Генераторы проблемно-ориентированных программ

Сесин И.Ю., Болбаков Р.Г.

***Аннотация*— Технология GPGPU (англ. general-purpose computing on graphics processing units) позволяет проводить расчёты общего назначения на графических процессорах. Эта технология широко используется для решения задач, требующих больших вычислительных ресурсов, а именно большого количества параллельно выполняющихся расчётов. Программные средства, использующие данную технологию, в процессе развития рано или поздно сталкиваются со следующей проблемой: с ростом объёма функционала программных средств, их быстродействие начинает существенно снижаться. В работе предлагается метод борьбы с данной проблемой, позволяющий увеличить производительность для подобных программ. Этот метод заключается в генерации проблемно-ориентированной программы из исходной программы. Вводится понятие проблемно-ориентированной программы, рассматривается их соотношение с оригинальной программой, описывается предпочтительная степень специализации для проблемно-ориентированных программ. Рассматриваются различные аспекты внедрения генератора проблемно-ориентированных программ. Производится сравнение со сходными технологиями увеличения производительности, рассмотрение их сходства и различий с предложенным методом, а также их применимость в рамках программ, запускаемых на графических процессорах.**

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

I. Введение

Компьютерные программы, использующие технологию GPGPU, как правило, направлены на решение конкретного класса вычислительных задач, требующих проведения большого количества параллельных расчётов [1, 2].

Тем не менее, за редкими исключениями, большинство подобных программ не используют все существующие участки кода в процессе работы над типовыми входными данными. Иначе говоря, такие программы обладают широким функционалом, из которого в типичном сценарии использования задействуется только часть. Но даже если часть программы не используется, за её наличие приходится платить производительностью.

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).

Это наиболее сильно выражается при работе с операторами ветвления – особенно, если подразумевается наличие взаимоисключающих режимов работы программы.

Графические процессоры, в подавляющем большинстве имеют архитектуру SIMD (англ. Single Instruction, Multiple Data, один поток команд, много потоков данных) по Флинну [3], из-за чего в условиях работы программы на GPU (графическом процессоре) происходит исполнение обеих ветвей оператора ветвления и последующее маскирование результатов той ветви, которую не следовало выполнять. Данная особенность архитектуры GPU называется предикацией веток или просто предикацией, и она имеет как преимущества, так и недостатки [4, с. 172].

Соответственно, программы, изобилующие операторами ветвления, могут испытывать существенное падение производительности.

II. Понятие проблемно-ориентированных программ

Предполагается, что язык, использованный для написания GPGPU программы, может быть определён контекстно-свободной грамматикой, что верно для большинства языков программирования. В таком случае программу возможно представить в виде абстрактного синтаксического дерева [5, с. 40].

Для конкретного набора входных параметров и параметров конфигурации будет существовать некоторое поддерево из узлов для выполнения программы.

Программу, соответствующую такому поддереву, можно назвать проблемно-ориентированной в узком смысле. На практике обеспечить полное соответствие данному условию сложно, поэтому следует определить проблемно-ориентированное программы в более широком плане – под проблемно-ориентированной программой будет пониматься специализация существующей программы под ряд входных условий.

Иначе говоря, если обычные программы ориентированы на общее решение некоторой задачи и принимают во внимание все возможные вариации и частные случаи этой задачи, то проблемно-ориентированные программы являются производными от них программами, решающими только частные случаи этой задачи.

Проблемно-ориентированные программы образуются от обычных программ посредством удаления неиспользуемого кода или упрощения существующего кода, то они заведомо будут не менее производительны,

чем оригинальные программы, разумеется, в рамках решения такого частного случая, под который проблемно-ориентированная программа была специализирована.

Стоит обратить особое внимание на тот факт, что проблемно-ориентированные программы могут иметь различную степень специализации. Доведённая до абсолютного логического предела проблемно-ориентированная программа просто возвращает заранее вычисленный ответ для конкретных условий. Однако, очевидным образом, практической пользы такой подход не имеет, в том числе и потому что такое поведение эквивалентно интерпретации и исполнению исходной программы на этапе её обработки.

Интерпретация программы не может быть быстрее исполнения скомпилированного машинного кода программы на аппаратном обеспечении (специально для этого предназначенном) – будь то центральный процессор (CPU) или графический процессор (GPU).

С другой стороны, недостаточная специализация программы может не дать достаточного прироста в производительности, чтобы возместить время, потраченное на предварительную обработку.

Таким образом, целесообразно добиваться некоторой “золотой середины” в процессе специализации программы, при которой прирост производительности стоил бы потраченного на него времени.

При таком подходе основной целью при формировании проблемно-ориентированной программы является устранение заведомо неисполняемых веток (в условиях конкретных входных данных) вместе с порождающими их операторами ветвления из кода программы. Однако, возможны и иные преобразования программы, способствующие увеличению производительности.

Таким образом, под проблемно-ориентированной программой в рамках научной статьи будет пониматься специализированная программа, направленная на решение частного случая более общей задачи. Важно отметить, что проблемно-ориентированная программа не обязательно должна обладать крайней степенью специализации.

Что касается области применения, то, в первую очередь, данный метод направлен на увеличение производительности программ для ряда архитектур, которые не могут извлечь пользу из методов оптимизации, применяемых на CPU, в частности, программ, использующих технологию GPGPU для проведения расчётов на графическом процессоре.

Максимальную пользу из данного подхода могут извлечь программы, обладающие следующими характеристиками:

– программы не являющиеся интерактивными, то есть, принимающие ряд параметров на вход и осуществляющие дальнейшую деятельность без вмешательства пользователя;

– выполняющие множество однотипных расчётов, в частности, имеющие участки кода, исполняющиеся от 106 раз и более;

– программы, имеющий широкий функционал, но не использующие в типичном сценарии его целиком;

– программы, имеющие широкие возможности конфигурации работы, в частности, имеющие опции с взаимоисключающими вариантами.

Примером такой программы может выступать рендерер (специализированная программа, получающая изображение на основе данных о трёхмерной сцене) на базе метода трассировки пути [6, 7]. Он относится к группе методов пре-рендеринга, характеризующейся большим временем работы и отсутствием интерактивности.

Данный тип рендерера требует большого количества однотипных расчётов в процессе решения интегрального уравнения рендеринга [6] методом Монте-Карло, что делает любую оптимизацию этих расчётов кратно эффективной, и, как следствие, очень востребованной.

Подобные рендереры, как правило, включают в себя возможности по работе с различными геометрическими примитивами, типами поверхностей и типами источников света, но далеко не всегда всё предложенное многообразие востребовано.

Так же рендереры этой группы могут предоставлять возможность конфигурации используемого генератора псевдослучайных чисел [8] или двулучевых функций отражательной способности для различных поверхностей или материалов.

Это создаёт плодотворную почву для применения метода повышения производительности такого типа рендереров, но метод может быть применён для любых GPGPU программ, обладающих ранее перечисленным списком характеристик.

III. Генерация проблемно-ориентированных программ

Естественно, “ручное” изготовление проблемно-ориентированных программ, по меньшей мере, не эффективно.

Вместо этого, для более эффективного применения проблемно-ориентированных программ следует автоматизировать процесс их создания из кода исходной программы.

Введём понятие “генератор проблемно-ориентированных программ”. Под генератором проблемно-ориентированных программ будет пониматься программа или программный комплекс, выполняющая функции формирования конечного кода проблемно-ориентированной программы, на основе входных данных, параметров конфигурации и кода программы, решающей общую задачу.

На практике процесс интеграции может различаться в зависимости от предполагаемого уровня специализации программы, а также от применяемых языков программирования – как со стороны генератора, так и со стороны генерируемой программы.

Так, например, не обязательно жёсткое разделение кода генератора и генерируемой программы. Если в качестве языка генерируемой программы выступает C-подобный язык, то возможно применение как прямой

сборки кода программы, так и оказание косвенного влияния посредством вставки в исходный код генерируемой программы директив препроцессора [9], модифицирующих результирующий код в процессе работы препроцессора компилятора.

Зачастую имеет смысл использовать сильные стороны языка программирования генерируемой программы, нежели усложнять саму логику работы программы-генератора.

Если рассматривать генератор как часть большего программного средства, то наиболее перспективным местом его внедрения является участок, ответственный за контроль входных данных, т.е. проверку на их нахождение в допустимом диапазоне, корректном формате и т.д.

Это позволяет избежать дополнительных проходов по всем входным данным, получая возможность вместо этого сразу же оценить и сформировать список неиспользуемых участков кода для дальнейшего использования в процессе формирования проблемно-ориентированной программы.

В рамках исследования применимости данного подхода был проведён следующий эксперимент. Была создана трёхмерная сцена, включающая в себя рассеивающие, отражающие и преломляющие поверхности. На базе этой сцены были созданы три вариации:

1. Рассеивающие и отражающие поверхности.
2. Рассеивающие и преломляющие поверхности.
3. Рассеивающие поверхности.

Этот набор сцен идентичен за исключением параметров поверхностей и представляет собой набор данных для тестирования предложенного подхода, от худшего случая к лучшему случаю. Был создан генератор проблемно-ориентированной программы для OpenCL трассировщика пути, ответственный за усечение веток условий для отражающих и преломляющих поверхностей. Следует заметить, что в рамках чистоты эксперимента было принято решение не исключать вызов генератора псевдо-случайных чисел (ГПСЧ), используемого трассировщиком для стохастического определения поведения луча при контакте с поверхностью. Строго говоря, в случае наличия в сцене только одного типа поверхности этот вызов не имеет смысла и его можно исключить для увеличения производительности, однако т. к. ГПСЧ является функцией с побочным эффектом, исключение вызова даст другую последовательность чисел для последующих вызовов ГПСЧ и создаст иные условия эксперимента для проблемно-ориентированной программы.

Был проведён ряд тестов, сравнивающих время исполнения программы без оптимизаций и проблемно-ориентированной программы. Для каждой из сцен было проведено 20 замеров времени работы.

В первом тесте программам на вход подаются данные о трёхмерной сцене (Рисунок 1), содержащей отражающие, преломляющие и рассеивающие примитивы.

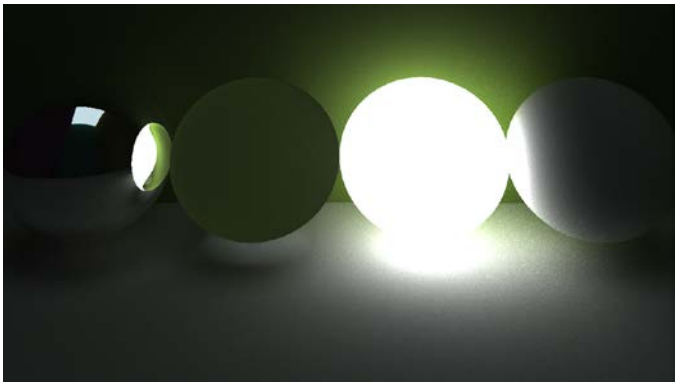

Рис 1. Сцена, использующая отражающие, преломляющие и рассеивающие типы поверхностей.

С точки зрения генератора проблемно-ориентированной программы это худший вариант, где нельзя убрать ни одну из ветвей, и данный подход бесполезен и не даст выигрыша в производительности.

Таб. 1. Замеры времени работы программ в худшем случае.

№	Время работы оптимизированной программы	Время работы неоптимизированной программы
1	67.69383978843689	67.68077969551086
2	67.7481107711792	67.71867775917053
3	67.75866794586182	67.74704170227051
4	67.76502275466919	67.77008438110352
5	67.75839138031006	67.76238822937012
6	68.04152631759644	67.75497221946716
7	67.7351930141449	67.74987649917603
8	67.73225784301758	67.73545145988464
9	67.71065664291382	67.71636772155762
10	67.69042921066284	67.69520139694214
11	67.68016123771667	67.69505834579468
12	67.92466402053833	67.67826437950134
13	67.70048260688782	67.70342421531677
14	67.7513074874878	67.6932487487793
15	67.69020318984985	67.68527340888977
16	67.69049000740051	67.68019676208496
17	67.68030953407288	67.68236207962036
18	67.67529344558716	67.67372870445251
19	67.6802122592926	67.68969888822937
20	67.70993542671204	67.6952133178711

Из представленных данных (Таблица 1) видно, что в худшем случае GPU программа имеет такую же производительность, что и её неоптимизированный оригинал, в полном соответствии с высказанным ранее утверждением.

Во втором тесте программам на вход подавались данные о трёхмерной сцене, включающей только отражающие и рассеивающие поверхности, но не преломляющие типы поверхностей (Рисунок 2).

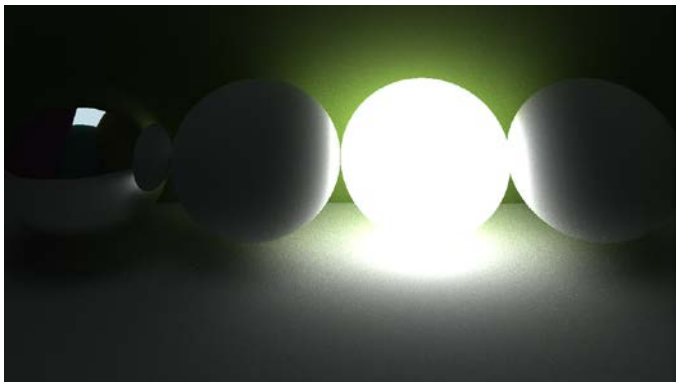

Рис 2. Сцена, использующая отражающие и рассеивающие типы поверхностей.

Таб. 2. Замеры времени работы программ в условиях наличия только отражающих и рассеивающих поверхностей.

№	Время работы оптимизированной программы	Время работы неоптимизированной программы
1	67.12907695770264	67.44728446006775
2	67.13515400886536	67.4463222026825
3	67.15006399154663	67.47111678123474
4	67.16546392440796	67.48140239715576
5	67.1836085319519	67.48531651496887
6	67.18495035171509	67.49968147277832
7	67.20991110801697	67.5191068649292
8	67.21035289764404	67.53541493415833
9	67.2100555896759	67.53375935554504
10	67.20858788490295	67.54740834236145
11	67.22032642364502	67.52974057197571
12	67.22488737106323	67.54736852645874
13	67.23392105102539	67.54783797264099
14	67.23501586914062	67.55026507377625
15	67.2450258731842	67.56052780151367
16	67.2400131225586	67.56713581085205
17	67.24040150642395	67.54514837265015
18	67.23219799995422	67.56395077705383
19	67.25527834892273	67.56786298751831
20	67.24336409568787	67.56792569160461

Согласно указанным данным (Таблица 3) о времени работы программ, оптимизированная программа справлялась в среднем на 0.47% быстрее чем оригинал.

В рамках третьего теста программам на вход подавались данные о трёхмерной сцене, включающей только рассеивающие и преломляющие поверхности, но не отражающие типы поверхностей (Рисунок 3).

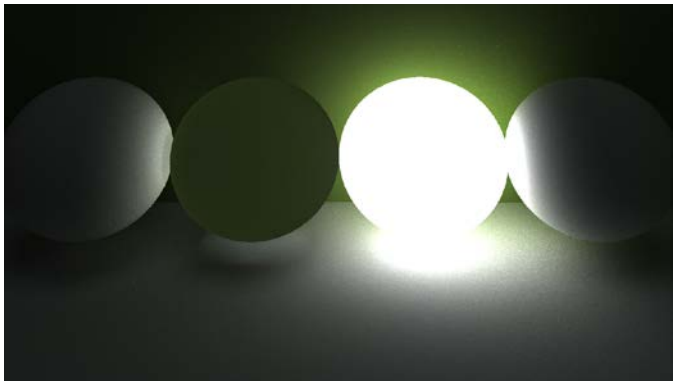

Рис 3. Сцена, использующая преломляющие и рассеивающие типы поверхностей.

Таб. 3. Замеры времени работы программ в условиях наличия только преломляющих и рассеивающих поверхностей.

№	Время работы оптимизированной программы	Время работы неоптимизированной программы
1	68.09544777870178	68.12025117874146
2	68.09525299072266	68.11012721061707
3	68.08532738685608	68.08511638641357
4	68.08010768890381	68.11027336120605
5	68.10957956314087	68.10525846481323
6	68.1055338382721	68.11656975746155
7	68.12319755554199	68.12482047080994
8	68.14990901947021	68.17044258117676
9	68.14908242225647	68.16076302528381
10	68.14505457878113	68.19411063194275
11	68.16586136817932	68.1694540977478
12	68.1799201965332	68.19512844085693
13	68.18312096595764	68.20532464981079
14	68.2203722000122	68.2050154209137
15	68.22965264320374	68.23560452461243
16	68.24225854873657	68.23049116134644
17	68.21346950531006	68.23717021942139
18	68.20378184318542	68.23343682289124
19	68.22351098060608	68.2402036190033
20	68.21530961990356	68.23506665229797

Согласно указанным данным (Таблица 3) о времени работы программ, оптимизированная программа справлялась в среднем на 0.02% быстрее чем оригинал. Результат ещё более скромный, нежели у предыдущего теста.

В рамках четвёртого теста программам на вход подавались данные о трёхмерной сцене, включающей только рассеивающие типы поверхностей (Рисунок 4).

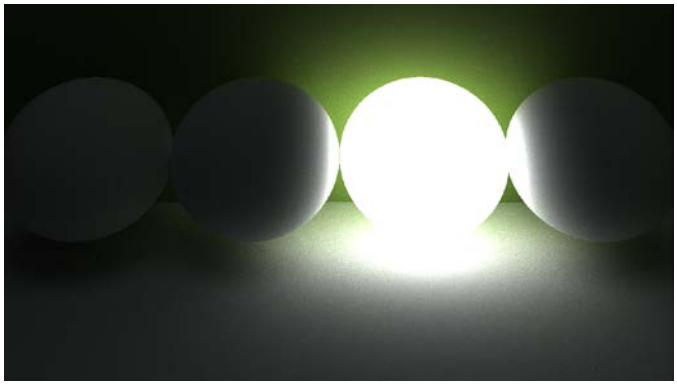

Рис 4. Сцена, использующая только рассеивающие типы поверхностей.

Таб. 4. Замеры времени работы программ в условиях наличия только рассеивающих поверхностей.

№	Время работы оптимизированной программы	Время работы неоптимизированной программы
1	64.463134765625	67.92571592330933
2	64.42191338539124	67.88872289657593
3	64.45050096511841	67.89494442939758
4	64.43252539634705	67.92813491821289
5	64.43793201446533	67.9198796749115
6	64.42062473297119	67.888507604599
7	64.42781448364258	67.88778853416443
8	64.43004536628723	67.90471982955933
9	64.43829321861267	67.88992357254028
10	64.43882441520691	67.87528872489929
11	64.4398546218872	67.88001585006714
12	64.41863489151001	67.86310172080994
13	64.40146160125732	67.83882975578308
14	64.39030313491821	67.82001161575317
15	64.39737343788147	67.79887986183167
16	64.40522170066833	67.79530382156372
17	64.4053225517273	67.80038285255432
18	64.39343333244324	67.79812145233154
19	64.39030313491821	67.79009890556335
20	64.38456153869629	67.79908657073975

Исходя из данных о быстродействии (Таблица 4), оптимизированная программа работала в среднем на 5.07% быстрее, чем неоптимизированная.

Полученный результат интересен в первую очередь тем, что полученный от исключения двух веток выигрыш в производительности больше, чем сумма выигрышей от исключения отдельных веток.

Этот результат можно было улучшить посредством исключения дополнительного вызова к ГПСЧ, на базе которого принимается решение об определённом поведении трассируемого луча – т.е. отражение, рассеивание или преломление. Тем не менее, в рамках чистоты эксперимента данный вызов был оставлен на месте, так как он не входил ни в одну из веток обработки попадания в поверхность.

Но в реальном применении данного метода на программе, использующей метод Монте-Карло исключение «лишнего» вызова к ГПСЧ должно дать прирост производительности при отсутствии отрицательных эффектов. Это связано с тем, что на достаточно больших выборках совершенно не принципиально, какое случайное число было выбрано на каком-либо этапе – пока выбранные числа укладываются в определённое распределение.

IV. Сравнение с существующими программно-аппаратными средствами повышения производительности

На сегодняшний день имеется существенный пласт самых разнообразных подходов к повышению производительности программного обеспечения, включающий как программные, так и аппаратные подходы.

Проведём сравнение предложенного в этой статье метода с наиболее функционально и принципиально схожими методами повышения производительности.

Технология JIT-компиляции [10] (англ. Just-In-Time, иногда переводится как динамическая компиляция) очень схожа с изложенным методом и обладает массой параллелей в своих подходах. JIT-компиляция является технологией повышения производительности, в первую очередь, для интерпретируемых языков. Вместо использования виртуальной машины, программы, использующие JIT-компиляцию преобразуют своё промежуточное представление (англ. intermediate representation) в нативный машинный код непосредственно во время исполнения [11], по мере надобности. Это позволяет программе, написанной на интерпретируемом языке исполняться с такой же скоростью или даже быстрее, чем программы, написанные на компилируемых языках. Вместе с этим JIT-компиляция привносит и ряд проблем, включая проблемы безопасности [12,13] возникающие из-за ошибок в алгоритмах генерации бинарного кода, потребность в отдельном специальном модуле, осуществляющем компиляцию под целевую платформу, а также непредсказуемые моменты простоя программы, пока происходит компиляция нового участка программы.

Для борьбы с этим была предложена альтернативная вариация, иногда называемая Pre-JIT, в которой интерпретируемая программа компилируется в бинарный код заранее, или, в некоторых случаях, сразу после запуска программы. Это устраняет непредсказуемые простои программы во время

динамической компиляции, что сглаживает производительность программы в целом. С другой стороны, в случае, когда предварительная компиляция невозможна, такой подход будет вызывать существенную задержку при запуске программы.

Несложно видеть, что некоторые положения предложенного метода схожи с подходами как JIT-компиляции, так и её Pre-JIT вариации, однако сходство имеет поверхностный характер и исходит преимущественно от опоры на те же общие принципы повышения производительности, а именно: не делать работу, которую можно не делать и вынос затратных операций на предварительный этап.

Основное отличие предложенного метода от JIT-компиляции заключается в использовании входных данных программы для прогнозирования используемых участков кода. Помимо этого, в предлагаемом методе не происходит динамической компиляции по причине ограничения аппаратной платформой записи в страницы памяти исполняемого кода любых данных программой для GPU.

Также существует технология специализации алгоритма во время исполнения, тоже весьма схожая в своих началах с генераторами проблемно-ориентированных программ. В рамках этой технологии для сложных, многократно запускаемых алгоритмов в автоматическом режиме создаются специализации, которые являются упрощёнными вариациями исходного алгоритма, получаемого при определённой комбинации входных данных. Для часто используемых данных могут быть созданы закэшированные результаты вызова специализируемой функции. Данный подход находит применение в системах автоматического доказательства теорем [14].

Предложенный в этой статье метод тоже определённо можно назвать специализацией алгоритма, но ключевым отличием является предварительность сего действия. Так как GPU программы не могут перезаписывать сами свой исполняемый код в силу аппаратных ограничений, специализация во время исполнения попросту невозможна в условиях работы программы на графических процессорах. Таким образом, специализация алгоритма переносится на этап, находящийся перед непосредственным запуском GPU программы, но после получения входных данных для этой программы. Именно тогда становится возможно получить определённые сведения о входных данных, на базе которых возможно исключить ненужные для них части программы.

В то время как специализация алгоритма во время исполнения создаёт вариации упрощённых версий одного алгоритма или функции для часто используемых частных случаев, генераторы проблемно-ориентированных программ специализируют всю программу в целом под конкретные входные данные, исключая участки кода, которые не будут запущены.

V. Заключение

В статье рассмотрена концепция проблемно-ориентированной программы, а также её реализация в виде генератора проблемно-ориентированных программ в качестве способа увеличения производительности программ, использующих технологию GPGPU.

Рассмотрена область применимости подхода, отмечены характеристики программ, которые смогут извлечь наибольшее количество пользы от данного подхода.

Проведено сравнение подхода к генерации проблемно-ориентированных программ с рядом сходных технологий увеличения производительности, отмечены как сходные, так и отличающие характеристики. Обоснована невозможность использования приведённых методов оптимизации в области использования генераторов проблемно-ориентированных программ.

Дальнейшее развитие научной статьи подразумевает исследование возможностей применения подхода для других задач, а также рассмотрение эффективных методов построения средств с использованием генераторов проблемно-ориентированных программ.

Библиография

[1] Mike Houston, General Purpose Computation on Graphics Processors (GPGPU), ATI HD 2000 Series Launch, Tunis, Tunisia (2007) URL: https://graphics.stanford.edu/~mhouston/public_talks/R520-mhouston.pdf (Дата обращения: 30.03.2021)

[2] Kim, H., Vuduc, R., Baghsorkhi, S., Choi, J., & Hwu, W. M. (2012). Performance analysis and tuning for general purpose graphics processing units (GPGPU). Synthesis Lectures on Computer Architecture, 7(2), 1-96. URL: https://core.ac.uk/download/pdf/205695507.pdf (Дата обращения: 30.03.2021)

[3] Flynn M. J. Very high speed computers // Proc IEEE, 1966, 54. — P. 1901—1901.

[4] Joseph A. Fisher, Paolo Faraboschi, Cliff Young. Embedded Computing - A VLIW Approach to Architecture, Compilers, and Tools. 2004.

[5] Aho, Alfred V., Ravi Sethi, and Jeffrey D. Ullman. "Compilers, principles, techniques." Addison wesley 7.8 (1986): 9.

[6] Kajiya, J. T. (1986, August). The rendering equation. In Proceedings of the 13th annual conference on Computer graphics and interactive techniques (pp. 143-150).

[7] Lafortune, E, Mathematical Models and Monte Carlo Algorithms for Physically Based Rendering, (PhD thesis), 1996.

[8] Сесин, И.Ю. Выбор генератора псевдослучайных чисел для использования в рендериге методом трассировки пути / Сесин И.Ю., Нечаев В.В. — INJOIT, v. 5, № 8 (2017)

[9] Stallman, Richard M., and Zachary Weinberg. "The C preprocessor." Free Software Foundation (1987). URL: https://scicomp.ethz.ch/public/manual/gcc/6.3.0/cpp.pdf (Дата обращения: 30.03.2021)

[10] Aycock, J. (June 2003). "A brief history of just-in-time". ACM Computing Surveys. 35 (2): 97–113. CiteSeerX 10.1.1.97.3985. doi:10.1145/857076.857077 URL: https://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.97.3985 (Дата обращения: 10.03.2021)

[11] 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: 10.03.2021)

[12] De Groef, W., Nikiforakis, N., Younan, Y., & Piessens, F. (2010). Jitsec: Just-in-time security for code injection attacks. In Benelux Workshop on Information and System Security (WISSEC 2010), Date: 2010/11/29-2010/11/30, Location: Nijmegen, The Netherlands. URL: https://lirias.kuleuven.be/retrieve/132387 (Дата обращения: 30.03.2021)

[13] Rohlf, C., & Ivnitskiy, Y. (2011). Attacking clientside JIT compilers. Black Hat USA. URL: https://paper.bobylive.com/Meeting_Papers/BlackHat/USA-2011/BH_US_11_RohlfIvnitskiy_Attacking_Client_Side_JIT_Compilers_WP.pdf (Дата обращения: 30.03.2021)

[14] A. Riazanov. Implementing an Efficient Theorem Prover// PhD thesis, The University of Manchester, 2003. URL: https://www.researchgate.net/publication/2906405_Implementing_an_Efficient_Theorem_Prover (Дата обращения: 10.03.2021)

Problem-oriented program generators

Sesin I.Y., Bolbakov R.G.

***Abstract*—GPGPU (General Purpose computing for Graphical Processing Units) technology allows one to harness the computational power of a GPU (Graphical Processing Unit) and apply it to practically any computationally-intensive task benefiting from parallelization.**

Software relying on GPGPU inevitably runs in performance problems as the complexity of the program grows and new functionality is introduced. This paper proposes a method to alleviate that particular issue, improving overall GPU program performance. Proposed method entails the creation of problem-oriented programs from the code of the original program.

A concept of problem-oriented program is introduced, and the key parts differentiating them from original programs are discussed. The preferable degree of program's specialization is covered.

Various aspects of practical application of this approach are presented. Comparison with existing methods for enhancing the software performance is made, presenting the similarities and differences between proposed approach and said methods, as well their general applicability on GPU.

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

REFERENCES

[1] Mike Houston, General Purpose Computation on Graphics Processors (GPGPU), ATI HD 2000 Series Launch, Tunis, Tunisia (2007) URL: https://graphics.stanford.edu/~mhouston/public_talks/R520-mhouston.pdf (Data obrashhenija: 30.03.2021)

[2] Kim, H., Vuduc, R., Baghsorkhi, S., Choi, J., & Hwu, W. M. (2012). Performance analysis and tuning for general purpose graphics processing units (GPGPU). Synthesis Lectures on Computer Architecture, 7(2), 1-96. URL: https://core.ac.uk/download/pdf/205695507.pdf (Data obrashhenija: 30.03.2021)

[3] Flynn M. J. Very high speed computers // Proc IEEE, 1966, 54. — P. 1901—1901.

[4] Joseph A. Fisher, Paolo Faraboschi, Cliff Young. Embedded Computing - A VLIW Approach to Architecture, Compilers, and Tools. 2004.

[5] Aho, Alfred V., Ravi Sethi, and Jeffrey D. Ullman. "Compilers, principles, techniques." Addison wesley 7.8 (1986): 9.

[6] Kajiya, J. T. (1986, August). The rendering equation. In Proceedings of the 13th annual conference on Computer graphics and interactive techniques (pp. 143-150).

[7] Lafortune, E, Mathematical Models and Monte Carlo Algorithms for Physically Based Rendering, (PhD thesis), 1996.

[8] Sesin, I.Ju. Vybor generatora psevdosluchajnyh chisel dlja ispol'zovanija v renderige metodom trassirovki puti / Sesin I.Ju., Nechaev V.V. — INJOIT, v. 5, # 8 (2017)

[9] Stallman, Richard M., and Zachary Weinberg. "The C preprocessor." Free Software Foundation (1987). URL: https://scicomp.ethz.ch/public/manual/gcc/6.3.0/cpp.pdf (Data obrashhenija: 30.03.2021)

[10] Aycock, J. (June 2003). "A brief history of just-in-time". ACM Computing Surveys. 35 (2): 97–113. CiteSeerX 10.1.1.97.3985. doi:10.1145/857076.857077 URL: https://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.97.3985 (Data obrashhenija: 10.03.2021)

[11] Croce, Louis. "Just in Time Compilation". Columbia University. URL: http://www.cs.columbia.edu/~aho/cs6998/Lectures/14-09-22_Croce_JIT.pdf (Data obrashhenija: 10.03.2021)

[12] De Groef, W., Nikiforakis, N., Younan, Y., & Piessens, F. (2010). Jitsec: Just-in-time security for code injection attacks. In Benelux Workshop on Information and System Security (WISSEC 2010), Date: 2010/11/29-2010/11/30, Location: Nijmegen, The Netherlands. URL: https://lirias.kuleuven.be/retrieve/132387 (Data obrashhenija: 30.03.2021)

[13] Rohlf, C., & Ivnitskiy, Y. (2011). Attacking clientside JIT compilers. Black Hat USA. URL: https://paper.bobylive.com/Meeting_Papers/BlackHat/USA-2011/BH_US_11_RohlfIvnitskiy_Attacking_Client_Side_JIT_Compilers_WP.pdf (Data obrashhenija: 30.03.2021)

[14] A. Riazanov. Implementing an Efficient Theorem Prover// PhD thesis, The University of Manchester, 2003. URL: https://www.researchgate.net/publication/2906405_Implementing_an_Efficient_Theorem_Prover (Data obrashhenija: 10.03.2021)