ИПМ им.М.В.Келдыша РАН • Электронная библиотека

Препринты ИПМ • Препринт № 58 за 2023 г.

ISSN 2071-2898 (Print)
ISSN 2071-2901 (Online)

В.А. Фролов, В.А. Галактионов

Программирование массивно-параллельных архитектур без API на примере параллельного алгоритма построения дерева

Рекомендуемая форма библиографической ссылки: Фролов В.А., Галактионов В.А. Программирование массивно-параллельных архитектур без API на примере параллельного алгоритма построения дерева // Препринты ИПМ им. М.В.Келдыша. 2023. № 58. 54 с.
https://doi.org/10.20948/prepr-2023-58
https://library.keldysh.ru/preprint.asp?id=2023-58

Ордена Ленина
ИНСТИТУТ ПРИКЛАДНОЙ МАТЕМАТИКИ
имени М. В. Келдыша
Российской академии наук

В.А. Фролов, В.А. Галактионов

Программирование массивно-параллельных архитектур без API на примере параллельного алгоритма построения дерева

Москва — 2023

В.А. Фролов, В.А. Галактионов.

Программирование массивно-параллельных архитектур без API на примере параллельного алгоритма построения дерева

Аннотация. В работе предлагается подход к программированию параллельных архитектур без использования интерфейсов параллельного программирования (API) и параллельных директив для различных приложений численного моделирования и компьютерной графики. Основная цель этого подхода — разрешение фундаментального противоречия между кросс-платформенностью и аппаратным ускорением при разработке высокопроизводительных программ. Это противоречие разрешается путём автоматизации процесса разработки: алгоритмическое описание на C++, лишённое каких-бы то ни было специфичных параллельных конструкций, автоматически транслируется в реализацию того же самого алгоритма и генерируется реализация на некотором существующем API программирования параллельных архитектур (C++ транслируется в SPIR-V для GPU, и в векторные инструкции для CPU). Далее, при необходимости добавления специфического аппаратного ускорения программист может заместить отдельные части сгенерированного кода при помощи замены ядер и замещения виртуальных функций в сгенерированном коде. Такое замещение позволяет перегенерировать код и не потерять изменения, сделанные пользователем под конкретную аппаратную платформу. Разработанная система работает как белый ящик, позволяя программисту читать и отлаживать сгенерированный код так же, как если бы этот код был написан вручную. Это позволяет нам легко отличать ошибки транслятора от ошибок пользователя и, кроме того, не создаёт зависимости в проектах от разработанной системы. Применение разработанной технологии рассматривается на примере алгоритмов параллельного построения деревьев, одной из самых сложных и неудобных для GPU задачи.

Ключевые слова: Программирование GPU, алгоритм Карраса.

V.A. Frolov, V.A. Galaktionov.

A no-API approach to massive-parallel architectures

Abstract. The work proposes an approach to programming parallel architectures without using parallel programming interfaces (APIs) and parallel directives for various numerical simulation and computer graphics applications. The primary goal of this approach is to resolve the fundamental conflict between cross-platform compatibility and hardware acceleration when developing high-performance programs. This conflict is resolved through the automation of the development process: algorithmic descriptions in C++, free from any specific parallel constructs, are automatically translated into the implementation of the same algorithm, and a realization is generated on an existing parallel architecture programming API (C++ translates to SPIR-V for GPUs, and vector instructions for CPUs). Furthermore, if specific hardware acceleration is required, the programmer can replace individual parts of the generated code by substituting kernels and virtual functions in the generated code. Such substitution allows for code regeneration without losing user-made changes tailored to a particular hardware platform. The developed system operates as a white box, allowing the programmer to read and debug the generated code as if it were written manually. This enables us to easily distinguish translator errors from user errors and, moreover, does not create dependencies on the developed system in projects. The generated code can always be manually rewritten when necessary. The application of the developed technology is considered using the example of one of the most complex and inconvenient problems for GPUs — parallel algorithms for constructing trees.

Key words: GPU programming, Karras algorithm.

Содержание

1 Введение

Разработка высокопроизводительных програмных и программно-аппаратных систем на сегодняшний день является сложной проблемой. Когда разработчики полагаются на одного производителя вычислительного оборудования (например, Nvidia), вопрос выбора технологии программирования, как правило, остро не стоит. Разработчики использует технологии программирования, которые этот производитель поддерживает наилучшим образом [1, 2, 3]. Однако такой подход возможен далеко не всегда.

Кросс-платформенная разработка (под разные вычислительные системы, центральные процессоры с различными векторными архитектурами, графические процессоры от разных производителей и т.д.) ставит перед современными разработчиками сложную проблему, для которой в настоящий момент нет хорошего решения. Необходимо сделать выбор в пользу определённых технологий, проанализировав потребности в имеющемся на текущий момент (а также с прицелом на будущее) аппаратном ускорении, а затем практически всецело довериться им. Разработчик вынужден строить своё решение на базе выбранного API и/или компилятора параллельной технологии программирования и ожидает, что в выбранных технологиях нужные ему возможности будут поддерживаться (а в идеале и развиваться) на достаточно широком классе оборудования достаточно продолжительное время. На практике это почти всегда не выполняется.

Кроме того, фиксация технологии параллельного программирования зачастую приводит к ряду принципиальных проблем.

- В технологиях, ориентированных на высокую производительность и, как правило, основанных на API:

    1. Трудоёмкость разработки почти всегда вырастает в несколько раз, т.к. логика алгоритма перемешивается с логикой взаимодействия с API, что приводит к запутанному, трудно-отлаживаемому и трудно-поддерживаемому коду;

2. Время жизни программной системы сильно сокращается: выход нового “перспективного” API приводит к мгновенному устареванию существующей кодовой базы;
3. Зачастую приходится поддерживать несколько API, что ещё больше увеличивает стоимость и трудоёмкость разработки. Это особенно остро ощущается в приложениях, ориентированных на индивидуальных конечных пользователей с неизвестным “зоопарком” целевых ЭВМ: графические редакторы, просмотрщики, игры, программы численного моделирования.

- В технологиях программирования, ориентированных на высокую продуктивность разработки и, как правило, основанных на компиляторах, появляется сильная зависимость от этой технологии:

  1. Программная система становится ограничена в возможностях поддержки аппаратного ускорения: нужную разработчику прикладного приложения функциональность в технологию программирования может добавить только разработчик самой технологии программирования;
  2. Собрать и использовать прикладную систему без выбранной технологии параллельного программирования становится практически невозможно: если в этой технологии встречается ошибка или нужно перенести программу, например, на другую ОС, разработчики заходят в тупик;
  3. Зачастую при получении низкой производительности своего решения в таких системах программист ничего не может с этим сделать: ни понять причину, ни как-либо повлиять на результат работы компилятора не представляется возможным.

Мы предлагаем относительно легковесное решение этих проблем за счёт фундаментально иного подхода. Мы не вводим новые языковые конструкции или директивы для параллельной обработки в

язык. Напротив, мы ограничиваем возможности языка определенными шаблонами программирования и вызовами библиотек, делая "чистый код" одновременно и высокопроизводительным. В отличие от многих других технологий программирования наше решение не делает проекты, использующие его, зависимыми от него самого: (1) исходный код представляет собой обычный код на C++, который может быть обработан любым существующим компилятором C++11 и также может использовать любые технологии на основе директив внутри него (например, OpenMP); (2) выходной код на технологии программирования нижнего уровня представляет собой читаемую реализацию того же самого алгоритма на некотором API параллельного программирования нижнего уровня (ISPC или Vulkan), которая предназначена для работы путем замещения различных виртуальных функций и ядер, если это необходимо.

В результате этот подход имеет важное преимущество перед существующими: даже если наш транслятор имеет ограниченную поддержку аппаратного обеспечения, пользователь не ограничен этими функциями. Это происходит потому, что оптимизации могут быть вручную добавлены в сгенерированный код, и наш подход предполагает, что пользователь будет делать это таким образом, чтобы перегенерация кода не нарушила пользовательских изменений. Поэтому мы позиционируем нашу систему как помощника для разработчиков, которым необходим контроль над производительностью и понимание (через тщательный анализ низкоуровневого кода).

Замечание: В нашей работе мы ориентируемся на т.н. массивно-параллельное программирование, не рассматривая традиционный кластерный параллелизм вроде MPI. Теоретически, наши идеи можно применить и к этому виду параллельных программ, однако эти исследования находятся за рамками нашей работы.

2 Предыдущие работы

Во введении мы упомянули, что разделяем современные технологии массивно-параллельного программирования (далее просто ТП) на 2 типа: ориентированные на производительность и ориентированные на продуктивность разработки. Это разделение не случайно, поскольку любые ТП определяются прежде всего целью, для которой они создавались.

2.1 ТП, ориентированные на производительность

Как правило, технологии программирования из этой группы создаются производителями оборудования или комитетами по стандартизации, такими как Khronos Group. Сюда можно отнести: CUDA, OpenCL, DirectX9–DirectX12, Metal, Vulkan, OpenGL3, OpenGL4, PSGL и аналоги.

GPGPU. В настоящее время CUDA является доминирующей ТП благодаря лидерству компании Nvidia. Тем не менее, помимо отсутствия кросс-платформенности, у нее есть множество недостатков: CUDA является мощной технологией с огромным количеством функций и версий (на данный момент 11 версий), которые не всегда работают одинаково хорошо на различных видеокартах Nvidia. Перенос большой программной системы с одной версии CUDA на другую часто занимает много времени. В попытках сделать CUDA кросс-платформенной были разработаны несколько открытых реализаций CUDA, использующих OpenCL или SIMD-инструкции центральных процессоров [14, 15, 16], а также AMD HIP [17]. Однако этот подход плох в своей основе: отсутствие аппаратной поддержки определенной функциональности в OpenCL (или другом бэкэнде) приводит к тому, что программное обеспечение либо вообще не будет работать, либо будет работать медленно из-за программной эмуляции аппаратно ускоренной функциональности. В то же время главным недостатком OpenCL является отсутствие поддержки широкого спектра аппаратной функциональности современных GPU.

Например, в OpenCL3.0 отсутствует непрямой вызов ядра (он же динамический параллелизм) [18]. Эта функциональность критична для алгоритмов с нетривиальным потоком управления, в которых логика выполнения вычислительных ядер зависит от результатов вычислений на графическом процессоре (т.е. запуск вычислительных ядер происходит без контроля со стороны CPU).

Графические API (DirectX,Metal,Vulkan и др.), несмотря на название, не являются узкоспециальными, а 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 CUDA. При этом, помимо вычислительных шейдеров в них, как правило, присутствует несколько видов графического конвейера и конвейера трассировки лучей. Бурное развития GPU в последние 20 лет не было эволюционным, а скорее наоборот, как сами GPU, так и API взаимодействия с ними менялись кардинально. Нет ни одной причины считать, что эти изменения остановились. Однако на сегодняшний момент мы можем ориентироваться на Vulkan как на наиболее кросс-платформенный и универсальный API, в котором в то же самое время доступно максимальное количество аппаратного ускорения. Цена за это — высокая трудоёмкость разработки программных систем на таком API.

В отличие от предшественников, Vulkan значительно более явно декларирует возможности управления графическим процессором. Это неоднородность памяти (что необходимо для реализации эффективных механизмов копирования данных и аллокации ресурсов); буферы команд (что позволяет выполнять без участия центрального процессора заранее записанные длинные последовательности команд API); синхронизация на основе барьеров, что позволяет более эффективно распараллеливать разные вычислительные ядра, не зависящие друг от друга, и более эффективно передавать данные между зависящими друг от друга ядрами, не выгружая эти данные в DRAM из L2; управление расположением данных внутри текстур (т.н. лэй-

ауты текстур), что может быть критично для производительности алгоритмов, работающих с изображениями и/или осуществляющих рендеринг в текстуру; сжатие текстур с возможностью аппаратной декомпрессии на лету; аппаратное ускорение трассировки лучей; конвейер трассировки лучей и таблица виртуальных функций (в настоящий момент эти функциональности связаны); управление графическим конвейером; взаимодействие между CPU и GPU во многих потоках CPU асинхронно.

ISPC [38] — это сокращение для Implicit SPMD Program Compiler. Изначально этот компилятор был разработан для векторизации CPU-кода, но позже была добавлена поддержка графических процессоров Intel. Основными преимуществами ISPC являются высокая производительность и контроль над сгенерированным векторизованным CPU-кодом как для архитектуры x64, так и для ARM. Он предоставляет программисту огромное количество подсказок по оптимизации кода, обладает встроенной поддержкой т.н. SOA-расположения данных (Structure Of Arrays) и множеством других функций [38]. Еще одним преимуществом является то, что для CPU-реализации нет необходимости копировать данные в другое адресное пространство, что обычно требуется делать для графических процессоров. ISPC ограничивается только графическими процессорами Intel, и поддерживаемые аппаратные возможности GPU для него даже меньше, чем в OpenCL, поэтому мы рассматриваем его только в контексте использования для векторизации кода на CPU, что не очень хорошо работает в Vulkan.

MLIR [55] — это одна из первых технологий промежуточного представления программ (как LLVM), ориентированная на внедрение алгоритмических оптимизаций компилятором. Основная концепция заключается в том, что элементарные операции MLIR могут быть достаточно высокоуровневыми, и, кроме того, они могут быть произвольными. Это позволяет строить процесс трансляции в виде постепенного процесса понижения уровня абстракции (т.н. lowering), пре-

вращая высокоуровневые элементарные операции в низкоуровневые и относительно легко внедряя различные алгоритмические оптимизации под разное оборудование. В настоящий момент MLIR активно используется для кроссплатформенной реализации нейросетевых моделей. Использование MLIR, несмотря на имеющиеся преимущества, сопряжено с рядом трудностей: MLIR не позволяет получить программисту полный контроль над процессом трансляции и производительностью в той же самой мере, в которой он, например, получает этот контроль при использовании графических API, CUDA или ISPC, т.к. процесс оптимизации спрятан внутри компилятора, а результат работы компилятора уже невозможно редактировать.

Близкие аналоги. Circle [24] — это экспериментальный компилятор C++ на базе LLVM, позволяющий осуществлять трансляцию C++ в SPIR-V и обладающий возможностями писать шейдеры в том числе для графического конвейера, имеющий поддержку CUDA бэкенда и различных возможностей, не входящих в стандарт C++. На наш взгляд Circle – это очень перспективное решение, попадающее, тем не менее, в классическую ловушку подобных технологий программирования: поскольку Circle – это компилятор C++ в традиционном понимании, работающий как чёрный ящик, его использование чревато возникновением ошибки, которую невозможно будет ни исправить, ни обойти. А с учётом того, что Circle — закрытый проект, не предоставляющий исходный код, его использование в значительной мере ограничено.

Работа [39] позволяет создавать шейдеры для графического конвейера на C++ в рамках унифицированной структуры, где код хоста и устройства смешаны. Отличительной чертой работы [39] является то, что она использует наследование классов C++ для интеграции сгенерированного кода с существующим и также для статической диспетчеризации шейдеров и их специализации. Это делает идеи работы [39] тесно связанными с нашими. Еще одной общей чертой нашей работы является то, что [39] позволяет использовать HLSL поверх сгенерированного кода напрямую в случае, если некоторые

аппаратные возможности не поддерживаются их транслятором. В то же время наша работа отличается от [39] в нескольких аспектах: (1) авторы [39] использует "игровой движок" Unreal Engine4 для хост-бэкенда и ориентирована на разработчиков игр, а не на общее вычисление; (2) работа [39] не позволяет выполнять входной код вне их среды на центральном процессоре, что является одной из основных целей нашего подхода; (3) для бэкенда Vulkan наш подход имеет другой набор функций по сравнению с [39]: аппаратное ускорение трассировки лучей, реализация вызова виртуальных функций, автоматическое обнаружение динамического параллелизма и генерация непрямых вызовов ядер, генерация кода параллельных примитивов программирования: редукция, префиксная сумма, сортировка.

Компилятор clspv [34] 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.

2.2 ТП, ориентированные на продуктивность

В этой группе существует огромное количество научных работ, создающих технологии как общего назначения, так и специализированные. Мы рассмотрим лишь наиболее близкие к нашей концепции технологии.

Языки параллельного программирования. Язык Halide [20, 21] изначально создавался для обработки изображений. Пользователь пишет программу на одном языке, а на выходе может быть сгенерирован C++ с шейдерами под разные API нижнего уровня. Наиболее примечательной особенностью Halide является наличие

встроенных алгоритмических оптимизаций с учетом предварительных знаний об алгоритмах обработки изображений. Taichi [41] использует статически типизированный язык, который выглядит как Python, но на самом деле это “не-Python”, как Numba [42], где все типы должны быть статически определены. И Halide, и Taichi являются довольно зрелыми технологиями промышленного уровня, которые в настоящее время имеют множество различных бэкендов для CPU и GPU.

Тем не менее для разработчика существует высокий риск стать зависимым от такой технологии, так же как и в случае Circle, поскольку они не являются каким-либо индустриальным стандартом и, по факту, существует единственная в мире реализации как для Halide, так и для Taichi. Поэтому, когда происходит портирование существующей кодовой базы с C++ на Taichi, разработчику приходится принимать трудное решение о прекращении разработки текущей кодовой базы на C++ и выборе Taichi или Halide в качестве основной технологией для проекта.

Языковые расширения и параллельные директивы. Менее рискованный подход — добавить знание о параллельном выполнении поверх существующей программы с использованием директив или языковых расширений. Сюда входят OpenMP, OpenACC [3], C++ AMP (Microsoft), Spearmint [4], OP2 [5], работы [6, 7], DVMH [8] и многие другие. У этого подхода есть два ключевых недостатка: (1) неявное копирование данных может негативно сказаться на производительности; (2) крайне слабая поддержка аппаратных возможностей GPU (даже текстуры не поддерживаются). Интересно отметить особенности системы DVM/DVMH, которая использует 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.

Использование C++ как есть. Другой подход заключается в использовании существующих языковых возможностей, специфических контейнеров и типов данных для распараллеливания алгоритмов, после чего создается новый компилятор C++ с различными бэк-эндами для CPU и GPU. К этой группе можно отнести скелетное программирование (SkePU/SkelCL [10, 11, 12]), новые стандарты C++ (NVC++ [43], SYCL [13] и OneAPI [44]); PACXX [22, 23]. На практике эти подходы не слишком отличаются от параллельных директив, о которых мы говорили в предыдущем разделе, и имеют ровно те же недостатки. Пример Nvidia весьма показателен: несмотря на наличие трех разных решений для C++ на GPU [43], — NVC++, OpenACC и CUDA, — нет эффективных способов смешивать код на этих технологиях и переходить между ними. Например, невозможно написать код для NVC++ и вставить ядра CUDA в некоторые места, критичные с точки зрения производительности, избегая дорогого копирования данных между CPU и GPU. И хотя, скорее всего, рано или поздно такая функция появится, эти технологии поддерживают только оборудование Nvidia (как мы обсуждали ранее, AMD HIP, например, имеет очень урезанный набор возможностей CUDA).

Предметно-ориентированные ТП. Nvidia OptiX [29], DirectML [30], AMD MIOpen; Taichi, Halide и другие DSL (языки предметной области) также можно рассматривать как DSL. Эти технологии имеют множество преимуществ в своей области благодаря узко ориентированным целям и специфическому дизайну. Кроме того, некоторые из них на самом деле более ориентированы на производительность и в то же время являются проприетарными (DirectML, OptiX). Поскольку намного проще писать трассировку лучей в OptiX, по

сравнению с Vulkan, например, почти все промышленные системы рендеринга перешли на OptiX: Octane, VRay, IRay, RedShift, Cycles, Thea и многие другие. Недостатком технологий, ориентированных на определенную предметную область, является то, что алгоритмы, выраженные на них, сложно переносить в другие области и интегрировать с остальным программным обеспечением, которое не использует DSL.

Таким образом, в настоящее время нет технологий, которые бы с одной стороны обладали высокой продуктивностью разработки, с другой стороны были бы кроссплатформенными и, наконец, позволяли бы осуществить доступ к любому имеющемуся и перспективному аппаратному ускорению без внесения модификаций в саму технологию программирования. Именно этот пробел заполняет наша работа.

3 Предложенный подход

Наш подход в некотором смысле “идёт против течения”, поскольку построен на идеях, противоположных многим из рассмотренных:

1. Мы используем существующий язык (C++) на входе, но не добавляем новых расширений в него, а, наоборот, ограничиваем возможности пользователя, чтобы не допустить кода, который невозможно ускорить на массивно-параллельных системах принципиально;

2. На выходе мы генерируем реализацию алгоритма на ТП нижнего уровня (Vulkan, ISPC или др.), применяя алгоритмические оптимизации прямо поверх элементов AST дерева;

3. Программист видит сгенерированный код как белый ящик, и имеет возможность его дорабатывать, причём так, что при перегенерации доработки не пропадают (рис. 1).

Наша технология программирования использует сопоставление с шаблоном (или т.н. “паттерны”). Паттерны не определяют аппарат-

ных функций и не задают параллельных конструкций, а выражают алгоритмические и архитектурные знания. Таким образом, “чистый код”, подходящий под паттерны, одновременно является и высокопроизводительным. При этом за аппаратное ускорение отвечает транслятор, а не разработчик.

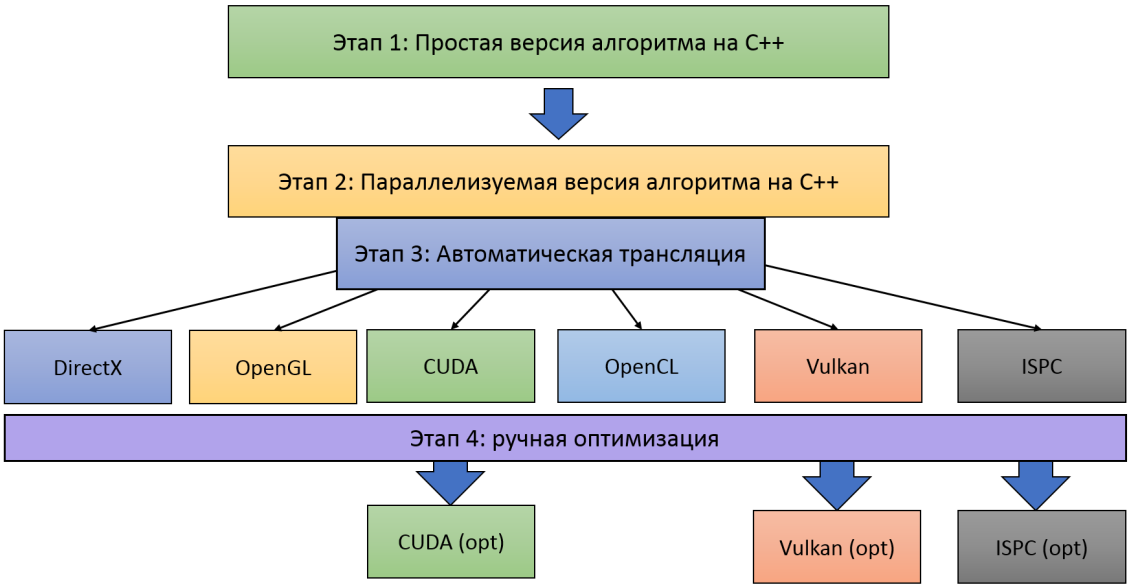


Рис. 1. Схема работы с предложенной технологией. На первом этапе программист разрабатывает свой алгоритм на C++ без каких-либо ограничений. На втором этапе он производит постепенное изменение исходного кода так, чтобы алгоритм был в принципе параллелизуем (на данном этапе можно, например, пробовать использовать OpenMP). На третьем этапе он запускает наш транслятор, и, следуя его указаниям, дорабатывает свой исходный код так, чтобы трансляция прошла без ошибок. После этого уже имеется работающая версия алгоритма на GPU, и в принципе здесь можно остановиться. При необходимости программист дорабатывает сгенерированную версию на 4 этапе, замещая отдельные вычислительные ядра или виртуальные функции сгенерированного класса.

3.1 Принципы работы транслятора

Единицей “трансляции” в нашем подходе является класс, имя которого транслятору нужно явно подать на вход. Далее от него генерируется производный класс, замещающий базовую реализацию массивно-параллельной реализацией с теми же самыми алгоритмами (рис. 2). Интеграция сгенерированного класса в существующий код, таким образом, происходит простой заменой указателя на исходный класс указателем на сгенерированной класс. Далее работает обычный механизм замещения виртуальных функций.

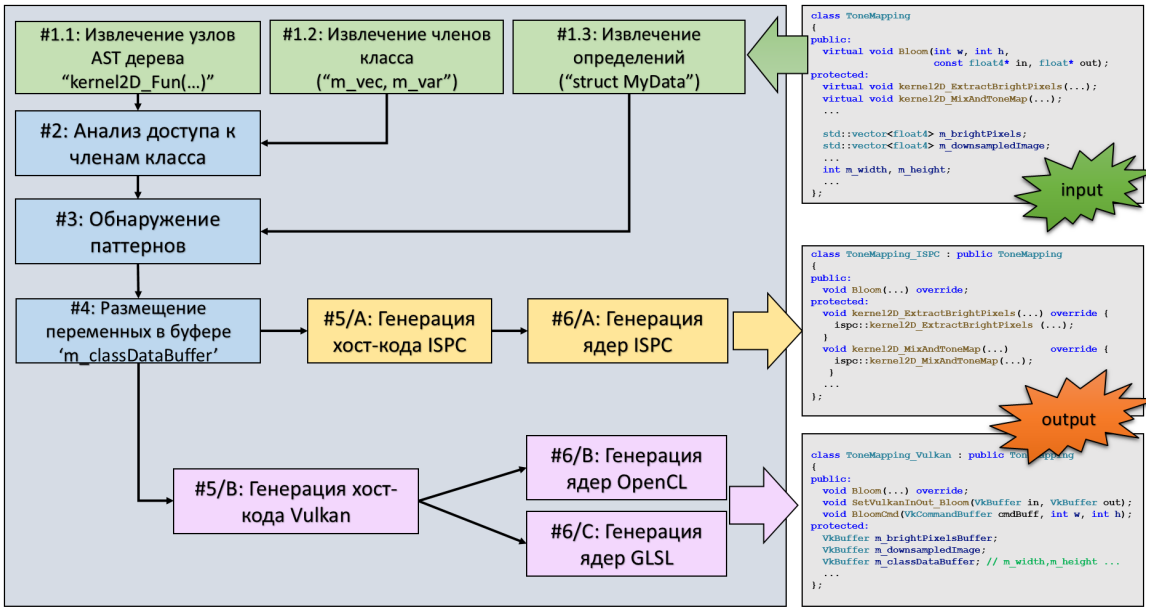


Рис. 2. Блок-схема предложенного транслятора. Обратите внимание, что в сгенерированном коде на Vulkan класс имеет функции-члены (“SetVulkanInOut_Bloom” и “BloomCmd”), которые предназначены для работы с существующими данными в GPU памяти.

Рассмотрим этапы, изображённые на рис. 2. На первом проходе (зелёные прямоугольники сверху) мы извлекаем узлы AST дерева для *ядер* и *управляющих функций*, имена и типы членов класса, константы и функции. Ядра — это функции-члены класса, обязательно содержащие внутри себя некоторое количество вложенных циклов.

Они будут преобразованы в вычислительные ядра для GPU. Управляющие функции — это функции-члены класса, которые вызывают ядра и, таким образом, управляют их запуском. В сгенерированном классе эти функции будут замещены кодом, управляющим привязкой ресурсов и запуском ядер на GPU.

В последующих проходах мы проводим анализ исходного кода ядер (синие прямоугольники): к каким членам класса ядра осуществляют доступ и каким образом это происходит внутри циклов. Кроме того, мы анализируем и некоторое другое поведение. Если говорить более точно, мы обнаруживаем паттерны параллельного программирования в последовательном коде: непрямой вызов ядра, редукция, вызовы виртуальных функций [40], доступ к текстурам, добавление элемента в конец масива, трассировка лучей, а также использование стандартной функциональности в управляющих функциях — memset, memcopy, sort, scan. Мы делаем это путём статического анализа AST дерева исходного кода, используя clang.

Последние два этапа на рис. 2 выполняют рендеринг текста по шаблону с помощью библиотеки “{{inja}}” [26] с использованием различных шаблонов для ISPC (желтые прямоугольники) и Vulkan (пурпурные прямоугольники). Таким образом, выходные классы можно использовать так же, как если бы они были реализованы в Vulkan или ISPC вручную.

3.2 Транслятор C++ в SPIR-V

Для Vulkan мы реализовали два варианта компиляции вычислительных ядер. В первом варианте мы преобразуем исходный код вычислительных ядер в OpenCL ядра, после чего подаём их на вход компилятору clspv [34]. Исходно мы считали этот метод более надёжным, поскольку в нём выполняются минимальные преобразования для исходного кода на C++. К сожалению, на практике мы столкнулись с двумя недостатками этого подхода:

1. clspv использует ряд расширений SPIR-V, которые поддерживают не все производители GPU. Это в первую оче-

редь SPV_KHR_variable_pointers для поддержки указателей и SPV_KHR_non_semantic_info для некоторых внутренних нужд проекта clvk [58], реализующего OpenCL поверх Vulkan.

2. Внедрение некоторой произвольной аппатной функциональности в clspv (например, ускорение трассировки лучей или поддержка массивов текстур VK_EXT_descriptor_indexing) представляется большой задачей, что влечёт за собой необходимость тесной кооперации с проектом clspv, либо поддержку отдельной ветки clspv, что также довольно трудоёмко.

Во втором варианте мы преобразуем код вычислительных ядер из C++ в GLSL и далее используем существующие компиляторы GLSL (glslangValidator [56] или glslc [57]), чтобы получить SPIR-V. Это потребовало от нас существенно большей работы по трансформации исходного C++ кода в GLSL, осбенно в части работы с указателями. Поскольку в GLSL не только отсуствуют указатели, но даже нет возможности передать ссылку на буфер или текстуру в функцию, мы были вынуждены разработать альтернативное решение.

Мы выполняем статический рекурсивный анализ графа вызовов для того, чтобы обнаружить в C++ коде функции, в которые передаются указатели на данные, находящиеся в глобальных буферах. Затем мы заменяем эти указатели на данные в буфере соответствующими смещениями от начала буфера. Например, если программист вызывает функцию f в двух местах внутри ядра: “f(vec1.data() + 5)” и “f(vec2.data() + 10)”, наша реализация сопоставит фактические аргументы формальным и сгенерирует две новые функции f_vec1 (которая обращается к буферу vec1) и f_vec2 (которая обращается к буферу vec2). Вызовы будут преобразованы в f_vec1(5) и f_vec2(10) соответственно, а смещения 5 и 10 будут переданы внутрь функций и применены ко всем местам, где осуществляется доступ к соответсвующему буферу. Что касается указателей и ссылок на локальные переменные вычислительных ядер (передаваемые в функции), они преобразуются в inout аргументы функций в GLSL.

3.3 Аппаратное ускорение

Наш транслятор отображает определённые типы данных (классы C++) и библиотечные функции в аппаратно-ускоренную функциональность на GPU. Например, каждый член класса типа std::vector преобразуется в VkBuffer. Чтобы добавить поддержку изображений, пользователь должен использовать специальный шаблонный класс Image2D, который является библиотечной реализацией изображений на C++. Чтобы использовать аппаратное ускоренние для трассировки лучей, пользователь вызывает функциональные члены определенного интерфейса ISceneObject. На центральном процессоре этот интерфейс реализован нами через Embree [49], а на графическом процессоре транслятор генерирует код GLSL с аппаратно ускоренными запросами трассировки лучей.

Здесь необходимо сделать несколько уточнений: Во-первых, наш транслятор фиксирует только интерфейсы “std::vector”, “Image2D” и “ISceneObject” — его совершенно не интересует их реализация на центральном процессоре, которая может быть абсолютно любой. Например, если пользователь не может использовать библиотеку Embree по какой-либо причине, он может создать собственную реализацию ISceneObject. Во-вторых, для GPU, у которых нет определённой аппаратной подержки (например, трассировки лучей), наш транслятор может генерировать код с программной реализацией этой же самой функциональности. Для этой цели мы разработали специальный механизм замещения неподдерживаемой аппаратной функциональности на основе паттерна “композиции” классов.

Поддержка старых GPU. Рассмотрим 3 шага, которые предпринимает программист для того, чтобы заместить неподдерживаемую аппаратную функциональность некоторого интерфейса (пусть это будет интерфейс трассировки лучей ISceneObject) собственной алгоритмической реализацией, которая точно так же будет переноситься на GPU нашим транслятором, как и остальной пользовательский код.

1. На первом шаге необходимо использовать паттерн программирования “композиции” и поместить указатель на интерфейс ISceneObject внутрь своего класса MyRender (листинг 1).

2. На втором шаге программист должен создать собственную реализацию ISceneObject (назовём её MyRayTracer) на C++ и отладить в своей программе на центральном процессоре.

3. На третьем шаге через аргументы командной строки программист должен явно указать транслятору, что он хочет в классе MyRender использовать для реализации интерфейса ISceneObject собственный класс MyRayTracer.

Наш транслятор объединяет классы MyRender и MyRayTracer так, как будто все данные и функции MyRayTracer просто объявлены внутри MyRender. Для того чтобы избежать конфликта имён, используется переименование: для всех данных и функций класса MyRayTracer мы добавляем префикс в виде имени переменной “m_pRayTracer_” (листинг 1).

```
struct MyRender {
  void kernel2D_DoSomeRayTrace(...) {
        m_pRayTracer->NearestHit(...);
  }
  std::shared_ptr<ISceneObject> m_pRayTracer;
};
```

Листинг 1: Пример использования паттерна композиции

Поддержка перспективных GPU. Поскольку вариантов различного аппаратного ускорения очень много, то невозможно поддержать их все в нашем трансляторе. Кроме того, мы бы хотели иметь возможность в будущем добавлять в программу перспективное аппаратное ускорение, то есть то, которое ещё не реализовано в современных GPU. Для того чтобы решить эту проблему, можно создать производный класс от сгенерированного нашим транслятором класса. В этом производном классе пользователь может делать всё

что ему нужно, работая напрямую с Vulkan или ISPC путём замещения отдельных ядер и виртуальных функций. Таким образом, при изменении исходного кода и перегенерации класса, реализщующего алгоритм на GPU, пользовательские изменения не пропадают.

3.4 Паттерны и их обнаружение

Для нас паттерн — это любой набор часто используемых конструкций языка и/или библиотечных вызовов в программе, которые должны как-то по-особому реализовываться на GPU (или другой массивно-параллельной вычислительной системе). Перечислим несколько основных паттернов, которые обнаруживает наш транслятор:

1. Параллельная редукция, реализованная при помощи обнаружения доступа к переменным класса специальным образом (листинг 4);

2. Параллельная префиксная сумма и сортировка, которые выглядят как вызов библиотечных функций std::inclusive_scan, std::exclusive_scan, std::sort в управляющих функциях;

3. Вызовы определённых библиотечных интерфейсов: обращение к текстуре, запрос на трассировку луча, доступ к массиву текстур, конверсия битовых представлений данных, работа с типом комплексных чисел;

4. Вызов виртуальных функций, которые исследованы в отдельной работе [40];

5. Различные случаи использования указателей: в примере из раздела 3.2 операция сложения указателя и смещения в аргументе функции $f(vec1.data()+5)$ — паттерн, который мы вынуждены отдельно обрабатывать, чтобы поддержать хотя бы в ограниченном виде функциональность указателей в GLSL.

Таким образом, добавляя поддержку новых достаточно произвольных паттернов, мы имеем возможность дорабатывать нашу технологию программирования под заказчика.

4 Параллельное построение дерева

Рассмотрим использование предложенной ТП на примере построения дерева по алгоритму Карраса [60]. Он показателен, поскольку построение иерархических структур на GPU является не просто сложной, а одной из самых сложных и неудобных задач для GPU. Это наглядно демонстрирует, что наш подход может, а что нет.

Для начала ответим на следующий вопрос: можем ли мы записать построение дерева традиционным способом, не задумываясь о параллельности вообще, осуществляя рекурсивное разбиение входного множества данных на подмножества так, как мы обычно это делаем на C++ или другом языке программирования. Краткий ответ — **нет**, в настоящий момент такой способ неизвестен. Существующие алгоритмы [60, 61, 62, 63, 64, 65, 66, 67] стараются свести задачу построения дерева к набору примитивных операций, которые можно эффективно реализовать на массивно-параллельной системе — префиксная сумма, сортировка, редукция. И вот эти примитивы мы **уже можем** записать по-простому, в 1 потоке, а затем автоматически преобразовать в массивно-параллельную реализацию тех же самых алгоритмов на GPU.

Поэтому, используя предложенную нами технологию программирования, нужно будет точно так же реализовывать алгоритм Карраса (а не просто “наивное” построение дерева сверху-вниз), как и в случае, например, непосредственного использования графических API. Тем не менее на начальном этапе разработки мы действительно можем иметь произвольную, простую реализацию алгоритма и далее мы можем двигаться к цели постепенно.

Итак, объявим интерфейс *IBVHBuilder*, с которым наша программа далее будет работать.

```
struct BVHNode  {
  float3 boxMin; uint leftOffset;
  float3 boxMax; uint rightOffset;
};
struct Box4f  {
  float4 boxMin; // 4 component for aligment
  float4 boxMax; // 4 component for aligment
};
struct IBVHBuilder  {
  virtual void BuildFromBoxes(const Box4f* in_boxes,
                              int boxNum,
                              BVHNode* out_tree) = 0;
  virtual void CommitDeviceData() {}
  virtual void GetExecutionTime(char* name, float time[4]){}
  virtual size_t Reserve(size_t a_inPrimsNum) = 0;
};
```

Листинг 2: Интерфейс для алгоритмов построения BVH деревьев

Мы сохраняем не указатели, а смещения от начала массива. Это существенное ограничение. Дело не в том, что GPU “не понимает” наших указателей, а скорее в том, что вопрос распределения памяти в массивно-параллельной системе должен решаться отдельно, и это ограничение принципиальное. Наша ТП **не может** его обойти, т.к. это ограничение ТП нижнего уровня (Vulkan, ISPC, OpenCL, CUDA и т.п). Поэтому мы должны с самого начала строить наши алгоримы так, чтобы распределение памяти в них делалось до начала работы “рассчётной части” (например, можно объявить массив типа std::vector<BVHNode> и зарезервировать память в нём). Поэтому в нашем интерфейсе есть функция *Reserve*.

Как мы обсуждали ранее, алгоритмов построения массивно-параллельного BVH деревьев существует довольно много. Весьма вероятно, что программист захочет реализовать некоторые из них, чтобы сравнивать качество построенного дерева и скорость построения для различных реализаций. Пусть это будут, например, классы LBVH_Hierarchical (соответствующий работе [63]) и LBVH_Karras (соответствующий [60]), которые будут наследоваться от IBVHBuilder и замещать его виртуальные функции. Таким обра-

зом программист сможет проверять и сравнивать между собой работу различных алгоритмов построения дерева в рамках некоторой единой инфраструктуры.

Функции-члены CommitDeviceData и GetExecutionTime в интерфейсе объявлены пустыми. Если их нет, наш транслятор попросит программиста определить эти функции, поскольку он собирается их замещать в сгенерированном классе, производном от LBVH_Karras.

4.1 Алгоритм Карраса

Итак, мы будем обрабатывать только класс LBVH_Karras. Сам алгоритм построения дерева на основе алгоритма Карраса состоит из нескольких шагов, каждый из которых обладает практически неограниченной возможностью для распараллеливания (т.е. производительность его отдельных этапов, как и алгоритма в целом растёт линейно с ростом количества ядер массивно-параллельной системы):

1. Вычисление кодов Мортона [59]. На этом шаге для каждого входного примитива вычисляется трёхмерный пространственный индекс для центров входных примитивов (боксов в нашем примере).

    - На первый взгляд кажется, что это шаг довольно прост. Однако это не так. Для того чтобы вычислить корректный код Мортона, нужно задать ограничивающий все входные примитивы прямоугольный параллелепипед (бокс);
    - Поэтому данный шаг на самом деле начинается с параллельной редукции для вычисления минимумов и максимумов координат по всем входным примитивам, и лишь затем уже только можно вычислить коды Мортона, перемешав по 10 бит (для 32 битного кода) от дискретизованных целочисленных координат по 3 измерениям.

2. Сортировка Кодов Мортона с использованием алгоритмов битонической параллельной либо поразрядной сортировки. На

практике обычно удобно сортировать пары ключ-значение, где ключ – код Мортона, а значение представляет собой изначальный индекс (до сортировки) некоторого объекта в массиве, который мы хотим отсортировать.

3. Формирование листьев дерева. На этом этапе необходимо использовать параллельную префиксную сумму, чтобы объединить все примитивы с одинаковыми кодами Мортона в один лист. Таким образом, в выходном массиве коды Мортона после этого этапа будут уникальны. Кроме того, на этом этапе вычисляются ограничивающие боксы для листьев.

4. Параллельное построение иерархии дерева по алгоритму Карраса [60]. Основная суть этой работы заключается в том, чтобы назначить узлам дерева специальные префиксные коды. Тогда можно построить все уровни дерева параллельно, в отличие от более ранних работ, которые строили их уровень за уровнем снизу-вверх. Здесь есть два нюанса:

   (a) Этот этап является ключевым моментом работы [60], и он не такой простой, каким кажется на первый взгляд. Иерархия должна быть построена таким образом, чтобы коды Мортона, сильно отличающиеся друг от друга, были объединены в один узел лишь на самых верхних уровнях дерева. Если не принимать в расчёт это требование, получится низкокачественное дерево, не пригодное для трассировки лучей, как в работе [65].

   (b) Из-за использования префиксных кодов порядок узлов дереве в памяти фиксирован. Именно поэтому дерево и можно построить быстрее, чем в более ранних подходах.

5. Пересчёт ограничивающих боксов узлов (т.н. refit). На данном шаге необходимо обойти дерево параллельно снизу-вверх для того, чтобы корректно вычислить ограничивающий бокс узла. Это шаг нетривиален из-за предыдущего пункта, а именно порядка узлов в памяти в работе [60]. Он такой, что он

не позволяет разбить массив узлов на непересекающиеся уровни иерархии. И листья, и промежуточные узлы на совершенно разных уровнях иерархии могут располагаться рядом. В целях упрощения изложения мы остановимся на алгоритме пересчёта боксов из работы [60], который обрабатывает все узлы дерева как почти независимые элементы массива в нескольких параллельных проходах. Более эффективное решение этой проблемы (позволяющее пересчитать узлы дерева за 1 параллельный проход) можно найти в работе [66].

4.2 Реализация построения дерева

Объявим далее класс *LBVH_Karras*, реализующий интерфейс *IBVHBuilder*:

```
class LBVH_Karras : public IBVHBuilder  {

  void BuildFromBoxes(const Box4f* in_boxes [[size("boxNum")]],
                      int boxNum,
                      BVHNode* out_tree [[size("boxNum*2-1")]])
                      override;

  void CommitDeviceData() override {}
  void GetExecutionTime(char* name, float time[4]) override;

  size_t Reserve(size_t a_inPrimsNum) override;

protected:

  float4 bboxMin;
  float4 bboxMax;

  std::vector<uint2> runCodesAndIndices;
  std::vector<uint>  codesEq;
  std::vector<uint>  prefixCodeEq;
  std::vector<uint>  compressedCodes;
  std::vector<uint>  newIndices;

  virtual void kernel1D_Reduction(const Box4f* a_boxes,
```

```
                                        uint a_size);
    virtual void kernel1D_EvalMC(const Box4f* a_inBoxes,
                                 uint a_boxCount,
                                 uint2* outRunCodes);

    virtual void kernel1D_AppendInit(uint2* in_runCodes,
                                     uint a_boxCount,
                                     uint* a_codesEq,
                                     uint* a_codesPref);

    virtual void kernel1D_AppendComplete(uint* a_codesEq,
                                         uint a_boxCount,
                                         uint* a_codesPref);

    virtual void kernel1D_MakeLeaves(const Box4f* a_inBoxes,
                                     uint a_boxCount,
                                     BVHNode* a_nodes);


    int delta(int i, int j);
    virtual void kernel1D_Karras12(BVHNode* a_nodes);

    virtual void kernel1D_RefitInit(const BVHNode* a_nodes,
                                    uint2* a_LR);
    virtual void kernel1D_RefitPass(BVHNode* a_nodes,
                                    uint2* a_LR);

};
```

Листинг 3: Реализация алгоритма Карраса в классе

Предопределённые функции. Функция GetExecutionTime(...), в отличие от интерфейса в классе реализации, не пуста. Предполагается, что пользователь в своём классе может самостоятельно измерить время выполнения различных функций на CPU, которое затем можно получить, вызвав функцию следующим образом: GetExecutionTime("BuildFromBoxes", ...). В классе, реализующем алгоритм на GPU, для каждой управляющей функции, подобной BuildFromBoxes, будет автоматически сгенерирован код, необ-

ходимый для измерения времени выполнения алгоритма на GPU (time[0]), времени копирования данных на GPU (time[1]), времени обратного копирования данных (time[2]) и времени накладных расходов на взаимодействие с API (time[3]).

Контракты. В управляющей функции $BuildFromBoxes$ у указателей in_boxes и out_treeData появились конструкции вида $[[size("boxNum")]]$ и $[[size("boxNum * 2 - 1")]]$ [1]. Это единственное расширение, которое нам пришлось добавить в язык. Следует отметить, что это расширение **не является конструкцией параллельного программирования**, а должно рассматриватьтся скорее как **контракт** между пользователем класса и его разработчиком. Разработчик класса обещает, что в функции $BuildFromBoxes$ он прочитает не более boxNum элементов из массива, на который указывает in_boxes, и запишет не более чем boxNum*2-1 элементов в массив, на который указывает out_treeData. Видя подобный контракт в функции BuildFromBoxes, его пользователь, с другой стороны, точно знает, сколько памяти необходимо зарезервировать для входных и выходных данных соответственно. Средства верификации корректности программ могли бы использовать этот контракт для статического или динамического анализа исходного кода, чтобы убедиться, что контракт соблюдается с обеих сторон. Нам же он необходим для того, чтобы корректно сгенерировать код, осуществляющий копирование входных данных на GPU из указателя in_boxes, и обратное копирование результата с GPU в обычную память по указателю out_treeData. Следует дополнительно сказать, что использование подобного контракта не было бы необходимо, если бы мы, например, использовали ссылку на контейнер std::vector для задания входа и выхода, поскольку вектор знает свой размер. В C++ коде, однако, использование указателей нужно очень часто, поэтому мы должны были предоставить возможность программисту исползовать их.

[1]Начиная с 11 стандарта C++ двойные квадратные скобки используются для введения пользовательских расширений в язык

Необходимо отдельно сказать, что наш транслятор обрабатывает все константные указатели как строго входные данные (генерируя для них копирование с CPU на GPU), а все неконстантные как строго выходные (генерируя для них копирование в обратном направлении). Мы не поддерживаем “in-out” указатели в аргументах, т.к., если это нужно, пользователь всегда может передать один и тот же указатель в два разных аргумента.

Данные. Все члены класса, к которым осуществляется доступ из ядер или функций-членов класса, которые вызываются из ядер, будут перемещаться на GPU при вызове пользователем специальной функции *CommitDeviceData*. Переменные, которые представляют из себя т.н. POD-типы (от словосочетания Plain Old Data), будут размещены в специальном буфере m_classData (bboxMin и bboxMax). Каждая переменная типа std::vector будет иметь свою зеркальную копию на GPU в отдельном буфере (VkBuffer). Это переменные runCodesAndIndices, codesEq, prefixCodeEq,compressedCodes, newIndices.

Ядра и функции члены класса. Внутри ядер содержатся циклы, которые предполагают параллельную реализацию на GPU. Из ядер могут вызываться функции-члены класса с любым желаемым уровнем вложенности. В нашем примере это функция *delta* из работы [60], которая принимает два кода Мортона i и j, и возвращает длину общего префикса двух ключей. Рассмотрим далее реализацию первого шага алгоритма построения дерева, описанного в разделе 4.1. В листинге 4 необходимо обратить анимание на несколько вещей.

```
void LBVH_Karras::kernel1D_Reduction(const Box4f* a_boxes,
                                     uint a_size){
  bboxMin = float4(+10000, +10000, +10000, 0);
  bboxMax = float4(-10000, -10000, -10000, 0);
  for (uint boxId = 0; boxId < a_size; boxId++) {
     const float4 triBoxMin = a_boxes[boxId].boxMin;
     const float4 triBoxMax = a_boxes[boxId].boxMax;
```

```
        bboxMin = min(bboxMin, triBoxMin);
        bboxMax = max(bboxMax, triBoxMax);
    }
}
```

Листинг 4: Редукция для вычисления ограничивающего примитивы бокса

Во-первых, наш транслятор разделяет код внутри вычислительного ядра на 3 части: пролог (до параллельного цикла, в котором происходит основная работа), сам цикл и эпилог (код после цикла). Для пролога генерируется код, выполняющийся на GPU в 1 потоке. В данном примере это позволяет нам проинициализировать переменные bboxMin и bboxMax начальными значениями.

Во-вторых, данный код сам по себе сугубо последовательный. Он предполагает, что итерации цикла выполняются одна за другой, а значения переменных bboxMin и bboxMax последовательно изменяются с каждой итерацией. Тем не менее мы знаем, что в конце выполнения цикла результат будет такой же, как если бы мы заменили последовательные выражения на параллельную редукцию. Именно это и делает наш транслятор. Он обнаруживает в исходном коде паттерны редукции (в данном примере это "x = f(x, ...)") и заменяет их на параллельную реализацию на GPU.

В третьих, стоит обратить внимание на условие выхода из цикла (boxId < a_size). Это некоторое выражение, зависящее от аргумента ядра a_size. Для нашего транслятора это тоже важный паттерн. Он говорит нам о том, что значение этого выражения/аргумента приходит к нам "извне", а именно, из управляющей функции. Логика управляющей функции остаётся на CPU, поэтому и значение a_size и будет передано нам со стороны CPU: то есть это обычный запуск ядра. Далее мы увидим пример, где это будет не так (параграф Непрямой вызов ядра).

Управляющие функции содержат логику, запускающую ядра на выполнение. То есть они *управляют* запуском ядер. В нашем примере это функция $BuildFromBoxes$ (листинг 5).

```
void LBVH_Karras::BuildFromBoxes(const Box4f* in_boxes,
                                 uint a_boxCount,
                                 BVHNode* out_tree) {
  // (0) input => (bboxMin, bboxMax)
  kernel1D_Reduction(a_boxes, a_boxCount);
  // (1) input => runCodesAndIndices
  kernel1D_EvalMC(a_boxes, a_boxCount,
                  runCodesAndIndices.data());
  // (2) sort runCodesAndIndices by morton code
  std::sort(runCodesAndIndices.begin(),
            runCodesAndIndices.end(),
            [](uint2 a, uint2 b) { return a.x < b.x; });
  // (3) make leafes array
  // runCodesAndIndices => (codesEq, prefixCodeEq)
  kernel1D_AppendInit(runCodesAndIndices.data(), a_boxCount,
                      codesEq.data(), prefixCodeEq.data());
  std::exclusive_scan(prefixCodeEq.begin(),
                      prefixCodeEq.end(),
                      prefixCodeEq.begin(),0);
  // (codesEq, prefixCodeEq) => (newIndices, compressedCodes)
  kernel1D_AppendComplete(codesEq.data(), a_boxCount,
                          prefixCodeEq.data());
  // a_boxes => a_outNodes
  kernel1D_MakeLeaves(a_boxes, a_boxCount, a_outNodes);
  // (4) Karras algorithm from his paper
  kernel1D_Karras12(out_tree);
  // (5) Refit
  // a_outNodes => runCodesAndIndices (used as temp buffer)
  kernel1D_RefitInit(a_outNodes, runCodesAndIndices.data());
  // (a_outNodes,runCodesAndIndices) =>
  // (a_outNodes,runCodesAndIndices)
  for(int pass = 0; pass < 32; pass++)
    kernel1D_RefitPass(a_outNodes, runCodesAndIndices.data());
}
```

Листинг 5: Управляющая функция, осуществляющая запуск вычислительных ядер и реализующая алгоритм построения дерева при помощи алгоритма Карраса

Управляющие функции могут содержать различную логику: ветвления, вызов других управляющих и обычных функций, вычисление различных выражений. Однако мы накладываем определённое

ограничение на взаимодействие управляющих функций и ядер:

1. Вычисления, производимые в выражениях внутри управляющих функций, всегда производятся на CPU. Их результаты могут передаваться в вычислительные ядра через аргументы, что наряду с запуском самих ядер является некоторой формой “управления”.

2. Передача данных между управляющей функцией и ядром возможно только через аргументы ядра, поскольку передача данных через аргументы является паттерном для нашего транслятора. Нельзя, например, записать значение переменной-члена-класса в управляющей функции, а прочитать его в ядре и наоборот.

3. Ядра могут общаться друг с другом через буферы, передаваемые в аргументы, и через любые данные, являющиеся членами класса.

Эти ограничения вызваны тем фактом, что данные, с которыми работают ядра в сгенерированном коде, находятся на GPU в отдельной области памяти (как бы в своей отдельной песочнице) и, кроме того, алгоритм на GPU в Vulkan может выполняться асинхронно, не ожидая каких-либо вычислений на CPU.

Раскрытие паттернов. Мы уже обсудили шаг вычисления кодов Мортона (**шаг №1**). Сортировка (**шаг №2**), как не трудно заметить, заключается в вызове библиотечной функции $std :: sort$. Наш транслятор обнаруживает вызов библиотечной функции, забирает исходный код лямбда-функции и генерирует соответсвующий набор шейдеров для параллельной битонической сортировки.

Теперь обратим внимание на следующий шаг алгоритма: формирование листьев (**шаг №3**). Он записывается тремя вызовами: $AppendInit$, $exclusive\_scan$, $AppendComplete$ (после чего в ядре $MakeLeaves$ происходит вычисление ограничивающих боксов для листьев, которые записываются в итоговый массив out_treeData).

То, что мы на самом деле хотим сделать на этом шаге, изображено в листинге 6. Это добавление элементов в конец буферов compressedCodes и newIndices, если соседние коды Мортона не равны. Мы могли бы именно так записать этот код, и наш транслятор с успехом преобразовал бы вызовы *push_back* в параллельную операцию добавления элемента в конец буфера при помощи атомарных инструкций.

```
for (uint32_t i = 0; i < prefixCodeEq.size()-1; ++i) {
  if ((runCodesAndIndices[i].x!=runCodesAndIndices[i+1].x))){
    compressedCodes.push_back(runCodesAndIndices[i].x);
    newIndices.push_back(i);
  }
}
```

Листинг 6: Логика алгоритма формирования листьев

К сожалению, это не то, что нам нужно, т.к. нам не только необходимо добавить элементы в конец буфера, но ещё крайне важно **сохранить их порядок**. В теории мы могли бы сгенерировать код, реализующий параллельное добавление элементов в буфер при помощи параллельной префиксной суммы (на наш взгляд, это вопрос дизайна ТП, и это можно сделать). Однако в данном случае, чтобы не запутывать программиста и не нарушать концепцию белого ящика для нашего транслятора, мы приняли более прозрачное решение: “push_back” трансформируется в параллельную операцию добавления элементов в буфер, а если необходимо сохранить порядок элементов, программист должен записать алгоритм добавления элементов в буфер **явно через префиксную сумму** (листинг 6).

```
void LBVH_Karras::kernel1D_AppendInit(uint2* in_runCodes,
                                      uint a_boxCount,
                                      uint* a_codesEq,
                                      uint* a_codesPref) {

  for (uint i = 0; i < a_boxCount; ++i) {
    uint codeEQ = 1;
    if(i > 0)
      codeEQ = (in_runCodes[i - 1].x!=in_runCodes[i].x)?1:0;
```

```
    a_codesEq   [i] = codeEQ;
    a_codesPref[i] = codeEQ;
  }
}

void LBVH_Karras::kernel1D_AppendComplete(uint* a_codesEq,
                                          uint* a_codesPref,
                                          uint a_boxCount) {

  uint leafNumber = a_codesPref[a_boxCount - 1] + \
                    a_codesEq   [a_boxCount - 1];
  newIndices.resize(leafNumber);
  compressedCodes.resize(leafNumber);

  for (uint i = 0; i < a_boxCount; i++){
    // Parallel Append, transformed via prefix summ
    if (a_codesEq[i] != 0) {
      const uint writeId = a_codesPref[i];
      // newIndices.push_back(i);
      newIndices      [writeId] = i;
      // compressedCodes.push_back(runCodesAndIndices[i].x);
      compressedCodes[writeId] = runCodesAndIndices[i].x;
    }
  }
}
```

Листинг 7: Алгоритма формирования листьев, разбитый на 2 вычислительных ядра

В листинге 7 необходимо обратить внимание на пролог ядра *AppendComplete*. В нём происходит вычисление числа листьев дерева на основании значений последних элементов массивов codesEq и prefixCodeEq. Наш транслятор хранит в буфере m_classData в памяти GPU размер (size) и вместимость (capacity) для каждого вектора, преобразованного в VkBuffer. Поэтому вызов $newIndices.resize()$ транслируется в изменение значения определённой переменной в памяти GPU (назовём её newIndices_size). Это важный момент, который мы будем использовать далее.

Непрямой вызов ядра и динамический параллелизм. Ядро $Karras12$ является непосредственной реализацией алгорритма Карраса из работы [60] (**шаг №4**). По этой причине мы не будем приводить его здесь целиком. Однако обратим внимание на интересную деталь, присутствующую в нём и в последующих ядрах ($RefitInit$ и $RefitPass$), а именно условие выхода из цикла (листинг 8).

```
void LBVH_Karras::kernel1D_Karras12(BVHNode* a_nodes) {

  for (int tid = 0; tid < newIndices.size()-1; tid++) {
    int d = sign(delta(tid, tid+1) - delta(tid, tid-1));
    int delta_min = delta(tid, tid - d);
    ...
    a_nodes[tid].leftOffset  = ... ;
    a_nodes[tid].rightOffset = ... ;
  }
}
```

Листинг 8: Логика алгоритма Карраса

В листинге 8 условие выхода из цикла зависит от выражения, которое определяется значением переменной, являющейся членом класса (размер вектора newIndices или переменная newIndices_size). Поскольку члены класса могут быть модифицированы ядрами, это означает, что для GPU реализации значение переменной newIndices_size будет известно только в момент выполнения кода на GPU. То есть это не обычный вызов ядра, а так называемый непрямой вызов, который в Vulkan может быть реализован с помощью vkDispatchIndirect. Количество потоков, которые будут запущены на выполнение, определяется предыдущими вычислениями, произведёнными GPU в ядре $AppendComplete$, которое изменяет размер вектора.

Refit (шаг №5) является более-менее тривиальной, хотя и не самой эффективной и очевидной реализацией параллельного обхода дерева в ширину снизу-вверх. Он начинает работу с листьев дерева и затем с каждым проходом (вызовом ядра $RefitPass$) продвигается выше на 1 уровень дерева. При этом для всех остальных уровней

дерева осуществляется ранний выход из ядра, т.к. не выполняется условие $if(lr.x! = -1)$.

```
void LBVH_Karras::kernel1D_RefitInit(const BVHNode* a_nodes,
                                     uint2* a_leftRight) {
  for(int idx = 0; idx < newIndices.size()-1; idx++) {
    BVHNode node      = a_nodes[idx];
    a_leftRight[idx] = uint2(node.leftOffset,
                             node.rightOffset);
  }
}

void LBVH_Karras::kernel1D_RefitPass(BVHNode* a_nodes,
                                     uint2* a_leftRight) {
  for(int idx = 0; idx < newIndices.size()-1; idx++) {
  const uint2 lr = a_leftRight[idx];
  if(lr.x != -1) { // early exit from kernel
    uint2 lr_left  = uint2(-1, -1);
    uint2 lr_right = uint2(-1, -1);
    if(int(lr.x) < int(newIndices.size()) - 1)
      lr_left  =  a_leftRight[lr.x];
    if(int(lr.y) < int(newIndices.size()) - 1)
      lr_right = a_leftRight[lr.y];
    if(lr_left.x == -1 && lr_right.x == -1) {
       BVHNode leftNode  = a_nodes[lr.x];
       BVHNode rightNode = a_nodes[lr.y];
       BVHNode currNode;
       currNode.boxMin=min(leftNode.boxMin,rightNode.boxMin);
       currNode.boxMax=max(leftNode.boxMax,rightNode.boxMax);
       currNode.leftOffset  = lr.x;
       currNode.rightOffset = lr.y;
       a_nodes     [idx] = currNode;
       a_leftRight[idx] = uint2(-1, -1); // node is processed
       if(idx == 0) // processed the root, stop other passes
         newIndices.resize(0);
      }
    }
  }
}
```

Листинг 9: Вычисление ограничивающих боксов

В листинге 9 для каждого узла во временном массиве (для

которого теперь используется вектор runCodesAndIndices) сохраняются ссылки на дочерние узлы дерева, которые затем с каждым проходом ядра $RefitPass$ помечаются как обработанные ($a\_leftRight[idx] = uint2(-1, -1)$), если для текущего узла был вычислен ограничиваюший его бокс. Обратим внимание на инструкцию $newIndices.resize(0)$ в конце алгоритма. Она изменяет размер массива newIndices (значение переменной newIndices_size), от чего зависит выполнение самого ядра $RefitPass$. Это некоторая оптимизация. Поскольку мы не можем заранее знать глубину дерева, нам приходится отправлять на запуск ядро $RefitPass$ некоторое достаточно большое число раз (32 в нашем случае, т.к. мы используем 32-битные коды Мортона). Однако, если на каком-то проходе мы достигли корня дерева, последующие запуски ядра не нужны. Наш транслятор корректно обрабатывает такую ситуацию, поскольку изменение значения переменной newIndices_size ведёт и к изменению соотвествующего значения в т.н. indirectBuffer, который в свою очередь и определяет, сколько потоков необходимо запускать на выполнение. Если это число становится равным нулю, драйвер имеет право пропустить все последующие вызовы ядра $RefitPass$.

4.3 Обсуждение

Отметим ряд особенностей предложенной технологии программирования, с которыми мы встретились при реализации алгоритма параллельного построения дерева.

1. Исходный код построения дерева написан на обычном C++11 и компилируется любым поддерживающим этот язык компилятором (а при незначительных изменениях даже и C++98).

2. Мы сохранили описание алгоритма в относительно чистом однопоточном виде, не используя ни одной конструкции параллельного программирования и ни одной параллельной директивы. Хотя наш транслятор не запрещает добавлять такие директивы (например, OpenMP).

3. Тем не менее сам алгоритм построения дерева нам пришлось сделать параллельным в нескольких местах: это параллельное добавление элементов в буфер, которое нам пришлось записать явно при помощи префиксной суммы (листинг 7), алгоритм Карраса [60], который мы использовали для формирования иерархии дерева и, наконец, параллельный алгоритм обхода дерева в ширину снизу-вверх для вычисления ограничивающих боксов в узлах дерева (листинг 9). Можно ли избежать подобных преобразований?

   (a) Мы показали, что это возможно для параллельной редукции, автоматически превратив однопоточный алгоритм в его параллельный аналог.

   (b) Мы считаем, что при желании это можно сделать как для добавления элементов в буфер, так и для обхода дерева, и для других "параллелизуемых" алгоритмов, которые можно автоматически обнаруживать в пользовательском коде. Однако если параллельная редукция, префиксная сумма и сортировка являются простыми паттернами, которые почти всегда записываются одинаковым образом, то по крайней мере обход дерева уже не является тривиальным примером. Например, наша первоначальная версия пересчёта боксов обходила дерево в глубину, а параллельная версия по факту уже обходит его в ширину. Причём, если бы у нас была возможность по-другому расположить узлы дерева в памяти (по уровням дерева в ширину), мы могли бы сделать алгоритм обхода на последнем шаге более эффективным. К сожалению, в рассмотренном примере это невозможно, т.к. расположение узлов в памяти продиктовано алгоритмом Карраса.

   (c) Таким образом, этот вопрос дискуссионный. Поскольку мы позиционируем свою технологию программирования как ориентированную на производительность, мы стараемся выполнять меньше неявных преобразований и предо-

ставить пользователю более прозрачное решение.

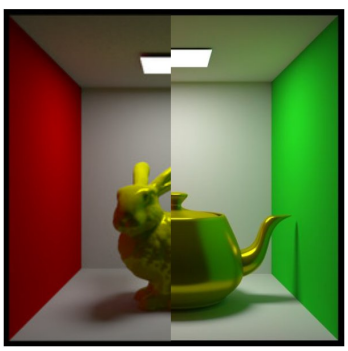
4. По сравнению с технологиями, основанными на директивах, в нашем подходе разработчик более явно управляет копированием данных. В нашей технологии входные и выходые данные алгоритма чётко отделены *естественными границами управляющих функций*, их аргументами. Эти данные будут копироваться между CPU и GPU. Все переменные, буферы и текстуры, объявленные внутри класса, будут располагаться на GPU, и таким образом гарантируется отсутствие лишних копирований. В технологиях, основанных на параллельных директивах, копирование либо происходит неявно, либо задаётся в виде отдельных прагм-директив, что несколько труднее контролировать, поскольку программисты начинают об этом задумываться на более поздней стадии разработки (когда уже оценивают производительность), в результате чего легко может получиться код, где доступ со стороны CPU и GPU постоянно чередуется.

5. Наконец, при желании, отладив алгоритм с данными, находящимися в обычной памяти CPU, программист имеет возможность интегрировать его в код остальной программы, которая была написана вручную на технологии программирования нижнего уровня и таким образом полностью избегать копирования.

5 Результаты

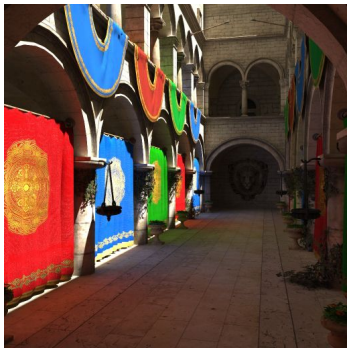
Мы реализовали алгоритм построения и обхода BVH дерева на нашей технологии в задаче трассировки лучей на нескольких трёхмерных сценах (рис. 3) и провели сравнения обоих алгоритмов с реализацией быстрого построения дерева в библиотеке Embree (HLBVH [62, 64]) от компании Intel (таблицы 1, 2) и реализацией того же самого, что и у нас, алгоритма построения LBVH по Каррасу из Hippie [67] от сотрудников компании AMD, выполненной на CUDA/HIP.

На CPU в нашей версии алгоритма и для Embree мы использовали OpenMP для автоматического распараллеливания циклов.



bunny/teapot

sponza



cry-sponza

conference

dragon

car

Рис. 3. Изображения тестовых сцен с ракурса, который использовался в алгоритме трассировки лучей. Сами изображения получены Монте-Карло трассировкой путей.

Во всех таблицах в колонках “Наш (CPU)” и “Наш (GPU)” указано время трассировки лучей на основе некоторой нашей программной реализации интерфейса ISceneObject на CPU (реализована на) и GPU (сгенерирована автоматически) соответственно. В колонке “Наш (RTX)” показано время на основе аппаратной трассировки лучей, сгенерированной нашим транслятором в том же тестовом программном окружении.

5.1 Сравнение с Embree

Embree [49] является популярной библиотекой трассировки лучей, развиваемой компанией Intel на протяжении многих лет. Такие задачи, как построение дерева и его обход лучами, оптимизированы в этой библиотеке в достаточно высокой степени, поэтому при сравнении с реализацией на CPU Embree является хорошей точкой отсчёта.

Как мы обсуждали ранее, построение дерева является сложной и неудобной задачей для GPU, упирающейся в пропускную способность подсистемы памяти. Обход дерева в трассировке лучей, напротив, легко параллелизуется и хорошо ускоряется (хотя также зачастую упирается в пропускную способность памяти). Поэтому, для того чтобы иметь представления о причинах высокой или низкой производительности, наряду с построением дерева (таблицы 1, 2, 5), мы сравнивали и скорость его обхода лучами (таблицы 3, 4, 6).

Сцена	N прим.	Embree	Наш (CPU)	Наш (GPU)	ускор.
teapot	25K	6.67 мс	5.48 мс	**1.41 мс**	3.9
sponza	66K	7.48 мс	9.71 мс	**2.47 мс**	3.9
bunny	144K	13.5 мс	21.6 мс	**4.54 мс**	4.8
cry-sponza	243K	14.6 мс	30.0 мс	**4.80 мс**	6.2
dragon	871K	61.7 мс	149 мс	**20.2 мс**	7.3
car	1.6M	88.0 мс	199 мс	**35.4 мс**	5.6

Таблица 1. Сравнение с Embree. Время построения дерева на стационарной ЭВМ по алгоритму LBVH. Оборудование CPU: Intel Core i7-9700K 3.60GHz (8 ядер); GPU: Nvidia RTX2070. ОС Ubuntu Linux 23.

Сцена	N прим.	Embree	Наш (CPU)	Наш (GPU)	ускор.
teapot	25K	9 мс	7.7 мс	**4.6 мс**	1.6
sponza	66K	13.9 мс	**11.8 мс**	14.9 мс	0.8
bunny	144K	**31.3 мс**	34.0 мс	32.4 мс	1.1
cry-sponza	243K	40.71 мс	42.8 мс	**35.0 мс**	1.2
dragon	871K	120 мс	165 мс	**109 мс**	1.5
car	1.6M	265 мс	379 мс	**210 мс**	1.8

Таблица 2. Сравнение с Embree. Время построения дерева на мобильной ЭВМ по алгоритму LBVH. Оборудование CPU: Qualcomm Snapdragon 8 Gen 1 (arm64, 8 ядер); GPU: Adreno 730. ОС Android 13 Tiramisu.

Сцена	N прим.	Embree	Наш (CPU)	Наш (GPU)	Наш (RTX)
teapot	25K	84 мс	123 мс	2.4 мс	**2.0 мс**
sponza	66K	126 мс	365 мс	7.0 мс	**2.5 мс**
bunny	144K	87 мс	161 мс	2.71 мс	**2.1 мс**
cry-sponza	243K	177 мс	486 мс	14.3 мс	**3.4 мс**
dragon	871K	138 мс	186 мс	4.7 мс	**1.9 мс**
car	1.6M	118 мс	319 мс	8.6 мс	**2.4 мс**

Таблица 3. Сравнение с Embree. Время обхода дерева для примерно 4 миллионов (2048x2048) первичных лучей на стационарной ЭВМ для дерева, построенного при помощи алгоритма LBVH. Оборудование CPU: Intel Core i7-9700K 3.60GHz (8 ядер); GPU: Nvidia RTX2070. ОС Ubuntu Linux 23.

Сцена	N прим.	Embree	Наш (CPU)	Наш (GPU)
teapot	25K	159 мс	202 мс	**13.0 мс**
sponza	66K	223 мс	490 мс	**33.8 мс**
bunny	144K	135 мс	207 мс	**15.8 мс**
cry-sponza	243K	357 мс	860 мс	**69.0 мс**
dragon	871K	163 мс	215 мс	**44.0 мс**
car	1.6M	212 мс	527 мс	**48.9 мс**

Таблица 4. Сравнение с Embree. Время обхода дерева для примерно 4 миллионов (2048x2048) первичных лучей на мобильной ЭВМ для дерева, построенного при помощи алгоритма LBVH.. Оборудование CPU: Qualcomm Snapdragon 8 Gen 1 (arm64, 8 ядер); GPU: Adreno 730. ОС Android 13 Tiramisu.

5.2 Сравнение с Hippie

Hippie [67] представляет собой небольшой фрейм-ворк (программное окружение) трассировки лучей, построенный сотрудниками AMD Daniel Meister и Jiří Bittner в 2022 году на базе работы сотрудника Nvidia Timo Aila 2009 года [68]. Hippie реализует различные алгоритмы построения и обхода дерева на GPU. Для наших сравнений мы брали алгоритм построения и обхода LBVH дерева по алгоритму Карраса 2012 года [60]. Сам фрейм-ворк реализован таким образом, что его можно собрать как для Nvidia CUDA, так и для AMD HIP.

Сцена	N прим.	Hippie (CUDA)	Наш (GPU)
teapot	25K	2.2 мс	**1.4 мс**
conference	144K	**3.6 мс**	4.2 мс
cry-sponza	243K	**4.5 мс**	4.8 мс
dragon	871K	**12 мс**	20 мс

Таблица 5. Сравнение с Hippie. Время построения дерева на стационарной ЭВМ по алгоритму LBVH. Оборудование CPU: Intel Core i7-9700K 3.60GHz (8 ядер); GPU: Nvidia RTX2070. ОС Ubuntu Linux 23.

Сцена	N прим.	Hippie (CUDA)	Наш (GPU)	Наш (RTX)
teapot	25K	2.6 мс	2.4 мс	**2.0 мс**
conference	280K	6.00 мс	6.20 мс	**3.0 мс**
cry-sponza	243K	11.2 мс	14.3 мс	**3.4 мс**
dragon	871K	4.49 мс	4.70 мс	**1.9 мс**

Таблица 6. Сравнение с Hippie. Время обхода дерева для примерно 4 миллионов (2048x2048) первичных лучей на ноутбуке. Для дерева, построенного при помощи алгоритма LBVH. Оборудование GPU: Nvidia RTX2070. ОС Windows 11.

5.3 Обсуждение результатов

Для начала обратим внимание на скорость построения дерева. Наша реализация построения дерева на GPU, полученная автоматически из C++ кода, обладает примерно такой же скоростью построения дерева, как и оптимизированная вручную реализация на CUDA из Hippie [67] (таблица 5).

Некоторое различие в скорости построения обусловлено тем, что Hippie использует более оптимальный алгоритм поразрядной сортировки со сложностью $O(N * k)$, чем наша реализация, использующая битоническую сортировку — $O(N * log^2(N))$ (аналогично и при сравнении с Embree нашего алгоритм на CPU). Кроме того, одним из основных недостатков битонической сортировки является необходимость увеличивать размер сортируемого массива до ближайшей степени двойки. Поэтому для 243 тысячи примитивов (cry-sponza) мы сортируем 256 тысяч примитивов, а уже для 280 тысяч примитивов (conference) мы вынуждены сортировать 512 тысяч элементов. Аналогично для 66 тысяч примитивов (sponza) нам приходится сортировать 128 тысяч элементов, что объясняет некоторое падение производительности на сцене sponza в таблицах (1, 2). Битоническая сортировка удобна, т.к. она использует произвольный оператор сравнения в лямбда-выражении, однако её использование не является

принципиальным ограничением нашей технологии программирования. Мы могли бы добавить генерацию кода поразрядной сортировки для некоторых типов данных.

Для GPU в стационарной машине (таблица 1) нам удаётся добиться ускорения в 4–7 раз по сравнению с параллельной CPU версией, что более-менее кореллирует с разницей в пропускной способности подсистемы памяти CPU и GPU. На мобильном GPU (таблица 2) при этом существенного ускорения добиться не получается, что, на наш взгляд, объясняется более слабой подсистемой памяти мобильных GPU и, как мы уже обсуждали, некоторой потерей эффективности в сортировке.

Ситуация меняется для обхода дерева в трассировке лучей. Здесь уже есть ощутимое количество арифметических операций. Поэтому по сравнению с CPU реализацией можно добиться ускорения до 40 раз для программной и до 130 раз для аппаратной реализации на GPU для первичных лучей на стационарной машине и 10-15 раз на мобильном GPU.

При этом, как и в случае с построением дерева, скорость нашей программной реализации на GPU сравнима с оптимизированной реализацией в Hippie (таблица 6). Некоторая разница в скорости обусловлена тем, что в Hippie трассировка лучей реализована в одноуровневом дереве, в то время как в нашей реализации дерево двух-уровневое. Кроме того, в Hippie данные вершин треугольников хранятся в буфере последовательно, а в нашей реализации доступ к ним осуществляется по индексам, что добавляет одно косвенное обращение в память при чтении каждой вершины треугольника.

6 Выводы

Мы предложили решение фундаментального противоречия между кросс-платформенностью и аппаратным ускорением. Мы продемонстрировали как, это можно сделать на примере трассировки лучей: вызовы виртуальных функций определённого интерфейса заменяются на аппаратно ускоренную реализацию трассировки лучей в

шейдерах. Если же такая реализация недоступна на целевом GPU, программист может определить свою собственную реализацию интерфейса, которая не только будет работать на CPU, но и успешно оттранслируется в шейдеры на GPU, заместив неподдерживаемую аппаратную реализацию. Мы также показали, что нашу технологию программирования можно применить на довольно сложном алгоритме построения и обхода дерева, который включает в себя различные паттерны параллельного программирования: редукцию, сортировку, префиксную сумму, динамический паралеллизм. Производительность полученного решения (как построения дерева, так и его обхода) в среднем отстаёт не более чем на 5% от аналогичного решения, написанного на CUDA и оптимизированного вручную квалифицированными специалистами.

Список литературы

[1] NVIDIA. *NVC++: a C++17 compiler for NVIDIA GPUs and AMD, Intel, OpenPOWER, and Arm CPUs.* //

[2] NVIDIA, Vingelmann, P. & Fitzek, F.H.P., 2020. CUDA, release: 10.2.89, Available at: https://developer.nvidia.com/cuda-toolkit.

[3] "OpenACC", 2021 URL: https://www.openacc.org/

[4] Jacobsen, Niklas. "LLVM supported source-to-source translation-Translation from annotated C/C++ to CUDA C/C++." MS thesis. University of Oslo, 2016.

[5] Balogh, Gabor Daniel, et al. "Op2-clang: A source-to-source translator using clang/llvm libtooling", IEEE/ACM 5th Workshop on the LLVM Compiler Infrastructure in HPC (LLVM-HPC). IEEE, 2018.

[6] Yang, Po, et al. "Improving utility of GPU in accelerating industrial applications with user-centered automatic code translation", IEEE Transactions on Industrial Informatics 14.4, 2017, 1347-1360.

[7] Pienaar, Jacques A., Srimat Chakradhar, and Anand Raghunathan. "Automatic generation of software pipelines for heterogeneous parallel systems", SC'12: Proceedings of the International Conference on HPC, Networking, Storage and Analysis. IEEE, 2012.

[8] Wenzel Jakob, Sebastien Speierer, Nicolas Roussel, and Delio Vicini. 2022. DR.JIT: a just-in-time compiler for differentiable rendering. ACM Trans. Graph. 41, 4, Article 124 (July 2022), 19 pages. https://doi.org/10.1145/3528223.3530099

[9] Bakhtin V., Krukov V. "DVM-Approach to the Automation of the Development of Parallel Programs for Clusters." Programming and Computer Software. 45. 121-132. 10.1134/S0361768819030034.

[10] Ohberg, Tomas. "Auto-tuning Hybrid CPU-GPU Execution of Algorithmic Skeletons in SkePU", MS thesis. Linkoping University, 2018.

[11] Ernstsson, August, Lu Li, and Christoph Kessler. "SkePU 2: Flexible and type-safe skeleton programming for heterogeneous parallel systems", International Journal of Parallel Programming 46.1, 2018, 62-80.

[12] Steuwer, Michel, Philipp Kegel, and Sergei Gorlatch. "Skelcl-a portable skeleton library for high-level gpu programming", 2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum. IEEE, 2011.

[13] SYCL, cross-platform abstraction layer, 2021. URL: https://www.khronos.org/sycl/

[14] Han, Tianyi David, and Tarek S. Abdelrahman. "hiCUDA: High-level GPGPU programming", IEEE Transactions on Parallel and Distributed systems 22.1, 2010, 78-90.

[15] Wu, Jingyue, et al. "gpucc: an open-source GPGPU compiler", Proceedings of the 2016 International Symposium on Code Generation and Optimization. 2016.

[16] Sathre, Paul, Mark Gardner, and Wu-chun Feng. "On the portability of cpu-accelerated applications via automated source-to-source translation", Proceedings of the International Conference on High Performance Computing in Asia-Pacific Region. 2019.

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

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

[19] Mammeri, Nadjib, and Ben Juurlink. "Vcomputebench: A vulkan benchmark suite for gpgpu on mobile and embedded gpus", IEEE International Symposium on Workload Characterization (IISWC). IEEE, 2018.

[20] Ragan-Kelley, Jonathan, et al. "Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines". ACM Sigplan Notices 48.6, 2013, 519-530.

[21] Adams, Andrew, et al. "Learning to optimize halide with tree search and random programs", ACM Trans. Graph. (TOG)38.4,2019:1-12.

[22] Haidl, Michael, and Sergei Gorlatch, "PACXX: Towards a unified programming model for programming accelerators using C++14", LLVM Compiler Infrastructure in HPC. IEEE, 2014.

[23] Haidl, Michael, et al. "Pacxxv2+ RV: an LLVM-based portable high-performance programming model", Proceedings of the Fourth Workshop on the LLVM Compiler Infrastructure in HPC, 2017.

[24] Sean Baxter. "Circle C++ shaders", 2021. URL: https://github.com/seanbaxter/shaders

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

[26] “Inja, template engine for modern C++”, 2021. URL: https://github.com/pantor/inja

[27] Sidelnik, Albert, et al, “Performance portability with the chapel language”, IEEE 26th international parallel and distributed processing symposium. IEEE, 2012.

[28] Baghdadi, Riyadh, et al. “Tiramisu: A polyhedral compiler for expressing fast and portable code”, IEEE/ACM International Symposium on Code Generation and Optimization (CGO), IEEE, 2019.

[29] “Nvidia OptiX”, 2021, URL: https://developer.nvidia.com/optix

[30] MS “DirectML”, 2021: https://github.com/microsoft/DirectML

[31] Hegarty, James, et al. “Darkroom: compiling high-level image processing code into hardware pipelines”, ACM Trans.Graph.33.4(2014):144-1.

[32] Rasch, Ari, Richard Schulze, and Sergei Gorlatch, “Developing High-Performance, Portable OpenCL Code via Multi-Dimensional Homomorphisms”, Proceedings of the International Workshop on OpenCL, 2019.

[33] Huang, Tsung-Wei, et al, “Taskflow: A General-purpose Parallel and Heterogeneous Task Programming System”, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 2021.

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

[35] Huang, Shan Shan, David Zook, and Yannis Smaragdakis. “Morphing: Safely shaping a class in the image of others”, European Conference on Object-Oriented Programming, Springer, Berlin, Heidelberg, 2007.

[36] Parker, Steven G., et al. “Optix: a general purpose ray tracing engine”, ACM transactions on graphics (tog) 29.4 (2010): 1-13.

[37] Laine, Samuli, Tero Karras, and Timo Aila. “Megakernels considered harmful: Wavefront path tracing on GPUs”, Proceedings of the 5th High-Performance Graphics Conference. 2013.

[38] Matt Pharr, William R. Mark. “spc: A SPMD Compiler for High-Performance CPU Programming”

[39] Kerry A. Seitz Jr., Theresa Foley, Serban D. Porumbescu, John D. Owens. “Supporting Unified Shader Specialization by Co-opting C++ Features”. arXiv:2109.14682

[40] Frolov Vladimir, Sanzharov Vadim, Galaktionov Vladimir, Scherbakov Alexandr. An auto-programming approach to Vulkan. // Proceedings of the 31th International Conference on Computer Graphics and Machine Vision, CEUR Workshop Proceedings, Vol 3027, pp. 150-165. URL: http://ceur-ws.org/Vol-3027/paper14.pdf

[41] “Taichi Programming Language”. https://www.taichi-lang.org/

[42] “Numba”. JIT compiler. https://numba.readthedocs.io/en/stable/

[43] Nvidia. “NVIDIA HPC Fortran, C++ and C Compilers with OpenACC”.

[44] J. Gold. “OneAPI: Software Abstraction for a Heterog. Comp. World”.

[45] Wenzel Jakob, Sebastien Speierer, Nicolas Roussel, and Delio Vicini. 2022. “DR.JIT: a just-in-time compiler for differentiable rendering.” ACM Trans. Graph. 41, 4, Article 124 (July 2022), 19 pages. https://doi.org/10.1145/3528223.3530099

[46] Yuanming Hu, Luke Anderson, Tzu-Mao Li, Qi Sun, Nathan Carr, Jonathan Ragan-Kelley, Fredo Durand. “DiffTaichi: Differentiable Programming for Physical Simulation.” ICLR 2020.

[47] Tzu-Mao Li, Michael Gharbi, Andrew Adams, Fredo Durand, and Jonathan Ragan-Kelley. “Differentiable programming for image processing and deep learning in halide.” ACM Trans. Graph. 37, 4, Article 139, 2018, 13 pages. https://doi.org/10.1145/3197517.3201383

[48] Reinhard, E., Ward, G., Pattanaik, S., and Debevec, P. “High Dynamic Range Imaging: Acquisition, Display and Image-Based Lighting”, 2005, chap. 6, pp. 187–221. Morgan Kaufmann Publishers Inc., first edition.

[49] Ingo Wald, Sven Woop, Carsten Benthin, Gregory S. Johnson, and Manfred Ernst. “Embree: a kernel framework for efficient CPU ray tracing.” ACM Trans. Graph. 33, 4, Article 143 (2014), 8 pages. https://doi.org/10.1145/2601097.2601199

[50] Ray Tracing Systems, Keldysh Institute of Applied Mathematics, Moscow State Uiversity. “Hydra Renderer. Open source rendering system”, 2019. https://github.com/Ray-Tracing-Systems/HydraAPI

[51] Frolov Vladimir, Sanzharov Vadim, Galaktionov Vladimir. kernel_slicer. https://github.com/Ray-Tracing-Systems/kernel_slicer

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

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

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

[55] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman,

Nicolas Vasilache, Oleksandr Zinenko. *MLIR: A Compiler Infrastructure for the End of Moore's Law* // arXiv:2002.11054

[56] KhronosGroup. glslangValidator // reference shader compiler for Vulkan. URL = https://github.com/KhronosGroup/glslang

[57] Google. Shaderc // A collection of tools, libraries and tests for shader compilation. URL = https://github.com/google/shaderc

[58] Kevin Petit. *A prototype implementation of OpenCL 3.0 on top of Vulkan using clspv as the compiler.* // https://github.com/kpet/clvk

[59] Morton G.M. *A Computer Oriented Geodetic Data Base: and a New Technique in File Sequencing* // Research Report IBM Ltd., Ottawa, ON, Canada, 1966.

[60] Tero Karras. Maximizing Parallelism in the Construction of BVHs, Octrees, and k-d Trees. // High Performance Graphics (2012)

[61] Tero Karras and Timo Aila. *Fast parallel construction of high-quality bounding volume hierarchies.* // In Proceedings of the 5th High-Performance Graphics Conference (HPG '13). 2013. Association for Computing Machinery, New York, NY, USA, 89–99. https://doi.org/10.1145/2492045.2492055

[62] Christian Lauterbach, Michael Garland, Shubhabrata Sengupta et al. *Fast BVH Construction on GPUs* // Computer Graphics Forum. — 2009.

[63] Yan Gu, Yong He, Kayvon Fatahalian, Guy Blelloch. *Efficient BVH Construction via Approximate Agglomerative Clustering* // Proceedings of the 5th High-Performance Graphics Conference. — HPG '13. — Association for Computing Machinery, 2013. — P. 81–88. https://doi.org/10.1145/2492045.2492054.

[64] Kirill Garanzha, Jacopo Pantaleoni, and David McAllister. *Simpler and faster HLBVH with work queues.* // In Proceedings of the ACM

SIGGRAPH Symposium on High Performance Graphics (HPG '11). 2011. Association for Computing Machinery, New York, NY, USA, 59–64. https://doi.org/10.1145/2018323.2018333

[65] Chitalu, Francis M. *Binary Ostensibly-Implicit Trees for Fast Collision Detection* // Computer Graphics Forum. — 2020. https://doi.org/10.1111/cgf.13948.

[66] Ciprian Apetrei. *Fast and Simple Agglomerative LBVH Construction* // EG UK Computer Graphics and Visual Computing (2014).

[67] Daniel Meister and Jiri Bittner. *Performance Comparison of Bounding Volume Hierarchies for GPU Ray Tracing* // Journal of Computer Graphics Techniques (JCGT), vol. 11, no. 4, 1-19, 2022. Available online http://jcgt.org/published/0011/04/01/

[68] Timo Aila and Samuli Laine. *Understanding the Efficiency of Ray Traversal on GPUs.* // Proc. High-Performance Graphics 2009.