DOI: 10.15514/ISPRAS-2021-33(5)-11

Автоматизация разработки на Vulkan: предметно-ориентированный подход

[1,2] *В.А. Фролов, ORCID: 0000-0001-8829-9884 <vfrolov@graphics.cs.msu.ru>*
[1,2] *В.В. Санжаров, ORCID: 0000-0001-6455-6444 <vadim.sanzharov@graphics.cs.msu.ru>*
[1] *В.А. Галактионов, ORCID: 0000-0001-6460-7539 <vlgal@gin.keldysh.ru>*
[2] *А.С. Щербаков, ORCID: 0000-0002-5360-4479 <alex.shcherbakov@graphics.cs.msu.ru>*

[1] *Институт прикладной математики имени М.В. Келдыша РАН (ИПМ РАН), 125047, Россия, Москва, Миусская площадь 4*
[2] *Московский государственный университет имени М.В. Ломоносова, 119991, Россия, Москва, Ленинские горы, д. 1*

Аннотация. Предлагается новый высокоуровневый подход к разработке приложений на GPU на основе Vulkan API. Цель работы – снижение трудоемкости разработки и отладки приложений, реализующих сложные алгоритмы на GPU при помощи Vulkan. Предлагаемый подход использует технологию генерации кода путем трансляции программы на языке C++ в оптимизированную реализацию на Vulkan, включающую автоматически генерацию шейдеров, привязывание ресурсов и использование механизмов синхронизации. Предлагаемое решение не является технологией программирования общего назначения, а специализируется на конкретных задачах, но обладает при этом расширяемостью, позволяющей адаптировать решение под новые задачи. Для одной входной программы на языке C++, мы можем сгенерировать несколько реализаций для разных случаев (опций транслятора) или разного аппаратного обеспечения. Например, вызов виртуальных функций может быть реализован либо через конструкцию switch в одном вычислительном ядре, либо через сортировку потоков и непрямой вызов в разных ядрах, либо через т. н. callable shaders в Vulkan. Таким образом, предлагаемая технология позволяет обеспечить кроссплатформенность решения за счёт генерации разных реализаций одного и того же алгоритма под разные GPU. В то же время за счёт этого она позволяет обеспечить доступ к специфической аппаратной функциональности, необходимой в приложениях компьютерной графики.

Ключевые слова: генерация кода; программирование GPU; Vulkan

Для цитирования: Фролов В.А., Санжаров В.В, Галактионов В.А., Щербаков А.С. Автоматизация разработки на Vulkan: предметно-ориентированный подход. Труды ИСП РАН, том 33, вып. 5, 2021 г., стр. 181-204. DOI: 10.15514/ISPRAS–2021–33(5)–11

Благодарности: Работа выполнена при поддержке Российского научного фонда (РНФ) № 21-71-00037.

Development in Vulkan: a domain-specific approach

[1,2] *V.A. Frolov, ORCID: 0000-0001-8829-9884 <vfrolov@graphics.cs.msu.ru>*
[1,2] *V.V. Sanzharov, ORCID: 0000-0001-6455-6444 <vadim.sanzharov@graphics.cs.msu.ru>*
[1] *V.A. Galaktionov, ORCID: 0000-0001-6460-7539 <vlgal@gin.keldysh.ru>*
[2] *A.S. Shcherbakov, ORCID: 0000-0002-5360-4479 <alex.shcherbakov@graphics.cs.msu.ru>*

[1] *Keldysh Institute of Applied Mathematics Russian Academy of Science, 4, Miusskaya sq., Moscow, 125047, Russia*
[2] *Lomonosov Moscow State University, GSP-1, Leninskie Gory, Moscow, 119991, Russia*

Abstract. In this paper we propose a high-level approach to developing GPU applications based on the Vulkan API. The purpose of the work is to reduce the complexity of developing and debugging applications that implement complex algorithms on the GPU using Vulkan. The proposed approach uses the technology of code generation by translating a C++ program into an optimized implementation in Vulkan, which includes automatic shader generation, resource binding, and the use of synchronization mechanisms (Vulkan barriers). The proposed solution is not a general-purpose programming technology, but specializes in specific tasks. At the same time, it has extensibility, which allows to adapt the solution to new problems. For single input C++ program, we can generate several implementations for different cases (via translator options) or different hardware. For example, a call to virtual functions can be implemented either through a switch construct in a kernel, or through sorting threads and an indirect dispatching via different kernels, or through the so-called callable shaders in Vulkan. Instead of creating a universal programming technology for building various software systems, we offer an extensible technology that can be customized for a specific class of applications. Unlike, for example, Halide, we do not use a domain-specific language, and the necessary knowledge is extracted from ordinary C++ code. Therefore, we do not extend with any new language constructs or directives and the input source code is assumed to be normal C++ source code (albeit with some restrictions) that can be compiled by any C++ compiler. We use pattern matching to find specific patterns (or patterns) in C++ code and convert them to GPU efficient code using Vulkan. Pattern are expressed through classes, member functions, and the relationship between them. Thus, the proposed technology makes it possible to ensure a cross-platform solution by generating different implementations of the same algorithm for different GPUs. At the same time, due to this, it allows you to provide access to specific hardware functionality required in computer graphics applications. Patterns are divided into architectural and algorithmic. The architectural pattern defines the domain and behavior of the translator as a whole (for example, image processing, ray tracing, neural networks, computational fluid dynamics and etc.). Algorithmic pattern express knowledge of data flow and control and define a narrower class of algorithms that can be efficiently implemented in hardware. Algorithmic patterns can occur within architectural patterns. For example, parallel reduction, compaction (parallel append), sorting, prefix sum, histogram calculation, map-reduce, etc. The proposed generator works on the principle of code morphing. The essence of this approach is that, having a certain class in the program and transformation rules, one can automatically generate another class with the desired properties (for example, the implementation of the algorithm on the GPU). The generated class inherits from the input class and thus has access to all data and functions of the input class. Overriding virtual functions in generated class helps user to carefully connect generated code to the other Vulkan code written by hand. Shaders can be generated in two variants: OpenCL shaders for google “clspv” compiler and GLSL shaders for an arbitrary GLSL compiler. Clspv variant is better for code which intensively uses pointers and the GLSL generator is better if specific HW features are used (like hardware ray tracing acceleration). We have demonstrated our technology on several examples related to image processing and ray tracing on which we get 30-100 times acceleration over multithreaded CPU implementation.

Keywords: Code generation; GPU programming; Vulkan

For citation: Frolov V.A., Sanzharov V.V., Galaktionov V.A, Scherbakov A.S. Development in Vulkan: a domain-specific approach. Trudy ISP RAN/Proc. ISP RAS, vol. 32, issue 5, 2021, pp. 181-204 (in Russian). DOI: 10.15514/ISPRAS–2021–33(5)–11

Acknowledgements. This work is supported by the Russian Science Foundation (RSF) under grant #21-71-00037.

1. Введение

Существующие на сегодняшний день кроссплатформенные технологии программирования GPU, такие как OpenCL и OpenMP, не предоставляют доступ к новейшим возможностям современных GPU, таким как непрямой вызов вычислительных ядер, буферы команд, управляемая синхронизация, аппаратная трассировка лучей, вызов виртуальных функций и др. Технологии, предоставляющие такую возможность, являются либо закрытыми и ориентированными на конкретную платформу (CUDA, OptiX, Metal), либо трудоемкими в использовании (Vulkan, DirectX12).

Фундаментальной проблемой, которая является причиной высокой трудоемкости и стоимости разработки, является отсутствие алгоритмических оптимизаций в системе трансляции/компиляции в современных технологиях программирования GPU: многие эффективные алгоритмы для разных приложений реализуются на GPU схожим или одинаковым образом, но их реализация может зависеть от конкретного GPU. На практике разработчикам приходится поддерживать несколько вариантов одного и того же алгоритма для разных GPU для достижения высокого уровня производительности и совместимости. При этом различия в исходном коде и в производительности могут быть весьма значительными. Например, реализация алгоритмов обработки изображений может быть выполнена с использованием вычислительных шейдеров или с использованием графического конвейера с аппаратным альфа-смешиванием, или с использованием возможностей мобильных GPU с тайловой архитектурой. Суть алгоритма не меняется от того, как конкретно он реализован на GPU, но разработчикам необходимо реализовывать разные версии и работать с низкоуровневыми деталями, которые отличаются для разных графических процессоров. Таким образом, кроссплатформенная разработка ПО с доступом к аппаратному ускорению является фундаментальной проблемой на сегодняшний день.

Основная цель нашей технологии – обеспечение доступа к любым текущим и перспективным аппаратным возможностям современных GPU при сохранении кроссплатформенности и, таким образом, обеспечение технологической независимости разработчика программного обеспечения от производителя оборудования.

2. Существующие решения

Количество существующих технологий программирования GPU и другого специализированного аппаратного обеспечения (например, FPGA) в настоящее время огромное [1]. Это связано с высокой актуальностью задачи и существующими проблемами в области. Рассмотрим эти решения, группируя их по способу работы и их назначению.

2.1 Общие технологии

К общим технологиям отнесем решения, не ориентированные на задачи в конкретной области, а позиционируемые как универсальные средства разработки на GPU.

2.1.1 Технологии на основе директив компилятора (прагм)

Данная группа решений ориентирована на максимальное упрощение разработки и ускорение произвольного кода на GPU настолько, насколько это возможно автоматически. Потенциальными пользователями таких технологий являются учёные и разработчики, не интересующиеся деталями реализации алгоритма на GPU. Сюда можно отнести OpenMP, OpenACC [2], C++ AMP (Miscosoft), Spearmint [3], OP2[4], [5], [6], частично DVM/DVMH [7-9] (система DVMH представляет собой исключение и будет подробнее рассмотрена позже). Основной подход, который обычно используют подобные технологии, состоит в переносе реализации отдельных параллельных участков (обычно циклов) кода на GPU и решение

задачи эффективного копирования данных между CPU и GPU при помощи конвейерной обработки.

У этого подхода есть два ключевых недостатка. Первый – слабый контроль за операциями копирования и распределения данных между CPU и GPU, которые приходится делать автоматически, что становится узким местом и приводит к низкой производительности [10]. В решении этой проблемы интересные результаты достигнуты в работе [6], где была использован программный конвейер, сочетающий в себе стадии выполнения на CPU и GPU. Однако конвейер не всегда может помочь, поскольку низкая производительность может быть вызвана неделимостью некоторых этапов вычислений либо необходимостью передавать большой объём данных. Что более важно, гибридное выполнение приложений (одновременно на CPU и GPU) не всегда нужно. Кроме того, в определённых ситуациях, управление памятью и копирование должно строго контролироваться (например, в приложениях реального времени, для чего существует OpenGL SC и Vulkan SC).

Второй ключевой недостаток данной группы технологий заключается в исключительно слабой поддержке аппаратных возможностей GPU. Например, в OpenACC невозможно напрямую использовать разделяемую память, функции голосования (warp voting functions), отсутствует поддержка текстур (что неприемлемо для высокопроизводительных и энергоэффективных приложений компьютерного зрения и компьютерной графики). Аналогичные проблемы существуют и у других технологий данной группы.

2.1.2 Скелетное программирование

Технологии данной группы нацелены на то, чтобы иметь одну реализацию алгоритма на C++, которая может работать эффективно как на CPU, так и на GPU, либо выполняться гибридно (одновременно на CPU и GPU). Сюда можно отнести такие работы как SkePU[11, 12], SkelCL[13], SYCL [14] и его производные (Intel oneAPI).

Основная идея скелетного программирования состоит в том, чтобы конструировать код программы в виде комбинации определенных базовых примитивов и операций - «скелетов», которые могут быть реализованы эффективно на GPU и, кроме того, могут быть конвейеризованы. Например, такие комбинации, как map-filter и map-reduce.

Основным преимуществом, по сравнению с предыдущей группой решений, является более высокая эффективность в отдельных алгоритмах, которые активно используют базовые скелеты или их комбинации. Недостатком является существенное ухудшение читаемости, сопровождаемости кода и его гибкости, поскольку реализацию алгоритмов приходится подстраивать под использование базовых примитивов и операций [15]. Кроме того, скелетное программирование наследует недостатки предыдущей группы подходов, в частности, в плане поддержки специализированных аппаратных возможностей GPU. Однако, следует отметить, что в скелетном программировании применяются алгоритмические оптимизации на GPU: транслятор обладает знаниями об алгоритме, благодаря чему может сгенерировать эффективную реализацию, используя низкоуровневую технологию программирования как слой нижнего уровня (т.н. back-end).

2.1.3 Вычисления на GPU общего назначения (general purpose computing on GPU, GPGPU)

К этой группе относятся CUDA и OpenCL.

На сегодняшний день CUDA является доминирующей технологией благодаря технологическому лидерству Nvidia. Однако, несмотря на это, даже помимо отсутствия кроссплатформенности, у неё имеется много недостатков: это весьма тяжеловесная технология с огромным количеством функциональностей и версий (11 версий на текущий момент), которые не всегда одинаково хорошо работают на видеокартах разных поколений и архитектур. Перенос программной системы с одной версии CUDA на другую зачастую

сопряжён со значительными затратами времени (например, из-за прекращения поддержки определенных возможностей или существенных изменениях в них). Наконец, значительный недостаток CUDA состоит в наличии глобальных состояний (текстуры, константная память), что может быть удобно для небольших проектов, но становится проблемой при отладке большого проекта.

Технология OpenCL лишена вышеперечисленных недостатков CUDA, но обладает крайне слабой поддержкой существующих аппаратных возможностей современных GPU. Например, т.н. непрямой вызов вычислительных ядер [16], который так и не появился в недавно вышедшем стандарте OpenCL 3.0. Эта функциональность является критической для сложных алгоритмов, в которых поток управления зависит от результатов вычислений на GPU [17].

Также следует отметить, что существует несколько открытых реализаций технологии CUDA через OpenCL или SIMD инструкции CPU [18-20], а также AMD HIP [21]. Эти подходы обладают таким же недостатком, который можно назвать критическим, - отсутствие поддержки аппаратных возможностей в технологии, используемой в качестве слоя нижнего уровня. Этот недостаток приводит к тому, что алгоритм либо не заработает совсем, либо будет обладать низкой производительностью из-за программной эмуляции той или иной отсутствующей функциональности. Например, в OpenCL отсутствует поддержка атомарных операций для типов с плавающей точкой. Программная эмуляция такой функциональности существует [22], но она медленная. Однако в подавляющем большинстве случаев сохранить производительность всё-таки можно, если заменить атомарные операции на параллельную редукцию. Но для этого нужно переписать алгоритм, поэтому трансформация кода должна производиться на более высоком уровне нежели делается в этих работах.

2.1.4 Графические API

Сюда можно отнести такие технологии, как OpenGL4+, DirectX10--DirectX11, DirectX12, Metal, Vulkan.

Несмотря на название, графические программно-аппаратные интерфейсы не являются узкоспециальными и почти всегда предоставляют более широкие возможности, чем API общего назначения. Вычислительные шейдеры являются частью любого из упомянутых интерфейсов и позволяют получить доступ к той же функциональности, что и, например, CUDA, и широко используются в различных приложениях.

За последние пять лет программирование графических процессоров сильно изменилось в связи с появлением новых аппаратных возможностей в таких API, как Vulkan, DirectX12 и Metal. Далее будем рассматривать только Vulkan, потому что это кроссплатформенная технология, в отличие от DirectX12 и Metal.

Наиболее существенным отличием Vulkan от предыдущих интерфейсов программирования на GPU является наличие возможностей, которых нет в других технологиях совсем либо в той же степени. К таким функциям относятся: буферы команд, настраиваемая синхронизация на основе барьеров, управление компоновкой текстур через проходы рендеринга (render passes), управление памятью графического процессора, под-проходы (subpasses) для мобильных графических процессоров, конвейер трассировки лучей (что включает в себя шейдерные таблицы, а это, по сути, есть вызов виртуальной функции) и множество функций, которые представлены, например, в OpenGL, но не включены в технологии программирования общего назначения: непрямой вызов ядер, графический конвейер, меш-шейдеры, сжатие текстур, создание работы на GPU, управляемая страничная память (разреженный буфер/текстура) и некоторые другие.

Благодаря перекладыванию ответственности на разработчика Vulkan-драйвер решает меньше задач. Поэтому разработчики программных систем, умеющие тщательно использовать возможности Vulkan API, могут получить существенные преимущества как по производительности, так и по энергоэффективности создаваемых ими приложений [23].

Кроме того, поддержка Vulkan реализована в огромном числе современных настольных и мобильных устройств [24].

Однако, из широких возможностей для разработчика следует и высокая трудоёмкость создания программ, - до 10 раз [25] больше, чем для CUDA или OpenCL, что является следствием необходимости ручного управления описанными возможностями. Это на практике может затруднять достижение цели повышения производительности, т.к. модифицировать большой объем исходного кода на Vulkan трудно, и разработчики не успевают проверить все возможные оптимизации.

Традиционный способ снижения сложности кода заключается в создании вспомогательных библиотек общего назначения или ориентированных на целевое приложение. Однако, в них пользователю все равно приходится работать с теми же сущностями Vulkan, настраивая и соединяя связи между ними. Альтернативный способ - более тяжелые варианты библиотек или движков, которые инкапсулируют сущности Vulkan и обрабатывают некоторые вещи автоматически [26-28]. Подобные решения, по сути, выполняют работу драйвера, например, OpenGL, что может приводить к неоптимальной реализации и ошибкам, поскольку разработчик в значительной степени повторяет работу производителя этого драйвера.

2.1.5 Различные технологии

В данную группу входят различные уникальные решения.

В работе [29] используются так называемые многомерные гомоморфизмы, – формальное описание задачи на высоком уровне, позволяющее выразить вычисления при помощи параллельных паттернов, которые можно по-разному реализовать на различном оборудовании. Существенное ограничение работы [29] состоит в использование OpenCL в качестве слоя нижнего уровня (OpenCL код генерируется на выходе), что не позволяет использовать многие из новых возможностей современных GPU (в том числе мобильных) в силу ограниченности OpenCL. Описания алгоритмов в [29] производится на крайне ограниченном предметно-ориентированном языке, что также является существенным недостатком. Данная технология рассматривает довольно ограниченный набор оптимизаций параметров алгоритмов, - размеры блоков, порядок циклов и использование различных типов памяти.

В TaskFlow [30] предлагается модель, позволяющая выражать вычисления в виде статических и динамических графов задач для гетерогенных вычислительных систем. Задачи общаются потоками данных между собой по рёбрам графа через очереди, используя схему производитель-потребитель (producer-consumer). В TaskFlow есть возможность построения гетерогенных графов (использующих одновременно CPU и GPU) и их дальнейшего конвейерного выполнения аналогично работе [6]. Недостаток TaskFlow в том, что алгоритм необходимо явно описывать в терминах графов задач, что не очень удобно для разработки и отладки.

PACXX [31, 32] использует идеи скелетного программирования, но более удобен для использования в существующем коде. PACXX напрямую реализует конструкции современного C++ и транслирует использование STL контейнеров в буферы на GPU. Определенный нижний уровень (бэк-энд) выбирается при помощи дополнительных вызовов объекта исполнителя (Executor) и в случае реализации на CPU эти вызовы могут игнорироваться, а в случае реализации на GPU компилятор генерирует код на CUDA или OpenCL. Однако PACXX реализует лишь базовые операции программирования и не имеет доступа ко многим аппаратным возможностям (например, к текстурам) доступным в CUDA и OpenCL.

Chapel [33] ставит целью добиться кроссплатформенности, чтобы единственное описание алгоритма могло эффективно работать как на CPU, так и на GPU. Для этого предлагается новый язык общего назначения, изначально ориентированный на параллельное

программирование. Ключевой недостаток такого подхода в том, что пользователю приходится переносить описание значительной части своего алгоритма на новый язык, что обычно встречает сопротивление в индустрии. Причины такого сопротивления в том, что при использовании совершенно нового языка кроссплатформенность обеспечивается только компилятором этого языка. Из-за чего разработчик попадает в зависимость от единственного в мире разработчика компилятора этого нового языка.

Tiramisu [34] — многогранный компилятор (то есть рассматривающий различные варианты оптимизации одного и того же алгоритма), который специализируется на высокопроизводительной генерации кода для различных платформ. Этот компилятор ориентирован на кластерные вычисления на различных платформах. Таким образом, он поддерживает выполнение одного и того же кода на многопоточных CPU, GPU производства компании Nvidia, кластерах из нескольких вычислителей и FPGA. К достоинствам этой технологии можно отнести переносимость между различным аппаратным обеспечением и высокую скорость работы относительно других схожих подходов. В то же время, данный компилятор имеет ряд недостатков. Во-первых, он поддерживает только специализированный высокоуровневый язык, что затрудняет его внедрение в приложения на других языках и увеличивает временные затраты на перенос алгоритмов, написанных на других языках программирования. Во-вторых, данный компилятор адаптирован для кластерных вычислений и имеет существенные ограничения в плане аппаратного обеспечения.

Компилятор clspv [35] позволяет транслировать ядра OpenCL в промежуточное представление кода для GPU, называемое SPIR-V (используемое в Vulkan для задания шейдерных программ) и, таким образом, может быть использован при разработке на Vulkan. Однако, он поддерживает только вычислительные шейдеры и в настоящий момент официально находится в стадии прототипа (хотя уже является относительно стабильным, т.к. разрабатывается с 2017 года). Предлагаемая нами технология использует clspv как один из возможных компиляторов нижнего уровня.

Компилятор Circle появился в 2018 году, но лишь в 2020 году стал ориентированным на программирование для GPU [36]. В настоящий момент это один их двух компиляторов C++ в мире, поддерживающий графическую функциональность современных GPU (графический конвейер, конвейер трассировки лучей, меш шейдеры). Circle - традиционный компилятор, который сам по себе имеет все недостатки традиционного подхода: если разработчик начинает использовать Circle как основной инструмент (то есть не только для шейдеров, а для всего описания алгоритма), то он становится зависимым от него и сборка на любую платформу уже невозможна без Circle. В сочетании с тем фактом, что Circle является проектом с закрытым исходным кодом, его прямое использование неприемлемо в ряде случаев, т.к. опять же разработчик начинает зависеть от единственного закрытого компилятора.

Второй компилятор шейдеров на C++ для Vulkan появился совсем недавно [37] и ориентирован на использование в составе игрового движка Unreal Engine. Как и Circle, авторы [37] используют механизм атрибутов, появившийся в стандарте C++11 в качестве стандартизированного синтаксиса для расширений языка, для обозначения частей кода, которые должны быть обработаны на GPU. Например, атрибут *[[entry]]* определяет функцию как точку входа, где вызывается GPU код, а функции, которые должны вызываться на GPU обозначаются атрибутом *[[gpu]]*. При этом GPU код инкапсулируется в классы, а механизм виртуальных функций используется для генерации различных вариантов шейдеров транслятором, входящим в решение [37]. Транслятор, реализованный на основе Clang LibTooling [38], конвертирует C++ код в C++ и HLSL код, который уже обрабатывается существующими компиляторами этих языков.

Забегая вперёд, следует сказать, что работа [37] наиболее близка к нашей работе как по сути, так и по используемым средствам обработки исходного кода. Наиболее существенное

отличие работы [37] от нашей в том, что в [37] и “хостовый” (CPU) и “девайсовый” (GPU) код пишется централизованно в одном месте, что делает эту технологию в некоторой степени похожей на CUDA; в нашей технологии, с другой стороны, *по CPU коду генерируется его “зеркальный клон”* на GPU, который пользователю нужно связать со своим хостовым кодом на Vulkan самостоятельно. О преимуществах и недостатках такого подхода мы будем говорить в разделе про предлагаемое решение. Ещё одно существенное отличие в том, что в качестве слоя в [37] нижнего уровня используется существующий инструментарий “Unreal Engine 4” (UE4) и его специфические механизмы, что на наш взгляд в некоторой степени ограничивает кросс-платформенность т.е. UE4 – это очень тяжеловесное решение.

2.2 Специализированные технологии

В данном подразделе рассмотрены решения, ориентированные на решение узкого круга задач в конкретной области.

2.2.1 Библиотеки, предоставляющие реализации конкретных инструментов или алгоритмов

Сюда можно отнести такие известные библиотеки как TBB, Thrust [39], CUBLAS, IPP, NPP, MAGMA, [40], [41], HPX [42] и многие другие. Некоторые из них превращаются в довольно мощные программные решения, имеющие внутри себя специфические компиляторы предметно-ориентированных языков (Domain Specific Language, DSL) для нейросетей (например, PyTorch [43], TensorFlow [44] и [45]). Основной недостаток библиотек в их ограниченных возможностях. Кроме того, многие библиотеки намеренно лишены кроссплатформенности (например, TBB, Thrust, IPP, Intel Embree), поскольку выпускаются определённым производителем аппаратного обеспечения (CPU и/или GPU), в том числе, в целях продвижения собственного оборудования и именно поэтому делаются непереносимыми.

2.2.2 Предметно-ориентированные API

В эту группу входят такие технологии, как Nvidia OptiX [46] и Microsoft DirectML [47]. OptiX — это специализированная технология для ускорения приложений трассировки лучей GPU, которая принимает на вход на C++. OptiX включает две ключевые технологии: (1) аппаратно-ускоренную трассировку лучей и (2) ускорение вызовов виртуальных функций с помощью таблиц шейдеров. Эти 2 функции также доступны в Vulkan и DirectX12. Поскольку программировать в OptiX намного проще (а это единственная технология промышленного уровня, поддерживающая C++ для трассировки лучей на GPU), почти все промышленные системы рендеринга перешли на неё: Octane, VRay, IRay, RedShift, Cycles, Thea и другие. Хотя новейшие графические процессоры AMD имеют аппаратную поддержку трассировки лучей, на самом деле для пользователей нет альтернативы Nvidia. Аналоги [48, 49] существенно отстают от решений на основе OptiX.

2.2.3 Предметно-ориентированные языки и технологии (Domain Specific Language, DSL)

Данная группа решений ориентирована на класс разработчиков в конкретной предметной области, для которых важно добиться максимального упрощения и абстрагирования от деталей реализации алгоритма на GPU при высоком уровне производительности.

Например, языки Darkroom [50] и Halide [51-53] предназначены для создания алгоритмов обработки изображений. Кроссплатформенность в Halide достигается путём реализации большого количества слоёв нижнего уровня, а высокая эффективность за счёт применения оптимизации последовательностей фильтров. Одна из ключевых оптимизаций основывается на идее переупорядочивания операций: если рассматривать применение двух фильтров

последовательно к изображению, тогда зачастую изображение можно разбить на регионы и к каждому региону сразу применить всю цепочку фильтров. За счёт этого при передаче данных от одного фильтра к другому они не выгружаются из кэша (т.к. обрабатывается небольшой регион). Это именно тот тип оптимизаций, которые используются в под-проходах (subpasses) Vulkan и Metal. Хотя в них эта оптимизация реализована в более ограниченном варианте, поскольку необходимо дополнительно решать проблему перекрывающихся регионов изображения. Например, аналогично тому, как это делает система DVMH [9] на основе теневых границ. Следующая важная оптимизация, реализованная в Halide - блочное расположение данных изображения в памяти (что в Vulkan реализуют текстуры). Это дополнительно позволяет повысить эффективность кэша.

Основная мотивация создания нового языка в том, что в традиционных языках программирования недостаточно конструкций для выражения таких вещей как параллелизм и локальность данных. Поэтому помимо упрощения разработки, Halide на самом деле преследует цель повышения контроля над механизмом исполнения алгоритма и, как следствие, производительности: пользователь имеет возможность выбирать различные варианты реализации одной и той же цепочки фильтров и измерять полученную производительность, чтобы остановиться на лучшем варианте. Наконец, существенным преимуществом Halide является возможность генерировать код на C/C++ для целевой платформы на выходе. Таким образом, пользователь не должен подключать к своему проекту компилятор Halide на целевой платформе, а может использовать сгенерированный C++ код (+ шейдеры для GPU) и любую систему сборки.

Преимущества предметно-ориентированных технологий состоит в быстрой, высокоэффективной и иногда (Halide) кроссплатформенной разработке алгоритмов определенного класса. Отчасти это происходит благодаря ограничениям, которые предметно-ориентированные языки программирования привносят в процесс разработки. Это добавляет возможности для оптимизации и генерации высокопроизводительного кода.

Ключевой недостаток предметно-ориентированных языков состоит в том, что алгоритмы и знания, написанные на них, трудно переносить на другие области и трудно состыковать с остальной частью ПО, не использующей предметно-ориентированный язык. Кроме того, чем больше областей в программном решении затрагивается, тем больше различных DSL нужно будет использовать и состыковывать, что существенно усложняет разработку. Например, в современных приложениях компьютерной графики шейдеры по сути являются предметно-ориентированным языком. В настоящее время есть несколько вариантов графического конвейера и конвейера трассировки лучей, где один алгоритм, по сути, распределяется на нескольких программ (от 2 до 5), дополняемых настройкой состояний графического конвейера или конвейера трассировки лучей в C++ коде. Это делает процесс написания программы крайне неочевидным, поскольку нет одного связанного описания алгоритма, а вместо этого имеется множество разрозненных программ и настраиваемые связи между ними в отдельных местах.

DVM — это технология программирования на основе директив, однако же использующая ряд алгоритмических оптимизаций на нижнем уровне. Например, транслятор DVM может прозрачным образом менять расположение массива в памяти “по строкам/по столбцам”, что повышает эффективность выполнения ряда алгоритмов. DVM ориентирована на кластерные вычисления и расчёты на сетках с плотными структурами данных, поэтому можно сказать, что в некотором смысле это предметно-ориентированная технология программирования, хотя и с достаточно широкими “общими” возможностями. Интересной возможностью DVM системы является динамический режим выполнения с автоматическим подбором схемы распределения данных в каждом параллельном регионе.

Если DVM работает с плотными структурами данных, то Taichi [54-56] ориентирована на приложения, работающие с разреженными структурами данных, включая физическое

моделирование, трассировку лучей и нейронные сети. Taichi позволяет пользователям писать высокоуровневый код с использованием предлагаемого языка (интерфейс встроен в C++), как если бы они имели дело с обычными плотными многомерными массивами. Затем компилятор генерирует промежуточное представление, оптимизирует его и возвращает код C++ или CUDA. Taichi также занимается управлением памятью и использует функцию унифицированного доступа к памяти, доступную в CUDA. Возможность работать как с центральными процессорами (с использованием таких методов, как векторизация цикла), так и с графическими процессорами (хотя только с использованием CUDA) является сильной стороной этого решения. Тот факт, что Taichi нацелен на операции с конкретными структурами данных, позволяет ему создавать высоко оптимизированный и эффективный код, но в то же время ограничивает его потенциальные приложения. Будучи DSL, Taichi также имеет те же недостатки, однако тесная интеграция с C++ несколько смягчает их.

В дальнейшем Taichi развивался в нескольких различных направлениях, оставаясь при этом предметно-ориентированной технологией. Во-первых, это дальнейшее развитие физического моделирования и переход на Python-подобный язык (правда со статической типизацией) для описания алгоритмической составляющей моделей. Во-вторых, это дифференцируемое моделирование [55] и сжатие данных [56].

2.3 Резюме по существующим решениям

Технологии общего назначения не поддерживают достаточное количество аппаратных функций, что снижает производительность и энергоэффективность (например, PACXX, OpenACC, DVM, OpenCL, CUDA, TaskFlow). Промышленные API низкого уровня предоставляют такую поддержку, но разработка с их использованием чрезвычайно трудоемка (Vulkan, Metal, DirectX12). Предметно-ориентированные технологии и языки (Halide, Optix) являются хорошим решением как для графических процессоров, так и для других вычислительных систем (FPGA или ASIC). Однако их ключевой недостаток заключается в том, что алгоритмы и знания, реализованные с их помощью, трудно перенести в другие области и трудно интегрировать с остальным программным обеспечением, которое не использует язык, специфичный для предметной области.

Таким образом, вызов к технологии программирования заключается в том, чтобы обеспечить одновременно и доступ к специфическим функциям аппаратуры, и кроссплатформенность. Фундаментальное противоречие, вызванное этим требованием, необходимо решать. Наилучшие результаты в этом направлении, на наш взгляд, достигнуты в [29, 30, 34, 55].

3. Предлагаемое решение

Вместо создания универсальной технологии программирования для построения различных программных систем мы предлагаем расширяемую технологию, которая может быть настроена для определенного класса приложений. В отличие от, например, Halide [51-53], мы не используем предметно-ориентированный язык, а необходимые знания извлекаются из обычного кода на C++. Одним из основных преимуществ предлагаемой технологии является то, что исходный код не расширяется никакими новыми языковыми конструкциями или директивами. Предполагается, что исходный код является обычным исходным кодом на языке C++ (хотя и с некоторыми ограничениями), который может быть скомпилирован любым компилятором C++. Это значительно увеличивает возможность кроссплатформенной разработки с использованием предлагаемой технологии.

Мы используем сопоставление с образцом (pattern matching), чтобы находить *определённые шаблоны (или паттерны)* в C++ коде и преобразовывать их в эффективный код для GPU с использованием Vulkan. Шаблоны-паттерны выражаются через классы C++, функции-члены и отношения между ними.

Важно отметить разницу между предлагаемой технологией и большинством существующих технологий параллельного программирования, таких как DVM, CUDA, Taichi, Halide и другие: они расширяют язык программирования новыми конструкциями или предлагают новые языки, в которых параллельные конструкции отображаются в некоторую эффективную реализацию на определённом аппаратном обеспечении. Предлагаемый подход противоположен. Во-первых, мы не добавляем расширения для языка программирования, а, наоборот, вводим ограничения на использование его возможностей. Во-вторых, шаблоны-паттерны не определяют аппаратных функций, а выражают алгоритмические и архитектурные знания. Таким образом, за аппаратные функции отвечает транслятор, а не разработчик.

3.1 Шаблоны

Шаблоны (или паттерны) делятся на архитектурные и алгоритмические. Архитектурный шаблон определяет предметную область и поведение транслятора в целом (например, обработку изображений, трассировку лучей, нейронные сети, физическое моделирование и т. д.). Алгоритмические шаблоны выражают знания о потоках данных и управления и определяют более узкий класс алгоритмов, которые могут иметь эффективную аппаратную реализацию. Алгоритмические шаблоны могут встречаться внутри архитектурных шаблонов. Например, параллельная редукция, уплотнение (параллельное добавление элементов в буфер), сортировка, префиксная сумма, вычисление гистограммы, map-reduce и др.

Рассмотрим шаблоны, реализованные в текущей версии транслятора:

- Архитектурный шаблон *обработки изображений*. Этот шаблон предоставляет базовые возможности для т.н. GPGPU. Исходный код выглядит как обычный код на C++ с циклами в стиле OpenMP, за исключением того, что мы не используем директивы (прагмы). Вместо этого шаблон предполагает, что существует некий класс с управляющими функциями и функциями-ядрами (листинг 1). Функции-ядра содержат код внутри циклов, который предполагается переносить на GPU практически один в один. Управляющие функции — это функции, которые вызывают функции-ядра и, таким образом, определяют логику их запуска и привязку ресурсов (входных и выходных буферов и текстур). Функции-ядра имеют специальные префиксы (kernel1D_, kernel2D_, kernel3D_ и просто kernel_ в шаблоне трассировки лучей). Мы могли бы использовать для этого директивы компилятора или синтаксис [[kernel]] по аналогии с работами [36, 37]. Это хорошее решение, хотя на наш взгляд разница умозрительная. Используя префиксы, мы можем остаться в рамках стандарта C++ 98 и не вводить никаких расширения в язык, что было одной из наших целей.
- Архитектурный шаблон для *трассировки лучей*. Цель этого шаблона - предоставить доступ к трассировке лучей с аппаратным ускорением и эффективно вызывать виртуальные функции на графическом процессоре. Поэтому его можно рассматривать как кроссплатформенный аналог OptiX. Существенная разница между этим шаблоном и предыдущим заключается в том, что в шаблоне обработки изображений циклы предполагается размещать внутри функций ядра, в то время как в шаблоне трассировки лучей предполагается, что они находятся вне управляющей функции (и, следовательно, вне функций-ядер). Следовательно, шаблон трассировки лучей удобен, если для каждого потока или каждого обрабатываемого элемента данных используется сложный код, но цикл обработки данных прост. А шаблон обработки изображений удобен, когда количество потоков (обрабатываемых элементов данных) может изменяться во время работы алгоритма. Например, если нужно изменить размер изображения, можно обработать уменьшенную версию изображения, а затем снова масштабировать ее.
- Алгоритмический шаблон *параллельной редукции*. Для этого шаблона предлагаемое решение обнаруживает доступ к переменным данных класса (члены входного класса,

листинг 1) и генерирует код для параллельной редукции на GPU в соответствии с типом переменной и операций, участвующих в редукции (листинги 2 и 3).

- Алгоритмический шаблон *для параллельного добавления элементов в буфер* и связанный с ним *шаблон непрямого вызова ядра (indirect dispatch)*. Например, есть функция-член, которая обрабатывает входные данные и добавляет некоторые выходные данные в вектор через *«std::vector.push_back(...)»*. После чего выбранные данные обрабатываются в другой функции, при этом счетчик цикла зависит от *«vector.size()»*, и поэтому фактическое количество потоков на GPU должно быть другим: оно будет известно только после завершения первого ядра; поэтому здесь необходимо использовать непрямой вызов ядра.

В Листинге 1 показан пример входного кода, а в Листингах 2 и 3 – упрощенные примеры выходных данных.

```
1. class Numbers {
2. public:
3.   void CalcArraySumm(const int* a_data, uint a_dataSize) {
4.     kernel1D_ArraySumm(a_data, a_dataSize);
5.   }
6.   void kernel1D_ArraySumm(const int* a_data, size_t a_dataSize) {
7.     m_summ = 0;
8.     for(uint i=0; i<a_dataSize; i++) {
9.       int number = a_data[i];
10.      if(number > 0)
11.        m_summ += number;
12.    }
13.  }
14.  int m_summ;
15.};
```

Листинг 1. Пример исходного кода для архитектурного шаблона обработки изображений: вычисление суммы всех положительных чисел в массиве
Listing 1. Sample source code for an architectural imaging pattern: Calculate the sum of all positive numbers in an array

В примере Листинга 1 функция ядра и ее размеры (1D, 2D или 3D) извлекаются путем анализа имени функции (kernel1D_ArraySumm, строки 6-12). Управляющая функция извлекается путем анализа ее кода: если из этой функции вызывается хотя бы одна из функций-ядер, то это управляющая функция (CalcArraySumm, строки 3-5). Любые члены данных класса, к которым имеют доступ функции-ядра, помещаются в единственный «буфер данных класса» (m_summ, строка 14). Доступ для таких переменных анализируется дополнительно. Если в некотором ядре одна и та же переменная записывается на разных итерациях цикла определённым образом (строка 11), то генерируется параллельный код редукции в конце шейдера для этой переменной (листинг 3).

```
1. class Numbers_Generated : public Numbers {
2. public:
3.
   virtual void SetInOutFor CalcArraySumm(VkBuffer a_dataBuffer) { ... }
4.   virtual void CalcArraySummCmd(VkCommandBuffer a_commandBuffer,
                                   uint a_dataSize) {
5.     m_currCmdBuffer = a_commandBuffer;
6.     vkCmdBindDescriptorSets(a_commandBuffer, ... , ArraySummLayout,
                               &allGeneratedDS[0], ... );
7.     ArraySummCmd(a_dataSize);
8.   }
9. protected:
10.  virtual void ArraySummCmd(size_t a_dataSize) {
11.    ...
```

```
12.     vkCmdBindPipeline   (m_currCmdBuffer, ...
   , ArraySummInitPipeline);
13.     vkCmdDispatch       (m_currCmdBuffer, 1, 1, 1);
14.     vkCmdPipelineBarrier(m_currCmdBuffer, ... );
15.     vkCmdBindPipeline   (m_currCmdBuffer, ... , ArraySummPipeline);
16.     vkCmdDispatch       (m_currCmdBuffer, a_dataSize/256, 1, 1);
17.     vkCmdPipelineBarrier(m_currCmdBuffer, ... );
18.   }
19. ...
20. }
```

Листинг 2. Получив некоторый класс в качестве входных данных, предлагаемое решение генерирует интерфейс и реализацию для GPU версии алгоритмов, реализованных в управляющих управления
Listing 2. Having received a certain class as input, the proposed solution generates an interface and implementation for the GPU version of the algorithms implemented in controllers

Из-за особенностей Vulkan необходимо сгенерировать две функции для каждой функции управления вводом. Для CalcArraySumm генерируются функции: SetInOutFor_CalcArraySumm и CalcArraySummCmd. Первая создает набор дескрипторов для входного буфера (a_dataBuffer в примере) и сохраняет его в allGeneratedDS[0]. Заметим, что параметр входного указателя a_data был удален. В сгенерированном коде параметры указатели в управляющих функциях и ядрах не используются, т.к. все данные теперь находятся на графическом процессоре, и доступ к ним осуществляется через наборы дескрипторов в шейдерах. В этом примере были сгенерированы два разных шейдера: ArraySummInitPipeline, который выполняет инициализацию цикла (обнуление суммы), а второй - ArraySummPipeline, который выполняет тело цикла (листинг 3).

```
1.  __kernel void kernel1D_ArraySumm(__global const int* a_data,
                                     __global NumbersData* ubo, ...) {
2.    __local int m_summShared[256*1*1];
3.    ...
4.    int number = a_data[i];
5.    if(number > 0)
6.      m_summShared[localId] += number;
7.    ...
8.    barrier(CLK_LOCAL_MEM_FENCE);
9.    m_summShared[localId] += m_summShared[localId + 128];
10.   barrier(CLK_LOCAL_MEM_FENCE);
11.   m_summShared[localId] += m_summShared[localId + 64];
12.   barrier(CLK_LOCAL_MEM_FENCE);
13.   m_summShared[localId] += m_summShared[localId + 32];
14.   m_summShared[localId] += m_summShared[localId + 16];
15.   m_summShared[localId] += m_summShared[localId + 8];
16.   m_summShared[localId] += m_summShared[localId + 4];
17.   m_summShared[localId] += m_summShared[localId + 1];
18.   if(localId == 0)
19.     atomic_add(&ubo->m_summ, m_summShared[0]);
20. }
```

Листинг 3. Пример сгенерированного шейдера для компилятора clspv
Listing 3. An example of a generated shader for the clspv compiler

В примере Листинга 3 исходное тело цикла преобразуется в строки 4–6. Видно, что доступ к m_summ был переписан на m_summShared [localId], который в дальнейшем используется при параллельной редукции в конце шейдера. В строках 13-17 реализован оптимизированный вариант параллельной редукции для графического процессора Nvidia, предполагающий, что размер подгруппы равен 32 потокам. Этот размер меняется в зависимости от входных параметров нашего транслятора. Например, можно отключить оптимизацию или задать меньший размер подгруппы (8 для мобильных графических процессоров) или использовать subgroupAdd вместо 13-17 строк (доступно только в GLSL).

3.2 Генерация кода

Предлагаемый генератор работает по принципу морфинга кода [57]. Суть этого подхода в том, что, имея определенный класс в программе и правила преобразования, можно автоматически сгенерировать другой класс с желаемыми свойствами (например, реализацию алгоритма на GPU). Правила преобразования определяются указанными шаблонами, в рамках которых осуществляется обработка и трансляция текущего кода. Сгенерированный класс наследуется от входного класса и, таким образом, имеет доступ ко всем данным и функциям входного класса.

Входной исходный код обрабатывается с помощью clang и libtooling [38]. Практически все задачи в предлагаемом трансляторе выполняются за два прохода. На первом проходе происходит обнаружение паттернов и их частей с помощью libtooling: вложенные циклы внутри функций ядра, редукция, доступ к членам данных класса и т. д. На втором проходе происходит переписывание исходного кода с использованием clang AST Visitors. Окончательный исходный код создается с помощью генерации текста по шаблону [58]. Таким образом, предлагаемое решение реализуется исключительно за счет source-to-source преобразования, т.е. в отличие, например, от Circle, который работает с LLVM-представлением кода. Хотя этот подход имеет определенные ограничения (нельзя изменить входной язык программирования, что возможно при использовании LLVM), он также имеет значительные преимущества:

I. Сгенерированный исходный код для обоих типов шейдеров (OpenCL C для clspv [35] или GLSL) и C++ код с вызовами Vulkan выглядят как обычный код, написанный вручную. Его можно отлаживать, изменять или комбинировать с другим кодом любым способом. Таким образом, в отличие от многих существующих технологий программирования легко отличить ошибки нашего транслятора от ошибок пользователя. Это проблема, например, для OptiX, Taichi или Circle, потому что непонятно, что технология программирования на самом деле делает с входным кодом.

II. Возможность генерировать исходный код для шейдеров дает значительную гибкость, потому что можно легко добавлять поддержку различного аппаратного обеспечения. Ранняя версия нашего инструмента использовала только clspv [35] для шейдеров. Однако мы быстро поняли, что возможностей clspv недостаточно для аппаратного ускорения трассировки лучей, массивов текстур (dynamic descriptor indexing), виртуальных функций и многого другого. Конечно, можно добавить такую поддержку в clspv, чтобы получить желаемые функции в SPIR-V из исходного кода шейдера OpenCL C, но этот путь дорогой и трудный, поскольку работа с исходным кодом SPIR-V и clspv требует специальных знаний и значительных усилий. В то же время добавить поддержку новых возможностей аппаратного обеспечения непосредственно в сгенерированном коде на GLSL относительно легко.

3.2.1 Передача данных CPU <=> GPU

Как упоминалось в обзоре, многие существующие решения автоматически решают проблему копирования данных. В большинстве случаев для программного обеспечения, использующего Vulkan, это нельзя делать по многим причинам. В основном, потому что копирование в Vulkan – нетривиальная операция, которая зависит от сценария использования алгоритма и возможностей аппаратуры. В предлагаемом подходе выполняется генерация кода для выполнения алгоритмов на графическом процессоре и для выполнения копирования, а затем пользователю предоставляется возможность самостоятельно вызывать этот код. Созданная функция с именем «UpdateAll» выполняет эту задачу. Если пользователю нужны данные обратно на CPU из некоторых внутренних данных сгенерированного класса, то можно создать новый класс, который наследуется от сгенерированного. В этом классе может быть реализован любой дополнительный алгоритм или функция копирования.

Предлагаемое решение реализует весь сгенерированный алгоритм на графическом процессоре, поэтому, как правило, все сгенерированные переменные и буферы находятся на графическом процессоре. Однако, поскольку сгенерированный класс наследуется от исходного, он также содержит все исходные переменные и векторы на CPU под их собственными именами. Пользователь либо предоставляет собственную реализацию копирования (через реализацию интерфейса т.н. “ICopyEngine” с несколькими виртуальными функциями типа Update/Read), либо использует предоставляемую нами реализацию того же интерфейса. Таким же образом пользователь может вручную удалить ненужные данные на CPU после вызова метода UpdateAll в классе, наследуемом от сгенерированного, а при необходимости возможно детальное управление копированием.

3.2.2 Сопоставление с ближайшими аналогами

Таким образом, ещё раз отметим, что наш инструмент по CPU коду генерирует его “зеркальный клон” на GPU, который пользователю нужно связать со своим кодом на Vulkan самостоятельно. Это более трудоёмко, чем подход CUDA или работы [37], но имеет и свои преимущества, поскольку не зависит от тяжеловесных инструментариев вроде UE4 и не порождает зависимости от себя самого в используемых проектах: сгенерированный код всегда можно дописать или переписать вручную и, таким образом, постепенно избавиться от нашего решения, если оно пользователя не устраивает. Кроме того, такой подход упрощает отладку и поиск ошибок: мы всегда можем понять в каком именно месте произошла ошибка – на стороне пользователя или в генераторе, т.к. выход представляет собой обыкновенный читаемый и отлаживаемый код на Vulkan и C++.

4. Экспериментальная оценка

Мы протестировали наш подход на нескольких приложениях, для которых мы генерировали различные реализации (GPU v1 – v3), используя разные опции транслятора. Эти опции отвечают за выбор различных реализаций одного и того же алгоритма на GPU. Поэтому в некоторых случаях производительность сильно отличается.

Рис. 1. Демонстрация применения разработанной технологии
Fig. 1. Demonstration of the application of the developed technology

Табл. 1 и 2 демонстрируют полученные результаты. Рис. 1 демонстрирует приложения, на которых мы тестировались.

В среднем на GPU мы достигали ускорения в 30-100 раз относительно многопоточной версии на CPU (14 ядер, OpenMP), что является адекватным показателем для рассматриваемых задач. Однако в данной работе мы не ставили своей целью обогнать какую-либо реализацию или технологию программирования. Важнее другое: производительность была разной для разных

сгенерированных реализаций и разных GPU. Это означает, что на практике для достижения желаемого уровня производительности разработчику приходится поддерживать все возможные версии реализаций на Vulkan вручную. В нашем трансляторе выбор между этими реализациями осуществляется автоматически простым переключением флагов.

Редукция (№1 – №3). В этих примерах мы демонстрируем возможность обнаруживать редукцию и генерировать параллельную редукцию на GPU с различными реализациями: GPU v1 представляет собой полностью кроссплатформенную реализацию, GPU v2 знает размер подгруппы (warp), а GPU v3 не только знает размер подгруппы, но и использует т.н. операции внутри подгруппы.

Задача N тел (№4). Это классическая проблема GPGPU с квадратичной сложностью, реализованная в рамках шаблона обработки изображений. В данном случае мы не применяли никаких специальных алгоритмических оптимизаций.

Тонирующий оператор **(*Bloom, №5*)** и денойзинг **(*Non Local Means, №6*).** В этих примерах мы демонстрируем возможность транслятора изменять используемую структуру данных для хранения изображений в шаблоне обработке изображений с буфера (GPU v1) на текстуру (GPU v2) и текстуру с половинной точностью (GPU v3). Интересно отметить что на использованной нами видеокарте Nvidia (табл. 1) реализация через буфер работает быстрее, а на использованной нами видеокарте от AMD победила реализация через текстуру (табл. 3).

Монте-Карло трассировщик путей (№7) демонстрирует перенос алгоритма трассировки путей на GPU с использованием аппаратного ускорения трассировки лучей несколькими способами: GPU v1 – реализация всего алгоритма в нескольких отдельных ядрах (при этом обе версии вызова виртуальной функции через конструкцию switch и через indirect dispatch имели приблизительно одинаковую производительность). GPU v2 – реализацию всего алгоритма с использованием т.н. “callable shaders” и GPU v3 – реализацию всего алгоритма в 1 ядре с преобразованием вызова виртуальной функции в конструкцию switch. В этом примере интересно отметить тот факт, что более простой способ через конструкцию switch был в 3 раза быстрее специально предназначенной для этого функциональности “callable shaders”. Дальнейший анализ (разд. 4.1) показал, что причина этого заключается в том, что реализованный нами код обработки материалов относительно несложный, и как следствие, вычислительные блоки GPU недогружены. В результате варианты GPU v1 и GPU v2, более интенсивно работающие с памятью, оказались существенно медленнее. Однако ситуация может поменяться в дальнейшем при расширении функциональности материалов и при увеличении количества типов материалов. Поэтому на наш взгляд нельзя сказать наперёд, какой именно из этих вариантов лучше. Предложенное решение позволяет разработчику поддерживать и тестировать несколько вариантов с низкой трудоёмкостью, переключая лишь флаги транслятора.

Табл. 1. Время выполнения в миллисекундах для различных (v1–v3) реализаций
Table. 1. Execution time in milliseconds for different (v1-v3) implementations

App/ Impl	CPU (1 core) Input source	CPU (14 cores) OpenMP	GPU v1 Ours	GPU v2 Ours	GPU v3 Ours
(#1) Int Arr. Σ	1.263 мс	0.271 мс	0.095 мс	0.089 мс	**0.084 мс**
(#2) FloatArr. Σ	1.420 мс	0.342 мс	0.104 мс	**0.096 мс**	**0.096 мс**
(#3) Sph. Eval.	39.73 мс	2.931 мс	0.399 мс	0.364 мс	**0.320 мс**
(#4) NBody	250400 мс	11920 мс	**118.0 мс**	-	-
(#5) Bloom	711.8 мс	52.74 мс	**0.733 мс**	1.420 мс	0.841 мс
(#6) NLM	88440 мс	6851 мс	422.0 мс	571.1 мс	**351.0 мс**
(#7) Path Trace	188.4 мс	14.928 мс	4.790 мс	1.360 мс	**0.460 мс**

Первая строка табл. 1 – задача вычисления суммы 1 миллиона целых чисел, изображённая на Листингах 1–3. Вторая строка – такая же сумма, но уже чисел с плавающей точкой, для которой транслятор генерирует дополнительные проходы после основного ядра (т.к. атомарных операций с плавающей точкой нет). Третья строчка — редукция для вычисления

сферических гармоник по изображениям. Эта задача схожа вычислению гистограммы. Четвёртая строчка – задача N тел, 5 и 6 строчки — упомянутые ранее фильтры обработки изображений. Последняя строчка — трассировка путей для изображения в разрешении 512x512 пикселей (256 тысяч путей). Для реализации трассировки пути на CPU мы использовали трассировку лучей из библиотеки Embree. Используемый CPU: Intel Core i9 10940X, графический процессор – Nvidia RTX 2080. Для трассировки пути было визуализировано изображение размером 512x512 (т. е. 256 тысяч путей). Измерения времени проводились при помощи Nvidia Nsight.

Табл. 2. Количество строк кода
Table 2. Number of lines of code

App/Lines	C++ (input source)	Vulkan (generated)	Vulkan (compact)	Vulkan (shaders only)
(#1) Int Arr. Σ	80	640	500	195
(#2) Float Arr. Σ	80	780	650	285
(#3) Sph. Eval.	120	1280	1000	445
(#4) NBody	115	850	700	140
(#5) Bloom	300	1460	1050	250
(#6) NLM	330	1290	1000	250
(#7) Path Trace	800	4500	3200	1470
Количество строк кода во входном примере (первый столбец), количество сгенерированных строк на Vulkan (второй столбец), оцененное некоторым образом, количество строк при ручном кодировании задачи (третий столбец) и количество строк кода в шейдерах (последний столбец).				

Табл. 3. Время выполнения в миллисекундах для различных (v1–v3) реализаций на двух видеокартах: AMD Radeon Vega 10 / Kirin 980 (мобильный GPU)
Table 3. Execution time in milliseconds for different (v1 – v3) implementations on two video cards: AMD Radeon Vega 10 / Kirin 980 (mobile GPU)

App/ Impl	GPU v1 Ours	GPU v2 Ours	GPU v3 Ours
(#1) Int Arr. Σ	0.24 мс / 1.51 мс	0.22 мс / 1.51 мс	**0.21 мс / -**
(#2) FloatArr. Σ	0.26 мс / 1.62 мс	0.26 мс / 1.61 мс	0.26 мс / -
(#3) Sph. Eval.	23 мс / 24 мс	22 мс / **22 мс**	**17 мс / -**
(#4) NBody	1260 мс / 768 мс	-	-
(#5) Bloom	20 мс / 273 мс	16 мс / 280 мс	**11 мс / 196 мс**
(#6) NLM	4510 мс / -	6000 мс / -	**2580 мс** / -
(#7) Path Trace	63.5 мс / 80 мс	-	**39.7 мс** / **52 мс**

Для примера #7 трассировка лучей была реализована программно, а вызываемые шейдеры на Vega 10 и Kirin 980 не поддерживаются. Измерения времени проводились при помощи таймера std::chrono. Фильтр Non Local Means не заработал на Kirin 980 из-за большого времени ожидания вычислительного шейдера.

4.1 Детали реализации вызова виртуальной функции на GPU

Вызов виртуальной функции является алгоритмическим шаблоном для нашего транслятора. Проблема, которую мы хотим решить – это эффективная реализация ветвлений на GPU для достаточно тяжёлого кода, когда потери на т.н. branch divergence становятся существенными.

В рассматриваемом примере трассировке путей реализованы четыре типа материалов: Ламберт, модель GGX [59] (чайник и зеркальные объекты справа от него), Смесь Ламберта и GGX по маске, вычисляемой при помощи так называемого шума Вороново [60] (цилиндр с розовыми прожилками), и самосветящийся материал с аналогом шумом Перлина [61] (источник света). Общий объём передаваемых в виртуальную функцию аргументов – 120 байт на один поток.

На примере этого подраздела мы демонстрируем анализ производительности, который можем производить для различных реализаций. Итак, мы рассмотрели три возможных способа реализации вызова виртуальной функции.

1) Сгенерировать конструкцию switch в коде вычислительного ядра. Этот способ наиболее тривиальный и универсальный.
2) Отсортировать потоки при помощи параллельного подсчёта гистограммы (где каждая ячейка гистограммы отвечает за количество объектов данного типа), после чего использовать последовательность непрямых вызовов (indirect dispatch).
3) Использовать функциональность Vulkan API “callable shaders”.

Для того чтобы вызов виртуальной функции можно было реализовать эффективно на GPU, мы вводим ряд ограничений и рассматриваем только специальный случай.

1) Один уровень вложенности. Большее количество уровней можно было бы поддержать (например, если всегда преобразовывать на последующих уровнях вызовы виртуальных функций в конструкцию switch), однако мы не исследовали это в нашей работе.
2) Объекты (относительно которых ведётся диспетчеризация) можно создавать, только выделяя их из пула. В настоящий момент предполагается, что пользователь создаёт объекты на CPU после чего они один раз загружаются на GPU путём перемещения всего пула на GPU в сгенерированной функции UpdateAll. Здесь необходимо сделать два важных уточнения:
   a. все данные для каждого объекта должны находиться в линейном и непрерывном участке памяти внутри пула;
   b. при необходимости память в пуле можно было бы выделать и на GPU (т.е. создавать новые объекты на GPU) при помощи параллельной префиксной суммы, однако мы не реализовывали этот сценарий.
3) Объекты нельзя удалять по отдельности. Можно лишь очистить пул целиком.
4) В текущей реализации может быть только функция-ядро целиком. При этом указатель на объект можно получить лишь специальным образом из другой функции ядра, которая формирует его по так называемому идентификатору объекта (Листинг 4).

```
TestClass::ControlFunc(uint tid, float4* out_color) {
  IMaterial* pMaterial = kernel_XXX(tid, ...);
  pMaterial->kernel_YYY_or_ZZZ(tid, ...);
}
```

Листинг 4. Пример того, как выглядит вызов виртуальной функции в пользовательском коде
Listing 4. An example of what a virtual function call looks like in custom code

На Листинге 4 kernel_XXX – это специальная функция-ядро-конструктор, которая внутри обязуется вызвать другую функцию “MakeObjPtr”, на которую уже реагирует наш транслятор. Функция kernel_YYY_or_ZZZ – виртуальная.

Мы измерили время выполнения итогового кода трассировки путей с различными реализациями вызова виртуальной функции и отдельно время на вызов виртуальной функции в тех случаях, когда это можно было сделать, используя Nvidia Nsight (табл. 4).

Табл. 3. Время выполнения
Table 4. Time of execution

Вариант реализации	Время на весь алгоритм	Время на вызов вирт.функ.	Накладные расходы
v1, 1 ядро, switch	**0.46 мс**	—	—
v2, множество ядер, switch	1.52 мс	0.22 мс	—
v2, множество ядер, indirect dispatch	1.72 мс	0.21 мс	0.20 мс
v3, callable shaders	1.36 мс	—	—
Для разрешения изображения 512x512. Мы использовали небольшой размер изображения, чтобы снизить нагрузку на память и улучшить работу L2 кэша в данном эксперименте, а итоговое изображение мы обходили блоками. При увеличении размера блока нагрузка на память в табл. 4 растёт.			

Табл. 5. Нагрузка на различные блоки GPU
Table 5. Load on various GPU blocks

Нагрузка на блоки GPU	switch (множество ядер)	indirect dispatch (множество ядер)	callable
VRAM ↓	**60.3%**	64.8%	—
L2 кэш ↓	22.5%	**21.3%**	—
L1 кэш ↓	10.4%	**7.0%**	—
SM↓	40.4%	**5.7%**	—
SM unused warps↓	30.5%	**9.0%**	—
SM compute warps↑	62.3%	**87.5%**	—
Память (VRAM), L1 и L2 кэши, загрузка вычислительных блоков мультипроцессора (SM) и среднее количество ожидающих подгрупп warp для него (SM unused warps). Поскольку время выполнения было практически одинаково для рассматриваемых вариантов (switch и indirect dispatch), лучше если при одном и том же временя выполнения загрузка блоков будет меньше. Это означает что теперь за то же самое время мы можем выполнить больше вычислений. Стрелками в первом столбце показано в какую сторону должно измениться число для того, чтобы мы могли рассматривать это изменение как улучшение.			

Кроме того, мы провели анализ различных вариантов сгенерированного кода на загрузку по отдельным блокам GPU на Nvidia Nsight (табл. 5), и на основе этого анализа можем сделать следующие выводы.

1) В рассматриваемом относительно простом примере трассировки путей быстрее всего оказался наиболее простой и универсальный способ через сгенерированную конструкцию switch.
   a. Причина этого в том, что в рассматриваемом примере трассировки путей вычислительные блоки всё ещё недогружены (40.4% в табл. 5), а при реализации через множество ядер память является узким местом (60.3% в табл. 5). Проще говоря, используемый нами GPU оказался слишком “прожорливым” в плане вычислений, и реализованный код с шумами не смог загрузить его настолько, чтобы вычисления стали узким местом.
   b. Для данного примера накладные расходы на подсчёт гистограммы в 0.2 мс слишком большие.
2) Тем не менее мы можем видеть, что при одном и том же времени выполнения ядра нагрузка на вычислительные блоки упала в 7 раз (с 40.4% до 5.7%), а количество ожидающих групп warp сократилось с 30.5% до 9% (аналогично количество занятых подгрупп warp выросло с 62.3% до 87.5%).

   a. Таким образом, эффективность выполнения этого участка кода в некотором смысле выросла, и теперь за то же самое время мы можем добавить больше вычислений для реализации обработки материалов. Детальный анализ этого феномена является предметом наших будущих исследований.
3) В рассматриваемом нами примере узким местом является память. После сортировки потоков объединение запросов к памяти работает, очевидно, хуже. Поэтому и улучшения по времени выполнения нет.
4) По своим временным характеристикам реализация через специально введённую функциональность вызываемых шейдеров (callable shaders) в Vulkan не была существенно лучше, чем реализация всего алгоритма трассировки путей через множество ядер (1.36 мс против 1.52 мс). Реализация же в одном ядре и конструкция switch существенно впереди (0.46 мс).
   a. Это свидетельствует о том, что нет единственно лучшего способа для реализации вызова виртуальной функции на GPU, поскольку несмотря на то что вызываемые шейдеры специально предназначены для решения проблемы ветвлений, даже при наличии достаточно тяжёлого кода вычисления шума [55] в исходном коде материалов, сохранение данных в память оказывается дороже.
   b. Тем не менее при дальнейшем росте сложности и функциональности обработки материалов вызываемые шейдеры скорее всего окажутся впереди, т.к. специально для этого разрабатывались.

5. Выводы

Мы предложили решение, способное снизить трудоёмкость реализации алгоритмов на Vulkan и GPU. На входе нашего решения обыкновенный C++ код с рядом ограничений, на выходе – C++ с необходимыми вызовами Vulkan API и кодом шейдеров (OpenCL C или GLSL). В процессе генерации кода могут применяться различные оптимизации для создания нескольких реализаций в зависимости от специфики проблемы и/или заданных конфигураций GPU. Сгенерированный код можно читать, отлаживать и использовать точно так же, как и код написанный на Vulkan вручную. Это позволяет нам легко отделять ошибки пользователей от ошибок генератора.

Наконец, анализ производительности на GPU – это вопрос исследовательский. Наши прогнозы о производительности той или иной реализации для обработки изображений, трассировки путей, вызова виртуальных функций (или чего-то другого) могут легко не сбыться. При этом такой анализ является достаточно трудоёмкой работой, которую не всегда можно выполнить для большой программы в разумных временных рамках. Предложенная нами технология программирования позволяет снизить трудоёмкость подобных экспериментов для достаточно большой кодовой базы за счёт автоматической генерации разных реализаций для одного и того же алгоритма.

6. Ограничения

У предлагаемой технологии есть два основных ограничения. Во-первых её использование возможно только профессиональными разработчиками на GPU, хорошо владеющими Vulkan. Предполагается, что разработчик связывает сгенерированный код на Vulkan со своим кодом вручную. Это более трудоёмко чем, например, разработка на CUDA, однако всё-же существенно менее трудоёмко, чем реализация сложных алгоритмов на Vulkan целиком вручную.

Во-вторых, на практике очень легко выйти за пределы шаблонов, предоставляемых текущей версией нашей системы, что делает невозможным её применение без постоянной адаптации под конкретный класс приложений или даже конкретное приложение. Мы видим в этой

адаптации новую “нормальность” для разработчиков на Vulkan так же, как, например, рано или поздно может потребоваться расширение любой используемой библиотеки в рамках обычной разработки на C++.

Список литературы / References

[1] J. Fang, Сю Huang et al. Parallel programming models for heterogeneous many-cores: a comprehensive survey. CCF Transactions on High Performance Computing, vol. 2, issue 4, 2020, pp. 382-400.

[2] OpenACC. URL: https://www.openacc.org/.

[3] N. Jacobsen. LLVM supported source-to-source translation. Translation from annotated C/C++ to CUDA C/C++. Master’s Thesis. University of Oslo, Norway, 2016, 134 p.

[4] G.D. Balogh, G.R. Mudalige et al. Op2-clang: A source-to-source translator using clang/llvm libtooling. In Proc. of the IEEE/ACM 5th Workshop on the LLVM Compiler Infrastructure in HPC (LLVM-HPC), 2018, pp. 59-70.

[5] P. Yang, F. Dong et al. Improving utility of GPU in accelerating industrial applications with user-centered automatic code translation. IEEE Transactions on Industrial Informatics, vol. 14, issue 4, 2017, pp. 1347-1360.

[6] J. A. Pienaar, S. Chakradhar, A. Raghunathan. Automatic generation of software pipelines for heterogeneous parallel systems. In Proc. of the International Conference on High Performance Computing, Networking, Storage and Analysis, 2012, pp. 1-12.

[7] Н.А. Коновалов, В.А. Крюков. DVM-подход к разработке параллельных программ для вычислительных кластеров и сетей / N.A. Konovalov, V.A. Kryukov. DVM-approach to the development of parallel programs for computing clusters and networks. 2002. URL: https://www.keldysh.ru/dvm/dvmhtm1107/publishr/dvm-appr-OpSys.htm (in Russian).

[8] В.А. Бахтин, А.В. Воронков и др. Использование языка fortran-dvm/openmp для решения больших задач. Труды Всероссийской суперкомпьютерной конференции «Научный сервис в сети Интернет: решение больших задач», 2008 г., стр. 185-191 / V.A. Bakhtin, A.V. Voronkov et al. Using the fortran-dvm / openmp language for solving large problems. In Proc. of the All-Russian Supercomputer Conference on Scientific Service on the Internet: Solving Big Problems, 2008, pp. 185-191 (in Russian).

[9] В.А. Бахтин, В.А. Крюков. DVM-подход к автоматизации разработки параллельных программ для кластеров. Программирование, том. 45, no. 3. 2019 г., стр. 43-56 / V.A. Bakhtin, V.A. Krukov. DVM-Approach to the Automation of the Development of Parallel Programs for Clusters, Programming and Computer Software, vol. 45, no. 3, 2019, pp. 121-132.

[10] M.A. Mikalsen. OpenACC-based Snow Simulation. Master’s Thesis. Norwegian University of Science and Technolog, 2013, 124 p.

[11] T. Öhberg. Auto-tuning Hybrid CPU-GPU Execution of Algorithmic Skeletons in SkePU. Master’s Thesis. Linkoping University, Sweden, 2018, 80 p.

[12] A. Ernstsson, L. Lu, C. Kessler. SkePU 2: Flexible and type-safe skeleton programming for heterogeneous parallel systems. International Journal of Parallel Programming, vol. 46, issue 1, 2018, pp. 62-80.

[13] M. Steuwer, P. Kegel, S. Gorlatch. Skelcl-a portable skeleton library for high-level GPU programming. In Proc. of the IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum, 2011, pp. 1176-1182.

[14] SYCL, cross-platform abstraction layer. URL: https://www.khronos.org/sycl/.

[15] SYCL for CUDA developers, examples, 2020. URL: https://developer.codeplay.com/products/computecpp/ce/guides/sycl-for-cuda-developers/examples.

[16] Vulkan specification, indirect dispatch command, 2021. URL: https://www.khronos.org/registry/vulkan/specs/1.2-extensions/man/html/vkCmdDispatchIndirect.html.

[17] A. Sherin. Resident Evil 2 Frame Breakdown, 2019. URL: https://aschrein.github.io/2019/08/01/re2_breakdown.html.

[18] T.D. Han, T.S. Abdelrahman. hiCUDA: High-level GPGPU programming. IEEE Transactions on Parallel and Distributed Systems, vol. 22, issue 1, 2010, pp. 78-90.

[19] J. Wu, A. Belevich et al. gpucc: an open-source GPGPU compiler. In Proc. of the IEEE/ACM International Symposium on Code Generation and Optimization (CGO), 2016, pp. 105-116.

[20] P. Sathre, M. Gardner, W. Feng. On the portability of CPU-accelerated applications via automated source-to-source translation. In Proc. of the International Conference on High Performance Computing in Asia-Pacific Region, 2019, pp. 1-8.

[21] HIP, C++ Runtime API and Kernel Language, 2021. URL: https://github.com/ROCm-Developer-Tools/HIP.

[22] A. Hamuraru. Atomic operations for floats in OpenCL – improved, 2016. URL: https://streamhpc.com/blog/2016-02-09/atomic-operations-for-floats-in-opencl-improved/.

[23] A. Kapoulkine. Getting Faster and Leaner on Mobile: Optimizing Roblox with Vulkan. 2019. URL: https://www.youtube.com/watch?v=hPW5ckkqiqA.

[24] Non-official Vulkan hardware database, 2021. URL: http://vulkan.gpuinfo.org/.

[25] N. Mammeri, B. Juurlink. VComputeBench: A vulkan benchmark suite for GPGPU on mobile and embedded GPUs. In Proc. of the IEEE International Symposium on Workload Characterization (IISWC), IEEE, 2018, pp. 25-35.

[26] V-EZ, an open source, cross-platform wrapper, 2018. URL: https://github.com/GPUOpen-LibrariesAndSDKs/V-EZ.

[27] Vuh, A Vulkan-based GPGPU computing framework, 2020. URL: https://github.com/Glavnokoman/vuh.

[28] Kompute, The general purpose GPU compute framework for cross vendor graphics cards, 2021. URL: https://github.com/KomputeProject/kompute.

[29] A. Rasch, R. Schulze, S. Gorlatch. Developing High-Performance, Portable OpenCL Code via Multi-Dimensional Homomorphisms. In Proc. of the International Workshop on OpenCL, 2019, article no. 4.

[30] T.-W. Huang, D.-L. Lin et al. Taskflow: A General-purpose Parallel and Heterogeneous Task Programming System. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 2021.

[31] M. Haidl, S. Gorlatch. PACXX: Towards a unified programming model for programming accelerators using C++14. In Proc. of the Workshop on the LLVM Compiler Infrastructure in HPC, 2014, pp. 1-11.

[32] M. Haidl, S. Moll et al. Pacxxv2+ RV: an LLVM-based portable high-performance programming model. In Proc. of the Fourth Workshop on the LLVM Compiler Infrastructure in HPC, 2017, pp. 1-12.

[33] A. Sidelnik, S. Maleki et al. Performance portability with the chapel language. In Proc. of the IEEE 26th international parallel and distributed processing symposium, 2012, pp. 582-594.

[34] R. Baghdadi, J. Ray et al. Tiramisu: A polyhedral compiler for expressing fast and portable code. In Proceedings of IEEE/ACM International Symposium on Code Generation and Optimization (CGO), 2019, pp. 193-205.

[35] Google clspv. A prototype compiler for a subset of OpenCL C to Vulkan compute shaders, 2021. URL: https://github.com/google/clspv.

[36] S. Baxter. Circle C++ shaders, 2021. URL: https://github.com/seanbaxter/shaders.

[37] K.A. Seitz Jr, T. Foley et al. Unified Shader Programming in C++. arXiv preprint arXiv:2109.14682, 2021, 13 p.

[38] Clang documentation, 2021. URL: https://clang.llvm.org/docs/LibTooling.html.

[39] Thrust: a powerful library of parallel algorithms and data structures, 2021. URL: https://developer.nvidia.com/thrust.

[40] A. Kolesnichenko, C.M. Poskitt, S. Nanz. SafeGPU: Contract-and library-based GPGPU for object-oriented languages. Computer Languages, Systems & Structures, vol. 48, 2017, pp. 68-88.

[41] D. Beckingsale, R. Hornung et al. Performance portable C++ programming with RAJA. In Proc. of the 24th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, 2019, pp. 455-456.

[42] T. Heller, P. Diehl et al. Hpx – an open source c++ standard library for parallelism and concurrency. In Proc. of the Workshop on Open Source Supercomputing (OpenSuCo-2017), 2017, pp. 1-5.

[43] A. Paszke, S. Gross et al. Pytorch: An imperative style, high-performance deep learning library. arXiv preprint arXiv:1912.01703, 2019, 12 p.

[44] M. Abadi, A. Agarwal et al. Tensorflow: Large-scale machine learning on heterogeneous distributed systems. arXiv preprint arXiv:1603.04467, 2016, 19 p.

[45] T. Chen, M. Li et al. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. arXiv preprint arXiv:1512.01274, 2015, 6 p.

[46] Nvidia OptiX, 2021. URL: https://developer.nvidia.com/optix.

[47] DirectML, 2021. URL: https://github.com/microsoft/DirectML.

[48] Wisp renderer, 2021. URL: https://github.com/TeamWisp/WispRenderer.

[49] Projects using RTX, 2021. URL: https://github.com.cnpmjs.org/vinjn/awesome-rtx.

[50] J. Hegarty, J. Brunhaver et al. Darkroom: compiling high-level image processing code into hardware pipelines. ACM Transactions on Graphics, vol. 33, no. 4, 2014, pp. 1-11.

[51] J. Ragan-Kelley, C. Barnes et al. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In Proc. of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation, 2013, pp. 519-530.
[52] R. T. Mullapudi, A. Adams et al. Automatically scheduling halide image processing pipelines. ACM Transactions on Graphics, vol. 35, no. 4, 2016, pp. 1-11.
[53] A. Adams, K. Ma et al. Learning to optimize halide with tree search and random programs. ACM Transactions on Graphics, vol. 38, no. 4, 2019, pp. 1-12.
[54] Y. Hu, T.-M. Li et al. Taichi: a language for high-performance computation on spatially sparse data structures. ACM Transactions on Graphics, vol. 38, no. 6, 2019, pp. 1-16.
[55] Y. Hu, Luke Anderson et al. Difftaichi: Differentiable programming for physical simulation. arXiv preprint arXiv:1910.00935, 2019, 20 p.
[56] Y. Hu, J. Liu et al. QuanTaichi: A Compiler for Quantized Simulations. ACM Transactions on Graphics, vol. 40, no. 4, 2021, pp. 1-16.
[57] S.S. Huang, D. Zook, Y. Smaragdakis. Morphing: Safely shaping a class in the image of others. Lecture Notes in Computer Science, vol. 4609, 2007, pp. 399-424.
[58] Inja, template engine for modern C++, 2021. URL: https://github.com/pantor/inja.
[59] W. Bruce, S.R. Marschner et al. In Proc. of the 18th Eurographics conference on Rendering Techniques, 2007, pp. 195-206
[60] Voronoi Noise, 2018. URL: https://www.ronja-tutorials.com/post/028-voronoi-noise/.
[61] Inigo Quilez. Fast 3D Noise, 2013. URL: https://www.shadertoy.com/view/4sfGzS.

Информация об авторах / Information about authors

Владимир Александрович ФРОЛОВ – кандидат физико-математических наук, старший научный сотрудник ИПМ РАН, научный сотрудник факультета ВМК МГУ. Сфера научных интересов: реалистичная компьютерная графика, моделирование освещённости, разработка программных систем оптического моделирования, параллельные и распределённые вычисления.

Vladimir FROLOV – PhD in computer graphics, senior researcher at Keldysh Institute of Applied Mathematics and researcher in computer graphics at Moscow State University. Research interests: realistic computer graphics, light transport simulation, elaboration of optical simulation software systems, GPU computing.

Вадим Владимирович САНЖАРОВ – младший научный сотрудник факультета ВМК МГУ, научный сотрудник ИПМ РАН. Сфера научных интересов: компьютерная графика, системы фотореалистичного синтеза изображений, параллельное и распределённое программирование.

Vadim SANZHAROV – junior researcher in computer graphics at Moscow State University and researcher at Keldysh Institute of Applied Mathematics. Research interests: realistic computer graphics, elaboration of optical simulation software systems, concurrent and distributed computing.

Владимир Александрович ГАЛАКТИОНОВ – доктор физико-математических наук, профессор, заведующий отделом компьютерной графики и вычислительной оптики. Сфера научных интересов: компьютерная графика, оптическое моделирование, создание программных систем оптического моделирования.

Vladimir GALAKTIONOV – Doctor of Science in physics and mathematics, Professor, Head of Computer graphics department. Research interests: computer graphics, optical simulation, elaboration of optical simulation software.

Александр Станиславович ЩЕРБАКОВ – аспирант 3-го года обучения факультета ВМК МГУ, ведущий разработчик в Gajin Entertament. Сфера научных интересов: компьютерная графика, системы фотореалистичного синтеза изображений, параллельное и распределённое программирование.

Alexander Stanislavovich SHCHERBAKOV – 3rd year postgraduate student at the Faculty of Computational Mathematics and Cybernetics, Moscow State University, Lead Developer at Gaijin Entertainment. Research interests: realistic computer graphics, elaboration of optical simulation software systems, concurrent and distributed computing.