Санкт-Петербургский государственный университет

КИРИЧЕНКО Валерий Владимирович

Выпускная квалификационная работа

Моделирование затопления отсеков с использованием вычислений общего назначения на видеокарте

Уровень образования: бакалавриат
Направление 02.03.02 «Фундаментальная информатика и информационные технологии»
Основная образовательная программа СВ.5003.2016 «Программирование и информационные технологии»
Профиль «Автоматизация научных исследований»

Научный руководитель:
кандидат физ.-мат. наук, доцент кафедры
компьютерного моделирования и многопроцессорных систем
Ганкевич Иван Геннадьевич

Рецензент:
эксперт, Общество с ограниченной
ответственностью «ТСистемс Рус»
Иващенко Андрей Игоревич

Санкт-Петербург
2020 г.

Содержание

Введение

В настоящее время появляется всё больше задач, связанных с обработкой больших массивов данных. В связи с замедлившимся ростом тактовых частот и одноядерной производительности центральных процессоров в целом, становится необходимым использовать модель параллельного программирования, при которой задача разбивается на небольшие подзадачи, которые обрабатываются параллельно и независимо друг от друга.

Для решения таких задач обычно используют кластеры или грид-вычисления. Однако, не всегда возможно применять такие системы в силу значительных задержек обработки данных. Например, если требуется отображать пользователю анимацию результата вычислений в реальном времени, применение таких технологий может негативно сказаться на опыте использования ПО.

В связи с этим, набирает популярность технология использования графических процессоров для выполнения общих вычислений — general-purpose computing on graphics processing units (GPGPU). С её помощью становится возможным значительно ускорить выполнение части вычислений, которая обладает высокой степенью параллелизма. При этом, поскольку графические процессоры стали неотъемлемой частью каждого современного компьютера, запуск программы с использованием такого рода вычислений возможен на любом ПК пользователя.

В данной работе рассматривается применение описанного подхода для ускорения вычислений, связанных с моделированием затопления отсеков, в рамках проекта виртуального полигона.

Постановка задачи

Целью работы является реализация модели затопления отсеков жидкостью с использованием вычислений на графическом ускорителе. Для этого необходимо изучить предметную область, разработать алгоритм, описывающий модель, выявить и ускорить требовательные к производительности части этого алгоритма с использованием технологии OpenCL, проанализировать производительность по сравнению с выполнением алгоритма только на центральном процессоре.

Глава 1. Изучение предметной области

1.1 Технология OpenCL

OpenCL — открытый кроссплатформенный стандарт для параллельного программирования гетерогенных систем, разработанный Khronos Group. Он позволяет абстрагироваться от работы с конкретными средами выполнения и их наборами команд, тем самым предоставляя возможность исполнять написанный код на любой системе с поддержкой OpenCL. В число поддерживаемых устройств входят различные центральные (CPU), графические (GPU), цифровые сигнальные (DSP) процессоры, а также программируемые пользователем вентильные матрицы (FPGA).[1]

Несмотря на то, что OpenCL предназначен для кроссплатформенной работы с различными устройствами, производители устройств могут предоставлять расширения OpenCL, что позволяет добавлять дополнительные возможности и способы увеличения производительности без необходимости интеграции их в стандарт.

Код, который выполняется на OpenCL-устройствах, пишется на языке OpenCL C. Он является расширенной, но в то же время ограниченной версией языка C99, что позволяет запускать параллельные программы на множестве гетерогенных устройств. Компиляция этого кода происходит во время выполнения программы. Она осуществляется драйвером устройства, на котором планируется выполнение программы, что позволяет оптимизировать код для каждого устройства. Такое решение также позволяет запускать OpenCL-программы на устройствах, которые ранее были для этого не предназначены. Если производитель оборудования добавил поддержку OpenCL, то программа заработает на устройстве без дополнительных усилий со стороны разработчика.

Далее рассмотрим четыре основных модели, на которых строится OpenCL.

1.1.1 Архитектура OpenCL

Модель платформы

Эта модель описывает общее представление гетерогенной системы. Модель состоит из единственного хоста, к которому подключены одно или несколько OpenCL-устройств (compute devices). Эти устройства разбиты на вычислительные блоки (compute units), которые в свою очередь разделены на обрабатывающие элементы (processing elements). Вычисления на устройстве происходят в обрабатывающих элементах.

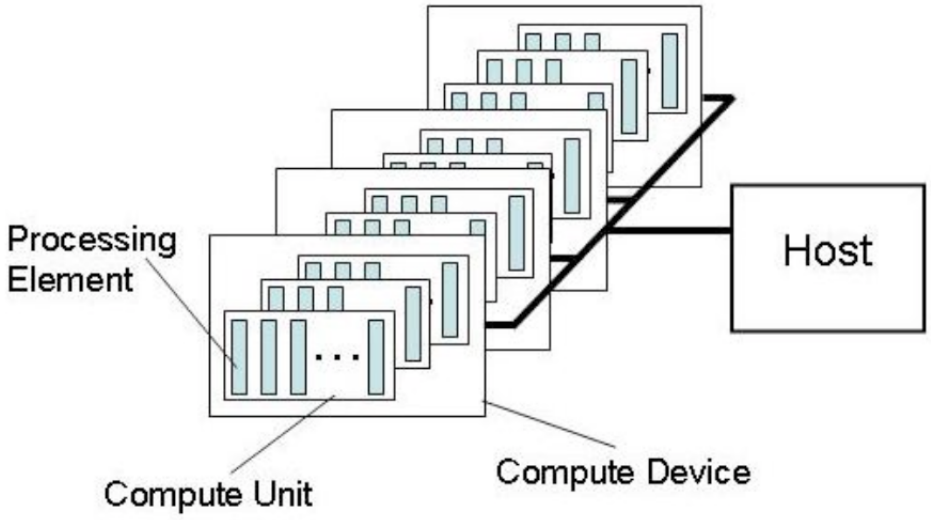

Рис. 1: Модель платформы с четырьмя OpenCL-устройствами.

Модель исполнения

Эта модель описывает взаимодействие программы, которая выполняется на хосте, с программой, которая выполняется на OpenCL-устройстве — ядром (kernel). Хост-программа задаёт контекст выполнения ядер и управляет их выполнением с помощью OpenCL API.

Контекст включает в себя используемые устройства, ядра (OpenCL-функции, исполняемые на устройствах), объекты программ (исходные и исполняемые коды, которые реализуют ядра) и объекты памяти.

Помимо контекста, создаётся очередь команд, с помощью которой происходит управление исполнением ядер. Хост добавляет команды в очередь,

которые затем исполняются на устройствах. Ими могут быть:

- Команды выполнения ядер
- Команды управления памятью
- Команды синхронизации

Поскольку команды выполняются асинхронно, в очереди команд возможно задать порядок их выполнения. В случае выполнения по порядку следующая команда начинается не раньше завершения предыдущей. Также возможно, чтобы команды не дожидались завершения предыдущих. В таком случае используются команды синхронизации для создания необходимых ограничений.

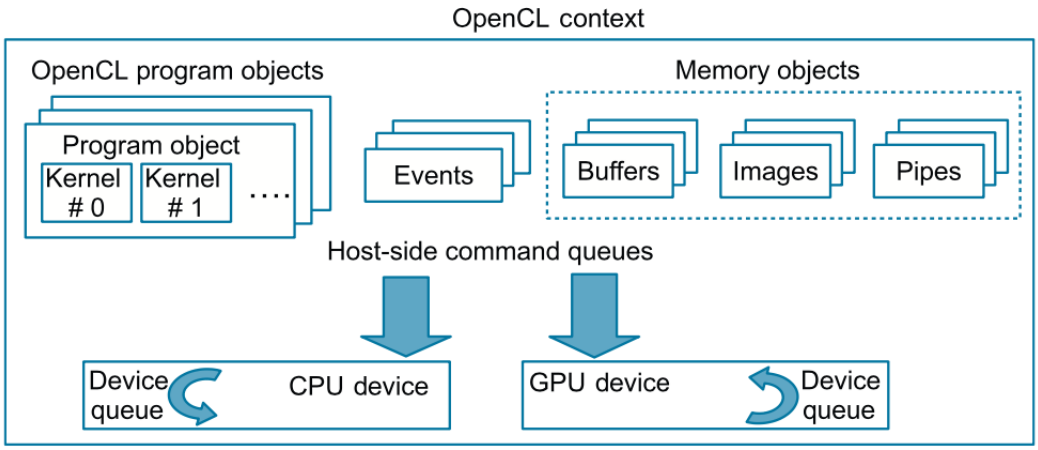

Рис. 2: Схематичное представление контекста.

При отправке ядра на выполнение, происходит определение индексного пространства. Экземпляр ядра, называемый work-item, выполняется для каждой точки этого пространства. Такая точка является глобальным идентификатором каждого work-item.

Экземпляры ядер объединяются в группы — work-groups. Группам назначается уникальный идентификатор из пространства той же размерности, что и индексное пространство work-items. Внутри группы каждому work-item назначается уникальный локальный идентификатор. Таким образом, work-item можно однозначно идентифицировать двумя способами: глобальным

идентификатором и комбинацией идентификатора группы и локального идентификатора внутри этой группы.

Индексное пространство в OpenCL называется NDRange. Поддерживаются одно-, дву- и трёхмерные пространства.

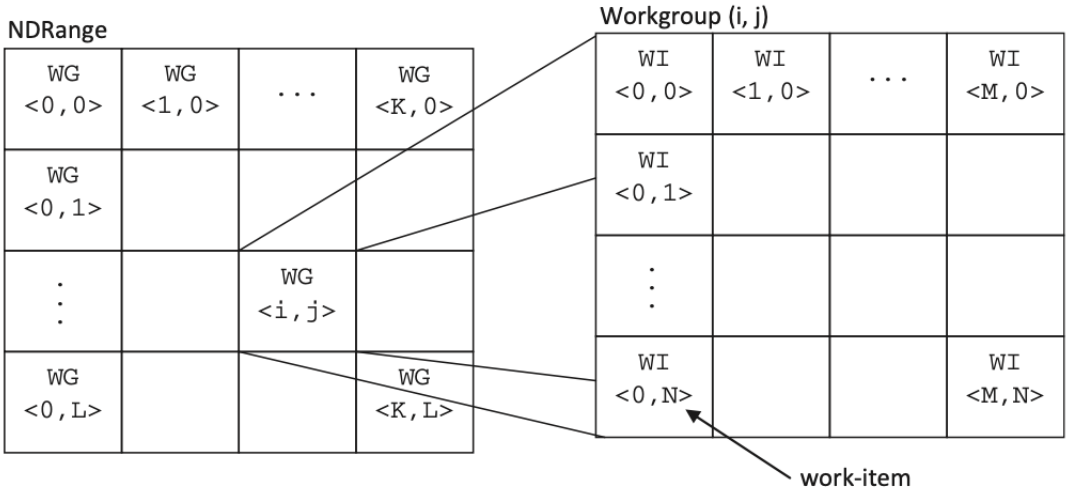

Рис. 3: Двумерное индексное пространство и группы в нём.

Важно отметить, что задачи на всех work-item внутри одной группы выполняются одновременно. Также внутри группы возможны барьерные синхронизации и доступ к общей памяти.

Такое разделение позволяет применять различные модели программирования.

Модель программирования ядра

Стандартом OpenCL поддерживаются модели параллелизма данных и параллелизма задач. Основной же моделью программирования является параллелизм данных.

В модели параллелизма данных вычисления происходят за счёт применения одних и тех же операций к разным участкам данных. В OpenCL это осуществляется за счёт индексного пространства, которым определяется как данные будут распределены по экземплярам ядер.

Модель параллелизма задач, в которой вычисления определяются как множество различных команд, выполняемых одновременно, поддерживается в

OpenCL с помощью одновременного выполнения различных ядер. Это может быть осуществлено с помощью одной или нескольких очередей команд.

Модель памяти

OpenCL разделяет всю память на две категории — память хоста и память устройств. Память хоста определяется вне OpenCL и управляется напрямую. С памятью устройств же взаимодействие осуществляется через OpenCL API. Она разделена на четыре региона:

- Глобальная память — доступна на чтение и запись всем work-item во всех группах;
- Константная память — часть глобальной памяти, которая остаётся константной во время выполнения ядра;
- Локальная память — часть памяти, доступная группе. Может быть использована для передачи данных между work-item в рамках одной группы. Также может быть ассоциирована с частью глобальной памяти;
- Приватная память — регион памяти, доступный только в рамках одного work-item.

Для передачи данных между хостом и устройствами, область память должна быть инкапсулирована в один и трёх объектов памяти:

- Буфер — хранит данные в памяти непрерывно, аналогично массивам языка Си, элементы буфера могут быть скалярными, векторными, а также структурами;
- Изображение — объект памяти, предназначенный для хранения текстур или любых других изображений. В отличие от буфера, не обязательно является одномерным и не гарантируется непрерывность размещения в памяти;
- Канал (pipe) — объект, хранящий упорядоченную последовательность данных, работает по принципу очереди. В любой момент времени лишь один экземпляр ядра может писать в канал, а другой — читать данные из канала.

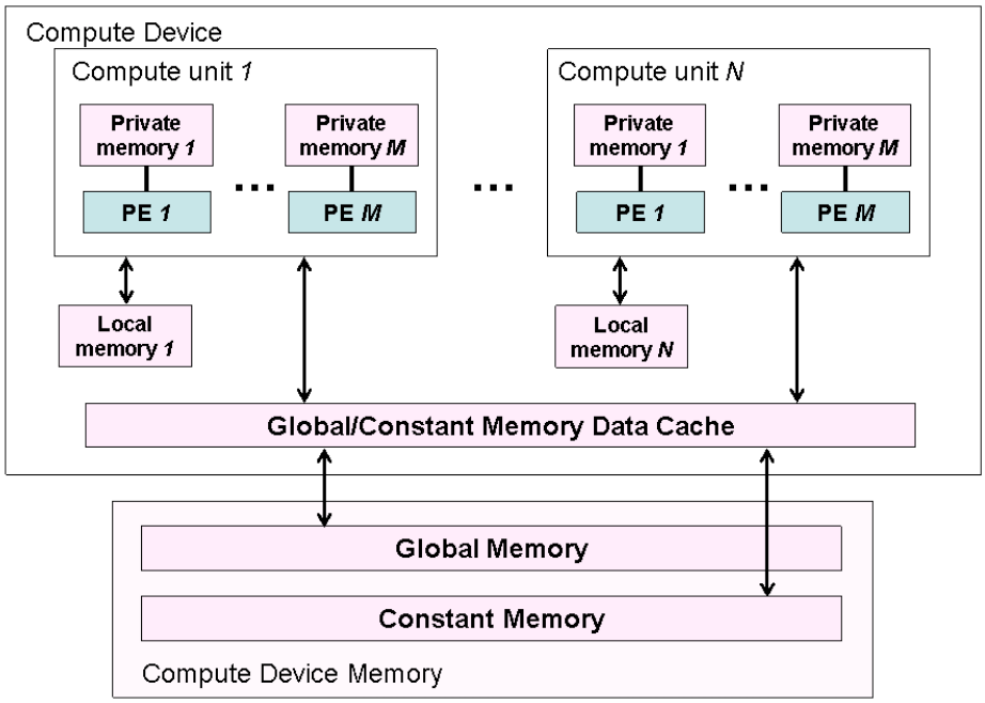

Рис. 4: Архитектура памяти в устройстве OpenCL.

1.1.2 Основные этапы выполнения вычислений с помощью OpenCL

Таким образом, после изучения архитектуры OpenCL можно выделить следующие этапы, необходимые для реализации и выполнения алгоритмов на OpenCL-устройствах:

1. Разработать ядра, реализующие алгоритмы или необходимые для ускорения части алгоритмов;
2. Определить имеющиеся платформы и устройства, а также поддерживаемые ими версии OpenCL;
3. Задать контекст с необходимыми устройствами для выполнения и параметрами;
4. Создать очередь команд;
5. Создать буферы для данных на устройстве;
6. Скопировать данные с хоста на устройство;

7. Прочитать код OpenCL-программы;
8. Скомпилировать программу для всех выбранных устройств;
9. Получить ядро из скомпилированной программы;
10. Выполнить ядро с необходимыми аргументами;
11. Скопировать результат работы с устройства на хост;
12. Освободить память устройства.

1.2 Виртуальный полигон

Виртуальный полигон — это программный комплекс, который предназначен для симуляции поведения морских волн, движения кораблей и затопления отсеков. Основной особенностью полигона является использование вычислений на графических ускорителях для повышения скорости проведения расчётов и визуализации в реальном времени. Это позволяет использовать её исследователями без необходимости получения доступа к высокопроизводительному оборудованию, например, кластерам, при этом обеспечивая достаточную производительность для комфортной работы.

Одно из предназначений виртуального полигона — моделирование экстремальных ситуаций, в числе которых повреждение обшивки корабля, при котором происходит полное или частичное затопление отсеков. Аналогично возможно моделировать затопление цистерн при погружении подводных лодок.

1.3 Задача моделирования движения свободной поверхности жидкости в частично затопленных отсеках и цистернах

Наличие свободной поверхности жидкости в судовых цистернах и отсеках приводит к ухудшению остойчивости[1] судна. Причём чем больше размеры отсека или цистерны, тем сильнее влияние свободной поверхности жидкости. В связи с этим, при проектировании морских судов важно исследовать влияние различных затопленных отсеков на запасы плавучести и остойчивости.[2]

Одним из способов проведения такого исследования является моделирование движения свободной поверхности жидкости. Частью задачи моделирования движения поверхности, решаемой в данной работе, является нахождение уровня и центра массы жидкости в затопленном отсеке при заданном объёме. Рассматривается модель, в которой свободная поверхность представляет собой плоскость.

[1] Остойчивость — способность плавучего средства противостоять внешним силам, вызывающим его крен или дифферент, и возвращаться в положение равновесия по окончании возмущающего воздействия.

Глава 2. Описание алгоритма

Входными данными алгоритма являются описание отсеков и их элементов как наборы вершин и треугольников, описанных индексами вершин, а также требуемые объёмы жидкости для затопления каждого отсека. При этом ориентация отсеков в пространстве уже скорректирована, что позволяет считать нормаль к поверхности жидкости коллинеарной оси Oz. Порядок вершин треугольников, составляющих элементы отсека, обратен таковому у отсеков. На выходе алгоритм возвращает уровни и центры масс жидкости в отсеке.

Вычисление уровня жидкости в затопленном отсеке происходит с помощью метода бисекции. Оптимизируется переменная z в уравнении $fluidVolume(z) - fluidVolumeIn = 0$, где $fluidVolumeIn$ — объём жидкости, которым затапливается отсек, $fluidVolume(z)$ — объём жидкости в отсеке на уровне z.

Для вычисления объёма жидкости на уровне z вычисляется сумма объёмов тетраэдров[3], составленных из затопленных граней отсека и его элементов и точки $(0, 0, z)$. Объёмы тетраэдров вычисляются со знаком, который зависит от порядка вершин, определяемых грань. Если грань полностью затоплена, то достаточно посчитать объём тетраэдра:

$$V = \frac{(\mathbf{a} - \mathbf{d}) \cdot ((\mathbf{b} - \mathbf{d}) \times (\mathbf{c} - \mathbf{d}))}{6},$$

где $\mathbf{a}$, $\mathbf{b}$ и $\mathbf{c}$ — вершины грани, а $\mathbf{d}$ – точка $(0, 0, z)$

Если же грань затоплена частично (одна или две вершины находятся над поверхностью жидкости), то сначала находятся точки пересечения рёбер грани и поверхности жидкости, затем, если две вершины выше уровня жидкости, то вычисляется объём тетраэдра, образованного точками пересечения рёбер и поверхности жидкости и точкой $(0, 0, z)$. Иначе вычисляются объёмы двух тетраэдров с вершиной в точке $(0, 0, z)$ и основаниями, сформированными точками пересечения рёбер и поверхности жидкости и, соответственно, вершинами треугольника, находящимися ниже поверхности жидкости.

Центр массы жидкости вычисляется как взвешенная сумма центров

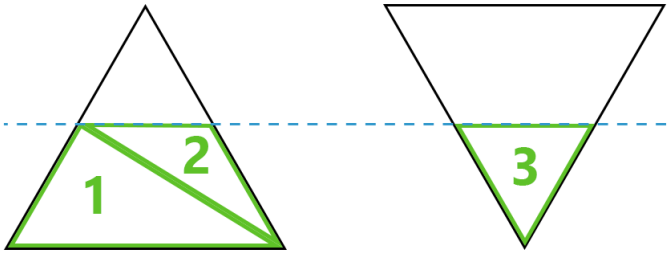

Рис. 5: Пример получающихся оснований тетраэдров (треугольники с номерами 1, 2 и 3) для частично затопленных граней

масс всех тетраэдров, получаемых на предыдущем шаге:

$$\mathbf{r_c} = \frac{\sum_i V_i \mathbf{r}_i}{\sum_i V_i},$$

где $\mathbf{r}_i = \frac{\mathbf{a}_i+\mathbf{b}_i+\mathbf{c}_i+\mathbf{d}_i}{4}$ — центры масс тетраэдров, а V_i — их объёмы соответственно.

Глава 3. Программная реализация

Реализация алгоритма представляет собой программный код на языках C++ (в части программы, исполняющейся на центральном процессоре) и OpenCL C (в части программы, исполняющейся на графическом ускорителе). При разработке программы наибольшее внимание было уделено минимизации количества данных и частоте их передачи между устройством и хостом, поскольку это оказывает существенное влияние на производительность[4].

В связи с этим, первой задачей стало определение необходимых данных и их форматов, которые будут передаваться между оперативной памятью компьютера и графического процессора. Поскольку в языке OpenCL C не реализован объектно-ориентированный подход, использованный в коде на C++, прежде чем передать необходимые для работы данные, требуется их преобразовать в поддерживаемый формат.

Благодаря поддержке языком OpenCL C встроенных векторных типов данных, эта задача не потребовала определения собственных структур данных. Для хранения вершин объектов был использован тип float3 (трёхмерный вектор со значениями с плавающей точкой одинарной точности), а для хранения индексов вершин, составляющих треугольники — тип int3 (целочисленный трёхмерный вектор).

При детальном изучении алгоритма было выявлено три потенциальных части, которые можно было бы ускорить с применением вычислений с использованием графического ускорителя:

1. Вычисление объёмов затопленных элементов отсека и их центров масс
2. Вычисление суммы объёмов тетраэдров и центра масс
3. Параллельное вычисление уровней жидкости в отсеках

Рассмотрим каждую часть отдельно.

3.1 Вычисление объёмов затопленных элементов отсека и их центров масс

Ускорение вычислений достигается за счёт параллельной обработки граней всех элементов отсека. Это наиболее трудоёмкая часть алгоритма, поскольку требует вычисление точек пересечения и объёмов образованных тетраэдров для каждой грани. Один рабочий элемент соответствует одной грани из всех объектов сцены.

С помощью функции языка OpenCL get_global_id получается номер обрабатываемой грани. Затем из глобальной памяти устройства по этому номеру запрашиваются индексы соответствующих вершин, при этом выполняется проверка, что номер грани меньше количества граней. Это необходимо, потому что количество элементов в общем случае не равно количеству граней из-за того, что количество граней не всегда делится нацело на количество элементов в рабочей группе, поэтому в последней группе могут быть «лишние» элементы.

Затем выполняются остальные шаги алгоритма практически без изменений. При этом используются встроенные в язык функции скалярного и векторного произведений, что может повысить производительность на устройствах, где эти операции ускоряются на аппаратном уровне.

Предыдущий этап оканчивается записью суммы объёмов и взвешенных центров масс тетраэдров в локальную память. Теперь необходимо получить итоговый объём жидкости и центр масс, сложив все промежуточные результаты. Для этого на графическом ускорителе был реализован параллельный алгоритм редукции в рамках каждой группы[5].

Алгоритм последовательно складывает числа парами на протяжении $log_2 N$ шагов, где N — число элементов в группе. Каждой паре соответствует свой рабочий элемент, при этом в конце каждого шага происходит синхронизация локальной памяти. После этого остаётся сложить K чисел, где K — количество рабочих групп. Этот этап выполняется на ЦП, поскольку по результатам тестирования передача и сложение такого количества данных практически не занимает времени, при этом позволяя освободить графический ускоритель для других задач. Иначе потребовалось бы запускать ещё

одно ядро по окончании работы предыдущего, потому что это единственный способ синхронизировать память между рабочими группами. Для вычисления центра масс получившаяся сумма делится на сумму объёмов.

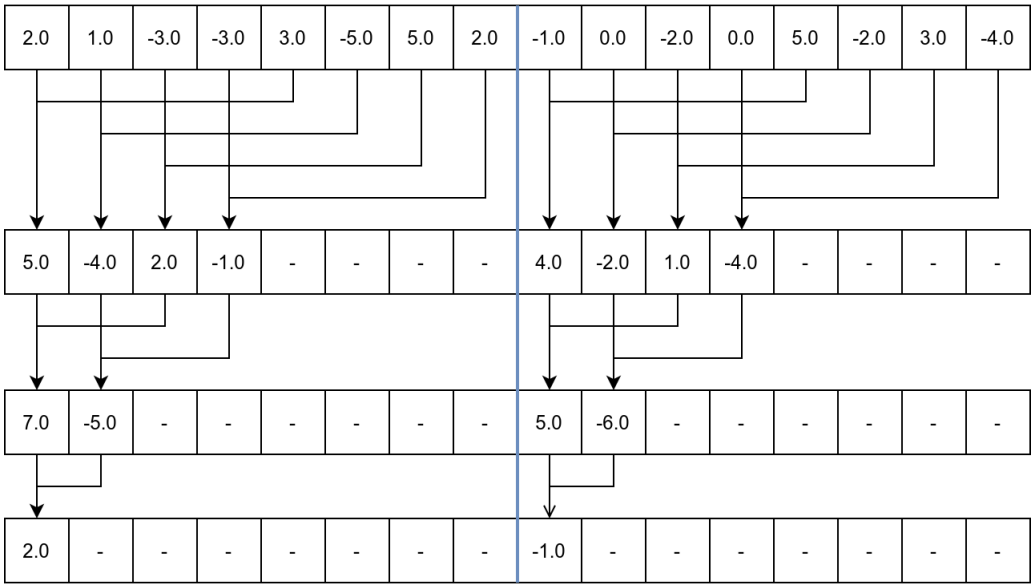

Рис. 6: Пример работы алгоритма с двумя рабочими группами

Как можно понять из описания, общая скорость выполнения редукции, включая передачу данных на хост и получение итоговой суммы на ЦП, существенно зависит от размера рабочих групп. Подробнее об этом написано в следующей главе.

3.2 Параллельное вычисление уровней жидкости в отсеках

Для параллельного вычисления уровней жидкости в отсеках необходимо запускать несколько ядер одновременно. Однако, это оказалось невозможно реализовать в рамках технологии OpenCL из-за нескольких факторов:

- Параллельный запуск ядер не поддерживается графическими процессорами. Технология OpenCL предоставляет возможность задать очередь с внеочередным исполнением задач, но видеокарты в настоящий момент поддерживают только последовательное исполнение ядер OpenCL. Кроме того, стандарт OpenCL 1.2 добавил возможность разбивать устройства на подустройства[1], тем самым предоставляя дополнительную функциональность для обеспечения параллельной работы ядер,

однако не все графические ускорители поддерживают это расширение стандарта.

- Из-за отсутствия возможности синхронизации глобальной памяти[2] невозможно полностью реализовать алгоритм на графическом ускорителе. Для этого в точках синхронизации требуется дождаться окончания выполнения ядра и добавить в очередь на выполнение другое ядро.

При этом, несмотря на приведённые выше ограничения, возможно запускать реализацию алгоритма в нескольких параллельных потоках для различных отсеков. В таком случае параллельно будут выполняться части программы на ЦП, но поскольку выполнение ядер будет происходить последовательно (а значит, один поток может ждать, пока закончится вычисление объёмов в другом потоке), существенного прироста производительности это не даст. В настоящий момент ускорение может достигаться лишь за счёт того, что возможны одновременная обработка данных и копирование данных между графическим ускорителем и хостом.

[2]Размер индексного пространства может превышать общее количество параллельных потоков графического процессора, из-за чего одни потоки могут ждать, пока другие исполняются. Если в это время исполняющиеся потоки попытаются синхронизироваться с ожидающими, то программа не сможет остановиться.

Глава 4. Тестирование производительности

Для проведения тестирования производительности, а также проверки корректности работы, было создано несколько моделей разной степени сложности. Каждый тест повторялся 5 раз, результаты усреднялись. Технически характеристики ПК, на котором проводились исследования, указаны в приложении А.

Таблица 1: Описание тестовых моделей

Номер	Количество вершин	Количество треугольников	Требуемый объём оперативной памяти ГП
1	8008	12 012	< 1 МиБ
2	26 008	48 012	≈ 1 МиБ
3	98 008	192 012	≈ 4 МиБ
4	386 008	768 012	≈ 17 МиБ
5	1 538 008	3 072 012	≈ 70 МиБ
6	6 146 008	12 288 012	≈ 281 МиБ
7	23 142 842	46 282 764	≈ 1059 МиБ

Первым этапом работы программы является определение OpenCL контекста и платформы, а также загрузка и компиляция OpenCL программы для целевой платформы. Этот этап не зависит от входных данных и в среднем для него требуется около 150 мс времени.

Далее необходимо преобразовать данные для передачи в память устройства. Для этого сначала выделяется область оперативной памяти хоста, затем данные конвертируются и записываются в эту область и в конце данные передаются в память графического ускорителя. Все эти этапы линейно зависят от количества вершин и треугольников. На небольших моделях они практически не сказываются на общем времени работы (таблица 2). Кроме того, эта подготовительная работа может выполняться лишь один раз при загрузке, а затем возможно переиспользовать уже записанные в память устройства буферы. По

этой причине в дальнейшем исследовании производительности это время учитываться не будет.

Таблица 2: Среднее время в миллисекундах, требуемое для этапов подготовки данных

Номер модели	Выделение памяти	Преобразование данных	Передача данных
1	0,09	0,06	0,77
2	0,35	0,19	0,94
3	1,41	0,80	2,69
4	5,64	3,14	8,72
5	21,32	13,36	33,48
6	88,23	51,80	132,03
7	350,25	198,86	493,64

Следующий этап работы программы — вычисление уровня жидкости. Для работы алгоритму необходимо задать объём жидкости, которым затапливается отсек. Для тестирования было выбрано по 5 значений объёма от практически полного отсутствия жидкости до почти полного затопления отсека.

Таблица 3: Тестовые объёмы жидкости

Номер модели	Объёмы жидкости для затопления
1	1; 14000; 28000; 42000; 56000
2	1; 15489; 30977; 46465; 61952
3	1; 15387; 30774; 46160,5; 61546
4	1; 15356,5; 30712; 46067,5; 61423
5	1; 15348,25; 30695,5; 46042,75; 61390
6	1; 15346,25; 30691,5; 46036,75; 61381
7	1; 1261; 2522; 3782.8; 5043.41

Как видно на рисунках 7 и 8, время выполнения алгоритма на графическом ускорителе практически не зависит от уровня жидкости в отличие от выполнения на ЦП. Это связано с тем, что программа, исполняющаяся на ЦП, обладает оптимизацией, недоступной в программе для видеокарты: пропускаются объекты, которые находятся выше уровня жидкости, что позволяет снизить количество работы, необходимой в случае низких уровней жидкости. Подобную оптимизацию можно реализовать и на ГП, но это потребует выделения и передачи дополнительной памяти.

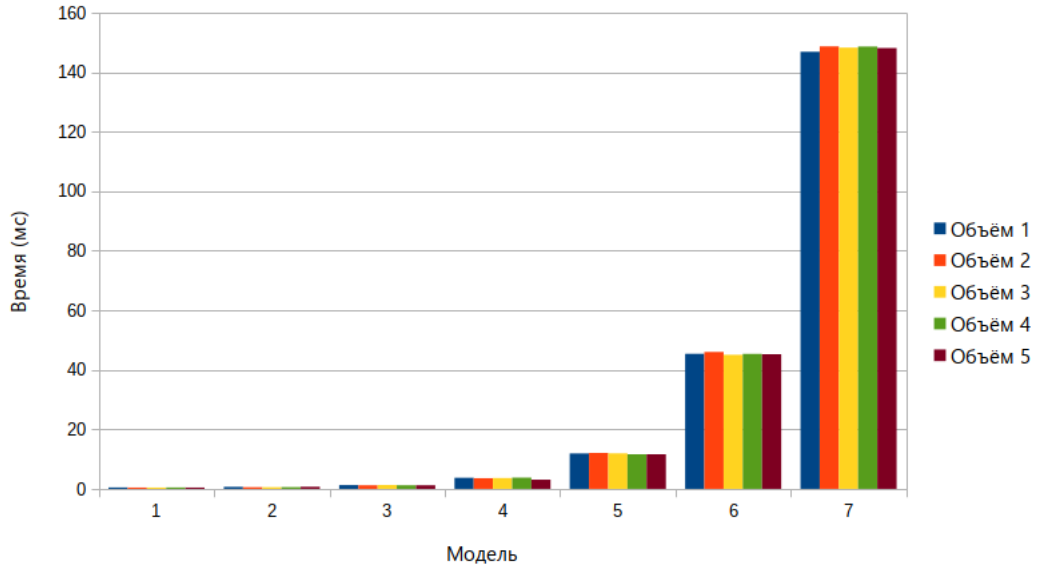

Рис. 7: Время работы алгоритма на ГП

Кроме того, на результат оказывает влияние то, как происходит выполнение потоков графического ускорителя[6]. Все потоки исполняются в группах по 32 (в случае GPU, разработанных компанией Nvidia), называемых варпами, которые управляются планировщиком варпов. Каждый варп исполняет одну общую инструкцию за один раз. Поэтому в момент ветвления количества затопленных вершин возможно, когда одни потоки в одном варпе ждут, пока отработает ветвь других потоков.

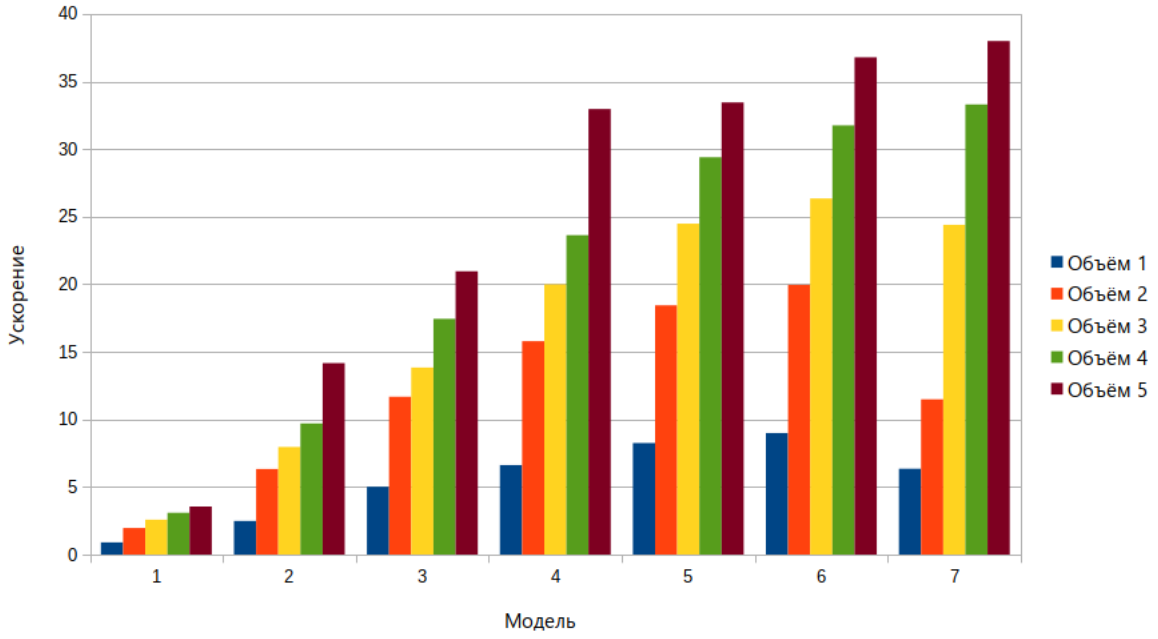

Рис. 8: Ускорение работы алгоритма на ГП по сравнению с ЦП

Также стоит отметить, что в случае первой модели в случаях затопления отсека небольшими объёмами жидкости, программа, выполняющаяся на графическом ускорителе, оказывается медленнее на 5% по сравнению с версией для ЦП. Однако, в связи с тем, что вычисление происходит очень быстро ($< 0,5$ мс), такая производительность не должна существенно повлиять на опыт использования виртуального полигона. В остальных же случаях ускорение составляет от 2,5 до 38 раз, что является достаточно хорошим результатом.

На примере модели №3 можно также показать, как сокращается время вычисления одного объёма жидкости в отсеке для заданного уровня. Время выполнения этого этапа оказывает наибольшее влияние на производительность программы, поскольку в методе бисекции на каждой итерации требуется его вычисление.

В программе, исполняемой на ГП, в среднем такое вычисление занимает около 240 мкс, из которых большую часть времени выполняется ядро OpenCL, а ещё 10% времени (≈ 25 мкс) занимает передача данных из памяти ГП в оперативную память компьютера (≈ 21 мкс) и вычисление окончательных сумм объёма и центра массы (≈ 4 мкс).

Время работы этого шага на ЦП зависит от заданного уровня жидкости, как было сказано ранее. Так, при почти полном заполнении отсека вычисление может занимать около 8000 мкс (в 33 раза больше, чем на ГП). При заполнении наполовину вычисление занимает около 5000 мкс (в 20 раз медленнее).

Рассмотрим влияние этапа вычисления окончательных сумм редукции от размера рабочей группы:

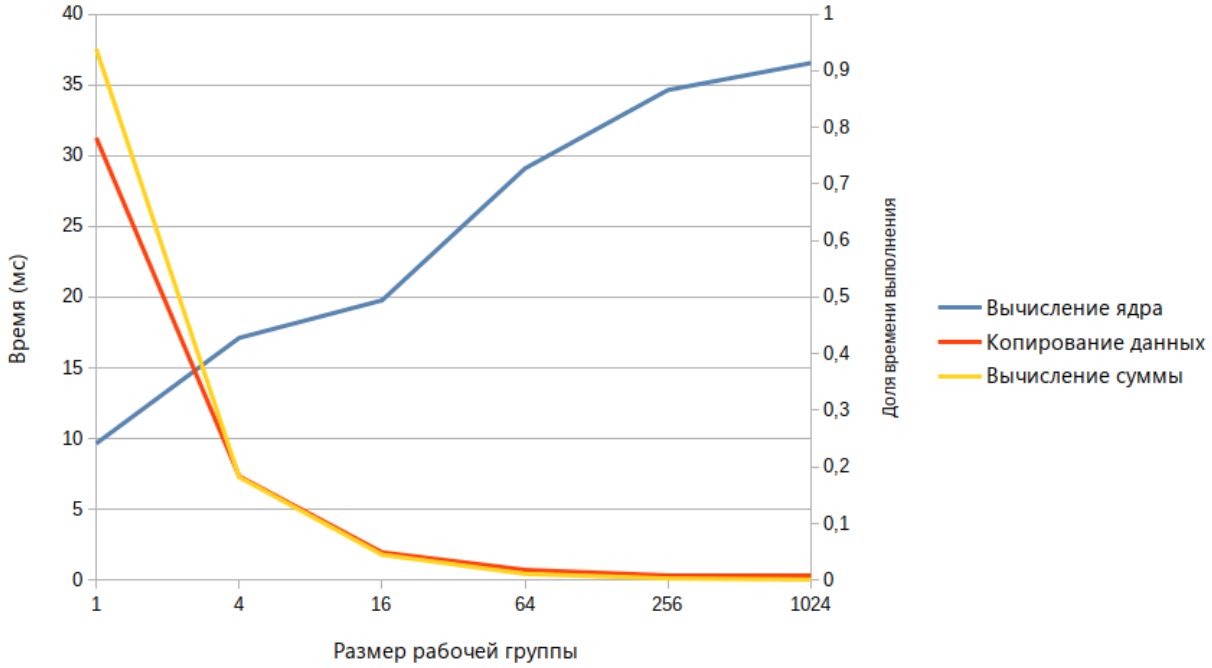

Рис. 9: Время выполнения этапов редукции на ЦП

Как можно видеть из графика, размер рабочей группы существенно влияет на скорость работы программы. Если в каждой рабочей группе находится всего один элемент, то суммарно этот этап может занимать до 75% всего времени вычисления объёма жидкости. При этом повышение размера рабочей группы способно снизить затрачиваемое рассматриваемым этапом время в 215 раз, что в конечном итоге позволяет вычислить объём в 25 раз быстрее:

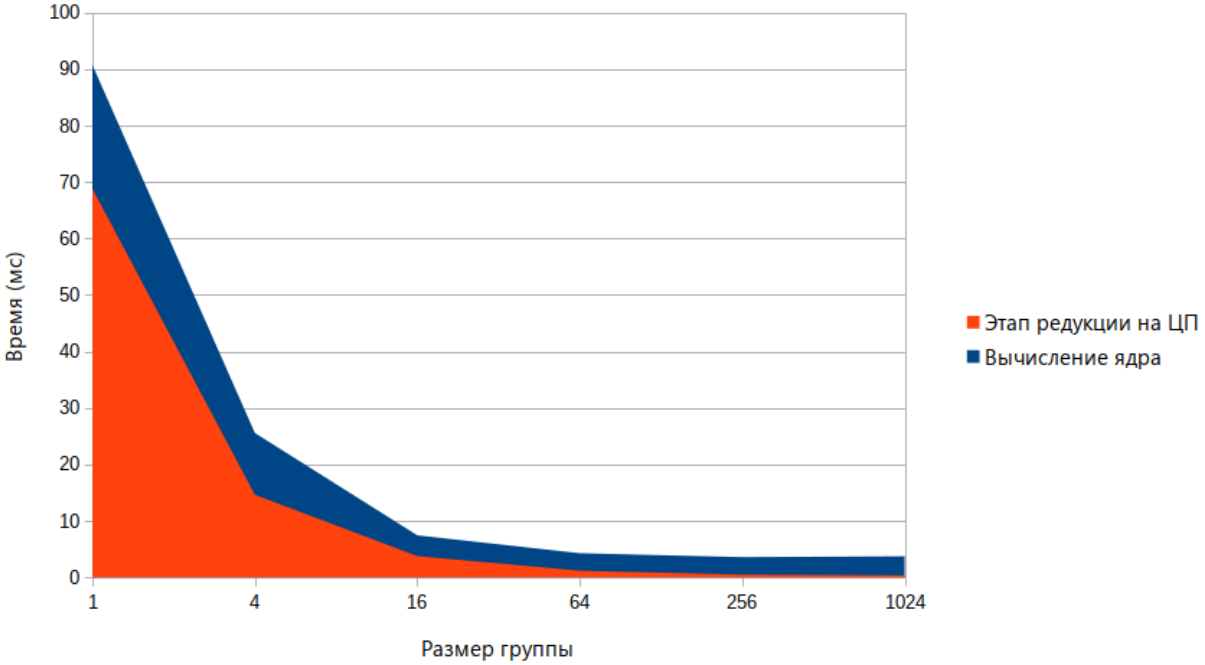

Рис. 10: Зависимость соотношения этапов вычисления объёма от размера группы

Стоит отметить, что даже встроенная в современные процессоры графическая подсистема (например, Intel Iris Graphics 6100, применяемая в процессоре на архитектуре Broadwell 2015 года) поддерживает до 256 рабочих элементов в одной группе.

Выводы

Использованием вычислений общего назначения на видеокарте позволяет существенно повысить скорость работы программы, однако сопряжено с рядом компромиссов, которые приходится решать в процессе разработки.

Одной из приоритетных задач является снижение количества обменов между центральным и графическим процессорами. Кроме того, сам объём передаваемых данных способен оказать существенное влияние на необходимое для передачи время. В работе для решения этой задачи был применён подход с переиспользованием буферов данных, что позволяет лишь один раз обработать и передать данные в память устройства.

Помимо передачи данных важным также является и правильное программирование ядер OpenCL. Даже в рамках устройства обращение ко глобальной памяти занимает достаточно много времени. Для минимизации влияния этой особенности все необходимые данные для работы ядра читаются из глобальной памяти в самом начале исполнения, а все промежуточные результаты записываются в локальную память рабочей группы.

Наконец, особенности синхронизации памяти устройства также становятся существенным препятствием для обеспечения параллельной работы алгоритмов. Невозможность синхронизации глобальной памяти не позволила перенести вычисление всего алгоритма на видеокарту. Однако, этапы, которые выполняются на ЦП, не требуют большого количества вычислительного времени.

Также в настоящее время графические процессоры не способны выполнять несколько ядер одновременно в рамках технологии OpenCL, что затрудняет параллельную обработку нескольких отсеков. Реализованное программное решение способно работать многопоточно, поэтому в случае, если видеокарты в будущем смогут запускать несколько ядер одновременно, можно ожидать увеличения производительности при работе со множеством отсеков с небольшим количеством геометрии.

Заключение

В рамках работы была разработана и протестирована программа для моделирования затопления отсеков с использованием вычислений общего назначения на видеокарте. По результатам тестирования применение технологии GPGPU позволило ускорить работу алгоритма до 38 раз по сравнению с исполнением этого же алгоритма на центральном процессоре. Кроме того, были предложены способы дальнейшей оптимизации времени работы программы за счёт увеличенного потребления оперативной памяти графического процессора.

Список литературы

[1] Khronos OpenCL Working Group: *The OpenCL 1.2 Specification*. https://www.khronos.org/registry/OpenCL/specs/opencl-1.2.pdf, 2012.

[2] П., Муру Н.: *Основы непотопляемости корабля*. Воениздат, 2-е изд., перераб. и доп., 1990.

[3] Lien, Sheue ling и James T. Kajiya: *A Symbolic Method for Calculating the Integral Properties of Arbitrary Nonconvex Polyhedra*. IEEE Computer Graphics and Applications, 4(10):35–42, 1984.

[4] *OpenCL Best Practices Guide*. http://developer.download.nvidia.com/compute/DevZone/docs/html/OpenCL/doc/OpenCL_Best_Practices_Guide.pdf, 2011.

[5] Scarpino, Matthew: *OpenCL in Action*. Manning Publications Co., 2012.

[6] *OpenCL Programming Guide for the CUDA Architecture*. http://developer.download.nvidia.com/compute/DevZone/docs/html/OpenCL/doc/OpenCL_Programming_Guide.pdf, 2012.

[7] Khronos OpenCL Working Group. https://www.khronos.org/registry/OpenCL/specs/opencl-2.0.pdf, 2015.

[8] Kaeli, David, Perhaad Mistry, Dana Schaa, и Dong Ping Zhang: *Heterogenous Computing with OpenCL 2.0*. Elsevier Inc., 2015.

Приложение А. Технические характеристики ПК, проводящего тестирование производительности

- ЦПУ: Intel Core i7 8700k (6 ядер на архитектуре Coffee Lake, 12 потоков), работал на частоте $\approx 4,4$ ГГц
- ГП: Nvidia GeForce GTX 1070 (8 ГиБ оперативной памяти GDDR5, 1920 ядер CUDA), работал на частоте $\approx 2,0$ ГГц
- Оперативная память: 2×8 ГиБ в двухканальном режиме на частоте 3200 МГц

Приложение Б. Исходный код ядра OpenCL

```
typedef struct {
    const float3 *a, *b, *c, *d;
} Tetrahedron;

inline float3 intersection_point(const float3 *a, const float3 *b, float z)
 ↪ {
    float s = (z - a->z) / (b->z - a->z);
    return *a + s * (*b - *a);
}

inline float det(const float3 v1, const float3 v2, const float3 v3) {
    return dot(cross(v1, v2), v3);
}

inline float t_volume(const Tetrahedron *t) {
    return det(*(t->a) - *(t->d), *(t->b) - *(t->d), *(t->c) - *(t->d)) /
     ↪  6.0f;
}

inline void add_volume(float* const o_volume, float3* const o_centre, const
 ↪  Tetrahedron *t) {
    float volume = -t_volume(t);
    *o_volume += volume;
    *o_centre += volume * (*(t->a) + *(t->b) + *(t->c) + *(t->d)) / 4.0f;
}

__kernel void calculate_volume(
   __global float3* vertices,
   __global int3* faces,
   __global float* volume_result,
   __global float3* centre_result,
   __local float* volume_sums,
   __local float3* centre_sums,
   const unsigned int total_faces,
   float fluid_level) {
    int face_num = get_global_id(0); // Номер грани для этого work item
    if (face_num >= total_faces) return;
```

```
35      int3 face = faces[face_num];
36
37      float3 origin = (float3) (0, 0, fluid_level);
38      float volume = 0;
39      float3 centre = (float3) (0);
40      float3 v[3] = {
41              vertices[face.x - 1],
42              vertices[face.y - 1],
43              vertices[face.z - 1]
44      };
45      const bool above[3] = {
46              v[0].z > fluid_level,
47              v[1].z > fluid_level,
48              v[2].z > fluid_level
49      };
50      int nabove = 0; // Количество вершин над поверхностью жидкости
51      for (int i = 0; i < 3; ++i) {
52          if (above[i]) {
53              ++nabove;
54          }
55      }
56      if (nabove == 0) {
57          // Все вершины под поверхностью, вычисляем объём тетраэдра
58          Tetrahedron t = {&v[0], &v[2], &v[1], &origin};
59          add_volume(&volume, &centre, &t);
60      } else if (nabove > 0 && nabove < 3) {
61          // Одна или две вершины над поверхностью, необходимо разбить грань
62          int outstanding, i1, i2;
63          const bool x01 = above[0] ^ above[1];
64          const bool x02 = above[0] ^ above[2];
65          const bool x12 = above[1] ^ above[2];
66          if (x01 && x02) { outstanding = 0, i1 = 1, i2 = 2; }
67          else if (x01 && x12) { outstanding = 1, i1 = 2, i2 = 0; }
68          else { outstanding = 2, i1 = 0, i2 = 1; }
69          const float3* outstandingVertex = &v[outstanding];
70          float3 v_new[3];
71          v_new[outstanding] = *outstandingVertex;
72          // Точки пересечения грани и поверхности жидкости
73          v_new[i1] = intersection_point(outstandingVertex, &v[i1],
            ↪  fluid_level);
```

```
74          v_new[i2] = intersection_point(outstandingVertex, &v[i2],
            ↪ fluid_level);
75
76          if (outstandingVertex->z > fluid_level) {
77              Tetrahedron t1 = {&v_new[i1], &v[i2], &v[i1], &origin};
78              add_volume(&volume, &centre, &t1);
79              Tetrahedron t2 = {&v_new[i1], &v_new[i2], &v[i2], &origin};
80              add_volume(&volume, &centre, &t2);
81          } else {
82              Tetrahedron t = {outstandingVertex, &v_new[i2], &v_new[i1],
                ↪ &origin};
83              add_volume(&volume, &centre, &t);
84          }
85      }
86
87      // Редукция
88      int lid = get_local_id(0);
89      int group_size = get_local_size(0);
90
91      volume_sums[lid] = volume;
92      centre_sums[lid] = centre;
93      barrier(CLK_LOCAL_MEM_FENCE);
94
95      for (int i = group_size/2; i>0; i >>= 1) {
96          if (lid < i) {
97              volume_sums[lid] += volume_sums[lid + i];
98              centre_sums[lid] += centre_sums[lid + i];
99          }
100         barrier(CLK_LOCAL_MEM_FENCE);
101     }
102     if (lid == 0) {
103         volume_result[get_group_id(0)] = volume_sums[0];
104         centre_result[get_group_id(0)] = centre_sums[0];
105     }
106 }
```