Санкт-Петербургский государственный университет

Кафедра системного программирования

Кирилл Петрович Смиренко

Разработка механизма использования OpenCL-кода в программах на F#

Курсовая работа

Научный руководитель:
к.ф.-м.н., доц. Григорьев С. В.

Санкт-Петербург
2017

Оглавление

Введение

Графические процессоры (GPU) являются общепринятым средством ускорения вычислений. Многоядерная архитектура видеопроцессоров даёт преимущество в высоконагруженных научных вычислениях, задачах компьютерного зрения, биоинформатики и других областей. Данный подход лёг в основу техники вычислений общего назначения на видеопроцессорах (GPGPU) [5, 7].

Существует ряд технологий для программирования видеопроцессоров. Наиболее распространённой является CUDA — платформа для параллельных вычислений на видеопроцессорах, разработанная компанией Nvidia в 2007 году [14]. Другим значимым проектом является Open Computing Language (OpenCL) — открытый стандарт кросс-платформенного, параллельного программирования различных процессоров, в том числе видопроцессоров, присутствующих в персональных компьютерах, серверах, мобильных и встраиваемых устройствах [8].

Упомянутые технологии предоставляют специальные языки программирования: CUDA C/C++, OpenCL C/C++. В то же время более удобным способом разработки для видеопроцессоров является использование более высокоуровневых языков, таких, как C# и F#. Эти языки чаще применяются для написания конечных приложений; кроме того, строгая типизация и статический анализ в интегрированных средах разработки повышают удобство прикладного программирования и надёжность разрабатываемого кода. Уже существует и активно используется ряд средств для высокоуровневого программирования видеопроцессоров [2, 4, 17].

При этом возникает потребность уметь переиспользовать в языках высокого уровня существующий код на специальных языках для видеопроцессоров. Это продиктовано следующими соображениями. Во-первых, переиспользование готового и проверенного кода вместо переписывания — стандартная инженерная практика. Во-вторых, низкоуровневый код для видеопроцессоров содержит специфические конструкции, например, барьеры и модификаторы памяти, часто исполь-

зуемые для оптимизаций. Пытаться выразить эти конструкции в языка высокого уровня лишь затем, чтобы потом транслировать их обратно в код целевой платформы (CUDA, OpenCL) для запуска на видеопроцессоре, не представляется рациональным решением.

Таким образом, является актуальной задача переиспользования низкоуровневого кода для GPU в высокоуровневом программировании. При этом отдельный интерес представляет возможность вызова низкоуровневых функций типизированным образом, что существенно повысило бы удобство прикладного программирования.

1. Обзор

1.1. Средства программирования видеопроцессоров

Существует ряд инструментов, предназначенных для программирования видеопроцессоров на языках программной платформы .NET. Наиболее известные из них представлены ниже.

- Alea GPU — коммерческий продукт от компании QuantAlea, предоставляющий средства разработки для платформы CUDA на языках C# и F# [17].
- Brahma.FSharp — проект с открытым исходным кодом, разрабатываемый на кафедре системного программирования математико-механического факультета СПбГУ. Это транслятор цитируемых выражений языка F# в код платформы OpenCL [2].
- FSCL — другой компилятор F#-кода в OpenCL C с открытым исходным кодом [4].

Также существуют программные проекты, позволяющие из C#-кода управлять запуском кода на специальных языках для видеопроцессора. К ним относятся:

- Alea GPU, предоставляющий возможность вызова ряда готовых низкоуровневых библиотек [17];
- CUSP — C++-библиотека для вычислений с разреженными матрицами и обработки графов [3];
- ManagedCUDA — проект, содержащий средства нетипизированного вызова скомпилированных CUDA-функций в коде на C# и C#-интерфейс для ряда наиболее популярных CUDA-библиотек. [11].

Однако все имеющиеся решения либо только транслируют высокоуровневый код на языках платформы .NET в низкоуровневый код платформ CUDA или OpenCL (Brahma.FSharp, FSCL), либо позволяют

использовать лишь фиксированный набор низкоуровневых библиотек (Alea GPU, CUSP). ManagedCUDA является наиболее близким решением, однако не предоставляет типизированных функций вызова низкоуровневого кода и не является транслятором .NET-кода в код платформы CUDA, позволяя работать лишь с предварительно скомпилированной кодовой базой.

В результате обзора можно сделать вывод, что ни одно из существующих решений не позволяет программировать видеопроцессоры на языке высокого уровня и при этом вызывать произвольный низкоуровневый код типизированным образом.

1.2. Провайдеры типов языка F#

Существует несколько способов интеграции низкоуровневого кода и среды исполнения .NET. Особый интерес представляют провайдеры типов F# [13]. Это механизм языка, который генерирует типы данных и встраивает в окружение времени исполнения как обычные типы F#. Провайдеры типов могут иметь типовые параметры, и, таким образом, имеется возможность во время разработки на F# пользоваться статической типизацией данных, которые были получены из динамических источников, например, из файла.

Традиционной и наиболее близкой альтернативой использованию провайдеров типов является кодогенерация. Однако по сравнению с ней провайдеры обладают рядом преимуществ:

- провайдеры типов обеспечивают тесную интеграцию с пользовательским контекстом: сгенерированные типы сразу находятся в одном пространстве имён с кодом, который их использует;
- генерация типов происходит во время компиляции пользовательского кода, что избавляет от опасности рассинхронизации с источником данных.

Провайдеры типов не лишены недостатков. В частности, затруднено тестирование проекта, в котором есть провайдеры типов: тесты, как и

любой код, использующий провайдеры, не могут быть включены в ту же сборку, что и провайдер, так как блокируют пересборку последнего. Кроме того, отладка провайдеров типов представляет собой достаточно сложную процедуру, отличающуюся от процесса отладки обычного кода [18]. Тем не менее, данные недостатки относятся к процессу разработки программного решения, но не затрудняют его использование.

Таким образом, именно механизм провайдеров типов F# был выбран для интеграции разобранного OpenCL-кода и платформы .NET.

2. Постановка задачи

Целью данной работы является добавление возможности переиспользования OpenCL C-кода в проект Brahma.FSharp. Для её достижения были поставлены следующие задачи:

- исследовать возможности механизма провайдеров типов F#;
- реализовать лексический и синтаксический анализатор заголовков OpenCL-функций;
- обеспечить возможность типизированного вызова OpenCL-функций в коде на F#;
- провести экспериментальное исследование представленного решения.

3. Механизм подгрузки OpenCL-кода

3.1. Архитектура

На схеме 1 представлена архитектура предлагаемого решения. Подгружаемый OpenCL C-файл подаётся последовательно лексическому, синтаксическому анализатору и провайдеру типов. На выходе генерируется F#-тип, содержащий подгруженные функции. Эти функции могут быть использованы в клиентском коде — цитируемом выражении F#, которое будет подано существующему транслятору Brahma.FSharp, а результат трансляции — передан драйверу OpenCL для запуска на видеопроцессоре.

Реализуемый в рамках настоящей работы модуль, таким образом, можно разделить на три составляющих: лексический анализатор, синтаксический анализатор, провайдер типов. Далее будет произведён их подробный обзор.

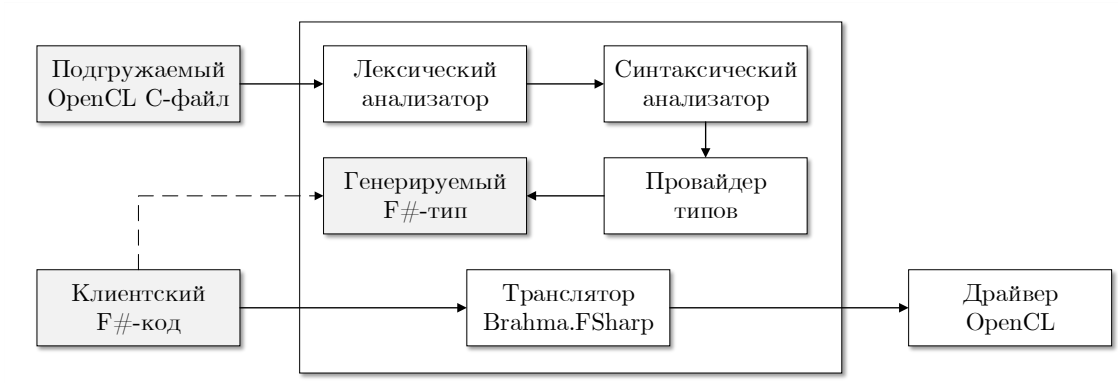

Рис. 1: Структура предлагаемого решения

3.2. Лексический и синтаксический анализатор

Для того, чтобы извлекать и переиспользовать сигнатуры функций OpenCL C, нужно уметь разбирать исходный код, т.е. проводить лексический и синтаксический анализ. Для реализации лексического анализатора был использован FsLex — инструмент с открытым исходным

кодом, генерирующий исходный код лексических анализаторов на языке F# по данному описанию языка в специальной нотации [6].

При написании синтаксического анализатора была использована библиотека YaccConstructor [22, 20]. Она предоставляет язык YARD [21] для описания синтаксических анализаторов. Этот язык имеет ряд преимуществ перед аналогами [1, 10] применимо к данной задаче: естественная интеграция с F#, возможность описывать грамматики в расширенной форме Бэкуса-Наура [19], поддержка S- и L-атрибутов, а также параметризованных правил вывода.

Синтаксический анализатор описан в виде S-атрибутной грамматики в расширенной форме Бэкуса-Наура. В качестве справочных материалов была использована формальная грамматика C99 [16] и спецификация языка OpenCL C [9]. Исходный код обоих анализаторов на F# генерируется упомянутыми инструментами при сборке проекта.

Представленная грамматика является упрощённой и не описывает весь язык OpenCL C: тот факт, что в данной задаче необходимо распознавать лишь заголовки функций, позволил при разборе игнорировать тела функций.

3.3. Провайдер типов для OpenCL-функций

В рамках работы был реализован провайдер типов для разобранных OpenCL-функций. Провайдер генерирует F#-тип, содержащий функции, которые типизированы так же, как исходные функции на OpenCL C. Провайдер параметризован путём к файлу, содержащему подгружаемый код на OpenCL C; файл может содержать неограниченное количество функций, однострочные комментарии и макросы препроцессора C. Вначале файл с исходным кодом на OpenCL C подвергается лексическому и синтаксическому анализу. Далее производится обход синтаксического дерева, полученного на этапе синтаксического анализа: из дерева извлекаются структурированные сигнатуры функций, по которым провайдер генерирует статические методы предоставляемого метода.

В C-подобных языках, включая OpenCL C, массивы чаще всего передаются по указателям. Поэтому для параметров функций, которые являются указателями, провайдер поддерживает два варианта отображения их в F#: как ссылочный тип и как массив. Это задаётся другим параметром провайдера. Пример подгрузки с помощью провайдера функции умножения матрицы на вектор представлен на изображении 2.

```
let matvec = KernelProvider<matvecPath, TreatPointersAsArrays=true>.matvec
```

Рис. 2: Пример использования провайдера типов

Также была проведена точечная доработка транслятора Brahma.FSharp. Транслятор одновременно с клиентским F#-кодом получает имя подгружаемого OpenCL C-файла, читает его содержимое и в текстовом виде передаёт драйверу OpenCL вместе с результатом трансляции F#-кода. Это обеспечивает необходимую функциональность.

4. Эксперименты

Помимо модульного тестирования представленного механизма, были произведены экспериментальные исследования работы механизма с высокопроизводительным кодом для видеопроцессора. Это отвечает изначальной мотивации данной работы. В целях эксперимента из стороннего проекта с открытым исходным кодом [12] был взят алгоритм перемножения вещественных матриц на видеопроцессоре, оптимизированный с использованием специфических конструкций OpenCL: барьеров и локальных групп.

Для сравнения производительности были также взяты неоптимизированные реализации наивного алгоритма перемножения матриц на F# и OpenCL C. Оптимизации, аналогичные таковым в первой рассматриваемой реализации, не представляются возможными в F# ввиду отсутствия в языке специфических низкоуровневых конструкций. Наивная реализация на OpenCL C участвует в эксперименте с целью сравнения скорости работы OpenCL-кода и аналогичного кода, транслированного из F#.

Запуск производился средствами Brahma.FSharp на видеокарте NVIDIA GeForce GT 755M с тактовой частотой графического процессора 980 МГц и памятью 2048 МБ. Результаты экспериментов представлены на рисунке 3.

Как видно на диаграмме, наивные реализации на F# и OpenCL C почти не отличаются по производительности, в то время как оптимизированная реализация показывает почти трёхкратный прирост производительности на больших матрицах. Это подтверждает целесообразность переиспользования OpenCL-кода, оптимизированного с помощью низкоуровневых конструкций, в высокоуровневом программировании видеопроцессоров.

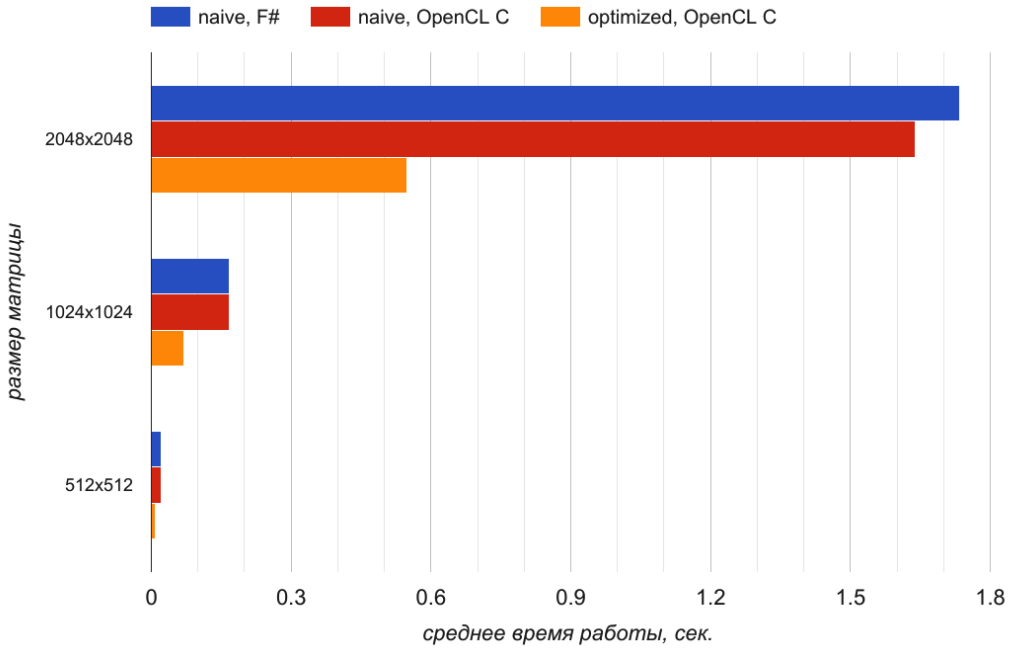

Рис. 3: Результаты экспериментов

Заключение

В ходе работы получены следующие результаты:

- исследован механизм провайдеров типов F#;
- реализован лексический и синтаксический анализатор заголовкой OpenCL-функций;
- реализован механизм типизированного вызова OpenCL-функций в коде на F# на основе провайдера типов;
- проведено экспериментальные исследование работы реализованного модуля.

Код реализованного модуля, включающего лексический, синтаксический анализатор и провайдер типов, находится на сайте `https://github.com/YaccConstructor/Brahma.FSharp`. В указанном репозитории автор принимал участие под учётной записью ksmirenko.

В дальнейшем планируется исследовать возможность применения реализованного механизма при решении задач синтаксического анализа

графов с использованием современного алгоритма перемножения разреженных матриц [15]. Соответствующий проект разрабатывается на кафедре системного программирования математико-механического факультета СПбГУ.

Список литературы

[1] ANTLR. Another Tool for Language Recognition. — 2014. — URL: http://http://www.antlr.org/ (online; accessed: 14.05.2017).

[2] Brahma.FSharp. Brahma.FSharp // Brahma.FSharp official page. — URL: http://yaccconstructor.github.io/Brahma.FSharp/ (online; accessed: 14.05.2017).

[3] CUSP. CUSP // CUSP official page. — URL: https://cusplibrary.github.io/ (online; accessed: 14.05.2017).

[4] Cocco Gabriele. FSCL: Homogeneous programming and execution on heterogeneous platforms : Ph. D. thesis / Gabriele Cocco ; University of Pisa. — 2014.

[5] From CUDA to OpenCL: Towards a Performance-portable Solution for Multi-platform GPU Programming / Peng Du, Rick Weber, Piotr Luszczek et al. // Parallel Comput. — 2012. — Vol. 38, no. 8. — P. 391–407.

[6] FsLexYacc. FsLex. — URL: http://fsprojects.github.io/FsLexYacc/fslex.html (online; accessed: 14.05.2017).

[7] Fung J., Tang F., Mann S. Mediated reality using computer graphics hardware for computer vision // Proceedings. Sixth International Symposium on Wearable Computers,. — 2002. — P. 83–89.

[8] Group Khronos. OpenCL // The open standard for parallel programming of heterogeneous systems. — URL: http://www.khronos.org/opencl/ (online; accessed: 14.05.2017).

[9] Group Khronos. The OpenCL C Specification. — 2016. — URL: https://www.khronos.org/registry/OpenCL/specs/opencl-2.0-openclc.pdf (online; accessed: 14.05.2017).

[10] Johnson S. C. Yacc: Yet Another Compiler Compiler // Computing Science Technical Report. — 1975.

[11] ManagedCUDA. ManagedCUDA // ManagedCUDA official page. — URL: https://kunzmi.github.io/managedCuda/ (online; accessed: 14.05.2017).

[12] MyGEMM. Code appendix to an OpenCL matrix-multiplication tutorial. — URL: https://github.com/CNugteren/myGEMM (online; accessed: 14.05.2017).

[13] Network Microsoft Developer. Type Providers. — 2016. — URL: https://docs.microsoft.com/en-us/dotnet/articles/fsharp/tutorials/type-providers/ (online; accessed: 14.05.2017).

[14] Nvidia. CUDA // Parallel Programming and Computing Platform. — URL: http://www.nvidia.com/object/cuda_home_new.html (online; accessed: 14.05.2017).

[15] Polok Lukas, Ila Viorela, Smrz Pavel. Fast Sparse Matrix Multiplication on GPU // Proceedings of the Symposium on High Performance Computing. — HPC '15. — Society for Computer Simulation International, 2015. — P. 33–40. — URL: http://dl.acm.org/citation.cfm?id=2872599.2872604.

[16] Programming languages – C : Standard / International Organization for Standardization, International Electrotechnical Commission : 1999.

[17] QuantAlea. Alea GPU // QuantAlea official page. — URL: http://www.quantalea.com/ (online; accessed: 14.05.2017).

[18] Tihon Sergey. F# Type Providers Development Tips. — 2016. — URL: https://sergeytihon.com/2016/07/11/f-type-providers-development-tips-not-tricks/ (online; accessed: 14.05.2017).

[19] Wirth Niklaus. What Can We Do About the Unnecessary Diversity of Notation for Syntactic Definitions? // Commun. ACM. — 1977. — Vol. 20, no. 11. — P. 822–823.

[20] YaccConstructor. YaccConstructor // YaccConstructor official page. — URL: `http://yaccconstructor.github.io` (online; accessed: 14.05.2017).

[21] YaccConstructor. YARD // YaccConstructor official page. — 2015. — URL: `http://yaccconstructor.github.io/YaccConstructor/yard.html` (online; accessed: 14.05.2017).

[22] Кириленко ЯА, Григорьев СВ, Авдюхин ДА. Разработка синтаксических анализаторов в проектах по автоматизированному реинжинирингу информационных систем // Научно-технические ведомости СПбГПУ: информатика, телекоммуникации, управление. — 2013. — no. 174. — P. 94–98.