«Санкт-Петербургский государственный электротехнический университет «ЛЭТИ» им. В.И.Ульянова (Ленина)»
(СПбГЭТУ «ЛЭТИ»)

Направление	01.03.02 – Прикладная математика и информатика
Профиль	Математическое и программное обеспечение вычислительных машин
Факультет	КТИ
Кафедра	МОЭВМ

К защите допустить

Зав. кафедрой Кринкин К.В.

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА БАКАЛАВРА

Тема: ОТЛАДЧИК ДЛЯ OPENCL

Студент		_____ *подпись*	Лосев М.Л.
Руководитель	доцент *(Уч. степень, уч. звание)*	_____ *подпись*	Беляева С.А.
Консультанты	ст. преподаватель *(Уч. степень, уч. звание)*	_____ *подпись*	Калишенко Е.Л.
	(Уч. степень, уч. звание)	_____ *подпись*	Лукин М.И.

Санкт-Петербург

2021

ЗАДАНИЕ
НА ВЫПУСКНУЮ КВАЛИФИКАЦИОННУЮ РАБОТУ

Утверждаю
Зав. кафедрой МОЭВМ
_____ Кринкин К.В.
«___»_____2021 г.

Студент Лосев М.Л. Группа 7383

Тема работы: Разработка отладчика для OpenCL

Место выполнения ВКР: СПбГЭТУ «ЛЭТИ», кафедра МО ЭВМ

Исходные данные (технические требования):
кратко указываются основные требования к ВКР

Содержание ВКР:
Кратко перечисляются основные разделы ВКР

Перечень отчетных материалов: пояснительная записка, иллюстративный материал, иные отчетные материалы

Дополнительные разделы: Обеспечение качества разработки, продукции, программного продукта

Дата выдачи задания
«___»_____20__ г.

Дата представления ВКР к защите
«___»_____20__ г.

Студент _____ Лосев М.Л.

Руководитель доцент _____ Беляев С.А.
(Уч. степень, уч. звание)

Консультант _____ Лукин М.И.
(Уч. степень, уч. звание)

КАЛЕНДАРНЫЙ ПЛАН ВЫПОЛНЕНИЯ
ВЫПУСКНОЙ КВАЛИФИКАЦИОННОЙ РАБОТЫ

Утверждаю
Зав. кафедрой МОЭВМ
_____ Кринкин К.В.
«___»_____20__ г.

Студент Лосев М.Л. Группа 7383

Тема работы: Разработка отладчика для OpenCL

№ п/п	Наименование работ	Срок выполнения
1	Обзор литературы по теме работы	00.00 – 00.00
2	Наименование раздела	00.00 – 00.00
3	Наименование раздела	00.00 – 00.00
4	Наименование раздела	00.00 – 00.00
5	Оформление пояснительной записки	00.00 – 00.00
6	Оформление иллюстративного материала	00.00 – 00.00

Студент(ка) _____ Иванов И.И.

Руководитель _____ Иванов И.И.
(Уч. степень, уч. звание)

Консультант _____ Иванов И.И.
(Уч. степень, уч. звание)

РЕФЕРАТ

ABSTRACT

Оглавление

ВВЕДЕНИЕ 6
1. ОБЗОР ПРЕДМЕТНОЙ ОБЛАСТИ 7
1.1. OpenCL 7
1.1.1. Модель платформы 7
1.1.2. Модель выполнения 8
1.1.3. Язык программирования OpenCL 9
1.1.4. Модель памяти 9
1.2. Отладчики 11
1.2.1. Классификация отладчиков 11
1.3. Обзор аналогов 12
1.2.1. Критерии сравнения 12
1.2.2. Существующие OpenCL-отладчики 12
1.2.3. Выводы по итогам сравнения 13
2. ФОРМУЛИРОВКА ТРЕБОВАНИЙ К ПРОГРАММЕ 14
2.1. Устройствонезависимость 14
2.2. Возможность интеграции 14
2.3. Интерфейс командной строки 14
2.4. Высокоуровневость 14
2.5. Платформонезависимость 15
3. ОПИСАНИЕ РЕШЕНИЯ 16
3.1.1. Структуры данных 16
3.1.2. Передача состояния устройства 20
3.1.3. Получение состояния устройства 24
2.1. Интерфейс пользователя 26
3. ОБЕСПЕЧЕНИЕ КАЧЕСТВА РАЗРАБОТКИ, ПРОДУКЦИИ, ПРОГРАММНОГО ПРОДУКТА (ДОП. РАЗДЕЛ) 27
3.2. Юнит-тесты 27
3.2.1. Тестирование парсинга данных 27
3.2.2. Тестирование парсинга типов 35
3.3. Интеграционные тесты 38
Заключение 41
Литература 42

ВВЕДЕНИЕ

Компания AMD прекратила поддержку инструмента разработки CodeXL для отладки и профилирования программ, использующих GPU, в 2019 году. Из-за этого отладка на новых устройствах, таких как Radeon RX 5700, невозможна. Поэтому нужны новые инструменты разработки под AMD-GPU.

Целью является создание отладчика для OpenCL, работающего со всеми поддерживаемыми устройствами, который позволяет пользователю видеть на каждом шаге исполнения программы построчно состояние регистров выбранного рабочего элемента (ядра GPU) и локальной памяти (кэш GPU).

Объектом исследования является алгоритм отладчика для модификации OpenCL-программы.

Предмет исследования – применение алгоритмов для получения состояния устройства (регистров и памяти) с целью его отображения.

Задачи:

1. Изучить подходы к разработке отладчиков.
2. Разработать алгоритм получения состояния устройства ("backend").
3. Реализовать вывод данных о состоянии устройства ("frontend").

Практическая значимость работы заключается в том, что результат работы сможет использоваться для повышения эффективности разработки OpenCL-приложений.

1. ОБЗОР ПРЕДМЕТНОЙ ОБЛАСТИ

1.1. OpenCL

OpenCL – это открытый стандарт параллельного программирования общего назначения на центральных, графических и иных процессорах [1]. Он включает в себя программный интерфейс приложений (API) для управления параллельными вычислениями на гетерогенных платформах и устройствах, а также одноименный язык программирования OpenCL [2].

1.1.1. Модель платформы

Схема модели платформы приведена на рисунке 1.1. Модель включает в себя одно или несколько устройств (Compute device), управляемых хостом (Host). Устройства состоят из одного или более вычислительных элементов (Compute units), которые делятся на один или несколько рабочих элементов (Processing element). Вычисления на устройстве выполняются рабочими элементами [1].

OpenCL-приложения выполняются на хосте в соответствии с моделью исполнения системы хоста (например, это могут быть исполняемые .exe-файлы для платформ семейства Windows). OpenCL-приложение отправляет команды на устройство, чтобы выполнить на нем параллельные вычисления.

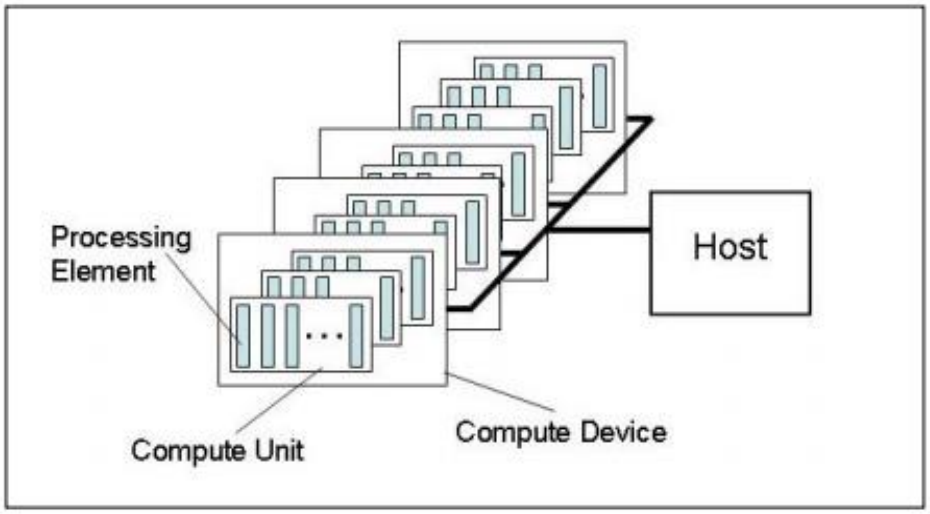


Рисунок 1 - Схема модели платформы

1.1.2. Модель выполнения

Выполнение OpenCL-программы происходит в двух частях: ядра (kernel), которое выполняется на одном или нескольких устройствах и хост-программа, которая выполняется на хосте. Хост-программа определяет контекст для ядер и управляет их выполнением.

Когда ядро отправляется хостом на выполнение устройством, определяется индексное пространство. Экземпляр ядра выполняется для каждой точки в этом индексном пространстве. Этот экземпляр ядра называется рабочим элементом (Work item) и определяется своей точкой в индексном пространстве, которая обеспечивает глобальный идентификатор рабочего элемента. Каждый рабочий элемент выполняет один и тот же код, но конкретный путь выполнения кода и обрабатываемые данные могут варьироваться в зависимости от рабочего элемента. Рабочие элементы организованы в рабочие группы (Work groups). Рабочие группы обеспечивают более крупное разложение индексного пространства. Рабочим группам присваивается уникальные идентификаторы с той же размерностью, что и индексное пространство, используемое для рабочих элементов. Заданиям присваивается уникальный локальный идентификатор в рабочей группе, так что отдельный рабочий элемент может быть однозначно идентифицирован по его глобальному идентификатору или комбинации его локального идентификатора и идентификатора рабочей группы. Рабочие элементы в данной рабочей группе выполняют одновременно элементы обработки одного вычислительного блока. Индексное пространство, поддерживаемое в OpenCL, называется NDRange. NDRange – это N-мерное индексное пространство, где N – натуральное число. NDRange определяется целочисленным массивом длины N, определяющим размер индексного пространства в каждом измерении, начиная с индекса смещения F (ноль по умолчанию). Глобальный идентификатор и локальный идентификатор каждого рабочего элемента представляют собой N-мерные кортежи. Глобальный идентификатор компоненты в некотором измерении - это значения в диапазоне от F до F плюс количество элементов в этом

измерении минус один. Рабочим группам присваиваются идентификаторы с использованием подхода, аналогичного тому, который используется для глобальных идентификаторов рабочих элементов.

На рисунке 1.2. показана схема индексации элементов и групп.

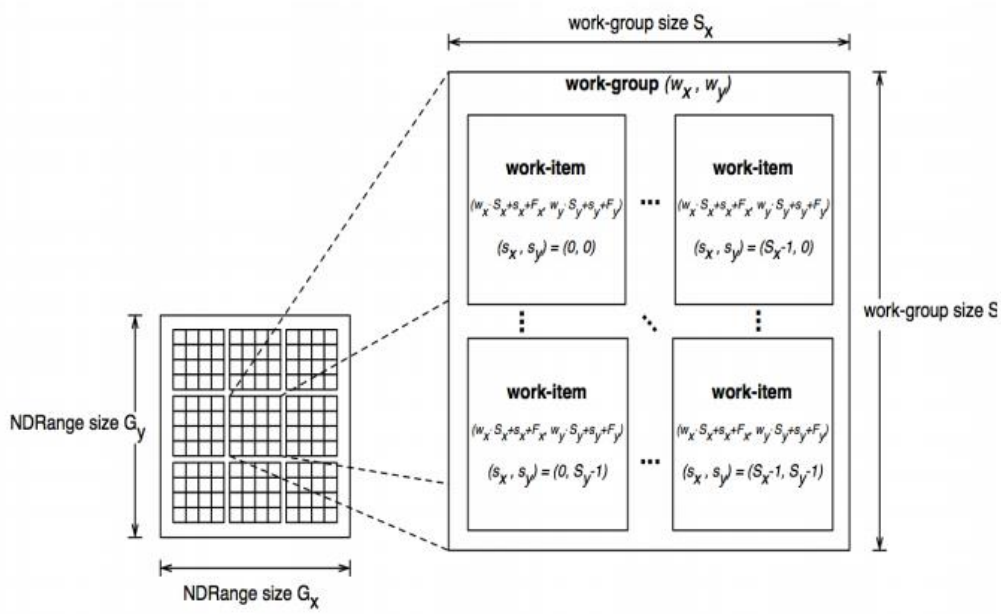


Рисунок 2 - Схема индексации элементов и групп

1.1.3. Язык программирования OpenCL

Язык программирования OpenCL C (также называемый OpenCL C) основан на спецификации языка C ISO / IEC 9899: 1999 (он же C99).

Язык поддерживает векторные типы данных, допускаются значения длины вектора: 2, 3, 4, 8 и 16.

Вставить таблицу векторных типов

Расписать за адресные пространства

1.1.4. Модель памяти

Рабочие элементы, выполняющие ядро, имеют доступ к четырем отдельным областям памяти:

- Глобальная память. Эта область памяти разрешает доступ для чтения / записи всем рабочим элементам во всех рабочих группах. Рабочие элементы могут читать или записывать в любой элемент объекта памяти.

Операции чтения и записи в глобальную память могут кэшироваться в зависимости от возможностей устройства.

- Постоянная память: область глобальной памяти, которая остается постоянной во время исполнения ядра. Хост выделяет и инициализирует объекты памяти, помещенные в постоянную память.
- Локальная память: область памяти, локальная для рабочей группы. Эта область памяти может быть использоваться для распределения переменных, которые являются общими для всех рабочих элементов в этой рабочей группе. Это может быть реализовано в виде выделенных областей памяти на устройстве OpenCL. В качестве альтернативы область локальной памяти может быть отображена на разделы глобальной памяти.
- Частная память: область памяти, принадлежащая рабочему элементу. Переменные, определенные в частной памяти рабочего элемента, не видны другому рабочему элементу.

На рисунке 1.3 приведена схема модели памяти.

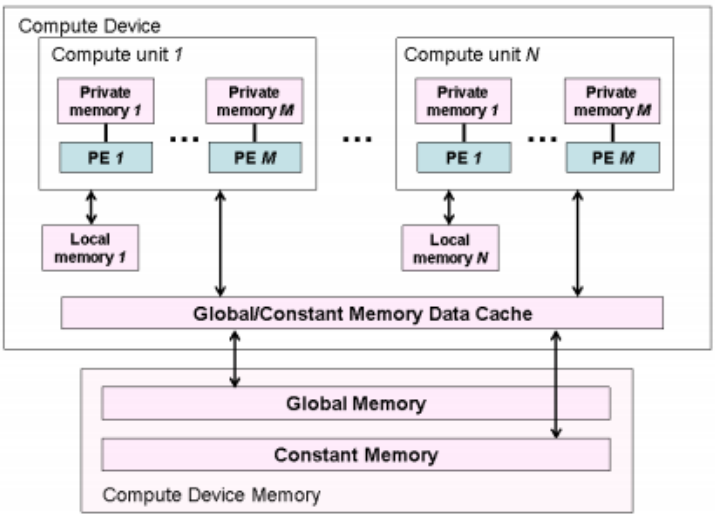


Рисунок 3 - схема модели памяти OpenCL

1.2. Отладчики

Отладка – поиск и устранение ошибок в компьютерной программе. Отладка связана с определением текущих значений переменных и путей выполнения, по которым идет программа.

Отладчик – это программное средство, контролирующее отлаживаемое (целевое) приложение таким образом, что программист может проследить за его выполнением и в выбранной точке посмотреть состояние программы, чтобы проверить ее корректность [1].

Автономные отладчики – это приложения, предназначенные исключительно для отладки. Примером такого отладчика служит Borland Turbo Debugger.

1.2.1. Классификация отладчиков

Интегрированные отладчики включены в среду разработки с целью повышения продуктивности разработчика.

Низкоуровневые отладчики позволяют выполнять пошагово инструкции машинного кода целевого приложения. Поэтому, отладка с их помощью должна производиться отдельно на различных аппаратных средствах. Они не совершают декомпиляцию, поэтому проще в реализации [1].

Высокоуровневые (символьные) отладчики позволяют следить за пошаговым выполнением исходного кода (на языках программирования высокого уровня), абстрагируясь от инструкций процессора. Однако, иногда программисту требуется опуститься на низкий уровень, чтобы понять, как программа выполняется на аппаратном обеспечении. Поэтому символьные отладчики также должны предоставлять низкоуровневую информацию. Обычно это достигается путем получения состояния регистров и дампа памяти. [1]

1.3. Обзор аналогов

1.2.1. Критерии сравнения

Кроссплатформенность. Поскольку OpenCL-программы зачастую компилируются драйвером видеокарты непосредственно перед выполнением, машинный код зависит от драйвера и платформы. Это значит, что в большинстве случаев отладку необходимо выполнять под разными ОС. Будем считать отладчик платформонезависимым, если он представлен в виде работающих дистрибутивов для Windows и Linux.

Устройствонезависимость. Так как OpenCL-программы могут выполняться на большом количестве различных устройств и часто появляются новые устройства, важно, чтобы корректность работы отладчика не зависела от устройства. Работа отладчика не зависит от устройства, если он работает корректно со всеми устройствами, включая новые, и не требует для этого добавления их поддержки от разработчиков.

Возможность интеграции в среды разработки (IDE) требуется в связи с современной практикой разработки, подразумевающей использование IDE. Она повышает удобство и производительность программиста и расширяет круг возможных пользователей.

1.2.2. Существующие OpenCL-отладчики

CodeXL – это отладчик для устройств AMD. Позволяет профилировать программы [1]. Предоставляет информацию о количестве ипользуемых регистров, загруженности шин памяти и т.п. Не поддерживается с 2019 года. Не работает с новыми устройствами. Недоступен в виде дистрибутива. Не работает под линуксами.

Oclgrind – симулятор OpenCL-устройства (использует виртуальные устройства) [2]. При наличии профиля (набора характеристик) видеокарты позволяет отладку под нее. Позволяет автоматически находить ошибки обращений к памяти, гонки потоков.

OpenCL API Debugger – отладчик для CPU и GPU производства Intel. Входит в пакет Intel SDK For OpenCL [3]. Не работает с AMD-GPU.

NVIDIA Nsight – отладчик для GPU производства Nvidia. Не работает с AMD-GPU. Позволяет работать не только с OpenCL-программами, но и с программами на Cuda [4].

gDEBugger CL позволяет отлаживать OpenCL- и OpenGL-программы, профилировать, изменять ядра во время их выполнения, анализировать доступы к памяти [5].

В таблице 1.1 приведено сравнение существующих OpenCL-отладчиков по критериям, сформулированным выше.

Таблица 1.1 – Сравнение существующих отладчиков

Отладчик	CodeXL	Oclgrind	OpenCL API Debugger	NVIDIA Nsight	gDEBugger CL
Платформонезависимость	-	+	+	+	+
Устройствонезависимость	-	+	-	-	+
Интеграция с IDE	+	-	+	+	-

1.2.3. Выводы по итогам сравнения

Только Oclgrind и gDEBugger работают со всеми устройствами. Но Oclgrind использует виртуальные устройства, что не позволяет учитвать все особенности архитектуры. Кроме того, это делает профилирование невозможным. Ни один из устройствонезависимых отладчиков не имеет плагина для популярных сред разработки. Все отладчики, кроме CodeXL, работают как под ОС Windows, так и под Linux. Как инструмент разработки лучше всего подходит gDEBugger: он представляет все возможности для отладки и профилирования, работает со всеми устройствами и популярными ОС. Но он выполнен в виде отдельного приложения с gui, что делает невозможным создание адаптера для интегрирования его в IDE.

2. ФОРМУЛИРОВКА ТРЕБОВАНИЙ К ПРОГРАММЕ

2.1. Устройствонезависимость

Устройствонезависимость требуется, чтобы программа позволяла отлаживать OpenCL-ядра на любых устройствах (частности, интерес представляют новые устройства фирмы AMD). Кроме непосредственно возможности отладки, это делает ее удобнее: например, в системе с несколькими гетерогенными устройствами программист сможет использовать только один инструмент отладки.

2.2. Возможность интеграции

Возможность интеграции отладчика со средами разработки повышает удобство пользователя. Кроме того, для современных отладчиков это фактически стало стандартом.

2.3. Интерфейс командной строки

Интерфейс командной строки (Command line interface, CLI) упрощает интеграцию с другими программами (в том числе IDE), но не исключает возможности добавления в отладчик собственного графического интерфейса.

2.4. Высокоуровневость

OpenCL позволяет программисту абстрагироваться от сложных наборов инструкций конкретного устройства, которые, зачастую, в большой мере архитектурно-специфичны и насчитывают сотни инструкций. Поэтому, низкоуровневая отладочная информация не обязательна для разработки OpenCL-приложений. Низкоуровневый подход подразумевает работу с машинным кодом. OpenCL-программы могут запускаться на различных устройствах с разными архитектурами и наборами инструкций (например, 5 архитектур AMD GCN и соответствующих наборов инструкций [?]), поэтому низкоуровневый подход исключает создание устройствонезависимого отладчика. Сим-

вольный подход позволяет достичь устройствонезависимости за счет работы с исходным кодом.

2.5. Платформонезависимость

OpenCL-приложения могут отлаживаться в системах под управлением различных операционных систем (например, семейств Linux и Windows). Поэтому, отладчик должен быть совместим с этими системами.

Обоснование постановки задачи (требований к решению) можно дать либо в данном либо в предыдщуем разделе.

Низкоуровневый подход подразумевает работу с машинным кодом. OpenCL-программы могут запускаться на различных устройствах с разными архитектурами и наборами инструкций (например, 5 архитектур AMD GCN и соответствующих наборов инструкций [?]), поэтому низкоуровневый подход исключает создание устройствонезависимого отладчика. Символьный подход позволяет достичь устройствонезависимости за счет работы с исходным кодом.

3. ОПИСАНИЕ РЕШЕНИЯ

Отладчик построен с использованием высокоуровневого подхода. Это значит, что он не работает с машинным кодом или кодом низкоуровневой виртуальной машины (LLVM), как это делают устройство-специфичные отладчики (например, CodeXL). Вместо модификации низкоуровневого кода целевого приложения (OpenCL-ядра), отладчик модифицирует исходный код, написанный на языке OpenCL.

Программа написана на языке программирования Python3, так как этот язык позволяет достичь высокой скорости и качества разработки.

3.1.1. Структуры данных

3.1.1.1. ClTypes

Класс `ClTypes` хранит в статических полях информацию о системе типов OpenCL:

- названия скалярных типов данных;
- названия векторных типов данных;
- названия целочисленных типов данных;
- названия знаковых целочисленных типов данных;
- названия беззнаковых целочисленных типов данных;
- названия вещественных типов данных;
- синоним для типа указателя;
- названия скалярных типов, которым соответствуют векторные типы;
- возможные длины векторов (переменных векторного типа);
- словарь, определяющий парсер для каждого примитивного скалярного типа;
- словарь printf-флагов для примитивных скалярных типов;
- статический метод, реализующий генерацию printf-флагов для любых встроенных типов данных (включая векторные).

Парсинг скалярных величин реализован в библиотеке numpy.

3.1.1.2. Declaration

Класс `Declaration` – базовый класс объявления переменной и поля структуры. Имеет методы:

- `is_struct()` – возвращает `True` или `False` если объект является объявлением структуры или нет соответственно;
- `words_num()` – возвращает количество слов, разделенных пробелом, которые требуются для текстового представления значения объявленной переменной в формате, используемом отладчиком для получения значений переменных OpenCL-программы.

3.1.1.3. VarDeclaration

Класс `VarDeclaration` хранит информацию об объявлении переменной в следующих полях:

- `var_name` – имя переменной;
- `full_type` – полная информация о типе;
- `address_space` – модификатор адресного пространства;
- `is_struct` – булево значение, определяющее, имеет ли переменная составной тип (структура);
- `var_type` – название типа переменной;
- `pointer_rank` – порядок указателя (если переменная является указателем);
- `var_shape` – форма массива (если переменная является массивом).

Конструктор получает имя переменной и полный тип, по которому определяет значения полей.

3.1.1.4. FieldDeclaration

Класс `FieldDeclaration` хранит информацию об объявлении переменной в следующих полях:

- `var_name` – имя переменной;
- `full_type` – полная информация о типе;
- `is_struct` – булево значение, определяющее, имеет ли переменная составной тип (структура);
- `var_type` – название типа переменной;
- `pointer_rank` – порядок указателя (если переменная является указателем);
- `var_shape` – форма массива (если переменная является массивом).

Отличается от `VarDeclaration` тем, что не содержит информации об адресном пространстве, так как адресное пространство полей структуры совпадает с оным у самой структуры и не указывается явно.

3.1.1.5. StructDeclaration

Класс `StructDeclaration` хранит информацию об объявлении структуры:

- поле `fields` представляет из себя словарь и хранит объявления полей структуры;
- поле `name` содержит имя структуры.

Так же как `Declaration`, имеет метод `words_num()` – возвращает количество слов, разделенных пробелом, которые требуются для текстового представления значения объявленной переменной в формате, используемом отладчиком для получения значений переменных OpenCL-программы.

3.1.1.6. Variable

Класс `Variable` хранит информацию об объявлении переменной в поле `decl` и ее значение в точке останова в поле value.

Конструктор получает объявление переменной и текстовое представление ее значения, полученное от целевого приложения, которому сопоставляется ее значение в виде, соответствующем типу. Для этого класс имеет следующие методы:

- `__parse_var` – возвращает значение переменной любого типа;
- `__parse_value` – возвращает значение одиночной переменной (не массива);
- `__parse_scalar_value` – возвращает значение переменной скалярного типа;
- `__parse_vector_value` – возвращает значение переменной векторного типа;
- `__parse_struct` – возвращает значение переменной пользовательского типа;
- `__parse_array` – возвращает значения элементов массива и указатель на его начало;
- `__parse_1d_array` – то же для одномерных массивов;
- `__parse_2d_array` – то же для двумерных массивов;
- `__parse_3d_array` – то же для трехмерных массивов;

Непосредственно в конструкторе вызывается только метод `__parse_var`, который, в свою очередь, вызывает остальные методы в зависимости от типа переменной. Иерархия вызовов методов представлена на рисунке 3.1

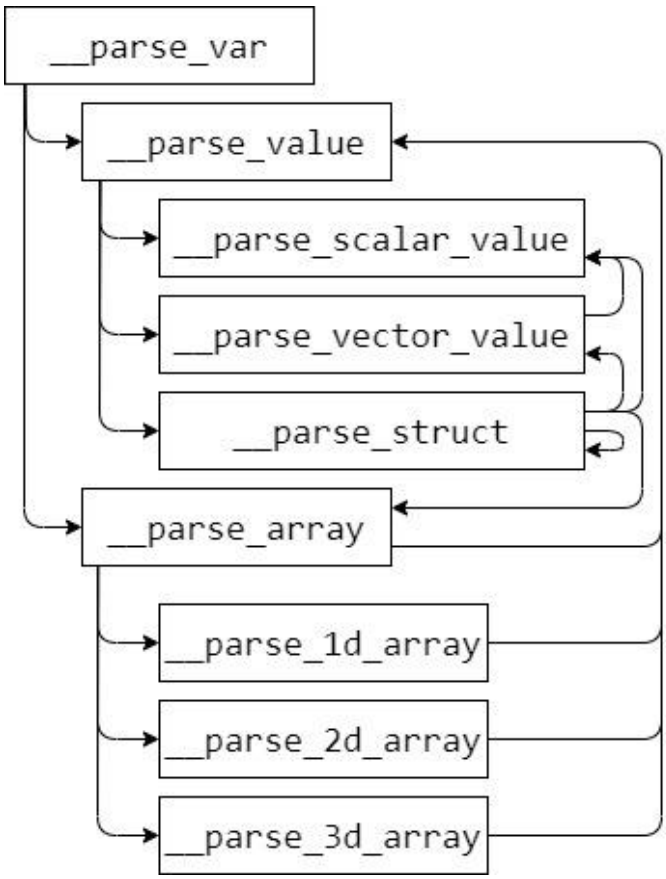


Рисунок 3.1 - Иерархия вызовов методов класса `Variable`

3.1.2. Передача состояния устройства

Для инспекции регистров и памяти OpenCL-устройства необходимо передать их состояния с устройства в систему, управляющую им. Сделать это можно тремя способами:

- Через пользовательские OpenCL-буферы – влечет возможность переполнения буферов, так как максимальный размер буфера, который может быть выделен в памяти устройства, определяется драйвером устройства;
- Через специальный printf-буфер (специальный буфер, определенный неявно, который служит для пользовательского отладочного вывода[?]) – уменьшает пропускную способность канала между устройством и управляющей им системой из-за кодирования данных как текста;
- С использованием API драйвера устройства – исключает устройствонезависимость отладчика, так как драйвера специфичны к устройству (или его производителю).

Отладчик использует printf-буффер, так как им управляет драйвер устройства, что позволяет получить с устройства данные любого размера.

Для этого он модифицирует исходный код OpenCL-ядра таким образом, что ядро при выполнении выводит в него целевую информацию.

Чтобы модифицировать исходный код ядра, используется библиотека Clang (libclang). Она позволяет построить абстрактное синтаксическое дерево (AST) кода. AST содержит информацию о типах переменных программы и объявленных структурах. С помощью этой информации код меняется таким образом, что при выполнении выводит в printf-буффер состояния переменных и областей памяти.

3.1.2.1. Формат данных

При передаче данных от целевого приложения к отладчику, они представляются в текстовом виде.

Переменная любого типа (включая массивы, структуры, числа, векторы, указатели) представляется одной строкой, завершающейся символом перевода строки. Строка всегда начинается с имени переменной.

Для переменных скалярных типов непосредственно после имени переменной идет ее значение:

- для целочисленных типов – шестнадцатеричное представление. Например, переменная, объявленная как “int a = 16” представляется как “a 0x10”
- для вещественных типов – десятичное представление. Например, переменная, объявленная как “float b = 0.1000” представляется как “b 0.1”

Для переменных векторных типов после имени переменной идут значения элементов вектора, разделенные запятой. Правила представления элементов вектора те же, что для скалярных типов. Например, переменная, объявленная как “int4 c = (int4)(1, 2, 3, 4)” представляется как “c 0x1,0x2,0x3,0x4”.

Для одномерных массивов после имени идет указатель на начало массива в шестнадцатеричной записи. В OpenCL указатели имеют длину 4 байта. После него через пробел идут значения элементов массива. Например, массив, объявленный как “char d[3] = {1, 2, 3}” может представляться как “d 0xffffffff 0x1 0x2 0x3”.

Двумерные массивы рассматриваются как массивы одномерных массивов, которые представляются по тем же правилам, что одиночные одномерные массивы. Правила представления трехмерных массивов аналогичны. . Например, массив, объявленный как “char e[3][1] = {1, 2, 3}” может представляться как “d 0x00000000 0x00000000 0x1 0x00000001 0x2 0x00000002 0x3”.

Массивов с размерностью большей, чем 3, в языке OpenCL нет.

Для структуры значение каждого поля представляется по правилам, указанным выше, последовательно через пробел. Если структура содержит другую структуру как поле, то правило представления структур принимает рекурсивный характер. Эта рекурсия не может быть бесконечной, так как число типов структур в программе конечно.

Например, если определено две структуры:

```
struct my_struct1 {
     int count = 1;
}
struct my_struct2 {
     my_struct1 f;
     int num = 4;
}
```

то переменная, объявленная как “`my_struct2 a;`” представляется как “a f count 1 num 4”.

3.1.2.2. Структура программной реализации

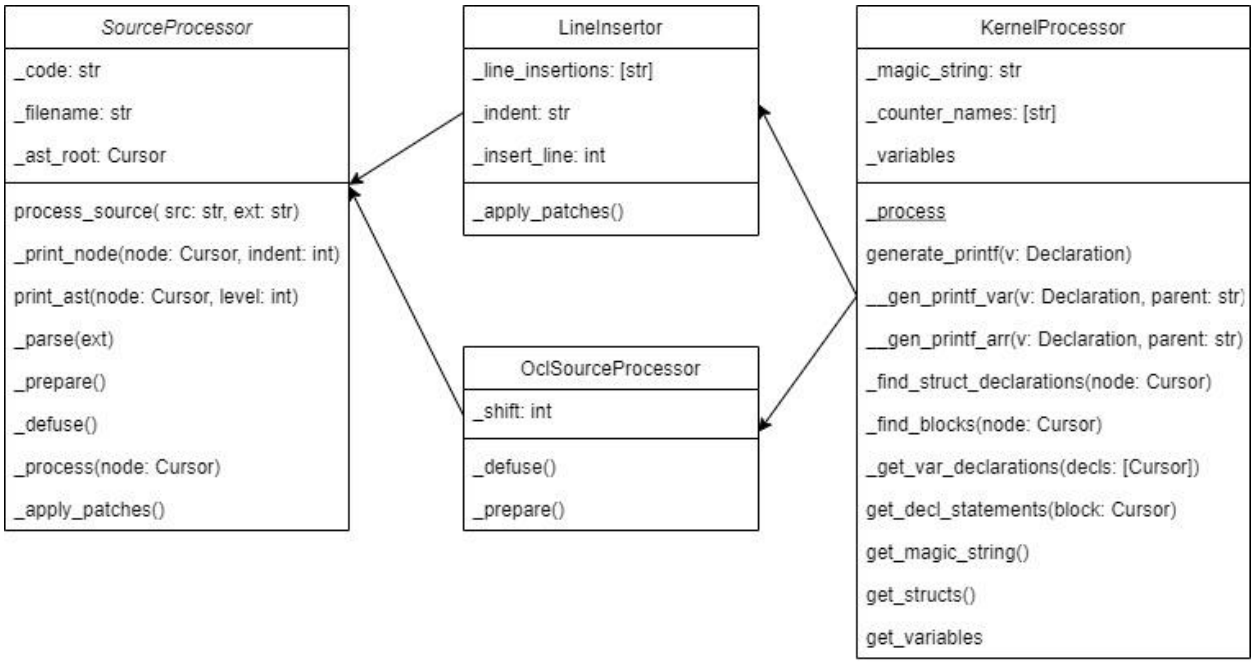
Построение AST кода, написанного на си-подобных языках, с помощью библиотеки Clang реализует класс `SourceProcessor.` Это абстрактный класс: он непосредственно реализует лишь методы построения AST и его текстового представления (которое требуется для отладки), оставляя реализацию методов изменения кода своим потомкам. Исходный код класса `SourceProcessor` представлен в приложении А.

Построение AST OpenCL-кода имеет свою специфику. Класс `OclSourceProcessor` является наследником класса `SourceProcessor` и реализует его абстрактные методы `_prepare()` и _defuse(), которые подготавливают код к построению AST средствами Clang: определяют встроенные типы языка OpenCL. Это необходимо, потому что обертка Clang для языка Python не предоставляет полного доступа к функционалу библиотеки. Исходный код класса `OclSourceProcessor` представлен в приложении Б.

Класс `LineInserter` является наследником класса `SourceProcessor` и реализует функционал добавления новых строк в исходный код. Он хранит номер строки, после которой вставляет новый код, в поле `_insert_line`, а код, подлежащий добавлению, в поле `_line_insertions` в виде массива строк. Метод `_apply_patches()` добавляет новые строки. Исходный код класса `LineInserter` представлен в приложении Б.

Класс `PrintfInserter` является наследником классов `OclSourceProcessor` и `LineInserter` по схеме ромбовидного наследования. Он реализует весь функционал, связанный с модификацией OpenCL-ядра. Сначала он обходит AST и сохраняет участки кода, содержащие точку останова (целевые), в иерархическом порядке в поле. Затем для каждого целевого участка кода он находит все переменные, объявленные в нем, и сохраняет данные об их объявлении. Таким образом, формируется список всех переменных (включая массивы, структуры и указатели), которые существуют

в программе в точке останова. После этого генерируется код, который будет вставлен в целевое OpenCL-ядро.



п*одписать*

Прямого (явного) доступа к целевому буферу OpenCL не предоставляет, поэтому он реализован как вызов функции printf() – специальной встроенной функции, которая передает хосту данные для вывода им в поток вывода приложения.

3.1.3. Получение состояния устройства

OpenCL-приложения обычно компилируют ядро для целевого устройства средствами его драйвера из исходного кода. Код ядра может как читаться из отдельного файла во время выполнения OpenCL-приложения, так и храниться непосредственно в памяти приложения (иногда, в зашифрованном виде). Поэтому после модификации исходного кода OpenCL-ядра целевое приложение нужно заново скомпилировать и слинковать.

После этого целевое приложение запускается отладчиком как дочерний процесс. Отладчик ждет, когда получит из потока вывода целевого приложения магическую строку – она сигнализирует о начале отладочного вывода. В качестве магической строки подбирается такая последовательность символов, которую целевое приложение никогда не выведет без модификации (то есть вероятность этого события должна быть пренебрежимо мала). Затем отладчик считывает данные и предоставляет пользователю.

Этот функционал реализует класс `OclDebugger`. Он имеет следующие методы:

- `__init__` - конструктор класса. Получает путь к файлу исходного кода ядра, путь к целевому приложению, команду для сборки целевого приложения.
- `_debug` – служебный метод. Модифицирует файл ядра с помощью объекта класса `PrintfInserter`, собирает целевое приложение, запускает его как свой подпроцесс, получает из его потоков вывода информацию о значениях переменных в точке останова. Запуск целевого приложения реализован асинхронно с помощью механизма async/await языка python. Это позволяет создать объект-генератор, выдающий строки из потоков вывода целевого приложения по мере их вывода. Генератор избавляет от необходимости получать весь вывод целевой программы целиком, разбивать его на строки и хранить в памяти отладчика. Это может снизить время простоя в ожидании данных от устройства, если инспектируется большое количество потоков.
- `_build_env` – служебный метод. Создает среду (словарь переменных окружения и их значений) для запуска целевого приложения.
- `_build` – реализует сборку целевого приложения с использованием механизма async/await языка python.
- `process_values` – создает объекты типа `Variable`, содержащие информацию о переменных ядра и их значениях в точке останова, полученную

от модифицированного целевого приложения. Получает данные из генератора `value_generator`.

- `value_generator` – метод-генератор, выдающий последовательно значения переменных, полученных от модифицированного целевого приложения.
- `safe_debug` – получает как параметр точку останова, возвращает список объектов типа `Variable`, содержащих значения переменных в точке останова. По сути является оберткой над методом `_debug`. Перед вызовом последнего сохраняет копию файла OpenCL-ядра целевого приложения, а после отменяет изменения, сделанные в этом файле. Благодаря механизму try/except/finally языка Python, модификации ядра будут отменены в любом случае: даже если метод `_debug` спровоцировал ошибку. Таким образом обеспечивается безопасность отладчика для исходного кода целевой программы.

2.1. Интерфейс пользователя

3. ОБЕСПЕЧЕНИЕ КАЧЕСТВА РАЗРАБОТКИ, ПРОДУКЦИИ, ПРОГРАММНОГО ПРОДУКТА (ДОП. РАЗДЕЛ)

Сначала определяются четкие требования к формату входных и выходных данных элемента (алгоритма, функции, метода). После этого создается набор тестов, проверяющих соответствие результата работы тестируемого элемента или всей программы сформулированным требованиям. Такой подход к разработке называется TDD (test driven development). Он был применен при создании программы, описанной в этой работе.

3.2. Юнит-тесты

Качество разработки отладчика (программного продукта) достигается путем тестирования нового функционала по мере его реализации. Тестирование реализовано в виде юнит-тестов – тестов, проверяющих корректность работы единицы функционала программы. Такой подход называется модульным тестированием. Он упрощает отладку за счет модульной структуры: модули отлаживаются независимо друг от друга. Юнит-тесты были сделаны с использованием библиотеки unittest.

3.2.1. Тестирование парсинга данных

Для тестирования парсинга данных, получаемых отладчиком от целевого приложения, создан набор юнит-тестов (который называется тест-кейсом). Каждый тест проверяет корректность определенного сценария.

В соответствии с форматом данных, определенным ранее (см п.??), значение переменной целочисленного типа передается в виде строки, содержащей название переменной и ее значение в шестнадцатиричном представлении, разделенные пробелом. Например, беззнаковая однобайтная переменная, объявленная, как “uchar c = 255” представляется как строка “c 0xff”, а переменная, объявленная как “char c = -1” представляется как строка “c 0xff”. То есть, знаковые и беззнаковые переменные могут представляться одинаково. Поэтому, для целочисленных типов данных тесты с отрицательными и положительными значениями проводятся отдельно.

Тест `test_char_positive` проверяет корректность парсинга значения переменной однобайтного целочисленного знакового типа: строка “a 12” записывает значение 0x12=18d переменной a.

- Входные данные: строка “a 12”
- Выходные данные: целое число 0x12

Тест `test_char_negative` проверяет корректность парсинга значения переменной однобайтного целочисленного знакового типа: строка “a ff” записывает значение 0xff=-1 переменной a, так как тип однобайтный, а значения представлены в регистрах (или памяти) в дополнительном коде.

- Входные данные: “a ff”
- Выходные данные: -1

Тест `test_char_prefix` проверяет корректность парсинга значения переменной однобайтного целочисленного знакового типа: строка “a 0x70” записывает значение 0x70=112 переменной a. Он показывает, что парсинг шестнадцатиричного числа может выполняться не только без префикса 0x, но и с ним.

- Входные данные: “a 0x70”
- Выходные данные: 0x70

Тест `test_char_too_much_digits` проверяет корректность парсинга значения переменной однобайтного целочисленного знакового типа в случае, если запись числа насчитывает слишком много цифр. В таком случае, значащими считаются младшие цифры (две шестнадцатиричные цифры в случае однобайтного целочисленного типа).

- Входные данные: “a 0x1f100”
- Выходные данные: 0x00

Тест `test_uchar_positive` проверяет корректность парсинга значения переменной однобайтного целочисленного беззнакового типа: строка “a 12” записывает значение 0x12=18d переменной a.

- Входные данные: “a 0x12”
- Выходные данные: 0x12

Тест `test_uchar_negative` проверяет корректность парсинга значения переменной однобайтного целочисленного беззнакового типа: строка “a ff” записывает значение 0xff=255 переменной a.

- Входные данные: “a 0xff”
- Выходные данные: 255

Тест `test_uchar_prefix` проверяет корректность парсинга значения переменной однобайтного целочисленного безнакового типа: строка “a 0x70” записывает значение 0x70=112 переменной a. Он показывает, что парсинг шестнадцатиричного числа может выполняться не только без префикса 0x, но и с ним.

- Входные данные: “a 0x70”
- Выходные данные: 0x70

Тест `test_uchar_too_much_digits` проверяет корректность парсинга значения переменной однобайтного целочисленного знакового типа в случае, если запись числа насчитывает слишком много цифр. В таком случае, значащими считаются младшие цифры (две шестнадцатиричные цифры в случае однобайтного целочисленного типа).

- Входные данные: “a 0x77700”
- Выходные данные: 0x00

Тест `test_int` проверяет корректность парсинга значения переменной четырехбайтного целочисленного знакового типа: строка “a 141” записывает значение 0x141 переменной a.

- Входные данные: “a 0x141”
- Выходные данные: 0x141

Тест `test_int_too_much_digits` проверяет корректность парсинга значения переменной однобайтного целочисленного знакового типа в случае, если запись числа насчитывает слишком много цифр. В таком случае, значащими считаются младшие цифры (восемь шестнадцатиричных цифр в случае четырехбайтного целочисленного типа).

- Входные данные: “a 0x777111111ff”
- Выходные данные: 0x111111ff

Тест `test_1d_arr` проверяет корректность парсинга значения одномерного массива целых четырехбайтных знаковых чисел длины 3. Такой массив представляется строкой, содержащей последовательно через пробел имя массива, указатель на его начало и три значения элементов массива. Например, строка “a 0xffffffff 1 2 3” записывает массив a = [1, 2, 3], начинающегося по смещению 0xffffffff – эти данные используются в тесте.

- Входные данные: “a 0xffffffff 1 2 3”
- Выходные данные: (0xffffffff, [1,2,3])

Тест `test_2d_arr` проверяет корректность парсинга значения двумерного массива (матрицы) целых четырехбайтных знаковых чисел формы (3,2). Такой массив представляется строкой, содержащей последовательно через пробел имя массива, указатель на его начало и три значения строк матрицы. Например, строка “`a 0x00000000 0x00000000 1 2 0x00000008 3 4`

`0x0000000C 5 6`” записывает массив a = [[1, 2], [3, 4], [5, 6]], начинающегося по смещению `0xffffffff` – эти данные используются в тесте.

- Входные данные: “a 0x00000000 0x00000000 1 2 0x00000008 3 4 0x0000000C 5 6”
- Выходные данные: (0x00000000, [(0x00000000, [1, 2]), (0x00000008, [3, 4]), (0x0000000C, [5, 6])])

Так как для чисел вещественного типа представление содержит знак, тесты для этих типов проводятся на положительных и отрицательных значениях.

Тест `test_float_positive` проверяет корректность парсинга положительного значения переменной четырехбайтного вещественного типа: строка “ a 0.1” записывает значение 0.1 переменной a.

- Входные данные: “a 0.1”
- Выходные данные: 0.1

Тест `test_float_negative` проверяет корректность парсинга положительного значения переменной четырехбайтного вещественного типа: строка “ a -0.133” записывает значение -0.13 переменной a.

- Входные данные: “a -0.133”
- Выходные данные: -0.133

Тест `test_double_positive` проверяет корректность парсинга отрицательного значения переменной четырехбайтного вещественного типа: строка “ a 0.1” записывает значение 0.1 переменной a.

- Входные данные: “a 0.1”
- Выходные данные: 0.1

Тест `test_double_negative` проверяет корректность парсинга отрицательного значения переменной четырехбайтного вещественного типа: строка “ a -0.133” записывает значение -0.13 переменной a.

- Входные данные: “a -0.133”
- Выходные данные: -0.133

Для тестов с векторными типами данных выбраны типы double2, double4, double8, double16.

Тест `test_double2` проверяет корректность парсинга положительного значения переменной четырехбайтного вещественного типа: строка “ a -0.133” записывает значение -0.13 переменной a.

- Входные данные: “ a -0.133,0.1”
- Выходные данные: [0.133, 0.1]

Тест `test_double4` проверяет корректность парсинга положительного значения переменной четырехбайтного вещественного типа: строка “ a -0.133” записывает значение -0.13 переменной a.

- Входные данные: “ a -0.133,0.1,0.2,0.3”
- Выходные данные: [0.133, 0.1, 0.2, 0.3]

Тест `test_double8` проверяет корректность парсинга положительного значения переменной четырехбайтного вещественного типа: строка “ a -0.133” записывает значение -0.13 переменной a.

- Входные данные: “ a -0.133,0.1,0.2,0.3,-0.133,0.1,0.2,0.7”
- Выходные данные: [-0.133, 0.1, 0.2, 0.3, -0.133, 0.1, 0.2, 0.7]

Тест `test_double16` проверяет корректность парсинга положительного значения переменной четырехбайтного вещественного типа: строка “ a -0.133” записывает значение -0.13 переменной a.

- Входные данные: “a -0.133,0.1,0.2,0.3,-0.133,0.1,0.2,0.7,-0.133,0.1,0.2,0.3,-0.133,0.1,0.2,0.7”
- Выходные данные: [-0.133, 0.1, 0.2, 0.3, -0.133, 0.1, 0.2, 0.7, -0.133, 0.1, 0.2, 0.3, -0.133, 0.1, 0.2, 0.7]

Для тестов, работающих со структурами, было определено пять структур:

```
struct my_struct_1
{
    int count;
    double2 v;
};

struct my_struct_2
{
    int count;
    my_struct_1 v;
};

struct my_struct_3
{
    int count;
    int a[3];
};
struct my_struct_4
{
    int count;
    int a[3][2];
};
```

```
struct my_struct_5
{
    int count;
    int a[3][2][1];
};
```

Это делается в методе `setUp()` класса, реализующего тест-кейс.

Тест `test_simple_struct1` проверяет корректность парсинга значения переменной составного типа (структуры) с именем s, содержащей только поля встроенных типов:

- Входные данные: “s count 0x1488 v 0.1,0.2”
- Выходные данные: {'count': 0x1488, 'v': [0.1, 0.2]}

Тест `test_simple_struct2` проверяет корректность парсинга значения переменной составного типа (структуры) с именем s, содержащей только поля встроенных типов:

- Входные данные: “s count 0x145 v 0.1,0.3”
- Выходные данные: {'count': 0x145, 'v': [0.1, 0.3]}

Тест `test_struct_with_struct_as_a_field` проверяет корректность парсинга значения переменной составного типа (структуры) с именем s, содержащей другую структуру как поле:

- Входные данные: “s count 0x145 v count 0x231 v 0.1,0.3”
- Выходные данные: {'count': 0x145, 'v': {'count': 0x231, 'v': [0.1, 0.3]}}

Тест `test_array1d_struct` проверяет корректность парсинга значения переменной составного типа (структуры) с именем s, содержащей одномерный массив как поле:

- Входные данные: “ `s count 0x1488 a 0x00000000 0x71 0x198 0x43`”
- Выходные данные: `{'count': 0x1488, 'a': (0, [0x71, 0x198, 0x43])}`

Тест `test_array2d_struct` проверяет корректность парсинга значения переменной составного типа (структуры) с именем s, содержащей двумерный массив как поле:

- Входные данные: “ `s count 0x1488 a 0x00000000 0x00000000 0x00000000 1 0x00000004 2 0x00000008 0x00000008 3 0x0000000C 4 0x00000010 0x00000010 5 0x00000014 6`”
- Выходные данные: {'count': 0x1488, 'a': (0x00000000, [(0x00000000, [(0x00000000, [1]), (0x00000004, [2])]), (0x00000008, [(0x00000008, [3]), (0x0000000C, [4])]), (0x00000010, [(0x00000010, [5]), (0x00000014, [6])])])})

3.2.2. Тестирование парсинга типов

Библиотека Clang представляет тип переменной как строку. Эта строка может иметь сложную структуру: она содержит информацию о типе, размерах (для массивов), является ли переменная указателем, а в OpenCL еще и модификатор адресного пространства, указывающий физическую область памяти переменной (регистры, кэш, оперативная память). Поэтому требуется получать эту информацию из строки, характеризующей тип переменной, которую предоставляет Clang в одном из полей узла AST, описывающего объявление переменной. Этот функционал реализован в классах `VarDeclaration` и `FieldDeclaration`.

Тесты для парсинга типов, реализованного в `VarDeclaration`, представлены классом `VariableDeclarationTest.`

Тест `test_private_int1` проверяет корректность распознавания целочисленной четырехбайтной переменной, содержащейся в приватной памяти рабочего элемента.

- Входные данные: “__private int”
- Выходные данные: {var_type: 'int', address_space: '__private', is_array: False, var_shape: None, pointer_rank: 0, words_num(): 2, is_struct(): False}

Тест `test_private_int2` проверяет корректность распознавания целочисленной четырехбайтной переменной, содержащейся в приватной памяти рабочего элемента. Отличается от предыдущего порядком слов: в OpenCL оба варианта допустимы.

- Входные данные: “__private int”
- Выходные данные: {var_type: 'int', address_space: '__private', is_array: False, var_shape: None, pointer_rank: 0, words_num(): 2, is_struct(): False}

Тест `test_private_1d_array` проверяет корректность распознавания массива целочисленных четырехбайтных величин, содержащегося в приватной памяти рабочего элемента.

- Входные данные: “__private int [41]”
- Выходные данные: {var_type: 'int', address_space: '__private', is_array: True, var_shape: [41], pointer_rank: 0, words_num(): 43, is_struct(): False}

Тест `test_private_2d_array` проверяет корректность распознавания двумерного массива целочисленных четырехбайтных величин, содержащегося в приватной памяти рабочего элемента.

- Входные данные: “__private int [41][5]”

- Выходные данные: {var_type: 'int', address_space: '__private', is_array: True, var_shape: [41, 5], pointer_rank: 0, words_num(): 248, is_struct(): False}

Тест `test_private_3d_array` проверяет корректность распознавания трехмерного массива целочисленных четырехбайтных величин, содержащегося в приватной памяти рабочего элемента.

- Входные данные: “__private int [2][3][1]”
- Выходные данные: {var_type: 'int', address_space: '__private', is_array: True, var_shape: [2, 3, 1], pointer_rank: 0, words_num(): 16, is_struct(): False}

Тест `test_local_1d_array` проверяет корректность распознавания массива целочисленных четырехбайтных величин, содержащегося в локальной памяти рабочего элемента.

- Входные данные: “__local int [41]”
- Выходные данные: {var_type: 'int', address_space: '__ local, is_array: True, var_shape: [41], pointer_rank: 0, words_num(): 43, is_struct(): False}

Тест `test_local_2d_array` проверяет корректность распознавания двумерного массива целочисленных четырехбайтных величин, содержащегося в локальной памяти рабочего элемента.

- Входные данные: “__ local int [41][5]”
- Выходные данные: {var_type: 'int', address_space: '__ local, is_array: True, var_shape: [41, 5], pointer_rank: 0, words_num(): 248, is_struct(): False}

Тест `test_local_3d_array` проверяет корректность распознавания трехмерного массива целочисленных четырехбайтных величин, содержащегося в локальной памяти рабочего элемента.

- Входные данные: “__ local int [2][3][1]”
- Выходные данные: {var_type: 'int', address_space: '__ local, is_array: True, var_shape: [2, 3, 1], pointer_rank: 0, words_num(): 16, is_struct(): False}

Тест `test_struct` проверяет корректность распознавания переменной типа my_struct_1, который является структурой, содержащегося в локальной памяти рабочего элемента.

- Входные данные: `'__private struct my_struct_1`”
- Выходные данные: {var_type: 'my_struct_1', address_space: '__private, is_array: False, var_shape: None, pointer_rank: 0, words_num(): 5, is_struct(): True}

  Для этого теста специально была определена структура:

```
struct my_struct_1
{
    int count;
    double2 v;
};
```

Тест `test_undefined_struct` проверяет корректность распознавания переменной типа my_struct1, который не был определен. Это должно повлечь исключение, что и проверяется тестом.

- Входные данные: `'__private struct my_struct1`”

3.3. Интеграционные тесты

Интеграционный тест проверяет корректность работы класса OclDebugger, который реализует весь функционал отладчика. Тест представлен в классе OclDebuggerTest. Он получает состояния переменных специально написанного OpenCL-ядра, код которого представлен в приложении ??. Для однозначности переменные проинициализированы явным образом при объ-

явлении. В коде объявлено 15 переменных различных типов (включая скалярные, векторные, массивы, структуры):

1. Переменная объявлена как “`char c = 1;`”. Ее ожидаемое значение – 1.
2. Переменная объявлена как “`uchar uc = 2;`”. Ее ожидаемое значение – 2.
3. Переменная объявлена как “short `s = 3;`”. Ее ожидаемое значение – 3.
4. Переменная объявлена как “ushort `us = 4;`”. Ее ожидаемое значение – 4.
5. Переменная объявлена как “int `i = 5;`”. Ее ожидаемое значение – 5.
6. Переменная объявлена как “uint `ui = 6;`”. Ее ожидаемое значение – 6.
7. Переменная объявлена как “long `l = 7;`”. Ее ожидаемое значение – 7.
8. Переменная объявлена как “ulong `ul = 8;`”. Ее ожидаемое значение – 8.
9. Переменная объявлена как “float `f = 14.41;`”. Ее ожидаемое значение – 14.31.
10. Переменная объявлена как “double `d = -147.1;`”. Ее ожидаемое значение – `-147.1`.
11. Переменная объявлена как “`arr_1d[2] = {14, 17}`”. Ее ожидаемое значение – (<любое число>, [14, 17]).
12. Переменная объявлена как “`arr_2d[2][2] = {{14, 17}, {15, 31}}`”. Ее ожидаемое значение – (<любое число>, [(<любое число> [14, 17]), (<любое число>, [15, 31])]).
13. Переменная объявлена как “`arr_3d[2][2][2] = {{{14, 17}, {15, 31}}, {{1, 2}, {3, 4}}}`”. Ее ожидаемое значение – (<любое чис-

ло>, [(<любое число>, [(<любое число> [14, 17]), (<любое число>, [15, 31])]), (<любое число>, [(<любое число> [1, 2]), (<любое число>, [3, 4])])]).

14. Переменная объявлена как “`struct my_struct s1 = {{{14, 17}, {15, 31}}, (double2)(0.1, 0.2)};`”. Ее ожидаемое значение – {count: `{{14, 17}, {15, 31}}, w1: [0.1, 0.2]`}.

15. Переменная объявлена как “ `struct my_struct2 s2 = {{{{14, 17}, {15, 31}}, (double2)(0.1, 0.2)}, 0.12}`”. Ее ожидаемое значение – {a: {count: `{{14, 17}, {15, 31}}, w1: [0.1, 0.2]`}, w2: 0.12}.

Для массивов не проверяется значение указателя на начало массива, так как оно может быть различным в зависимости от устройства и его состояния.

Заключение

Написать в последнюю очередь

Литература

1. The OpenCL Specification
2. http://asu-cs.donntu.org/sites/default/files/images/doc/opencl.programming.guide.pdf p.30-33
3. Jonathan B. Rosenberg (1996). How Debuggers Work: Algorithms, Data Structures, and Architecture

Отладчики:

1. https://en.wikipedia.org/wiki/CodeXL
2. https://github.com/jrprice/Oclgrind
3. https://software.intel.com/content/www/us/en/develop/articles/api-debugger-tutorial.html
4. https://developer.nvidia.com/nsight-visual-studio-edition
5. http://www.gremedy.com/tutorial/