Национальный исследовательский университет ИТМО

Факультет программной инженерии и компьютерной техники

Лабораторная работа №1

по дисциплине Проектирование вычислительных систем

Студент: Саржевский Иван

Группа: P3402

г. Санкт-Петербург

2020 г.

Radeon Asm Debugger Extension for Visual Studio

https://github.com/vsrad/radeon-asm-tools

Краткое описание проекта в свободной форме

Данный проект представляет собой реализацию средств разработки для написания шейдеров на ассемблере GCN для видеокарт AMD, выполненную в качестве расширения для Microsoft Visual Studio. Основныйе возможности - псевдо-дебаггер и подсветка синтаксиса, однако помимо этого реализовано множество дополнительных функций, призванных упростить процесс разработки.

В основе работы расширения лежит клиент-серверная модель взаимодействия двух изолированых компонентов - самого VS-расширения и сервера, способного исполнять команды, между которыми настраивается TCP-соединение. Клиент-серверная модель выбрана не случайно - дело в том, что особенностью отладки шейдеров для графического процессора на ассемблере является то, что в случае некорректного поведения он может привести к отказу машины, на которой происходит отладка. Это, в свою очередь, влечет за собой потерю времени и риск потерять последние изменения при отладке на хосте. Помимо этого, часто необходимо отладить шейдер на нескольких разных видеокартах, без клиент-серверной модели это было бы проблематично.

Выше было указано, что реализованный отладчик является псевдо-дебаггером. Дело в том, что настоящий дебаг шейдерного кода довольно проблематичен. Как минимум он требует аппаратной поддержки целевой видеокарты. В большинстве случаев такой поддержки нет, к тому же обычно требуется отладка на нескольких целевых платформах. Поэтому было решено реализовать определенные скрипты, суть которых заключается в том, чтобы 'пропатчить' исходный код шейдера - вставить после необходимой строки инструкции, которые сохраняют в заранее выделенные участки памяти значения интересующих пользователя переменных и останавливают выполнение шейдера. Таким образом, этот отладчик является псевдо-дебаггером, потому что на самом деле, при каждом шаге отладки шейдер запускается заново с новой строкой для остановки.

Недостаток такого подхода очевиден - процесс отладки был очень нетривиальным, нужно было переместить исходный код на целевую машину, запустить из командной строки последовательность скриптов с правильными аргументами, не говоря уже о том, что получившийся дамп памяти необходимо было как-то анализировать.

Поэтому созрел запрос на автоматизацию всех этих действий - интеграцию отладочных скриптов в используемую среду разработки (VS) и инструменты для корректной визуализации полученных в результате отладки данных. Эти проблемы призван решить данный проект.

Жизненный цикл

1. Планирование

   Роль в проекте: Оценка требований, анализ существующих подобных систем, оценка трудоемкости

   Методы и средства: Ознакомление с кодом других проектов на `github`, комуникация с заказчиком, изучение существующей системы отладки

2. Проектирование системы

   Роль в проекте: Определение архитектурных проблем и принятие решений по ним, построение архитектуры всех компонентов системы и архитектуры системы в целом

   Методы и средства: Создание entity-диаграммы, UML-диаграммы, прецедентов

3. Разработка мокапов интерфейсов

   Роль в проекте: Разработка мокапов, согласно которым будут верстаться пользовательские интерфейсы согласно требованиям системы и из соображений эргономики

   Методы и средства: Инструменты прототипирования

4. Реализация

   (a) Реализация сервера

   (b) Реализация расширения для VS

   Роль в проекте: Каждая часть системы должна быть реализована и оттестирована

   Методы и средства: Написание кода, Unit-тесты

5. Релиз

   Роль в проекте: Должна быть построена релизная сборка, которая должна быть протестирована на всех целевых платформах, написан change log, все это должно быть оформлено в виде github-релиза с версией отвечающей дате релиза

   Методы и средства: Ручное тестирование, github

6. Анализ текущего релиза

   Роль в проекте: Получение новых feature-реквестов и баг репортов

   Методы и средства: github issues

7. Планирование следующего релиза

   Роль в проекте: Определение списка новых опций и багов, которые необходимо исправить для того, чтобы произвести новый релиз

   Методы и средства: github issues, системы планирования задач

   После этого возвращаемся на пункт 4, реализация новых опций с последующим релизом

Архитектурные проблемы

1. Выбор модели взаимодействия между расширением и сервером.

   Теоретически, расширение для `VS` может самостоятельно запускать все необходимые скрипты на машине, на которой работает. Это позволило бы избежать проблем с обеспечением корректной передачи данных между хостом и сервером. Однако, в силу особенностей системы, изложенных в кратком описании проекта, было решено реализовать именно клиент-серверную модель взаимодействия.

2. Тип протокола взаимодействия расширения и сервера: `Stateful/Stateless`.

   Изначально было принято решение четко определить последовательность сообщений, которыми хост и сервер обмениваются в процессе отладки, однозначно определив сессию и, соответственно, сервер был спроектирован как `stateful` компонент. Однако в процессе поддержки появлялось все больше различных сценариев взаимодействия пользователя с системой, что требовало добавления все новых состояний и типов пользовательских сессий, что значительно затрудняло разработку и увеличивало количество возможных багов. Это подтолкнуло нас к редизайну сервера, теперь это `stateless` компонент, который обслуживает несколько базовых команд.

3. Пользовательский интерфейс конфигурации шагов отладки.

   Одним из требований системы было наличие т. н. "Профилей" отладки. Профиль включал в себя окружение и конфигурацию для основных шагов - дебага, запуска препроцессора, профайлера и дизассемблера. Благодаря наличию профилей можно быстро переключаться между целевыми платформами для отладки. Однако в процессе использования данный подход зарекомендовал себя как недостаточно гибкий - для более сложных сценариев требовалось реализовывать логику в вызываемом скрипте. Было принято решение внести изменение в дизайн профилей - отныне каждое детерменированное действие может строиться из набора простых действий в произвольном порядке. Помимо этого, пользователь теперь может определять кастомные действия и вызывать их в любой момент. Профиль теперь определяет совокупность всех определенных действий. Данное архитектурное решение позволило добиться необходимой гибкости работы пользователя при относительной простоте настройки расширения.

4. Выбор оптимального интерфейса представления данных с учетом специфики системы.

   Так как отладчик предназначен, в первую очередь, для отладки шейдеров на GPU, привычные интерфейсы для отображения значений переменных использовать невозможно. Дело в том, что из-за высокой степени параллелизма у каждого потока будет свое значение заданной переменной. Для решения этой проблемы было решено разработать собственный интерфейс отображения переменных, который представляет собой таблицу, строки которой являются переменной, а каждый столбец представляет один лейн.

5. Поддержка конфигураций нескольких проектов в рамках одного решения.

   VS поддерживает одновременную работу с несколькими проектами в рамках одного решения. Необходимо было сделать выбор каким образом это обрабатывать. Проще было бы иметь один сет профилей для всего решения, однако для большей гибкости настройки было принято решение сохранить сет профилей за каждым проектом. Активный сет профилей при этом определяется текущим `startup project` в решении.

Вывод

В ходе выполнения ланной лабораторной работы были приобретены навыки анализа архитектурных проблем, а также краткого описания системы и её жизненного цикла.