



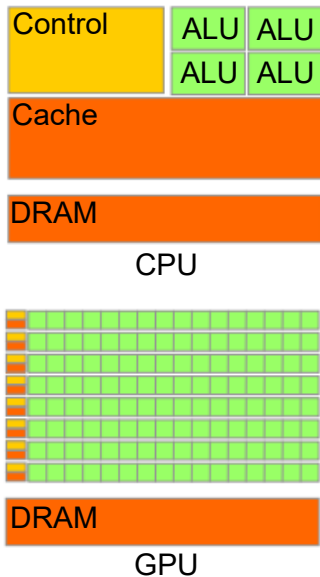
F# OpenCL C Type Provider

Kirill Smirenko, **Semyon Grigorev**

JetBrains Research, Programming Languages and Tools Lab
Saint Petersburg University

September 27, 2018

GPGPU



General purpose computations on graphical processor units

- (Almost) SIMD architecture
- Huge amount of “simple” ALUs on single chip
- Initially for computer graphic/games
- Good choice for massive data processing

General purpose applications of GPGPU

- Initially for scientific computations
 - ▶ Physics
 - ▶ Math
 - ▶ Chemistry
- But more and more for applications
 - ▶ Finance/Banking
 - ▶ Data Analytics and Data Science (Hadoop, Spark ...)
 - ▶ Security analytics (log processing)
 - ▶ Some scientific computations today are daily-used applications (bioinformatics, chemistry , ...)

High level languages and GPGPU

Low-level platforms and languages
for GPGPU programming

- NVIDIA CUDA: Cuda C, Cuda Fortran
- **OpenCL: OpenCL C**

High-level platform and languages
for applications

- C++
- Python, Haskell, OCaml, ...
- JVM: Java, Scala, ...
- .NET: C#, F#, ...

High level languages and GPGPU

Low-level platforms and languages
for GPGPU programming

- NVIDIA CUDA: Cuda C,
Cuda Fortran
- **OpenCL: OpenCL C**

High-level platform and languages
for applications

- C++
- Python, Haskell, OCaml, ...
- JVM: Java, Scala, ...
- .NET: C#, F#, ...

Interaction is a problem!

Possible solutions

- Translation of high-level language to GPGPU specific one
 - + Useful features of host language for GPGPU programming (type safety)
 - High performance GPGPU programs is inherently low-level
- Reusing of existing GPGPU libraries
 - + GPGPU optimized solution in low-level language
 - ? We need automatic generation of “well-typed” bindings

- F# quotations to OpenCL C translator
- Runtime
 - ▶ Command queue
 - ▶ Execution context management
 - ▶ Memory management
 - ▶ F# aliases for OpenCL-specific functions

F# type providers

- Compile-time metaprogramming for compile-time types creation
 - ▶ Type provider is a **function which constructs type**
- Design-time features in IDE
 - ▶ Completion
 - ▶ Type information
- Used for type-safe integration of external data with fixed schema
 - ▶ Type providers for XML, JSON, INI
 - ▶ R, SQL

Example of INI type provider

```
[Section1]
intSetting = 2
stringSetting = stringValue
[Section2]
floatSetting = 1.23
boolSetting = true
anotherBoolSetting = False
emptySetting =
stringWithSemiColonValue = DataSource=foo@bar;UserName=blah
```

```
open FSharp.Configuration
```

```
type Config = IniFile<"Config.ini">
```

```
Config.
```

ConfigFileName

Section1

Section2

type Section2 =

static member anotherBoolSetting : bool

static member boolSetting : bool

static member emptySetting : string

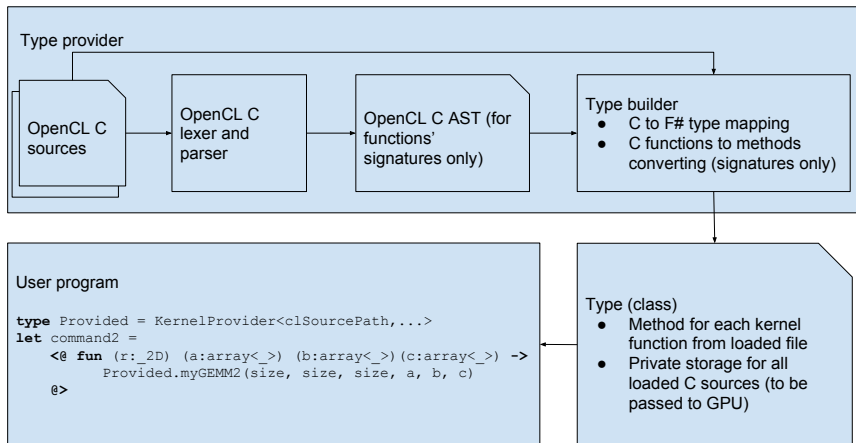
static member floatSetting : float

static member stringWithSemiColonValue : string

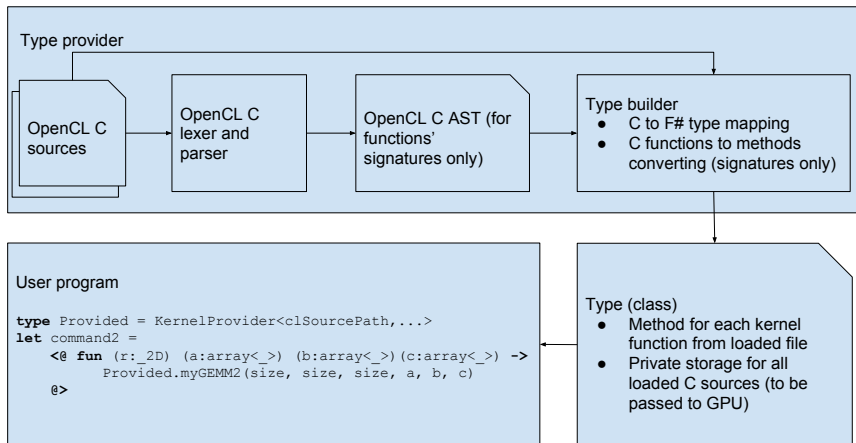
OpenCL C type provider

- We want to construct type-safe wrapper for existing library
- OpenCL standard declares source-level distribution with in place compilation
 - + We can work with source code, not with binaries
 - Existing library is a set of files includes *.h files
- It is enough to process functions signatures

OpenCL C type provider: architecture



OpenCL C type provider: architecture



Yes, it is typical type provider

Limitations

- Only (small) subset of OpenCL C
 - ▶ *.h files are not supported
 - ▶ preprocessor is not supported
 - ▶ only small subset of syntax is supported
- Very simple C to F# type mapping

Examples

```
// TypeProvider configuration
let constantsPath = __SOURCE_DIRECTORY__ + "/constants.h"
let [<Literal>] clSourcePath = __SOURCE_DIRECTORY__ + "/mygemm.c"
type ProvidedType = KernelProvider<clSourcePath, TreatPointersAsArrays=true>
```

Examples

```
// TypeProvider configuration
let constantsPath = __SOURCE_DIRECTORY__ + "/constants.h"
let [<Literal>] clSourcePath = __SOURCE_DIRECTORY__ + "/mygemm.c"
type ProvidedType = KernelProvider<clSourcePath, TreatPointersAsArrays=true>

let command2 =
    <@
        fun (r:_2D) (a:array<_>) (b:array<_>) (c:array<_>) ->
            ProvidedType.myGEMM2(newSize, size, size, a, b, c)
    @>
```

This expression was expected to have type
int
but here has type
float

Examples

```
// TypeProvider configuration
let constantsPath = __SOURCE_DIRECTORY__ + "/constants.h"
let [<Literal>] clSourcePath = __SOURCE_DIRECTORY__ + "/mygemm.c"
type ProvidedType = KernelProvider<clSourcePath, TreatPointersAsArrays=true>

let command2 =
    <@
        fun (r:_2D) (a:array<_>) (b:array<_>) (c:array<_>) ->
            ProvidedType.myGEMM2(newSize, size, size, a, b, c)
    @>
```

This expression was expected to have type
int
but here has type
float

```
let command2 =
    <@
        fun (r:_2D) (a:array<_>) (b:array<_>) (c:array<_>) ->
            ProvidedType.
    @>
```

- myGEMM1
- myGEMM2
- myGEMM3
- myGEMM5

KernelProvider<...>.myGEMM1(M: int, N: int, K: int, A: float32 [], B: float32 [], C: float32 []): unit

Future work

- Improve OpenCL C support
 - ▶ Lexer and parser
 - ▶ Translator
 - ▶ Types mapping
 - ▶ Headers files processing
 - ▶ ...
- Unify kernels on client side
 - ▶ Currently native Brahma.FSharp's kernel and kernel loaded by type provider are different types
- Improve mechanism of kernels composition

Summary

- F# OpenCL C type provider
 - ▶ Type-safe integration of existing OpenCL C code in F# applications
 - ▶ Prototype with limitations
- Source code on GitHub:
<https://github.com/YaccConstructor/Brahma.FSharp>
- Package on NuGet:
<https://www.nuget.org/packages/Brahma.FSharp/>

- Semyon Grigorev: s.v.grigoriev@spbu.ru
- Kirill Smirenko: k.smirenko@gmail.com
- Brahma.FSharp:
<https://github.com/YaccConstructor/Brahma.FSharp>

Thanks!