Автоматизация создания генераторов проблемно-ориентированных программ

И.Ю. Сесин, Р.Г. Болбаков

***Аннотация*—Генерация проблемно-ориентированных программ является подходом, ориентированным на улучшение производительности определённого класса программ, использующих технологию GPGPU (англ. General Purpose computing for Graphical Processing Units, расчёты общего назначения на графических процессорах).**

Ручное создание генератора проблемно-ориентированных программ это трудоёмкий и сложный процесс, требующий от программиста высокой квалификации и знания особенностей решаемой проблемы.

В этой статье рассматривается подход к автоматизации создания генераторов проблемно-ориентированных программ. Вводится понятие мета-генератора как программы, создающей на базе исходного кода GPU программы генератор проблемно-ориентированных программ. Описаны основные задачи, решаемые в процессе анализа программы мета-генератором и рассмотрены технологии, подходящие для применения в рамках решения поставленных задач вкупе с особенностями их внедрения.

Проведено сравнение между предлагаемым подходом и существующими идейно схожими методами улучшения производительности программного обеспечения, рассмотрены их ключевые различия.

***Ключевые слова*— генератор проблемно-ориентированных программ, расчёты общего назначения на графических процессорах, оптимизация компьютерных программ.**

I. Введение

Графические процессоры, или GPU (англ. Graphical Processing Unit) являются специализированным аппаратным обеспечением, выполняющим обработку графической информации.

Для борьбы с возникающими потерями производительности был выдвинут метод генераторов проблемно-ориентированных программ, подразумевающий создание отдельного этапа обработки, на котором на базе входных данных решается, какие части программы будут скомпилированы.

Тем не менее, создание генераторов проблемно-ориентированных программ является достаточно трудоёмкой задачей, требующей основательного анализа исходной программы и решаемой ей задачи программистом.

К тому же созданные генераторы привязаны к конкретным архитектурным решениям оригинальных программ и не могут быть использованы повторно для отличных целей.

В силу изложенных обстоятельств авторами настоящей статьи представляется целесообразным рассмотрение более общих подходов, имеющих целью частичную или полную автоматизацию создания генераторов проблемно-ориентированных программ.

II. Генерация генераторов проблемно-ориентированных программ

Для более полного понимания автоматизируемой задачи, разберём выполняемые человеком процессы и рассмотрим технические средства и подходы могущие помочь в полной или частичной автоматизации оных процессов.

Процесс «ручного» создания генераторов проблемно-ориентированных программ можно разбить на следующие этапы:

– анализ структуры исходного кода оригинальной GPGPU программы;

– внесение инструментации в исходный код или процесс сборки программы для достижения возможности управления тем, какие участки кода будут скомпилированы;

– написание процедуры для анализа входных данных программы, которая будет определять множество отсекаемых ветвей;

– включение созданной процедуры в цепочку взаимодействий между программой на хост-устройстве и программой на графическом процессоре.

Ключевой составляющей подхода к созданию генераторов проблемно-ориентированных программ является анализ входных данных программы с целью элиминации неиспользуемых ветвей.

Задачи анализа исходного кода GPGPU программы и создания процедур определения множества отсекаемых ветвей, могут быть вынесены в отдельную программу. Используя полученные данные, эта программа будет генерировать код непосредственно генератора проблемно-ориентированных программ для проанализированной оригинальной GPGPU программы. Назовём такую программу *мета-генератором*.

Так как задача получения проблемно-ориентированной программы переносится с плеч программиста на автоматику, становится возможным изменить подход к генерации программы. Вместо

Статья получена 19 июля 2021.

И. Ю. Сесин, РТУ МИРЭА (e-mail: isesin@protonmail.com).

Р. Г. Болбаков, к.т.н. доцент, зав. каф. ИППО института ИТ РТУ МИРЭА (e-mail: Bolbakov@mirea.ru).

использования конструкций управляющих генерацией кода на этапе препроцессора становится возможным управление исходным кодом в целом.

Иначе говоря, генераторы проблемно-ориентированных программ, образованные мета-генератором могут попросту генерировать сам код исходной программы вместо набора определений для препроцессора. Это существенно более гибкий и мощный подход, позволяющий делать преобразования с исходным кодом программы, которые препроцессор выполнять не может. С другой стороны, сложность программы, которая работает с преобразованием исходного кода вырастает экспоненциально — но большую часть этой сложности можно вынести в мета-генератор, оставляя в генераторах проблемно-ориентированных программ только проверки, зависящие от входных данных программы.

Процесс работы предлагаемого метода можно разбить на три этапа.

Первый этап является подготовительным, и на него выносятся самые сложные и ресурсоёмкие задачи. На этом этапе мета-генератор производит анализ исходного кода GPGPU программы и конструирует генератор проблемно-ориентированных программ, специализированный под указанную GPGPU программу. Этот этап необходимо провести лишь единожды, при условии, что код GPGPU программы не меняется.

Второй этап входит в процесс регулярного использования GPGPU программы и будет предприниматься каждый раз когда требуется вызвать GPGPU программу с новым набором данных. В ходе этого этапа производится вызов сформированной мета-генератором программы с входными данными программы для GPU. Эта программа (генератор проблемно-ориентированных программ) сформирует специализированный под переданные входные данные код GPU программы, после чего этот код можно будет передать на компиляцию. Этот этап целесообразно производить вместе с процессом валидации входных данных.

Третий этап это непосредственный запуск сгенерированной проблемно-ориентированной программы на GPU с предоставленными входными данными. Этот этап, как правило, предваряется вторым этапом в типичном сценарии использования, но может быть и самостоятельным в случае, когда входные данные не менялись и повторный вызов генератора проблемно-ориентированных программ не имеет смысла.

III. Устройство мета-генераторов

В задачи, возложенные на мета-генераторы входит анализ исходного кода программы, определение зависимости ветвей исполнения от входных данных GPGPU программы и формирование кода генератора проблемно-ориентированных программ.

Рассмотрим процесс анализа программы и установление зависимости ветвей условных операторов от входных данных.

В процессе анализа GPGPU программы должно осуществляться выражение логических условий условных операторов в контексте переменных, переданных исполняемой GPGPU программе. Для этого требуется проследить всю цепочку взаимодействий переменных от каждого условного оператора до входных данных программы и использовать полученные данные для генерации реинтерпретированных условий, на основе которых генератор проблемно-ориентированных программ будет включать определённые ветви в код программы для последующего запуска.

Процесс анализа GPGPU программы разбивается на две задачи — выделение цепочек связей и создание реинтерпретированных условий.

Выделение цепочек связей подразумевает определение связи между входными данными и активацией различных участков программы. Так как входные данные могут быть преобразованы и использованы в расчётах программой множество раз, требуется аккуратный учёт последствий каждого действия программы. Для решения этой задачи хорошо подходит технология символьного исполнения [1, 2, 3] (англ. symbolic execution).

Технология символьного исполнения позволяет определить, какие входные данные приводят к запуску тех или иных частей программы. Это осуществляется при помощи интерпретатора, который следует нормальному ходу выполнения программы, но с тем различием, что вместо константных входных данных используются символьные переменные. Таким образом вместо конкретных значений, программа аналитически оперирует выражениями, включающими исключительно символьные переменные и константы. Этот процесс весьма схож с процессом работы с неизвестными в ходе решения человеком уравнений.

Когда символьное исполнение достигает условного оператора, исполнение делится на две ветви, каждая из которых отражает состояние соответствующей ветви условного оператора, включая ограничения, которые были введены для удовлетворения (или наоборот, нарушения) логического условия оператора.

Символьное исполнение имеет ряд ограничений, таких как сложности работы с указателями и рекурсивными функциями, но, с другой стороны, для вышеуказанных целей не обязательно полное символьное исполнение программы — достаточно лишь определение условий и веток, на которые влияют эти условия.

В процессе решения поставленной задачи с использованием символьного исполнения мета-генератор фиксирует данные о каждом операторе ветвления, включая условия и границы областей его блоков. Используя эти границы, возможно разделить файл с исходным кодом на «срезы», часть из которых может быть исключена в зависимости от входных данных.

Но для того, чтобы стало возможно применить символьное исполнение сначала нужно преобразовать исходный код GPGPU программы в более удобную

форму для обработки, а именно в абстрактно-синтаксическое дерево (AST), отражающее структуру программы.

Для этого следует выполнить разбор самого текста кода, что, по сути, является одной из задач выполняемых компиляторами. Соответственно, целесообразно применение существующих инструментов, благо пласт знаний и наработок из теории компиляторов и смежных областей знаний применимы к GPGPU программам в такой же степени как и к программам, запускаемым на CPU.

В случае C-подобных языков (OpenCL, CUDA C) для разбора и превращения кода в AST часто используется комплекс программных средств для создания компиляторов, состоящий из Lex и Yacc [4]. Помимо этого, прежде чем приступать к разбору текста программы, в случае C-подобных языков нужно осуществлять обработку текста программы препроцессором, чтобы была осуществлена подстановка макросов и констант.

Второй частью задачи является вывод условий, которые будут встроены в генерируемый генератор проблемно-ориентированных программ.

Под этим понимается создание логических условий, которые будут применены ко входным данным с целью установления множества используемых частей программы в процессе генерации проблемно-ориентированной программы.

Задача формулирования условий через входные данные, на деле, решается в два этапа — основа работы закладывается в процессе символьного исполнения программы, где все переменные автоматически выражаются через символьные переменные входных данных и константы. По мере того, как программа символьно исполняется, формулируется цепочка преобразований, но эти цепочки могут быть весьма громоздки и требуют упрощения.

Для решения проблемы переформулирования и упрощения цепочек можно применить методы компьютерной алгебры [5, 6].

Системы компьютерной алгебры работают с математическими выражениями, применяя символьные вычисления, позволяя проводить операции с введёнными в программу математическими выражениями с константами и неизвестными. Они позволяют производить большое количество операций, включая:

– упрощение выражений или приведение их к стандартной форме;
– взятие первообразных и производных;
– операции с рядами;
– взятие пределов;
– автоматическое доказательство теорем.

Для решения задачи представляет интерес первый пункт. Несмотря на то, что это всего лишь часть системы компьютерной алгебры, инструментарий для упрощения выражения является обширной областью исследований и продвинутые системы компьютерной алгебры включают существенное количество различных стратегий, позволяющих упрощать те или иные аспекты сложных выражений или преобразовывать их в определённую стандартизированную форму.

Важно, что системы компьютерной алгебры имеют существенные отличия в том, как работают арифметические операции по сравнению со средой исполнения программы. В случае программы существуют такие проявления ограничений архитектуры как переполнение целочисленных переменных и ограниченная точность чисел с плавающей точкой.

В системах компьютерной алгебры эти ограничения были разрешены, но они по-прежнему актуальны для программ, которые будут прооптимизированы с помощью мета-генератора.

В то же время программы достаточно редко используют пограничные случаи поведения стандартных численных типов для реализации критического функционала, что на практике позволяет пренебречь этим несоответствием между расчётами систем компьютерной алгебры и аппаратной платформы. Строго говоря, для полной эмуляции поведения аппаратной составляющей требуется внесение существенных изменений в системы компьютерной алгебры, таких как создание отдельных типов данных для эмулируемых типов, с заданием нового набора правил их поведения, который бы детально воспроизводил характерные особенности, в т.ч. недокументированные особенности, конкретной аппаратной платформы.

В курсе решения поставленной задачи целесообразно использовать систему компьютерной алгебры, которая может быть внедрена в качестве библиотеки [7], нежели чем системы, выполненные в виде отдельного программного комплекса с пользовательским интерфейсом.

После проведения анализа мета-генератор должен сформировать код программы, которая будет генерировать специализированные версии анализируемой GPGPU программы.

Сгенерированная программа должна принимать те же входные данные, что и исходная GPGPU программа, обрабатывать их при надобности и применять выведенные мета-генератором условия, чтобы определить множество используемых частей программы.

Это множество затем используется в процессе линейной сборки исходного кода проблемно-ориентированной программы из частей («срезов»), на которую разбил анализируемую программу мета-генератор в ходе извлечения данных о ветвлении.

Таким образом, генератор проблемно-ориентированных программ существенно более прост и быстр, нежели мета-генератор. Это разбиение соотношения сложности позволяет сократить время работы наиболее часто используемых программ — генераторов и специализированных GPGPU программ, оставляя длительную обработку мета-генератором в качестве единоразовой платы за производительность.

.

IV. Сравнение с существующими программно-аппаратными средствами повышения производительности

Рассмотрим схожие методы и подходы среди существующих технологий увеличения производительности программного обеспечения.

В целом, предложенный подход идейно наиболее схож с принципами, применяемыми в JIT-компиляторах [8, 9] (Just-in-time компиляторах), но имеет существенные отличия. В то время как программы, использующие Just-In-Time компилируют интерпретируемый код в машинный код по мере надобности (т. е. при вызове участков кода, которые ещё не были скомпилированы), подход генерации проблемно-ориентированных программ подразумевает полную компиляцию программы, составленную только из используемых участков.

Это связано, в первую очередь, с невозможностью применения самомодифицирующегося кода в условиях графического процессора по ряду архитектурных причин, включая изоляцию страниц памяти с кодом и данными и плохую приспособленность SIMD архитектуры к использованию самомодифицирующегося кода.

В свою очередь, JIT имеет вариацию, называемую пре-JIT компиляцией (англ. pre-JIT), в которой весь код программы компилируется в машинный код заранее, или, в отдельных случаях, непосредственно перед запуском. Компиляция всей программы может занять существенное время, но это позволяет добиться существенно более плавной операции программы, т. к. при полностью скомпилированном коде программа не будет вызывать методы JIT компилятора по достижении кода, который не запускался раньше.

В определённом смысле такой подход перестаёт быть JIT компиляцией и переходит в разряд AOT (англ. Ahead-Of-Time, компиляция заранее) компиляции, однако он имеет свою нишу, ускоряя работу программ на интерпретируемых языках.

Существенное отличие этого пре-JIT метода от метода генераторов проблемно-ориентированных программ в отсутствии в пре-JIT анализа входных данных программы для определения используемых частей программы. Пре-JIT осуществляет полную компиляцию программы, в то время как генераторы проблемно-ориентированных программ выборочны, позволяя решить проблему падения производительности из-за предикации на графических процессорах.

Ещё одним методом, идейно близким к предложенному, является один из подходов, включаемых в арсенал оптимизирующих компиляторов, а именно устранение мёртвого кода [10] (англ. dead code elimination).

Устранение мёртвого кода подразумевает исключение из компилируемой программы заведомо неиспользуемых участков (т. н. «мёртвого кода») в ходе анализа структуры программы компилятором. Мёртвый код может возникать в силу множества причин и в самых разнообразных конструкциях кода, включая условные операторы. Используя данные о константах, компилятор определяет инвариантность условий к входным данным программы, и при их ложности просто не включает участок кода, зависящий от этой проверки.

Этот принцип работы очень близок к предложенному, но он решает иные задачи. В первую очередь применение подхода устранения мёртвого кода направлено на устранение участков кода, которые по недосмотру программиста остаются в программе, но не используются ей. Это могут быть как временные отладочные конструкции так и полноправные участки кода, которые были замещены в процессе развития и модификации программы.

Этот подход силён только когда дело касается заранее известных констант, объявленных в программе и не может быть применён с учётом сведений о входных данных.

У этого метода существует редкая вариация, известная как динамическое устранение мёртвого кода [11] (англ. dynamic dead code elimination). Под ним понимается динамическое определение зависимостей кода и подключение соответствующего кода (чаще всего в форме библиотеки), с последующим изменением кода для активации требуемой возможности. В типовом случае, это направлено на уменьшение размера занимаемого программой в оперативной памяти. В более широком понимании, этот метод подразумевает динамическое изменение программы во время исполнения для включения или отключения конкретных возможностей программы.

Полноправное применение этого метода для компилируемых языков невероятно редко ввиду его большой сложности. Тем не менее, этот подход, по сути, неявно реализуется для интерпретируемых языков [12] с JIT компилятором, как одна из самых основных стратегий применения JIT.

Как и в случае с JIT-компиляторами, данный подход реализуется за счёт динамического изменения исполняемой программы, что коренным образом отличает его от подхода генерации проблемно-ориентированных программ.

V. Заключение

В статье рассмотрен подход к автоматизации процесса создания генераторов проблемно-ориентированных программ для GPGPU программ с целью дальнейшего упрощения процесса решения проблемы уменьшения производительности программ для расчётов общего назначения на GPU, возникающей в ходе наращивания функционала оных.

Рассмотрены типовые этапы создания программистом подобного генератора в ходе ручного внедрения генераторов проблемно-ориентированных программ. Описываются изменения, которые следует внести в эти этапы для более успешной автоматизации.

В статье было введено понятие мета-генератора как программы, производящей генераторы проблемно-ориентированных программ на основе их исходного кода. Описывается структура программы мета-

генератора.

Рассмотрены аспекты анализа программ мета-генераторами, основные задачи, решаемые в процессе анализа кода GPU программ, и технологии, подходящие для решения этих задач.

Проведено сравнение данного метода с иными подходами к повышению производительности программного обеспечения, указаны их сходства и различия.

Дальнейшее развитие указанной в статье проблематики подразумевает внедрение описанной программы и рассмотрение смежных подходов, обладающих синергетическим потенциалом по отношению к предложенному методу.

Библиография

[1] Boyer, R. S., Elspas, B., & Levitt, K. N. (1975). SELECT—a formal system for testing and debugging programs by symbolic execution. ACM SigPlan Notices, 10(6), 234-245.

[2] King, J. C. (1976). Symbolic execution and program testing. Communications of the ACM, 19(7), 385-394. URL: http://www.cs.umd.edu/class/fall2014/cmsc631/papers/king-symbolic-execution.pdf (Дата обращения: 05.04.2021)

[3] Howden, W. E. (1976, June). Experiments with a symbolic evaluation system. In Proceedings of the June 7-10, 1976, national computer conference and exposition (pp. 899-908). (Дата обращения: 05.04.2021)

[4] Levine, John R., et al. Lex & yacc. " O'Reilly Media, Inc.", 1992.

[5] Cohen, J. S. (2003). Computer algebra and symbolic computation: Mathematical methods. CRC Press.

[6] Von Zur Gathen, Joachim, and Jürgen Gerhard. Modern computer algebra. Cambridge university press, 2013.

[7] Joyner, D., Čertík, O., Meurer, A., & Granger, B. E. (2012). Open source computer algebra systems: SymPy. ACM Communications in Computer Algebra, 45(3/4), 225-234.

[8] John Aycock. 2003. A brief history of just-in-time. ACM Comput. Surv. 35, 2 (June 2003), 97–113. DOI:https://doi.org/10.1145/857076.857077

[9] Croce, Louis. "Just in Time Compilation". Columbia University. URL: http://www.cs.columbia.edu/~aho/cs6998/Lectures/14-09-22_Croce_JIT.pdf (Дата обращения: 01.03.2021)

[10] Jens Knoop, Oliver Rüthing, and Bernhard Steffen. 1994. Partial dead code elimination. In Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation (PLDI '94). Association for Computing Machinery, New York, NY, USA, 147–158. DOI:https://doi.org/10.1145/178243.178256

[11] Butts, J. A., & Sohi, G. (2002). Dynamic dead-instruction detection and elimination. ACM SIGPLAN Notices, 37(10), 199-210.

[12] Rigger, M., Grimmer, M., Wimmer, C., Würthinger, T., & Mössenböck, H. (2016, October). Bringing low-level languages to the JVM: Efficient execution of LLVM IR on Truffle. In Proceedings of the 8th International Workshop on Virtual Machines and Intermediate Languages (pp. 6-15).

Automating Creation of Problem-Oriented Program Generators

I.Y. Sesin, R.G. Bolbakov

***Abstract*—Problem-oriented program generation is an approach to improving performance for certain class of GPGPU (General Purpose computing for Graphical Processing Units) programs.**

Creating problem-oriented program generators manually is a time-consuming and difficult task, one that requires the programmer to possess a certain degree of insight in the problem being solved by said program.

This paper covers an approach to automating the creation of problem-oriented program generators. Meta-generator is introduced as a program creating said generators from GPU program's source code. The key steps that meta-generator performs in the course of program analysis are discussed along with technologies most suitable for implementing on each step.

Lastly, a comparison between proposed approach and similar existing means to enhancing software performance was made, outlining their key differences.

***Keywords*— problem-oriented program generator, general-purpose computing for graphical processing units, program optimization.**

References

[1] Boyer, R. S., Elspas, B., & Levitt, K. N. (1975). SELECT—a formal system for testing and debugging programs by symbolic execution. ACM SigPlan Notices, 10(6), 234-245.

[2] King, J. C. (1976). Symbolic execution and program testing. Communications of the ACM, 19(7), 385-394. URL: http://www.cs.umd.edu/class/fall2014/cmsc631/papers/king-symbolic-execution.pdf (Дата обращения: 05.04.2021)

[3] Howden, W. E. (1976, June). Experiments with a symbolic evaluation system. In Proceedings of the June 7-10, 1976, national computer conference and exposition (pp. 899-908). (Дата обращения: 05.04.2021)

[4] Levine, John R., et al. Lex & yacc. " O'Reilly Media, Inc.", 1992.

[5] Cohen, J. S. (2003). Computer algebra and symbolic computation: Mathematical methods. CRC Press.

[6] Von Zur Gathen, Joachim, and Jürgen Gerhard. Modern computer algebra. Cambridge university press, 2013.

[7] Joyner, D., Čertík, O., Meurer, A., & Granger, B. E. (2012). Open source computer algebra systems: SymPy. ACM Communications in Computer Algebra, 45(3/4), 225-234.

[8] John Aycock. 2003. A brief history of just-in-time. ACM Comput. Surv. 35, 2 (June 2003), 97–113. DOI:https://doi.org/10.1145/857076.857077

[9] Croce, Louis. "Just in Time Compilation". Columbia University. URL: http://www.cs.columbia.edu/~aho/cs6998/Lectures/14-09-22_Croce_JIT.pdf (Дата обращения: 01.03.2021)

[10] Jens Knoop, Oliver Rüthing, and Bernhard Steffen. 1994. Partial dead code elimination. In Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation (PLDI '94). Association for Computing Machinery, New York, NY, USA, 147–158. DOI:https://doi.org/10.1145/178243.178256

[11] Butts, J. A., & Sohi, G. (2002). Dynamic dead-instruction detection and elimination. ACM SIGPLAN Notices, 37(10), 199-210.

[12] Rigger, M., Grimmer, M., Wimmer, C., Würthinger, T., & Mössenböck, H. (2016, October). Bringing low-level languages to the JVM: Efficient execution of LLVM IR on Truffle. In Proceedings of the 8th International Workshop on Virtual Machines and Intermediate Languages (pp. 6-15).