Вычислительные технологии, 2024, том 29, № 4, с. 95–109. © ФИЦ ИВТ, 2024
Computational Technologies, 2024, vol. 29, no. 4, pp. 95–109. © FRC ICT, 2024
ISSN 1560-7534
eISSN 2313-691X

ИНФОРМАЦИОННЫЕ ТЕХНОЛОГИИ

DOI:10.25743/ICT.2024.29.4.007

Тестирование операций графической обработки с эмуляцией графических ускорителей

А. М. Гиацинтов*, К. А. Мамросенко
Федеральный научный центр Научно-исследовательский институт системных исследований РАН, 117218, Москва, Россия
*Контактный автор: Гиацинтов Александр Михайлович, e-mail: giatsintov@niisi.ras.ru
Поступила 20 октября 2023 г., доработана 13 ноября 2023 г., принята в печать 20 ноября 2023 г.

Важными аспектами работы с графическими операциями являются достижение корректности выполнения операций и отладка подсистем и драйверов, связанных с отображением графической информации. В статье представлена архитектура эмулятора графического ускорителя, предназначенного для диагностики и последующего исправления сложно выявляемых ошибок в компонентах графической подсистемы. В отличие от известных решений, в архитектуре предусматривается возможность применения программного обеспечения как в составе эмуляторов аппаратного обеспечения системы в целом типа QEMU, так и в виде присоединяемой библиотеки к драйверу графического ускорителя. Применение эмулятора позволило обнаружить некоторые ошибки, которые на реальной аппаратуре имели характер “плавающих” и были тяжело воспроизводимы.

Ключевые слова: визуализация, графическая система, рендеринг, операционная система, эмуляция, QEMU.

Цитирование: Гиацинтов А.М., Мамросенко К.А. Тестирование операций графической обработки с эмуляцией графических ускорителей. Вычислительные технологии. 2024; 29(4):95–109. DOI:10.25743/ICT.2024.29.4.007.

Введение

Сегодня все большее число задач требует использования аппаратных ускорителей для уменьшения времени, необходимого для вычислений, а также для отображения результатов вычислений. Одним из значимых направлений является обработка графической информации на встраиваемых системах, используемых в авиационной, автомобильной и других отраслях промышленности. В качестве операционной системы во встраиваемых системах стал активно применяться Linux, например в автомобилях. Однако существует ряд встраиваемых систем, для которых важно своевременно реагировать на возникающие события. Как правило, для таких целей используются операционные системы реального времени (ОСРВ). В настоящей статье рассматривается отечественная ОСРВ [1], которая базируется на следующих общих принципах:

- использование стандартов;
- мобильность;
- развитие средств протоколирования диагностики и обработки ошибок;
- гибкие средства планирования;
- использование объектно-ориентированного подхода;
- управляемость (в частности, наличие средств конфигурирования);

– наличие кросс-средств разработки и отладки пользовательских приложений;
– значительное число пакетов окружения для создания графических приложений, баз данных, картографических систем.

При разработке этой операционной системы использованы международные стандарты и спецификации:

– стандарт C, описывающий язык и библиотеки языка C;
– стандарт POSIX 1003.1 на мобильные операционные системы (программный интерфейс) [2];
– спецификация ARINC 653 [3], определяющая интерфейс APEX (application executive) между операционной системой целевого модуля и прикладными программами [4].

Важными аспектами работы с графическими операциями являются достижение корректности их выполнения и отладка подсистем и драйверов отображения графической информации. Учитывая, что целевая операционная система предназначена для работы на процессорах, архитектура которых отличается от x86-64, запуск тестовых программ становится затруднительным. Один из способов отладки — запуск целевой операционной системы и тестовых программ на программном эмуляторе системы на кристалле (SoC). Однако для корректной работы таких компонентов, как драйверы графического ускорителя, требуется программная эмуляция GPU, которая не реализована в большинстве доступных программных эмуляторов.

Известно несколько программных эмуляторов, позволяющих моделировать абстрактный или определенный GPU. К ним можно отнести эмулятор gem5-gpu, созданный на базе gem5, который может эмулировать некоторые видеокарты NVIDIA с архитектурой Fermi и Maxwell. Необходимо упомянуть эмулятор Attila, который моделирует виртуальный GPU с поддержкой OpenGL и DirectX 9, а также предоставляет программный стек, включающий реализации OpenGL и DirectX. Эмулятор Attila позволяет сохранять статистику использования памяти, время выполнения функций реализации OpenGL и другие параметры. К сожалению, оба эмулятора более не развиваются.

В рамках этой статьи авторы пытаются ответить на вопрос, какие методы отладки графических операций можно применить при использовании операционных систем реального времени, а также какими функциональными характеристиками должен обладать эмулятор графического ускорителя, используемый при диагностике и исправлении сложно выявляемых ошибок в компонентах графической подсистемы.

Новизна работы заключается в следующем: разработана архитектура эмулятора графического процессора. В отличие от известных решений, в ней предусмотрена возможность применения программного обеспечения (ПО) как в составе эмуляторов аппаратного обеспечения системы в целом типа QEMU [5], так и в виде присоединяемой библиотеки к драйверу графического ускорителя. Для повышения производительности в архитектуре предложено использование аллокаторов памяти.

Приведен обзор смежных работ по графическим технологиям в операционных системах и эмуляторах. В разд. 2 дано описание разработанной архитектуры графической системы. В разд. 3 рассмотрены вопросы отладки графической подсистемы и операций прорисовки. Раздел 4 посвящен методам разработки эмулятора графического ускорителя, применяемого для отладки драйверов и подсистем операционной системы без использования реального аппаратного обеспечения. Результаты практической оценки и применения эмулятора представлены в разд. 5.

1. Анализ предшествующих работ

Использованию встроенных решений для обработки графической информации посвящен ряд статей. В работе [6] описано создание приборной панели самолета на основе операционной системы VxWorks и генератора интерфейсов Tilcon. Работа [7] посвящена созданию приборов с использованием ОСРВ, однако при этом не задействуются GPU, а все вычисления выполняются на CPU. В работе [8] описан подход к созданию приборов с использованием встраиваемых систем и стандартов ARINC 661 и OpenVG. В статье [9] приведено влияние различных факторов на производительность визуализации авиационных приборов при использовании графических API OpenGL ES и OpenGL SC.

Эмуляции встраиваемых систем и их отдельных компонентов, таких как графические чипы, также посвящено значительное количество работ. В статьях [10, 11] описан эмулятор GPU в контексте работы с API NVIDIA CUDA, однако в них рассмотрены вопросы только общих вычислений на GPU и не затронуты вопросы выполнения графических операций. Исследование [12] посвящено разработке нового GPU, а также использованию существующих эмуляторов GPU при создании нового устройства. Эмуляция процессора с архитектурой MIPS64 Octeon с использованием эмулятора с открытым исходным кодом QEMU описана в статье [13], однако в ней нет сведений о работе с графикой. Статья [14] посвящена разработке эмулируемого устройства на базе QEMU, а также созданию драйвера устройства в ОС Linux. Работы [15–20] посвящены вопросам разработки и отладки системного ПО с использованием QEMU. В статьях [21–23] рассмотрены вопросы безопасности при использовании эмуляторов, подобных QEMU.

2. Графическая система ОСРВ

Для обработки графической информации в ОСРВ разработана архитектура графической системы, состоящая из трех основных частей [24].

1. *Подсистема верхнего уровня.* Взаимодействует с пользовательскими приложениями, предоставляет набор компонентов для взаимодействия пользовательских приложений с графическим драйвером.

Основные задачи пользовательского уровня графической системы:

- управление соединениями (создание/закрытие, установка соединения и т. д.);
- управление поверхностями (создание/удаление поверхностей, получение/установка параметров поверхности и т. д.);
- управление дисплеем (включение/выключение, установка параметров — гамма, поворот и т. д.);
- управление режимом изображения (получение списка поддерживаемых режимов, добавление режима, установка режима отображения и т. д.);
- ускорение графических операций.

Структуры и функции, используемые в подсистеме пользовательского уровня, имеют префикс gs.

2. *Подсистема нижнего уровня.* Предоставляет различные функции и структуры для графических драйверов:

- основные функции драйвера (регистрация/отмена регистрации, получение параметров, управление драйвером и т. д.);
- управление выводом на экран (конвейер вывода, “миксер” поверхностей и т. д.);

– компоненты ускорения графических операций (планировщик операций, примитивы синхронизации и т. д.);
– получение режимов монитора (получение и разбор edid);
– управление видеопамятью;
– управление функциями платы (управление тактовыми генераторами, i2c и т. д).

Структуры и функции, используемые в подсистеме нижнего уровня, имеют префикс gsi.

3. *Подсистема ядерного уровня.* Взаимодействует с ядром ОС и выполняет низкоуровневые операции:

– хранение информации о графическом драйвере;
– выделение/освобождение памяти, доступной различным процессам;
– выделение/освобождение непрерывной физической памяти;
– преобразование физической памяти в виртуальную и наоборот;
– обработка прерываний.

Поскольку графическая система разрабатывается для ОСРВ с микроядерной архитектурой, основным аспектом проектирования архитектуры является взаимодействие уровней графической системы, графических драйверов и ядра ОС. Например, ядро Linux является монолитным, и большинство драйверов интегрированы непосредственно в ядро. Драйверы пользовательского пространства и приложения взаимодействуют с драйверами через файл устройства, расположенный в папке /dev. С другой стороны, QNX имеет микроядерную архитектуру, а управление драйверами осуществляется через серверы и менеджеры ресурсов. Приложения также взаимодействуют с драйверами через файлы устройств, но с помощью другого набора функций.

В настоящее время графическая система использует гибрид этих подходов. Подсистемы ядерного и нижнего уровней работают в ядре ОС в главном системном процессе. Драйверы графических устройств находятся в пакете поддержки микропроцессора, который может содержать исходный код или скомпилированную версию драйвера в виде статической библиотеки. Исходный код драйвера может быть скомпилирован при сборке бинарного образа ОС. Кроме того, пакет содержит файл с конфигурацией для определенного типа устройства, передаваемой графическим драйверам при инициализации. Графические драйверы также работают в главном системном процессе. Подсистема ядерного уровня создает в папке /dev файл устройства с именем graphics, который используется графическими драйверами для выполнения низкоуровневых операций, таких как преобразование физических адресов в виртуальные. Графические драйверы не создают отдельный файл устройства в папке /dev. Взаимодействие пользовательского приложения с графическими драйверами осуществляется через подсистему верхнего уровня.

Каждая подсистема компилируется в отдельную статическую библиотеку. Библиотека подсистемы верхнего уровня компонуется непосредственно в пользовательское приложение, а библиотеки ядерного и нижнего уровней компонуются в бинарный образ ОС. Учитывая, что каждая подсистема компилируется в свою собственную библиотеку и выполняется в разных процессах, необходимо было использовать механизм для передачи данных между разными процессами. Графическая система использует механизм очередей сообщений для организации связи между подсистемой нижнего уровня, выполняемой в главном системном процессе, и подсистемой верхнего уровня, выполняемой в пользовательском процессе.

При регистрации драйвера в графической системе создаются две очереди сообщений: одна для приема сообщений, другая для их передачи. Когда пользовательское

приложение инициализируется и устанавливается соединение, верхний уровень графической системы определяет количество доступных драйверов в системе и отправляет запрос в очередь сообщений конкретного драйвера для получения информации о возможностях драйвера.

Запросив информацию у подсистемы нижнего уровня, пользовательское приложение всегда ожидает ответа — операции взаимодействия между верхним и нижним уровнями графической системы синхронны. Подсистема нижнего уровня определяет тип входящего сообщения и выполняет соответствующую обработку. В большинстве случаев это означает взаимодействие с функциональностью драйвера, а не только с графической системой. Для каждого драйвера подсистема нижнего уровня создает свой собственный поток с циклом обработки сообщений.

3. Отладка графической системы и операций прорисовки

Важные аспекты работы над графической системой — достижение корректности выполнения операций и отладка подсистем и драйверов, связанных с отображением графической информации. Так как целевая операционная система предназначена для работы на процессорах, архитектура которых отличается от x86-64, запуск тестовых программ становится затруднительным. Тестовые программы могут быть запущены на целевой платформе и операционной системе, что требует доступа к аппаратному обеспечению, а это не всегда возможно. Время одной итерации может быть значительным, так как во многих случаях требуется перезапускать операционную систему и другие системные компоненты.

Описанный выше метод тестирования направлен в большей степени на окончательное тестирование всех компонентов графической подсистемы и тестирование взаимодействия различных подсистем в различных условиях. Однако значительная часть ошибок может быть выявлена в процессе разработки с помощью юнит-тестов. В большинстве случаев юнит-тесты во время разработки запускаются на машине разработчика, что может привести к трудностям, когда графическая система использует функции и структуры, отсутствующие в операционной системе, установленной на компьютере разработчика. Поскольку разработка графической системы ведется разными программистами на разных операционных системах (Windows, Linux), принято решение эмулировать отсутствующие типы данных и функции целевой операционной системы. Для этого внутри графической системы существует набор файлов с префиксом simulator_, который обеспечивает эмуляцию необходимой функциональности (например, функций распределения памяти, присущих целевой операционной системе). Используя эмулированные типы данных и функции целевой операционной системы, можно протестировать бóльшую часть функциональности графической системы на компьютере разработчика, что значительно ускорит разработку и тестирование.

Еще одним способом отладки является запуск целевой операционной системы и тестовых программ на программном эмуляторе системы на кристалле, например QEMU. Этот метод позволяет обнаружить большинство проблем. Однако необходимо учитывать, что эмулятор может не полностью соответствовать реальной системе на кристалле и проблемы, которые можно воспроизвести на аппаратном обеспечении, не воспроизводятся на эмуляторе. Кроме того, для корректной работы графической системы, драйверов и тестовых программ программный эмулятор должен обеспечивать эмуляцию графического чипа системы на кристалле, но эмуляторы с открытым исходным кодом редко предоставляют такую возможность.

4. Эмулятор графического ускорителя

Для решения указанных проблем разработан эмулятор графического чипа. Главными требованиями к эмулятору были низкая сложность внутреннего устройства, модульность и возможность работать как в виде присоединяемой библиотеки, так и в составе других эмуляторов, например QEMU. С учетом требования обеспечения возможности встраивания эмулятора в другие системы принято решение разрабатывать эмулятор на языке C.

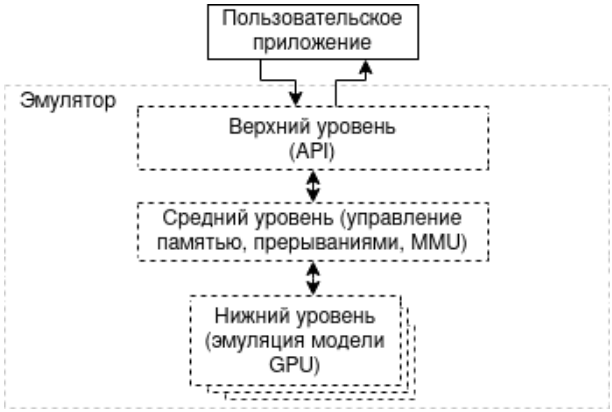
Для расширения возможностей отладки разработан дополнительный модуль (приложение-контейнер), позволяющий использовать функциональность графической системы, драйверы 3D-ускорителя и эмулятора в режиме присоединяемой библиотеки. Каждый из компонентов подключен к приложению в виде статической библиотеки. Кроме того, в приложение интегрируется тестовая программа, которая используется для отладки компонентов. Она также должна быть скомпилирована в виде динамической или статической библиотеки. При запуске приложения производится инициализация описанных выше компонентов, после чего запускается тестовая программа, во время работы которой возможны остановка и просмотр состояния драйвера 3D-ускорителя, графической системы и ее подсистем.

Архитектуру разработанного эмулятора можно разделить на несколько слоев:

- верхний уровень предоставляет API для взаимодействия прикладных программ с эмулятором (инициализация, программирование регистров виртуального устройства и т. д.);
- средний уровень выполняет сервисные функции, необходимые для корректной работы эмулятора (управление памятью, пулом потоков, проверка наличия запрошенного регистра у эмулируемого устройства, и т. д.);
- нижний уровень осуществляет эмуляцию конкретного графического чипа (см. рисунок).

С учетом требования к модульности эмулятора код эмуляции конкретного графического чипа (нижний уровень) собирается в отдельную библиотеку, что позволяет подключать и отключать различные эмулируемые графические чипы при необходимости. Рассмотрим каждый из слоев подробнее.

Для простоты работы с эмулятором API для взаимодействия с прикладными программами сделан минималистичным. Как и с реальными графическими чипами, управ-



Архитектура эмулятора
Emulator architecture

ление виртуальным чипом осуществляется посредством чтения и записи в регистры устройства. Для этих операций эмулятор предоставляет отдельные функции (на данный момент поддерживаются наиболее часто используемые операции чтения и записи по 4 байта). Основные параметры эмулятора задаются при инициализации, но ряд параметров можно задать при помощи механизма свойств. Например, при помощи свойств пользовательское приложение получает доступ к регистрам (указатель на память, отвечающую за регистры) эмулируемого устройства. Также при помощи свойств регулируется необходимость генерации прерываний виртуальным устройством.

Большинство графических чипов имеют блок управления памятью (memory map unit — MMU) для возможности работы с буферами памяти, которые не расположены непрерывно в оперативной памяти [25]. В этих случаях эмуляция функциональности MMU производится на нижнем уровне для конкретного блока. Когда функциональности MMU нет (например, этот блок в разработке) или по какой-то причине нет возможности ее задействовать, эмулятор предоставляет программную эмуляцию блока управления памятью.

Рассмотрим метод программной эмуляции MMU подробнее. При запуске программы, использующей эмулятор, задаются начальный адрес отсчета для виртуальных адресов и общий доступный размер, например начальный адрес 4 КБ, общий доступный размер — (4 ГБ – 4 КБ). Далее программа может добавлять, удалять или получать уже существующие отрезки памяти. Отрезок памяти состоит из начального адреса, размера отрезка памяти и указателя на пользовательские данные, которые хранятся в этом отрезке.

При добавлении информации об отрезке памяти в эмулятор происходит проверка на корректность параметров, а также на то, что этот отрезок уже был добавлен. Добавление отрезка не происходит, если указанные условия не выполнены. Далее происходит проверка, можно ли модифицировать уже существующий отрезок памяти или новый отрезок расположен отдельно от ранее добавленных. При модификации существующего отрезка могут быть изменены или начальная точка, или размер отрезка. В противном случае определяется, куда вставить новый отрезок памяти — перед существующими, между ними или после существующих.

При удалении отрезка памяти также проверяется, существует ли данный отрезок памяти в MMU. Удаление происходит только в случае, если отрезок существует. Далее происходит определение, откуда требуется удалить отрезок — с начала или конца существующего отрезка или из середины. При удалении из середины отрезка создаются два новых отрезка, у которых изменены размер и начальный адрес.

При получении отрезка памяти из MMU определяется, есть ли данный отрезок в MMU, и вычисляется позиция запрашиваемого отрезка памяти среди остальных отрезков памяти в MMU. Иногда требуется использовать адрес памяти, начинающийся не в начале отрезка памяти, а, например, в середине. В этом случае рассчитывается смещение относительно начала отрезка памяти и возвращаются пользовательские данные с рассчитанным смещением.

Общая функциональность эмулятора, используемая остальными его частями, составляет средний уровень эмулятора. Один из факторов, влияющих на скорость работы эмулятора, — эффективность управления памятью. Системное программное обеспечение в целом и эмуляторы в частности характеризуются частым выделением небольших объемов памяти, во многих случаях используемых для хранения промежуточных результатов. К сожалению, операция выделения памяти в большинстве современ-

ных систем достаточно ресурсоемкая. При использовании профилировщика процедуры работы с памятью часто занимают от 10 до 30 % от общего времени выполнения программы [26].

Чтобы уменьшить влияние функций обработки памяти на скорость выполнения, можно использовать несколько подходов. Один подход предполагает исключение или минимизацию использования функций выделения памяти в коде, выполняемом бóльшую часть времени жизни программы. Другой подход предполагает использование аллокаторов. Аллокатор заранее запрашивает у операционной системы определенный объем памяти (обычно несколько мегабайт), который затем используется приложением по мере необходимости. Функции распределения памяти аллокатора выполняются во много раз быстрее, чем функции распределения памяти, предоставляемые операционной системой. Если памяти, выделенной аллокатором, недостаточно, далается запрос дополнительной памяти у операционной системы. Аллокатор позволяет выделять память отдельно для каждого потока выполнения без необходимости использования блокирующего доступа, что также должно повысить производительность в ряде сценариев. Используемый аллокатор основан на решении с открытым исходным кодом rpmalloc.

Одной из задач, стоящих перед эмулятором, является предоставление регистров эмулируемого устройства для пользовательской программы. На реальном аппаратном обеспечении пользовательской программе доступны начальный адрес регистров устройства и общий размер регистров — эти параметры используются для отображения физических регистров в оперативную память (напрямую работу с физической памятью для чтения и записи регистров применяют реже, в основном в тестовых программах без использования операционной системы). Далее программа использует полученный виртуальный адрес в качестве начального адреса регистров и прибавляет требуемое смещение относительно начала для чтения или записи в конкретный регистр. Соответственно, эмулятор должен предоставлять подобную функциональность, а также реагировать на использование некорректных адресов регистров (например, устройство не имеет регистров между адресами 0x10 и 0x20, а программа обратилась по адресу 0x14).

Для реализации указанной выше функциональности предложен следующий метод. Во время инициализации эмулятора создаются виртуальные регистры эмулируемого устройства. После создания виртуальных регистров у эмулятора есть полная информация о том, какое смещение используется для каждого виртуального регистра относительно начального адреса. Эмулятор выделяет память, которая будет передана пользовательскому приложению с размером, равным общему размеру регистров устройства — данную память приложение будет использовать для чтения и записи в регистры. Далее, в выделенную память заносится информация о наличии или отсутствии регистра в конкретном смещении относительно начала выделенной памяти, которое принимается за нуль. Шаг смещения составляет 4 байта. Если в конкретном смещении (например, 0x8) имеется виртуальный регистр, то в память по данному смещению записывается значение 1, в противном случае записывается нуль. При использовании этой памяти пользовательским приложением для чтения или записи в виртуальный регистр происходит следующее: если в памяти с указанным смещением записано значение 1, то этот виртуальный регистр существует и продолжается запрошенное пользовательской программой действие (чтение или запись). В противном случае эмулятор выдает сообщение об обращении к несуществующему виртуальному регистру.

Для большинства устройств требуется эмуляция работы прерываний. При помощи прерываний устройство сообщает о произошедшем событии — ошибке, завершении по-

ставленной задачи и т. д. Эмулятор предоставляет два режима работы прерываний — основной и принудительный. Основной режим сообщает пользовательскому приложению о прерывании, следуя логике работы эмулируемого устройства. Принудительный режим заставляет эмулируемое устройство немедленно выдать прерывание требуемого типа. Второй режим используется в основном для тестирования различных ситуаций по обработке ошибок. Работа с прерываниями выполняется в отдельном потоке, что обеспечивает асинхронную работу эмулятора при обработке прерывания пользовательским приложением.

Для работы с прерываниями при инициализации эмулятора пользовательское приложение привязывает функцию, которая будет вызвана при срабатывании прерывания. Нижний уровень эмулятора может значительно различаться для каждого эмулируемого устройства, так как именно здесь происходит эмуляция поведения устройства. Однако можно выделить базовую функциональность, которая будет присутствовать в большинстве эмулируемых видеокарт, а именно: регистры устройства, обработка командного буфера и изменение состояния устройства, обработка инструкций.

При инициализации эмулятора создаются регистры эмулируемого устройства (название регистра, определение смещения регистра относительно начального адреса, значение при сбросе, возможность записи в регистр, функции-обработчики чтения и записи в регистр). Для большинства регистров применяются стандартные обработчики чтения и записи, которые осуществляют атомарное чтение или запись нового значения. Если, например, для регистра требуется определенное поведение при записи, то для него используется отдельный обработчик записи, который выполняет дополнительные действия помимо записи нового значения. Эмулятор также позволяет работать с регистрами устройства без использования некоторых механизмов проверки для внутренних нужд (например, запись значения в регистр прерывания, который запрещает запись извне).

Большинство видеокарт используют командные буферы для задания новых состояний устройства. Несмотря на то что представление командного буфера уникально для каждой архитектуры графического чипа, они используются для одного и того же — изменения параметров устройства и входных данных для выполнения расчетов или прорисовки в указанный буфер. Основные параметры, которые могут быть изменены командным буфером, схожи у большинства видеокарт: задание буферов вершин и индексов, задание шейдерных программ, задание параметров отображения (размер видимой области, границы области отсечения прорисовки (scissor) и т. д.), задание параметров прорисовки и т. д.

Важной частью эмуляции видеокарты является обработка инструкций шейдерной программы. Каждая архитектура графических чипов обладает уникальным набором команд, однако инструкции можно разделить на несколько категорий:

- арифметические (сложение, вычитание, умножение, деление, получение и сохранение значений и т. д.);
- логические (обработка условий, циклов, вызов подпроцедур и т. д);
- работа с текстурами (получение данных текстуры, различные преобразования этих данных).

Для эмуляции инструкций графического типа могут быть применены интерпретатор или JIT-компилятор. Интерпретатор инструкций проще в реализации, но производительность обработки инструкций может быть не очень высокой. Подход с использованием JIT-компилятора для обработки инструкций позволяет преобразовать эмулиру-

емую инструкцию в одну или несколько инструкций архитектуры используемого центрального процессора. Этот подход более трудоемкий, но позволяет получить лучшую производительность по сравнению с интерпретатором. На текущий момент эмулятор использует интерпретатор для эмуляции инструкций графического чипа.

Каждый графический чип имеет конвейер, который в той или иной степени соответствует конвейеру, используемому в высокоуровневых графических интерфейсах программирования (API), таких как OpenGL и DirectX. Конвейер состоит из таких этапов, как: обработка вершин вершинным шейдером, теселляция вершин (опционально), обработка примитивов геометрическим шейдером (опционально), растеризация треугольников, обработка растеризированного изображения пиксельным (фрагментным) шейдером. Для вычислительных задач существует отдельный конвейер. Для задач отрисовки используются вершинный и пиксельный шейдеры, остальные типы шейдеров применяются значительно реже. Соответственно, для задач отрисовки эмулятор должен реализовать, как минимум, обработку вершинных шейдеров, растеризацию и обработку пиксельных шейдеров.

При реализации конвейера прорисовки встает вопрос о визуализации результатов. Существует два основных варианта прорисовки — с использованием центрального процессора и аппаратного ускорения (обычно это 3D-ускорители). Первый вариант обеспечивает максимальную точность визуализации, но он медленный по скорости. Второй вариант позволяет отображать результаты прорисовки в реальном масштабе времени, но может отличаться от эталонного изображения в зависимости от применяемого аппаратного обеспечения и драйверов. На данный момент разрабатываемый эмулятор не обладает средствами отрисовки. В первую очередь будет разрабатываться вариант с использованием CPU, т. е. программная визуализация. В дальнейшем эмулятор будет предоставлять пользователю выбор — максимальная точность визуализации, но медленная скорость или возможные артефакты изображения, но высокая скорость отрисовки.

5. Результаты и обсуждение

Разработанный эмулятор использован при тестировании компонентов графической системы, связанных с ускорением графических операций. Несмотря на то что разрабатываемый эмулятор имеет ограниченную функциональность, он может использоваться для стабилизации компонентов графической системы и графических драйверов. Одним из его основных преимуществ является запуск драйвера графического ускорителя на инструментальной машине с возможностью пошаговой отладки. Применение эмулятора позволило воспроизвести ошибки, которые на реальной аппаратуре имели характер “плавающих” и были тяжело воспроизводимы.

Заключение

Разработанная архитектура эмулятора позволяет эмулировать графические чипы различной сложности и упрощает разработку и отладку программных компонентов, зависящих от графических ускорителей, таких как графические системы и драйверы. Кроме того, за счет возможности интеграции в существующие эмуляторы, такие как QEMU, разрабатываемый эмулятор может быть использован для тестирования значи-

тельного количества программного обеспечения, задействующего графические ускорители при помощи высокоуровневого API OpenGL, DirectX и им подобных. Полученные результаты позволяют решить ряд задач, которые затруднительно выполнить известными методами отладки для заданного типа программного обеспечения. В дальнейшем планируется доработать эмулятор для возможности отображения результатов эмуляции на экране и использовать его при создании новых перспективных графических ускорителей.

Благодарности. Публикация выполнена в рамках НИР ФГУ ФНЦ НИИСИ РАН по теме № FNEF-2024-0003 “Методы разработки аппаратно-программных платформ на основе защищенных и устойчивых к сбоям систем на кристалле и сопроцессоров искусственного интеллекта и обработки сигналов”.

Список литературы

[1] **Drozdov A.Y., Fonin Y.N., Perov M.N., Vedishcheva T.S., Novoselova Y.K.** An approach to cross-platform drivers development. Moscow: IEEE Press; 2015: 54–57. DOI:10.1109/EnT.2015.14.

[2] Embedded systems in the Internet of Things. Benisontech. Available at: `https://benisontech.com/embedded-systems-in-the-internet-of-things`.

[3] **Godunov A.N.** Real-time operating system Baget 3.0. Software & Systems. 2010; 4(92):16–19.

[4] **Godunov A.N., Soldatov V.A.** Baget real-time operating system family (features, comparison, and future development). Programming and Computer Software. 2014; 40(5):259–264. DOI:10.1134/S036176881405003X.

[5] **Chen I.-H., King C.-T., Chen Y.-H., Lu J.-M.** Full system emulation of embedded heterogeneous multicores based on QEMU. IEEE 24th International Conference on Parallel and Distributed Systems (ICPADS). 2018; 771–778. DOI:10.1109/PADSW.2018.8645045.

[6] **Wu Y., Xuejun Z., Huaxian L., Xiangmin G.** Design and realization of display control software in integrated avionic system for general aviation based on the VxWorks. IEEE 13th World Congress on Intelligent Control and Automation (WCICA). 2018; 1295–1299. DOI:10.1109/WCICA.2018.8630561.

[7] **Kui R., Huai-Liang T.** Design and implementation of a graphics display system based on RTOS. IEEE Fifth International Conference on Computational and Information Sciences (ICCIS). 2013; 84–87. DOI:10.1109/ICCIS.2013.30.

[8] **Yoon J., Baek N., Lee H.** ARINC661 graphics rendering based on OpenVG and its use cases with wireless communications. Wireless Personal Communications. 2017; 175–185. DOI:10.1007/s11277-015-3163-y.

[9] **Girard S.R., Legault V., Bois G., Boland J.-F.** Avionics graphics hardware performance prediction with machine learning. Scientific Programming. 2019; 1–15. DOI:10.1155/2019/9195845.

[10] **Gong X., Ubal R., Kaeli D.** Multi2Sim Kepler: a detailed architectural GPU simulator. IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS). 2017; 269–278. DOI:10.1109/ISPASS.2017.7975298.

[11] **Power J., Hestness J., Orr M.S., Hill M.D., Wood D.A.** gem5-gpu: a heterogeneous CPU-GPU simulator. IEEE Computer Architecture Letters. 2015; (14):34–36. DOI:10.1109/LCA.2014.2299539.

[12] **Dow H.-K., Huang C.-H., Lai C.-H., Tsao K.-H., Tseng S.-C., Wu K.-Y., Wu T.-H., Yang H.-C., Zhang Jain D.-J., Chang Y.-N., Haga S.W., Hsiao S.-F.,**

Huang I.-J., Kuang S.-R., Lee C.-N. An OpenGL ES 2.0 3D graphics SoC with versatile HW/SW development support. VLSI Design, Automation and Test (VLSI-DAT). 2015; 1–4. DOI:10.1109/VLSI-DAT.2015.7114496.

[13] **Mehmood M.A., Ain Q.U., Akram A., Qadeer A., Waheed A.** Emulating an Octeon MIPS64 based embedded system on X86 in QEMU. IEEE 19th International MultiTopic Conference (INMIC). 2016; 1–7. DOI:10.1109/INMIC.2016.7840110.

[14] **Bian X.** Implement a virtual development platform based on QEMU. IEEE International Conference on Green Informatics (ICGI). 2017; 93–97. DOI:10.1109/ICGI.2017.19.

[15] **Wei H.-L., King C.-T., Das B., Peng M.-C., Wang C.-C., Huang H.-L., Lu J.-M.** Application specific component-service-aware trace generation on AndroidQEMU. 30th IEEE International System-on-Chip Conference (SOCC). 2017; 316–321. DOI:10.1109/SOCC.2017.8226069.

[16] **Almeida R., Novais L., Naia N., Faria R., Cabral J.** Reliable software design aided by QEMU simulation. 22nd IEEE International Conference on Industrial Technology (ICIT). 2021; 797–804. DOI:10.1109/ICIT46573.2021.9453486.

[17] **Dileep K.P., Devesh G., Raghavendra Rao A., Suman M., Srikanth S.V.** Verification of Linux device drivers using device virtualization. 2nd International Conference on Computing for Sustainable Global Development (INDIACom). 2015; 694–698.

[18] **Markovic A., Markovic S., Rikalo A., Jovanovic P.** Adding support for nanomips architecture to QEMU emulator. 26th Telecommunications Forum (TELFOR). 2018; 1–4. DOI:10.1109/TELFOR.2018.8611981.

[19] **Reitz J., Gugenheimer A., Rosmann J.** Virtual hardware in the loop: hybrid simulation of dynamic systems with a virtualization platform. IEEE Winter Simulation Conference (WSC). 2020; 1027–1038. DOI:10.1109/WSC48552.2020.9383963.

[20] **Lupori L., Rosario V., Borin E.** Towards a high-performance RISC-V emulator. IEEE Symposium on High Performance Computing Systems (WSCAD). 2018; 213–220. DOI:10.1109/WSCAD.2018.00041.

[21] **Wang Z., Zheng F., Lin J., Fan G., Dong J.** SEGIVE: a practical framework of secure GPU execution in virtualization environment. IEEE 39th International Performance Computing and Communications Conference (IPCCC). 2020; 1–10. DOI:10.1109/IPCCC50635.2020.9391574.

[22] **Lee S., Yoo S.** Breaching GPU data on a cloud VM. IEEE 13th International Conference on Information and Communication Technology Convergence (ICTC). 2022; 205–207. DOI:10.1109/ICTC55196.2022.9952730.

[23] **Oliveira P.R., Meireles M., Maia C., Pinho L.M., Gouveia G., Esteves J.** Emulation-in-the-loop for simulation and testing of real-time critical CPS. IEEE Industrial Cyber-Physical Systems (ICPS). 2018; 258–263. DOI:10.1109/ICPHYS.2018.8387669.

[24] **Giatsintov A., Mamrosenko K., Bazhenov P.** Architecture of the graphics system for embedded real-time operating systems. Tsinghua Science and Technology. 2023; (28):541–551. DOI:10.26599/TST.2022.9010028.

[25] **Behera D., Jena U.R.** Detailed review on embedded MMU and their performance analysis on test benches. IEEE International Conference on Computational Intelligence for Smart Power System and Sustainable Energy (CISPSSE). 2020; 1–6. DOI:10.1109/CISPSSE49931.2020.9212265.

[26] **Weisner R.** How memory allocation affects performance in multi-threaded programs. Available at: `https://www.oracle.com/technical-resources/articles/it-infrastructure/dev-memalloc.html`.

Вычислительные технологии, 2024, том 29, № 4, с. 95–109. © ФИЦ ИВТ, 2024 ISSN 1560-7534
Computational Technologies, 2024, vol. 29, no. 4, pp. 95–109. © FRC ICT, 2024 eISSN 2313-691X

INFORMATION TECHNOLOGIES

DOI:10.25743/ICT.2024.29.4.007

Testing graphics processing operations with graphics accelerator emulation

A. M. Giatsintov*, K. A. Mamrosenko

Scientific Research Institute for System Analysis of the RAS, 117278, Moscow, Russia
*Corresponding author: Alexander M. Giatsintov, e-mail: giatsintov@niisi.ras.ru
Received October 20, 2023, revised November 13, 2023, accepted November 20, 2023.

Abstract

Nowadays, an increasing number of problems require the use of hardware accelerators to reduce the time required for calculations as well as for displaying their results. An important aspect of working with graphics operations is to achieve correctness for execution of operations and debugging of both subsystems and drivers involved in displaying of graphics information. This paper presents the architecture of an emulator of the graphics accelerator designed to diagnose and correct hard-to-detect errors within components of the graphics subsystem. Unlike the known solutions, the developed architecture allows using the emulator as a part of system emulators such as a QEMU and as an attached library for a driver of the graphics accelerator. The architecture of the developed emulator can be divided into several layers: the upper level provides API for application interaction with the emulator (initialization, programming of registers of the virtual device, etc.); the middle level performs service functions necessary for correct work of the emulator (memory management, thread pooling, checking if the emulated device has the requested register, etc.); the lower level performs emulation of a particular graphics chip. The virtual chip is controlled by reading and writing to the device registers — the emulator provides separate functions for these operations. One of the main advantages is the ability to run the graphics accelerator driver on a tool machine with capability for step-by-step debugging. The emulator allows detecting some intermittent errors that were hard to reproduce on real hardware.

Keywords: visualization, graphics system, rendering, operating system, emulation, QEMU.

Citation: Giatsintov A.M., Mamrosenko K.A. Testing graphics processing operations with graphics accelerator emulation. Computational Technologies. 2024; 29(4):95–109. DOI:10.25743/ICT.2024.29.4.007. (In Russ.)

Acknowledgements. The publication was carried under state contract FNEF-2024-0003 “Methods of development of hardware and software platforms based on secure and fault-tolerant systems-on-chips and coprocessors for artificial intelligence and signal processing”.

References

1. **Drozdov A.Y., Fonin Y.N., Perov M.N., Vedishcheva T.S., Novoselova Y.K.** An approach to cross-platform drivers development. Moscow: IEEE Press; 2015: 54–57. DOI:10.1109/EnT.2015.14.
2. Embedded systems in the Internet of Things. Benisontech. Available at: https://benisontech.com/embedded-systems-in-the-internet-of-things.
3. **Godunov A.N.** Real-time operating system Baget 3.0. Software & Systems. 2010; 4(92):16–19.
4. **Godunov A.N., Soldatov V.A.** Baget real-time operating system family (features, comparison, and future development). Programming and Computer Software. 2014; 40(5):259–264. DOI:10.1134/S036176881405003X.

5. **Chen I.-H., King C.-T., Chen Y.-H., Lu J.-M.** Full system emulation of embedded heterogeneous multicores based on QEMU. IEEE 24th International Conference on Parallel and Distributed Systems (ICPADS). 2018; 771–778. DOI:10.1109/PADSW.2018.8645045.
6. **Wu Y., Xuejun Z., Huaxian L., Xiangmin G.** Design and realization of display control software in integrated avionic system for general aviation based on the VxWorks. IEEE 13th World Congress on Intelligent Control and Automation (WCICA). 2018; 1295–1299. DOI:10.1109/WCICA.2018.8630561.
7. **Kui R., Huai-Liang T.** Design and implementation of a graphics display system based on RTOS. IEEE Fifth International Conference on Computational and Information Sciences (ICCIS). 2013; 84–87. DOI:10.1109/ICCIS.2013.30.
8. **Yoon J., Baek N., Lee H.** ARINC661 graphics rendering based on OpenVG and its use cases with wireless communications. Wireless Personal Communications. 2017; 175–185. DOI:10.1007/s11277-015-3163-y.
9. **Girard S.R., Legault V., Bois G., Boland J.-F.** Avionics graphics hardware performance prediction with machine learning. Scientific Programming. 2019; 1–15. DOI:10.1155/2019/9195845.
10. **Gong X., Ubal R., Kaeli D.** Multi2Sim Kepler: a detailed architectural GPU simulator. IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS). 2017; 269–278. DOI:10.1109/ISPASS.2017.7975298.
11. **Power J., Hestness J., Orr M.S., Hill M.D., Wood D.A.** gem5-gpu: a heterogeneous CPU-GPU simulator. IEEE Computer Architecture Letters. 2015; (14):34–36. DOI:10.1109/LCA.2014.2299539.
12. **Dow H.-K., Huang C.-H., Lai C.-H., Tsao K.-H., Tseng S.-C., Wu K.-Y., Wu T.-H., Yang H.-C., Zhang Jain D.-J., Chang Y.-N., Haga S.W., Hsiao S.-F., Huang I.-J., Kuang S.-R., Lee C.-N.** An OpenGL ES 2.0 3D graphics SoC with versatile HW/SW development support. VLSI Design, Automation and Test (VLSI-DAT). 2015; 1–4. DOI:10.1109/VLSI-DAT.2015.7114496.
13. **Mehmood M.A., Ain Q.U., Akram A., Qadeer A., Waheed A.** Emulating an Octeon MIPS64 based embedded system on X86 in QEMU. IEEE 19th International MultiTopic Conference (INMIC). 2016; 1–7. DOI:10.1109/INMIC.2016.7840110.
14. **Bian X.** Implement a virtual development platform based on QEMU. IEEE International Conference on Green Informatics (ICGI). 2017; 93–97. DOI:10.1109/ICGI.2017.19.
15. **Wei H.-L., King C.-T., Das B., Peng M.-C., Wang C.-C., Huang H.-L., Lu J.-M.** Application specific component-service-aware trace generation on AndroidQEMU. 30th IEEE International System-on-Chip Conference (SOCC). 2017; 316–321. DOI:10.1109/SOCC.2017.8226069.
16. **Almeida R., Novais L., Naia N., Faria R., Cabral J.** Reliable software design aided by QEMU simulation. 22nd IEEE International Conference on Industrial Technology (ICIT). 2021; 797–804. DOI:10.1109/ICIT46573.2021.9453486.
17. **Dileep K.P., Devesh G., Raghavendra Rao A., Suman M., Srikanth S.V.** Verification of Linux device drivers using device virtualization. 2nd International Conference on Computing for Sustainable Global Development (INDIACom). 2015; 694–698.
18. **Markovic A., Markovic S., Rikalo A., Jovanovic P.** Adding support for nanomips architecture to QEMU emulator. 26th Telecommunications Forum (TELFOR). 2018; 1–4. DOI:10.1109/TELFOR.2018.8611981.
19. **Reitz J., Gugenheimer A., Rosmann J.** Virtual hardware in the loop: hybrid simulation of dynamic systems with a virtualization platform. IEEE Winter Simulation Conference (WSC). 2020; 1027–1038. DOI:10.1109/WSC48552.2020.9383963.
20. **Lupori L., Rosario V., Borin E.** Towards a high-performance RISC-V emulator. IEEE Symposium on High Performance Computing Systems (WSCAD). 2018; 213–220. DOI:10.1109/WSCAD.2018.00041.
21. **Wang Z., Zheng F., Lin J., Fan G., Dong J.** SEGIVE: a practical framework of secure GPU execution in virtualization environment. IEEE 39th International Performance Computing and Communications Conference (IPCCC). 2020; 1–10. DOI:10.1109/IPCCC50635.2020.9391574.
22. **Lee S., Yoo S.** Breaching GPU data on a cloud VM. IEEE 13th International Conference on Information and Communication Technology Convergence (ICTC). 2022; 205–207. DOI:10.1109/ICTC55196.2022.9952730.
23. **Oliveira P.R., Meireles M., Maia C., Pinho L.M., Gouveia G., Esteves J.** Emulation-in-the-loop for simulation and testing of real-time critical CPS. IEEE Industrial Cyber-Physical Systems (ICPS). 2018; 258–263. DOI:10.1109/ICPHYS.2018.8387669.

24. **Giatsintov A., Mamrosenko K., Bazhenov P.** Architecture of the graphics system for embedded real-time operating systems. Tsinghua Science and Technology. 2023; (28):541–551. DOI:10.26599/TST.2022.9010028.
25. **Behera D., Jena U.R.** Detailed review on embedded MMU and their performance analysis on test benches. IEEE International Conference on Computational Intelligence for Smart Power System and Sustainable Energy (CISPSSE). 2020; 1–6. DOI:10.1109/CISPSSE49931.2020.9212265.
26. **Weisner R.** How memory allocation affects performance in multi-threaded programs. Available at: `https://www.oracle.com/technical-resources/articles/it-infrastructure/dev-memalloc.html`.