gmail.com

Semyon Grigorev
Associate Professor
St.Petersburg State University
St.Petersburg, Russia
semen.grigorev@jetbrains.com

Abstract

The popularity of GPGPU usage in applied software is growing but GPGPU utilization in high-level programming languages is still challenging. We present the OpenCL type provider: a way to use existing OpenCL C source code in F# in a strongly typed way.

CCS Concepts • **Software and its engineering → Functional languages**; Parallel programming languages;

Keywords Type Providers, Metaprogramming, Generic Programming, GPGPU, OpenCL

ACM Reference Format:
Kirill Smirenko and Semyon Grigorev. 2018. Extended Abstract: F# OpenCL Type Provider. In *Proceedings of The workshop on Type-Driven Development (TyDe'18).* ACM, New York, NY, USA, 3 pages. https://doi.org/

1 Introduction

General Purpose Graphical Processor Units, or GPGPUs, are commonly used for fast computations. Their multi-core architecture benefits high-load computations in applied science, computer vision, bioinformatics and other areas [7, 8].

Several frameworks for GPGPU programming are known. The most popular one is CUDA, a platform for parallel GPGPU computations developed by Nvidia in 2007 [13]. Another important project is Open Computing Language (OpenCL), an open standard for cross-platform parallel computing on different platforms, including GPGPU [10]. CUDA and OpenCL functions executed on a GPGPU device are called *kernels*.

The technologies mentioned provide special programming languages: CUDA C/C++, OpenCL C/C++. However, using higher-level languages, such as C# or F#, can be more convenient for GPGPU development, because they are used more often for general software development, and they are strongly and statically typed, which, with the help of integrated development environments (IDEs), facilitates development and improves the reliability of software. In this paper, in context of GPGPU development, we are going to call programming in special CUDA/OpenCL languages *lower-level development*, and coding in C#/F# *higher-level development.*

*The research was supported by a grant from JetBrains Research.

TyDe'18, September 27, 2018, St. Louis, Missouri, United States
2018.

There are several instruments for higher-level GPGPU development. AleaGPU [14] is a commercial product that allows using C# and F# for CUDA programming, and calling a limited number of included lower-level CUDA libraries. The work [3] presents FSCL, a limited subset of F# to OpenCL compiler, which allows developing OpenCL kernels within the .NET environment. Brahma.FSharp [1] is another F# to OpenCL compiler, but it differs from FSCL in that it focuses on translating F# *quotations*, not regular code. F# quotations are specific expressions that are compiled not in a regular way, but into objects that represent F# expressions, and can be evaluated later with F# language tools. Also, Brahma.FSharp is under active development, while FSCL has not been updated since January 2016.

There are also tools for managing GPGPU code launches from .NET. As mentioned earlier, Alea GPU allows reusing some CUDA libraries. CUSP [5] is a C++ library that enhances CUDA development with sparse linear algebra and graph computations. ManagedCUDA [11] library allows to run arbitrary pre-compiled CUDA kernels from C#, however the calls are untyped, which doesn't satisfy the need for a completely type-safe environment.

To our knowledge, neither of the existing solutions allows both programming for GPGPU in higher-level programming languages and calling arbitrary OpenCL C code in a strongly typed way. In our work we are trying to solve this problem by augmenting Brahma.FSharp project with a new component that loads OpenCL C source files with a mechanism called type provider.

2 F# type providers

A type provider [15] (TP) is a component of F# programming language that provides types, properties, and methods dynamically [12]. When imported as a DLL, a type provider adds the generated types to the client code's environment at compile-time. Type providers can have static parameters and thus receive configuration, data source etc. as arguments. Therefore, type providers allow the user to interact with data from dynamic sources (such as a file) in a statically typed way. In short, type providers are a form of compile-time metaprogramming.

The most popular usage of F# type providers is for integration with dynamic data sources, such as SQL, CSV, JSON, in a strongly typed way [6]. One relevant example is R Type Provider [2]. It enables interoperability between F# and R

by discovering installed R packages and making them available as .NET namespaces underneath the parent namespace RProvider. Another example is SQL Type Provider [4] that connects the F# environment in IDE to database sources and allows to explore them in a type-safe manner.

As an alternative to type providers, a common approach to loading external resources in a typed way is code generation. However, type providers ensure better integration with user context because they work at runtime, while the generated code must be replaced each time the data source is modified. Thus the threat of dissynchronization with the data source is eliminated.

Type providers also have disadvantages. First of all, testing type providers is very challenging because an IDE instance with any code that uses a TP, including unit tests, locks the DLL that contains the type providers and thus prevents rebuilding it. Also, debugging a type provider is a challenging process that requires two IDE instances and unusual setup, and editing TP is still impossible until the other IDE instance is closed.

3 OpenCL type provider

In order to simplify usage of existing OpenCL C kernels in applied software, we propose the OpenCL type provider which is implemented as a part of Brahma.FSharp and works as follows. The type provider loads the specified OpenCL C file, performs lexing and parsing of the loaded file, and generates an F# type with static functions that have the same signatures as the original OpenCL functions. After that one can use the provided functions inside code quotations, which will be compiled by Brahma.FSharp to OpenCL C. Type check of calls to referenced OpenCL functions will be performed during the compilation of F# code.

To use OpenCL type provider in a project, the user has to include the DLL containing the TP. Then, the KernelProvider type should be initialized with the proper static parameters, which there are two: *PathToFile* specifies the path to the OpenCL C source file being included; *TreatPointersAsArrays* defines whether the function parameters in OpenCL source code that are pointers are represented in F# environment with the corresponding array type or reference type (ByRef in F#).

The generated type provides all functions from the OpenCL C file as static F# functions. Listing 1 shows an example of TP usage: the file *mygemm.cl* is included, and the function myGEMM1 from that file is generated within the provided type in F#.

```
let [<Literal>] clSourcePath =
    __SOURCE_DIRECTORY__ + "/MyGEMM/mygemm.cl"
type Provided = KernelProvider<clSourcePath,
    TreatPointersAsArrays=true>
let cmd = <@ fun ... -> Provided.myGEMM1 ... @>
```

Listing 1. Example of TP usage

The screenshots below show examples of how the proposed type provider enhances the development process. Type information of the reused code becomes available, as well as code suggestions (figure 1). Compile-time type check of interactions with the reused code is performed (figure 2).

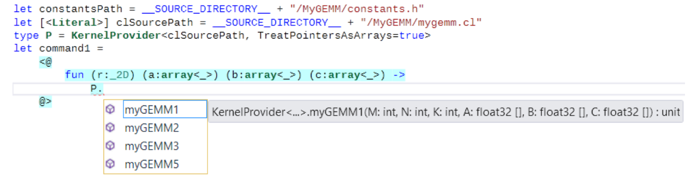

Figure 1. Code suggestions and type information in IDE

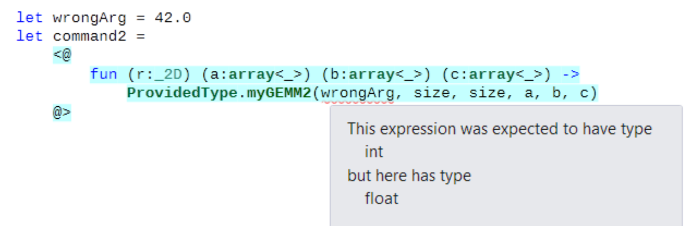

Figure 2. Compile-time type check with TP

4 Discussion

Our solution works on .NET and Mono (for Mac and Linux) and is published within NuGet package [9].

We can propose some possible directions for the future work. First of all, it is necessary to improve OpenCL C parser and translator which are used in type provider. This is required for complex kernels handling, and also for deeper and more transparent integration of OpenCL C and F# type systems. For example, running OpenCL code on different devices requires proper configuration of parameters such as grid size and tile size. Brahma.FSharp allows to set grid parameters but does not allow to pass arbitrary constants. Therefore, header files (.h) containing `#define` preprocessor macros may be required along with the reused OpenCL C code. We hope to find a way to mitigate this restriction in the future.

Another direction is unification of the kernel primitive provided by Brahma.FSharp and kernels loaded by TP. It looks natural for an end user to think that kernel primitives from Brahma.FSharp and OpenCL C kernels can be used in the same manner, but currently we have two different representations for them.

References

[1] Brahma.FSharp. 2016. Brahma.FSharp. Brahma.FSharp official page. (2016). http://yaccconstructor.github.io/Brahma.FSharp/ Date accessed: 27.03.2018.

[2] BlueMountain Capital. 2014. F# R Type Provider. (2014). http://bluemountaincapital.github.io/FSharpRProvider/ Date accessed: 27.03.2018.

[3] Gabriele Cocco. 2014. *FSCL: Homogeneous programming and execution on heterogeneous platforms*. Ph.D. Dissertation. University of Pisa.

[4] F# Community. 2014. SQLProvider. (2014). https://fsprojects.github.io/SQLProvider/ Date accessed: 27.03.2018.

[5] CUSP. 2015. CUSP. CUSP official page. (2015). https://cusplibrary.github.io/ Date accessed: 27.03.2018.

[6] F# Data. 2016. F# Data: Library for Data Access. (2016). https://fsharp.github.io/FSharp.Data/ Date accessed: 27.03.2018.

[7] Peng Du, Rick Weber, Piotr Luszczek, Stanimire Tomov, Gregory Peterson, and Jack Dongarra. 2012. From CUDA to OpenCL: Towards a Performance-portable Solution for Multi-platform GPU Programming. *Parallel Comput.* 38, 8 (2012), 391–407.

[8] J. Fung, F. Tang, and S. Mann. 2002. Mediated reality using computer graphics hardware for computer vision. In *Proceedings. Sixth International Symposium on Wearable Computers, (ISWC '02)*. IEEE Computer Society, Washington, DC, USA, 83–89. http://dl.acm.org/citation.cfm?id=862896.881093

[9] Semyon Grigorev. 2013. Brahma.FSharp NuGet package. (2013). https://www.nuget.org/packages/Brahma.FSharp Date accessed: 27.03.2018.

[10] Khronos Group. 2008. OpenCL. The open standard for parallel programming of heterogeneous systems. (2008). http://www.khronos.org/opencl/ Date accessed: 27.03.2018.

[11] ManagedCUDA. 2014. ManagedCUDA. ManagedCUDA official page. (2014). https://kunzmi.github.io/managedCuda/ Date accessed: 27.03.2018.

[12] Microsoft Developer Network. 2016. Type Providers. (2016). https://docs.microsoft.com/en-us/dotnet/articles/fsharp/tutorials/type-providers/ Date accessed: 27.03.2018.

[13] Nvidia. 2006. CUDA. Parallel Programming and Computing Platform. (2006). http://www.nvidia.com/object/cuda_home_new.html Date accessed: 27.03.2018.

[14] QuantAlea. 2010. Alea GPU. QuantAlea official page. (2010). http://www.quantalea.com/ Date accessed: 27.03.2018.

[15] Don Syme, Keith Battocchi, Kenji Takeda, Donna Malayeri, Jomo Fisher, Jack Hu, Tao Liu, Brian McNamara, Daniel Quirk, Matteo Taveggia, et al. 2012. Strongly-typed language support for internet-scale information sources. *Technical Report MSR-TR-2012–101, Microsoft Research* (2012).