МИНОБРНАУКИ РОССИИ
Федеральное государственное бюджетное образовательное учреждение
высшего образования
«САРАТОВСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ
ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ ИМЕНИ Н. Г. ЧЕРНЫШЕВСКОГО»
(СГУ)

Кафедра математической
кибернетики и компьютерных наук

ВЫЧИСЛЕНИЯ НА ВИДЕОКАРТАХ

КУРСОВАЯ РАБОТА

студента 3 курса 351 группы
направления 09.03.04 — Программная инженерия
факультета КНиИТ
Григорьева Алексея Александровича

Научный руководитель
доцент _____ М. С. Семенов

Заведующий кафедрой
к. ф.-м. н. _____ С. В. Миронов

Саратов 2019

СОДЕРЖАНИЕ

ВВЕДЕНИЕ 3
1 Краткая теория 5
1.1 Типовая модель видеокарты 5
1.2 Основные понятия OpenCL 6
2 Алгоритмы на видеокарте 9
2.1 Требования к алгоритмам 9
2.2 Настройка среды разработки 13
2.3 Инициализация OpenCL программы 14
2.4 Задачи на одномерных массивах 14
2.4.1 Вычисление суммы ряда 14
2.4.2 Нахождение максимального префикса 15
2.5 Задачи на двумерных массивах 16
2.5.1 Транспонирование матрицы 16
2.5.2 Умножение матриц 18
ЗАКЛЮЧЕНИЕ 21
СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ 22
Приложение А Листинг программы 23
Приложение Б Листинг сборочных файлов CMake 33

ВВЕДЕНИЕ

Ранние видеокарты использовались для решения узкоспециализированных задач по эффективному отображению графических элементов на экране. До определенного момента их развитие происходило параллельно с процессорами, и функционал определялся стремительно развивавшейся игровой индустрией. С появлением у разработчиков компьютерных игр желания самостоятельно программировать шейдеры, были разработаны программные средства, способные решать широкий спектр задач.

Одновременно с этим рост производительности процессоров сильно замедлился, и достигался либо за счет увеличения количество ядер, либо за счет некоторых оптимизаций в них. Однако, для задач с крупными вычислениями этого было недостаточно, и сравнения производительности видеокарт и процессоров показали большую разницу в показателях 1.

Причиной этого является различие в количестве ядер как следствие слабой масштабируемости процессоров.

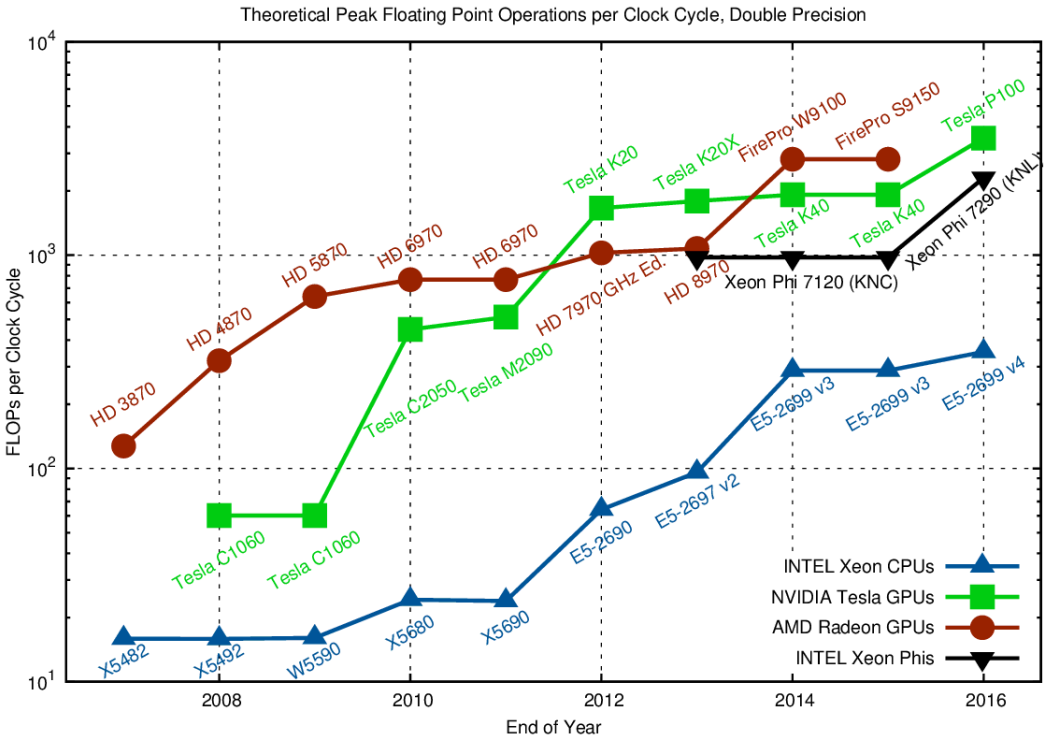


Рисунок 1 – Теоретический максимум количества операций с плавающей запятой в такт для GPU и CPU на 2016 год. График наилучшего по производительности процессора изображен снизу.

В связи с тем что видеокарты созданы под решение определенного класса задач, алгоритмы должны обладать определенными свойствами чтобы реше-

ние с использованием GPU было эффективней чем на процессоре. В данной работе рассматривается архитектура вычислительных устройств видеокарты и связанных с ними особенностями, которые влияют на проектирование алгоритмов.

В практической части рассматриваются не только реализация алгоритмов на видеокарте, но и возможные оптимизации по количеству вычислений и обращений к памяти.

При выполнении курсовой работы были поставлены следующие цели:

— ознакомиться с теорией, необходимой для написания эффективных алгоритмов для видеокарты с использованием OpenCL;
— понять свойства архитектуры видеокарты и тем самым научиться оптимизировать алгоритмы;
— получить практический опыт разработки программ на видеокартах с помощью OpenCL.

1 Краткая теория

Составление эффективных алгоритмов вычисления на видеокарте в значительной степени отличается от привычных алгоритмов, исполняющихся на процессоре. При составлении программного кода необходимо учитывать как и общие особенности видеокарт, так и, возможно, характеристики конкретного устройства, для которого программируется алгоритм.

В данном разделе будет рассмотрена типовая модель видеокарты и основные понятия OpenCL, с которыми будем оперировать в данной работе.

1.1 Типовая модель видеокарты

Рассмотрим следующую архитектуру вычислительного устройства, используемого в видеокартах Nvidia 2.

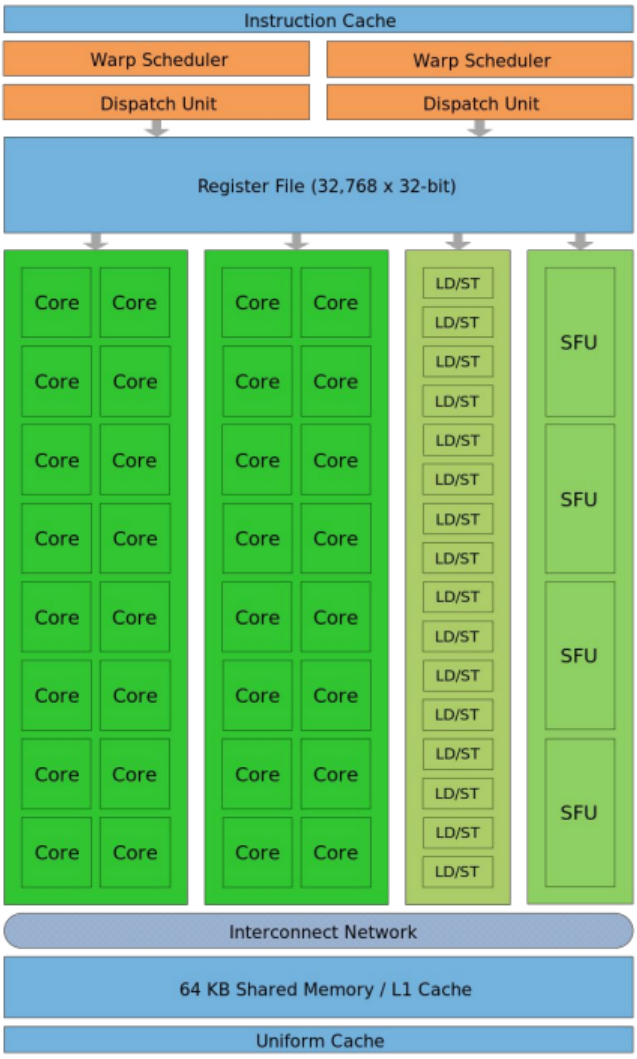


Рисунок 2 – Архитектура потокого мультипроцессора Fermi.

Вычислительное устройство в архитектуре Nvidia имеет 32 **ядра** (CUDA

cores), каждое из которых в состоянии работы является **потоком**. В отличии от процессора, ядра выполняют более узкий набор задач, что позволяет с меньшими затратами увеличить их количество в устройстве [1]. Для управления ими существует **warp scheduler**, выполняющий роль указателя на инструкции соответствуя архитектуре SIMD. Данные для вычислений потоки берут из **локальной памяти** (shared memory), общей для всех ядер. Достигается это с использованием **устройств загрузки и хранения** (load-store units), соответственно способных загружать, а также сохранять данные в локальную память. Между всеми 32 ядрами вычислительного устройства динамически распределяются **регистры**, самая быстрая память, доступная им. У мультипроцессора в наличии намного больше регистров, чем могло быть нужно для выполнения программы. Это сделано для сокрытия времени на загрузку памяти и быстрого переключения контекста, подробнее - в разделе 2.1.

Количество таких устройств в видеокарте определяется следующим образом:

Количество ядер в видеокарте / 32, в случае Nvidia

Количество ядер в видеокарте / 64, в случае AMD

В терминологии Nvidia, поток из всех (32) активных ядер вычислительного устройства образует **warp**, Например, видеокарта Nvidia Geforce GTX 1050 Ti имеет 768 ядер CUDA, и, соответственно, 24 warp.

1.2 Основные понятия OpenCL

OpenCL – открытый для свободного пользования программный интерфейс для создания параллельных приложений, использующих многоядерные структуры как и центрального процессора (CPU), так и графического (GPU). Использование API необходимо для обеспечения совместимости программы с различными устройствами [2].

При построении задач, определяется **рабочее пространство** (NDRange), представляющее собой все возможные в рамках задачи значения индексов потоков. Размер рабочего пространства определяется программистом на этапе инициализации OpenCL программы. Рабочее пространство может представлять:

- одномерный массив длиной N элементов;
- двумерную сетку размерности NxM;

— трехмерное пространство размерностью NxMxP.

Код, выполняющийся параллельно на ядрах процессора, называется **kernel**. Копия kernel выполняется для каждого индекса рабочего пространства и называется **work-item** с глобальным ID, соответствующим некоторому ID рабочего пространства. Kernel для всех work-item в рабочем пространстве имеют одинаковый код и входные параметры, но может иметь различный путь выполнения программы соответственно своему глобальному индексу - индекс в рабочем пространстве, полученному с использованием функции `get_global_id()`. Kernel в отличии от остальной программы полностью выполняется на видеокарте [3].

Группа work-item называется **work-group**, и за каждой группой закреплен собственный warp (см. предыдущий раздел), в рамках которого work-item могут синхронизироваться. Для каждой рабочей группы существует ее индекс в рабочем пространстве, и каждый work-item может узнать свой индекс внутри рабочей группы. Нетрудно заметить следующее соотношение:

$$\text{global ID} = \text{group ID} * \text{размер группы} + \text{local ID}$$

Размер рабочей группы аналогично рабочему пространству определяется программистом.

Каждое ядро, выполняя заданный kernel, является work-item в некоторой рабочей группой, на которые разделено рабочее пространство NDRange.

Рассмотрим на примере следующей схемы 3 другие виды сущностей, с которыми будет взаимодействие в OpenCL.

— Платформа — драйвер, модель взаимодействия OpenCL и устройства. Распространены платформы от следующих производителей: Nvidia, Intel, AMD.
— Программа — хостовая часть, организующая подготовку к вычислениям и набор kernel-подпрограмм.
— Kernel — программа, исполняющаяся на видеокарте в каждом ядре.
— Контекст — окружение, в котором исполняется kernel.
— Объект памяти — создаваемый в контексте объект.
— Буфер — произвольный массив данных.

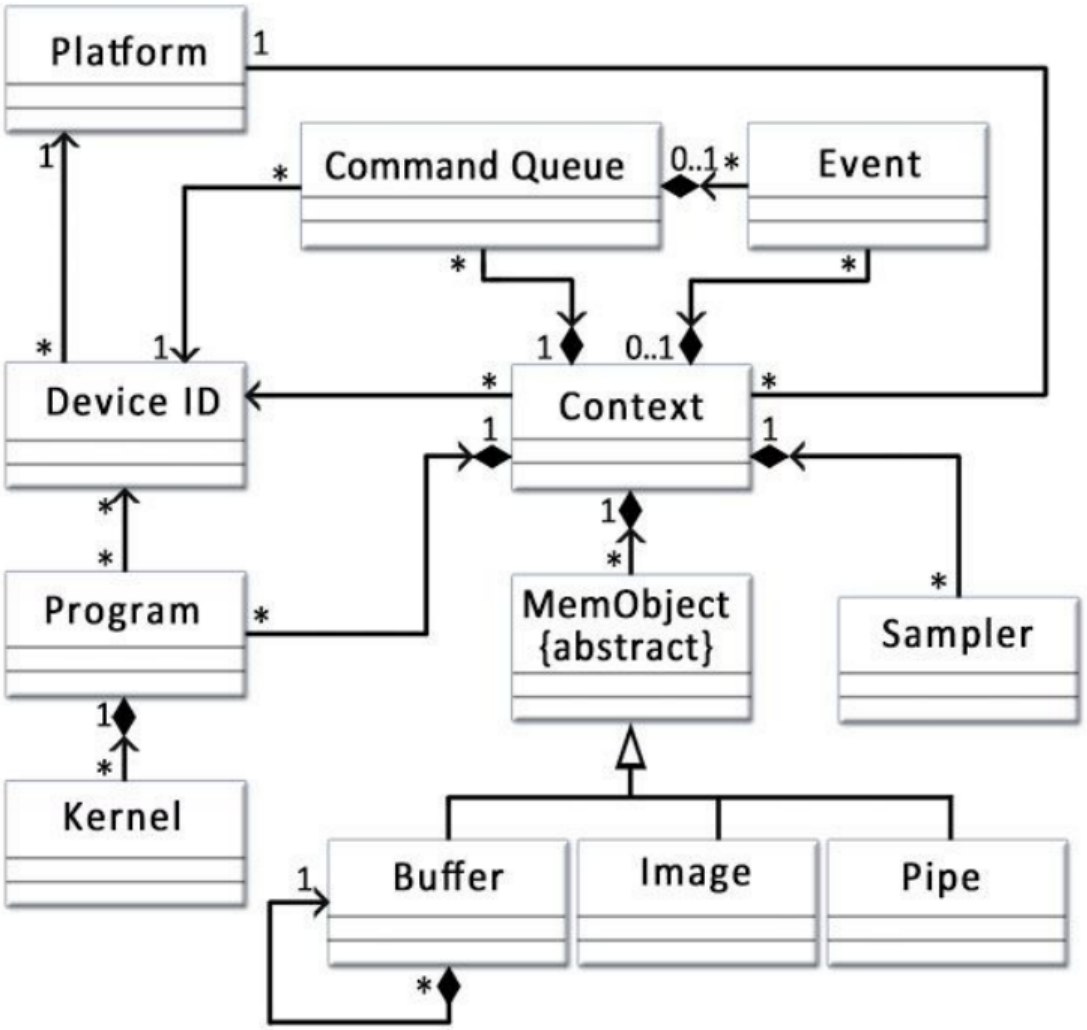


Рисунок 3 – Основные сущности в OpenCL.

2 Алгоритмы на видеокарте

В данном разделе будет рассмотрена анализ и практическая реализация алгоритмов на видеокарте, включая:

— описание общих требований к алгоритмам на основе доступа к памяти и параллельного исполнения;
— настройка среды разработки Microsoft Visual Studio 2017 под выполнение параллельных программ с использованием OpenCL;
— написание программ для задач, использующих входные данные разных размерностей.

2.1 Требования к алгоритмам

Любой алгоритм можно вычислить на видеокарте, но эффективность в сравнении с реализацией на центральном процессоре зависит от корректного построения алгоритма для видеокарты.

Основным требованием к составлению алгоритма на видеокарте считается наличие массового параллелизма. Он заключается в том что задачу можно разбить на рабочие группы так, что не будет требоваться постоянная синхронизация между work-item из разных рабочих групп.

Следует вспомнить, что все потоки в warp выполняют одинаковые инструкции в любой момент времени. Какая инструкция будет выполняться следующей определяется с помощью warp scheduler, единого для всех потоков в warp. Рассмотрим следующий фрагмент кода:

```
if (predicate) {
    value = x[i];
}
else {
    value = y[i];
}
```

Учитывая сказанное выше, все потоки при срабатывания if-части должны выполнить внутреннюю часть, однако это не совсем так, и если у потока предикат — `False`, он будет спрятан от выполнения внутренней части, аналогично и с `else`-частью. Однако несмотря на то что результат выполнения конструкции `if-else` будет верным, часть потоков будет простаивать, ожидая выполнение маскированных для них частей.

Данная ситуация называется code divergence, и она может стать причиной низкой производительности программы. Этого можно избежать, если организовать код таким образом чтобы для всех потоков предикат возвращал одинаковое значение, тогда конструкция не соответствующая ему будет пропущена указателем на инструкции. Если это невозможно, то для эффективного выполнения алгоритма рекомендуется отказаться от многочисленных сложных ветвлений, так как сложность выполнения фрагмента алгоритма будет вычисляться как сумма if- и else- частей вместо максимума как в последовательных программах.

При выборе размера рабочих групп стоит учитывать особенности алгоритма, однако, есть некоторые общие правила, которых необходимо придерживаться.

1. Размер рабочей группы не должен быть меньше warp.
2. Размер рабочей группы должен быть кратен 32 (64 если используется AMD).

В противном случае, некоторые потоки будут простаивать, ожидая пока остальные завершат свою работу

Как известно, операции с памятью являются одними из самых долгих по времени выполнения, в связи с этим было решено сделать разбиение задач на рабочие группы, в результате у видеокарт появился аналог имеющегося у процессоров hyper-threading. Он заключается в использовании каждым вычислительным устройством регистров для переключения контекста при задержке, созданной обращением к памяти (latency) [4].

Другими словами, warp может быстро сохранить состояние выполнения в данной рабочей группе и пока выполняется долгая операция обращения к памяти, вычислительное устройство может переключиться на другой warp в рабочей группе, и если второй warp хочет выполнить операцию обращения к памяти, то происходит возвращение к первому warp если доступ к памяти завершился, либо активируется третий warp и так далее. Следствие — высокая вычислительная мощность и большая пропускная способность видеокарты [5].

Количество одновременно активных warp в рабочей группе определяется как минимум из:

— количества регистров / количество используемых в warp регистров;
— количества локальной памяти / количество используемой локальной па-

мяти;

— максимально допустимого количества warp (~10).

В соответствии с этим существует величина **occupancy**, определяемая соотношением

среднее кол-во активных warp / максимальное кол-во активных warp

Не всегда высокий occupancy означает что программа имеет высокую производительность. Например, если доступ к памяти в программе очень быстрый, только из регистров, то необходимости в сокрытии задержки и переключения контекста нет, и occupancy будет низким.

Однако, низкий occupancy и высокая задержка при обращении к памяти может означать что программа написана не достаточно эффективно, и ей необходимы улучшения, если это возможно.

Чем больше на одном вычислителе warp — тем реже все warp оказываются в состоянии «ждем запрос памяти» и тем реже вычислитель будет простаивать, т.к. тем чаще у него находится рабочая группа в которой можно что-то посчитать [4].

Если потоки из одного warp делают запрос к памяти, то эти запросы склеются в столько запросов, сколькими кэш-линиями покрываются запрошенные данные.

Другими словами, если потоки запрашивают данные, которые в памяти лежат подряд, то достигнутая пропускная способность будет максимальная так как запросы «склеются». Размер кэш-линии обычно от 32 до 128 байт.

Если приложение использует OpenCL 1.x, то размеры NDRange должны нацело (без остатка) делиться на размеры рабочих групп. Там, где данные образуют NDRange с другим размером, необходимо самостоятельно изменить их чтобы выполнялось это условие, например, добавлением нулей или средних значений, которые не будут значимо влиять на результат вычислений [6].

В OpenCL 2.0 появилась новая возможность, в которой устранена данная проблема. Речь идет о так называемых неоднородных рабочих группах: выполняемый модуль OpenCL 2.0 может разделить NDRange на рабочие группы неоднородного размера по любому измерению. Если разработчик укажет размер рабочей группы, на который размер NDRange не делится нацело, выполняемый модуль разделит NDRange таким образом, чтобы создать как мож-

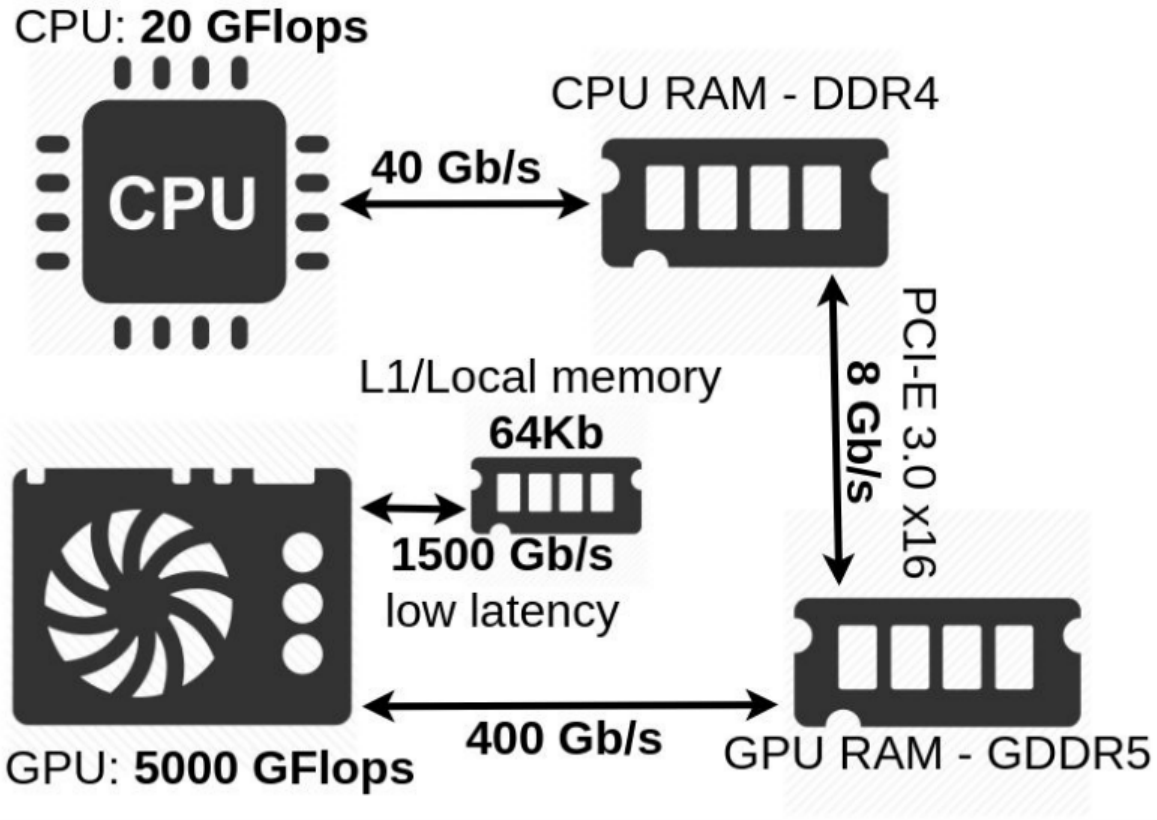


Рисунок 4 – Доступные ресурсы — память.

но больше рабочих групп с указанным размером, а остальные рабочие группы будут иметь другой размер. Например, для NDRange размером 1918x1078 рабочих элементов при размере рабочей группы 16x16 элементов среда выполнения OpenCL 2.0 разделит NDRange, как показано на приведенном ниже рисунке 5.

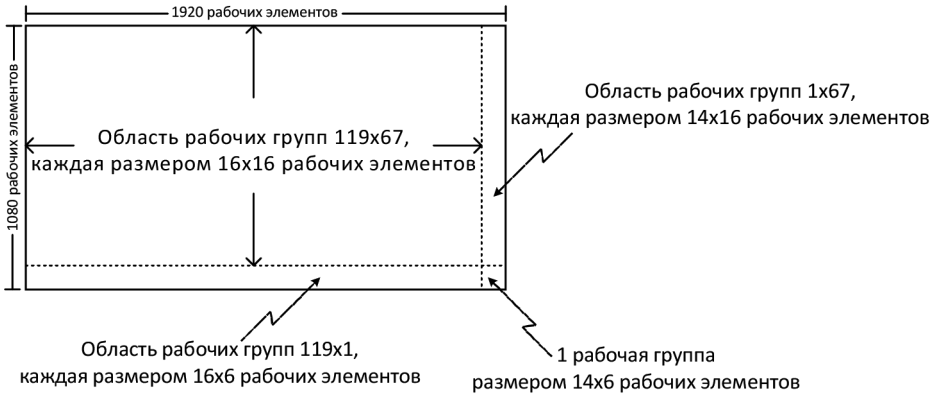


Рисунок 5 – Разделение NDRange на рабочие группы разных размеров.

2.2 Настройка среды разработки

В данном разделе будет рассмотрен процесс настройки среды разработки и создания первого OpenCL-проекта.

В качестве среды разработки для программирования с использованием OpenCL выбрана Microsoft Visual Studio, язык программирования — C++.

На компьютер была установлена реализация OpenCL от Nvidia — Nvidia GPU Computing SDK. А также программа CMake, являющаяся независимым от платформы инструментом для сборки проектов 6.

С помощью графического интерфейса выберем расположение файлов исходного кода и места сборки проекта. Директория с исходными файлами должны содержать текстовые файлы CMakeLists из приложения Б. Если OpenCL установлен корректно, то нажатие кнопки «Configure» выведет найденные на компьютеры файлы, связанные с OpenCL. Нажмем «Generate», и перейдем в папку с проектом, в котором можно увидеть созданный файл .sln проекта Microsoft Visual Studio, сконфигурированного под OpenCL.

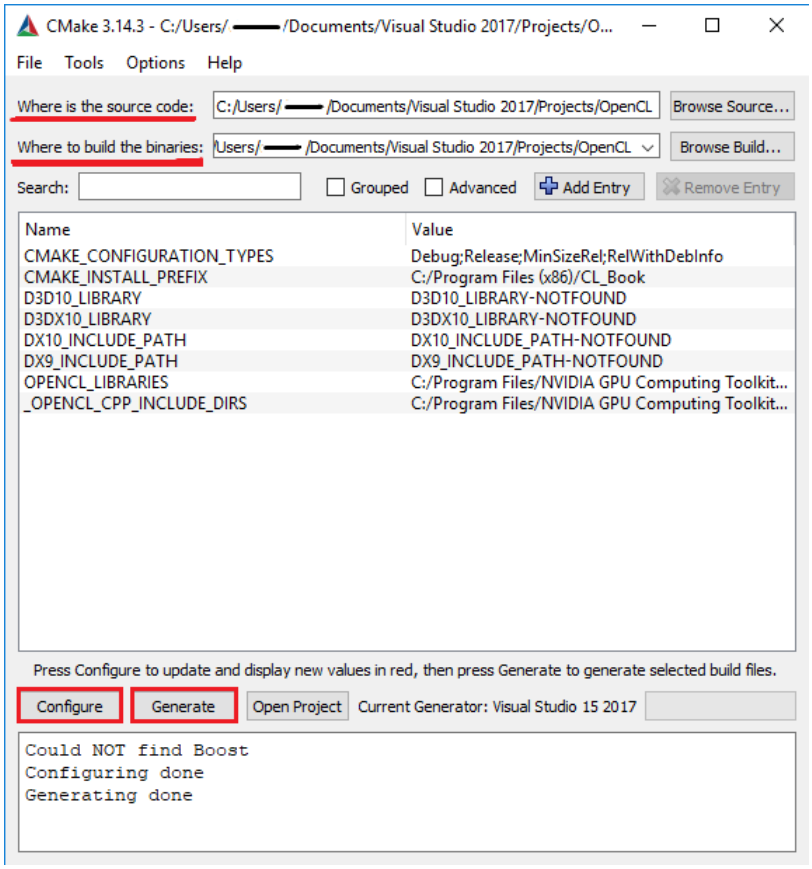


Рисунок 6 – Окно программы CMake-gui.

2.3 Инициализация OpenCL программы

В данном разделе будут рассмотрены базовые функции, необходимые для инициализации параллельной программы с использованием OpenCL. Данные функции будут предварять задачи из следующих разделов.

Рассмотрим пример, взятый из руководства по OpenCL [7]. С полным кодом, содержащим комментарии, переведенными на русский язык, можно ознакомиться в приложении A, файл HelloWorld.cpp. Обратим внимание на последовательность действий в функции main(). Многие понятия из данного раздела подробно описаны в 1.2.

Сначала с помощью функции `CreateContext()` создается контекст на основе первой найденной на компьютере платформы. Далее для первого доступного устройства в контексте создается командная очередь `clCreateCommandQueue()`, а в случае неудачи запускается функция очистки и программа завершается с кодом ошибки 1.

Из файла с исходным кодом kernel HelloWorld.cl создается OpenCL программа `clCreateProgramWithSource()`. После этого создается и сам kernel на основе созданной «программы» `clCreateKernel()`.

После этого создаются объекты памяти для конкретной задачи `clCreateBuffer()`, и каждый из них поочередно загружается в kernel с помощью `clSetKernelArg()`. Затем kernel ставится в очередь на выполнение `clEnqueueNDRangeKernel()`, и после завершения работы, выводим в консоль буфер-результат, являющимся результатом выполнения данной OpenCL программы `clEnqueueReadBuffer()`.

2.4 Задачи на одномерных массивах

Решим типовые задачи на одномерных массивах.

2.4.1 Вычисление суммы ряда

Одна из самых тривиальных задач — посчитать сумму двух векторов. Даны векторы **a** и **b**, на основе их суммы должен получиться вектор **result**.

```
1
2 __kernel void hello_kernel(__global const float *a,
3                                                 __global const float *b,
4                                                 __global float *result)
5 {
```

```
6     int gid = get_global_id(0);
7
8     result[gid] = a[gid] + b[gid];
9 }
```

Рассмотрим kernel для решения данной задачи. Каждый work-item, исполняя копию kernel, узнает свой глобальный ID в рабочем пространстве с помощью `get_global_id(0)`. Таким образом он читает данные из видеопамяти соответствуя своему индексу. В основной программе выбран NDRange размерности 1024 и `local_work_size`, размер рабочей группы, равный 32. Эти параметры переданы в функцию `clEnqueueNDRangeKernel()`

Пусть значения компонент первого вектор соответствуют номеру компоненты начиная с 0, а значения компонент второго вектора соответствуют удвоенному номеру компоненты в векторе. Результат работы данной программы представлен на изображении 7.

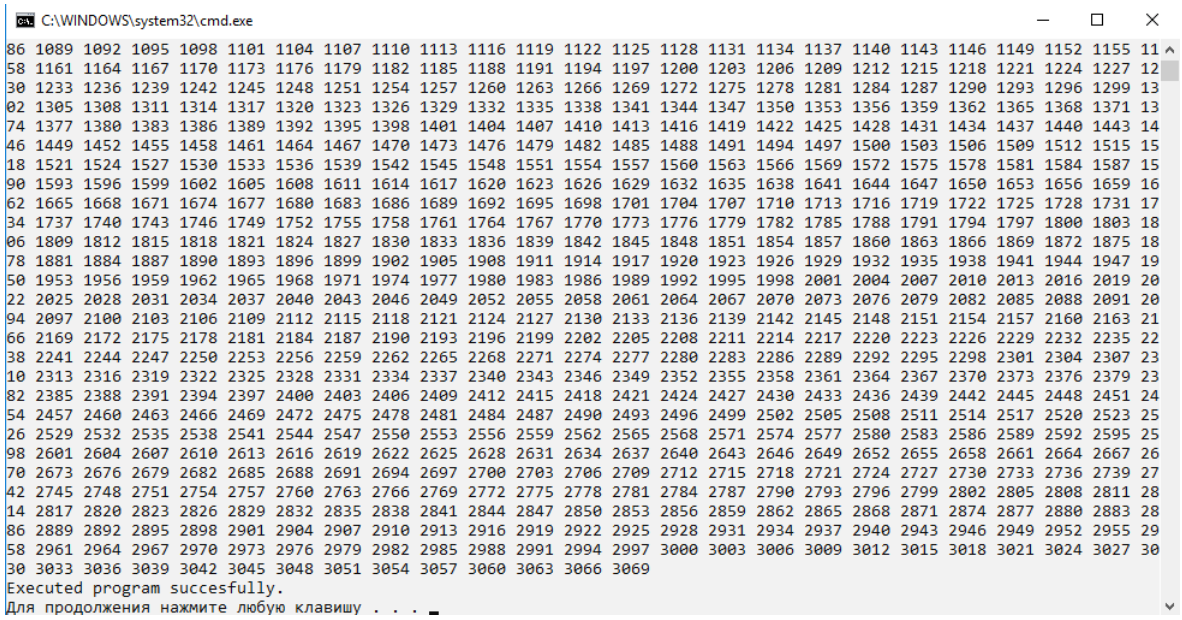


Рисунок 7 – Результат работы программы по суммированию двух векторов.

2.4.2 Нахождение максимального префикса

Усложним задачу, найдя для всего массива максимальную сумму на префиксе. То есть, значение префиксной суммы для каждого элемента в массиве должно определяться суммой всех предыдущих элементов массива включая его самого.

Данная задача просто решается в последовательных решениях, но требует значительной модификации алгоритма для параллельных вычислений на видеокарте.

—-

2.5 Задачи на двумерных массивах

Следующий набор задач использует в качестве входных данных двумерный массив.

2.5.1 Транспонирование матрицы

Решим задачу транспонирования матрицы. Задача сводится к считыванию и записи данных в память, но как было описано ранее в разделе 2.1, эти операции являются очень медленными.

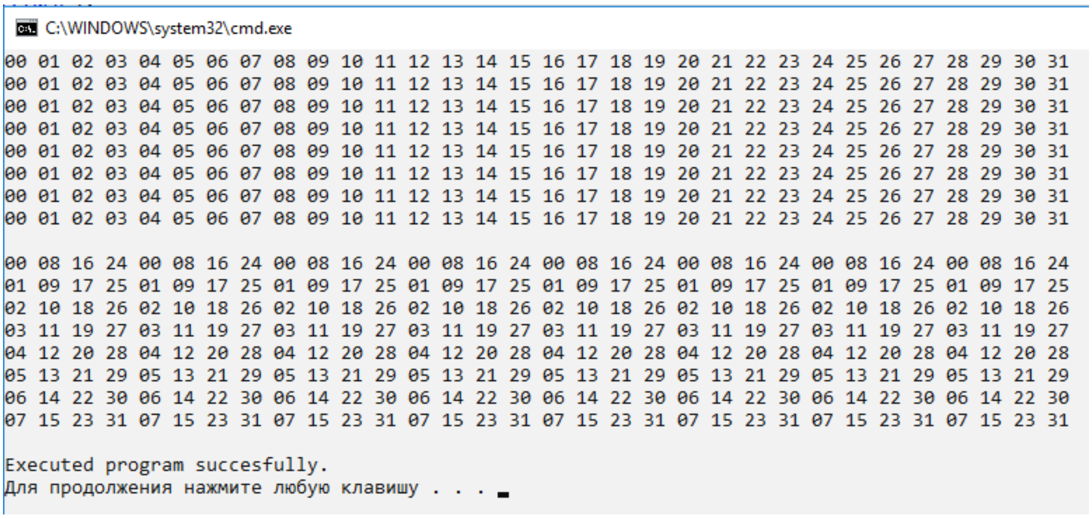
В простейшей реализация kernel будет выглядеть следующим образом:

```
1
2 __kernel void transpose1(__global float *a,
3                                                   __global float *at,
4                                                   unsigned int m, unsigned int n)
5 {
6     int i = get_global_id(0);
7         int j = get_global_id(1);
8
9         float x = a[j * k + i];
10    at[i * m + j] = x;
11 }
```

Результат работы программы представлен на изображении 8

В этом kernel данные считываются построчно, и записываются в столбец. Заметим, что операция считывания, очевидно, происходит в одной кэш-линии, и все work-item в рабочей группе за 1 глобальную операцию получат данные из исходной матрицы. Запись, напротив, происходит в разные строки, и данные не могут находиться в одной кэш-линии. Следовательно, на каждый активный warp произойдет 32 глобальные операции записи, и это сильно замедлит выполнение алгоритма.

Данную проблему можно решить использованием локальной памяти для транспонирования в соответствии с изображением 9. Задача переносится на tile



```
C:\WINDOWS\system32\cmd.exe
00 01 02 03 04 05 06 07 08 09 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
00 01 02 03 04 05 06 07 08 09 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
00 01 02 03 04 05 06 07 08 09 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
00 01 02 03 04 05 06 07 08 09 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
00 01 02 03 04 05 06 07 08 09 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
00 01 02 03 04 05 06 07 08 09 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
00 01 02 03 04 05 06 07 08 09 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
00 01 02 03 04 05 06 07 08 09 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31

00 08 16 24 00 08 16 24 00 08 16 24 00 08 16 24 00 08 16 24 00 08 16 24 00 08 16 24 00 08 16 24
01 09 17 25 01 09 17 25 01 09 17 25 01 09 17 25 01 09 17 25 01 09 17 25 01 09 17 25 01 09 17 25
02 10 18 26 02 10 18 26 02 10 18 26 02 10 18 26 02 10 18 26 02 10 18 26 02 10 18 26 02 10 18 26
03 11 19 27 03 11 19 27 03 11 19 27 03 11 19 27 03 11 19 27 03 11 19 27 03 11 19 27 03 11 19 27
04 12 20 28 04 12 20 28 04 12 20 28 04 12 20 28 04 12 20 28 04 12 20 28 04 12 20 28 04 12 20 28
05 13 21 29 05 13 21 29 05 13 21 29 05 13 21 29 05 13 21 29 05 13 21 29 05 13 21 29 05 13 21 29
06 14 22 30 06 14 22 30 06 14 22 30 06 14 22 30 06 14 22 30 06 14 22 30 06 14 22 30 06 14 22 30
07 15 23 31 07 15 23 31 07 15 23 31 07 15 23 31 07 15 23 31 07 15 23 31 07 15 23 31 07 15 23 31

Executed program succesfully.
Для продолжения нажмите любую клавишу . . .
```

Рисунок 8 – Результат работы программы по транспонированию матрицы. Для выравнивания элементам меньше 10 добавлены незначащие нули

(плитка), которые создаются в локальной памяти, доступ к которой происходит быстро.

Задача условно делится на 3 части, разделенные функцией «барьер»:

1. считывание из глобальной памяти в локальную (tile);
2. транспонирование в локальной памяти (tile);
3. запись из локальной памяти (tile) в глобальную.

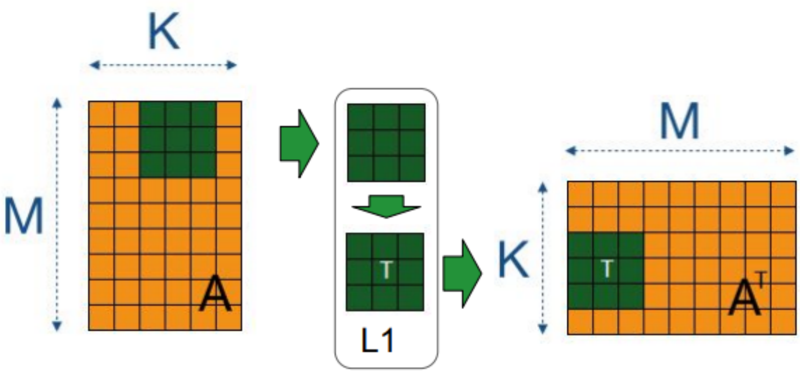


Рисунок 9 – Транспонирование матрицы с использованием «плиток».

Код оптимизированного kernel:

```
#define TILE_SIZE 32
__kernel void transpose2(__global float *a,
                                                __global float *at,
                                                unsigned int m, unsigned int n)
{
        int i = get_global_id(0);
        int j = get_global_id(1);
```

```
8
9     _local float tile[TILE_SIZE][TILE_SIZE];
10        int local_i = get_local_id(0);
11        int local_j = get_local_id(1);
12
13        tile[local_j][local_i] = a[j * k + i];
14        barrier(CLK_LOCAL_MEM_FENCE);
15
16        float tmp = tile[local_j][i];
17        tile[local_j][local_i] = tile[local_i][local_j];
18        tile[local_i][local_j] = tmp;
19        barrier(CLK_LOCAL_MEM_FENCE);
20
21        at[i * m + j] = tile[j * TILE_SIZE][i];
22 }
```

В данном коде используется функция `get_local_id()` для определения позиция внутри рабочей группы, и, соответственно, получения элемента плитки, с которым будет работать текущий work-item

2.5.2 Умножение матриц

Задача умножения матриц является одной из типовых задач, решение которых имеет огромное преимущество при использовании видеокарты для вычисления 10.

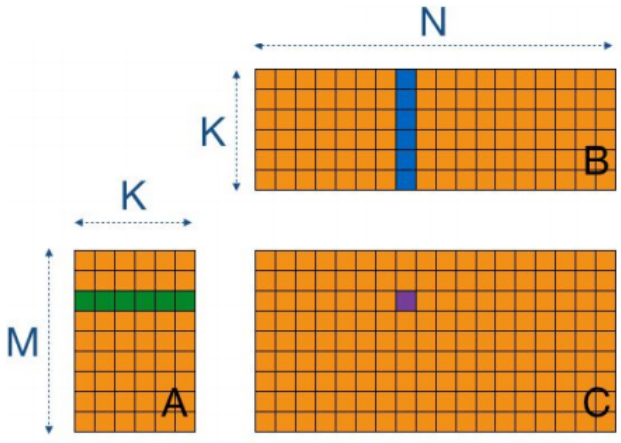


Рисунок 10 – Умножение матриц A ($M \times K$) и B ($K \times N$).

Как и в предыдущей задаче про транспонирование рассмотрим наиболее простое решение данной задачи. Каждый поток будет считывать для соответствующей ему ячейки результирующей матрицы строку и столбец исходной.

```
1
2 __kernel void matmul1(__global float *a,
3                 __global float *b,
4                 __global float *c,
5                 unsigned int M, unsigned int K, unsigned int N)
6 {
7     int i = get_global_id(0);
8         int j = get_global_id(1);
9
10        float sum = 0.0f;
11        for (int k = 0; k < K; k++) {
12                sum += a[j * K + k] * b[k * N + i];
13        }
14        c[j * N * i] = sum;
15 }
```

Можно заметить, что для каждой ячейки результирующей матрицы ($N \times M$ ячеек) происходит $2 \times K$ обращений к памяти, в массиве A он эффективный так как его элементы берутся последовательно, а в массиве B — нет, и при получении элементов из столбца может быть выполнено до 32 обращений к памяти вместо одного. Таким образом происходит O($M \times N \times K$) обращений к памяти, что является существенным фактором медленной работы алгоритма.

Оптимизируем алгоритм, использовав модифицированную версию подхода, представленную в предыдущей задаче. Разобьем задачу на фрагменты в локальной памяти, используемые потоками одной рабочей группы 11.

Преимущество подобного решения — снижение длительности операций считывания памяти за счет того что каждый warp работает в собственной локальной памяти, а также за счет считывания данных построчно (см. раздел 2.1).

Код kernel для эффективного умножения матриц:

```
1 #define TILE_SIZE 32
2 __kernel void matmul2(__global float *a,
3                 __global float *b,
4                 __global float *c,
5                 unsigned int M, unsigned int K, unsigned int N)
6 {
7     int i = get_global_id(0);
8         int j = get_global_id(1);
9         int local_i = get_local_id(0);
10        int local_j = get_local_id(1);
```

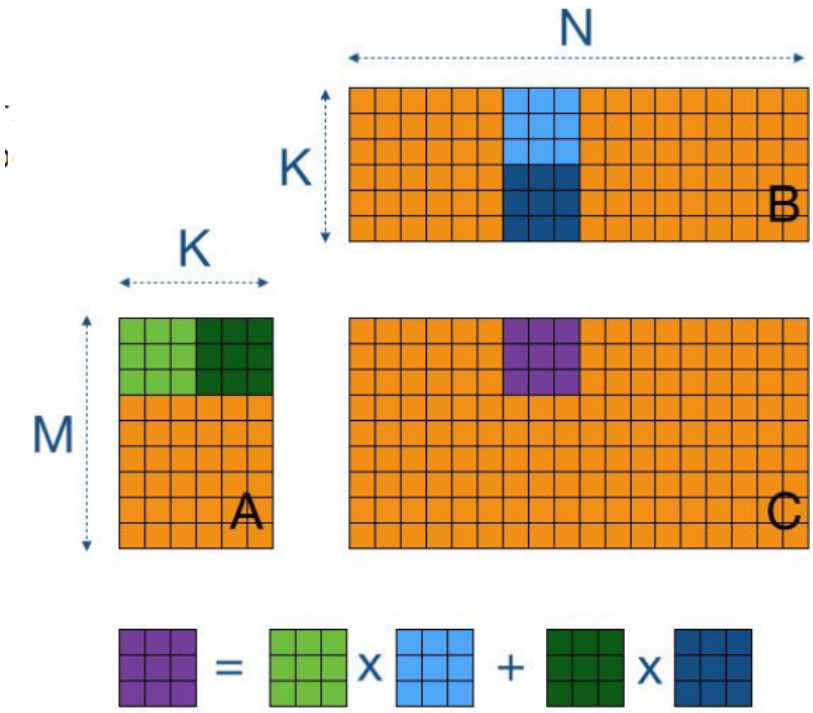


Рисунок 11 – Использование локальной памяти при умножении матриц.

```
11          __local float tileA[TILE_SIZE][TILE_SIZE];
12          __local float tileB[TILE_SIZE][TILE_SIZE];
13
14          float sum = 0.0f;
15          for (int tileK = 0; tileK * TILE_SIZE < K; tileK++) {
16                  tileA[local_j][local_i] = a[j * K + (tileK * TILE_SIZE + local_i)];
17                  tileB[local_j][local_i] = b[j * K + (tileK * TILE_SIZE + local_i)];
18                  barrier(CLK_LOCAL_MEM_FENCE);
19                  for (int k = 0; k < TILE_SIZE; k++) {
20                          sum += tileA[local_j * TILE_SIZE][k]
21                                          * tileB[local_i * TILE_SIZE][k];
22                  }
23          }
24          c[j * N * i] = sum;
25  }
```

ЗАКЛЮЧЕНИЕ

В настоящей работе была изучена технология OpenCL для параллельных вычислений на видеокарте. Корректно построенные параллельные алгоритмы на видеокарте показывают высокую производительность за счет использования большого числа ядер. Эффективность алгоритма зависит от поддержания массового параллелизма, корректного ветвления кода, а также правильного доступа к памяти.

Была изучена архитектура видеокарты и связанные с ней понятия OpenCL, а также решены некоторые задачи с исходными данных разных размерностей путем написания высокопроизводительных OpenCL программ, состоящих из хостовых и kernel частей.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1 [NVIDIA's Next Generation CUDA Compute Architecture: Fermi]. — URL: https://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf (Дата обращения 12.05.2019). Загл. с экр. Яз. англ.

2 [Введение | OpenCL]. — URL: http://opencl.ru/node/8 (Дата обращения 12.05.2019). Загл. с экр. Яз. рус.

3 [The OpenCL Specification]. — URL: https://www.khronos.org/registry/OpenCL/specs/ (Дата обращения 12.05.2019). Загл. с экр. Яз. англ.

4 [Введение в OpenCL. Архитектура видеокарты]. — URL: https://compscicenter.ru/courses/video_cards_computation/2018-autumn/classes/3980/ (Дата обращения 12.05.2019). Загл. с экр. Яз. рус.

5 [Achieved Occupancy]. — URL: https://docs.nvidia.com/gameworks/content/developertools/desktop/analysis/report/cudaexperiments/kernellevel/achievedoccupancy.htm (Дата обращения 12.05.2019). Загл. с экр. Яз. англ.

6 [Неоднородные рабочие группы OpenCL 2.0]. — URL: https://software.intel.com/ru-ru/articles/opencl-20-non-uniform-work-groups (Дата обращения 12.05.2019). Загл. с экр. Яз. рус.

7 *Aaftab Munshi Benedict R. Gaster, T. G. M. J. F. D. G.* OpenCL Programming Guide / T. G. M. J. F. D. G. Aaftab Munshi, Benedict R. Gaster. — Ann Arbor, Michigan: Edwards Brothers, 2012.

ПРИЛОЖЕНИЕ А
Листинг программы

Вычисление суммы двух массивов.

```
// HelloWorld.cpp
//
//    В данном примере продемонстрирована базовая установка и использование OpenCL
//

#include <iostream>
#include <fstream>
#include <sstream>

#ifdef __APPLE__
#include <OpenCL/cl.h>
#else
#include <CL/cl.h>
#endif

///
//  Константы
//
const int ARRAY_SIZE = 1000;

///
//  Создание OpenCL контекста на основе доступной платформы,
//  использующей GPU (в приоритете) или CPU
//
cl_context CreateContext()
{
    cl_int errNum;
    cl_uint numPlatforms;
    cl_platform_id firstPlatformId;
    cl_context context = NULL;

    // Выберем OpenCL платформу, на которой будет запущен код.
    // В данном примере выберем первую доступную платформу.
    errNum = clGetPlatformIDs(1, &firstPlatformId, &numPlatforms);
    if (errNum != CL_SUCCESS || numPlatforms <= 0)
    {
        std::cerr << "Failed to find any OpenCL platforms." << std::endl;
        return NULL;
```

```
39      }
40
41      // Создадим OpenCL контекст на заданной платформе.
42      // Попробуем создать основанный на GPU контекст и в случае
43      // неудача попробуем создать основанный на CPU контекст
44      cl_context_properties contextProperties[] =
45      {
46          CL_CONTEXT_PLATFORM,
47          (cl_context_properties)firstPlatformId,
48          0
49      };
50      context = clCreateContextFromType(contextProperties, CL_DEVICE_TYPE_GPU,
51                                        NULL, NULL, &errNum);
52      if (errNum != CL_SUCCESS)
53      {
54          std::cout << "Could not create GPU context, trying CPU..." << std::endl;
55          context = clCreateContextFromType(contextProperties, CL_DEVICE_TYPE_CPU,
56                                            NULL, NULL, &errNum);
57          if (errNum != CL_SUCCESS)
58          {
59              std::cerr << "Failed to create an OpenCL GPU or CPU context." << std::endl
60              return NULL;
61          }
62      }
63
64      return context;
65  }
66
67  ///
68  //  Создание командной очередь для первого доступного
69  //  устройства из контекста
70  //
71  cl_command_queue CreateCommandQueue(cl_context context, cl_device_id *device)
72  {
73      cl_int errNum;
74      cl_device_id *devices;
75      cl_command_queue commandQueue = NULL;
76      size_t deviceBufferSize = -1;
77
78      // Получить размер буфера устройства
79      errNum = clGetContextInfo(context, CL_CONTEXT_DEVICES, 0, NULL, &deviceBufferSize)
```

```
80      if (errNum != CL_SUCCESS)
81      {
82          std::cerr << "Failed call to clGetContextInfo(...,GL_CONTEXT_DEVICES,...)";
83          return NULL;
84      }
85
86      if (deviceBufferSize <= 0)
87      {
88          std::cerr << "No devices available.";
89          return NULL;
90      }
91
92      // Выделить память под буфер устройства
93      devices = new cl_device_id[deviceBufferSize / sizeof(cl_device_id)];
94      errNum = clGetContextInfo(context, CL_CONTEXT_DEVICES, deviceBufferSize, devices,
95      if (errNum != CL_SUCCESS)
96      {
97          delete [] devices;
98          std::cerr << "Failed to get device IDs";
99          return NULL;
100     }
101
102     // Выбор первого доступного устройства
103     commandQueue = clCreateCommandQueue(context, devices[0], 0, NULL);
104     if (commandQueue == NULL)
105     {
106         delete [] devices;
107         std::cerr << "Failed to create commandQueue for device 0";
108         return NULL;
109     }
110
111     *device = devices[0];
112     delete [] devices;
113     return commandQueue;
114 }
115
116 ///
117 //  Создание OpenCL программы из файла-kernel
118 //
119 cl_program CreateProgram(cl_context context, cl_device_id device, const char* fileNam
120 {
```

```
121     cl_int errNum;
122     cl_program program;
123 
124     std::ifstream kernelFile(fileName, std::ios::in);
125     if (!kernelFile.is_open())
126     {
127         std::cerr << "Failed to open file for reading: " << fileName << std::endl;
128         return NULL;
129     }
130 
131     std::ostringstream oss;
132     oss << kernelFile.rdbuf();
133 
134     std::string srcStdStr = oss.str();
135     const char *srcStr = srcStdStr.c_str();
136     program = clCreateProgramWithSource(context, 1,
137                                         (const char**)&srcStr,
138                                         NULL, NULL);
139     if (program == NULL)
140     {
141         std::cerr << "Failed to create CL program from source." << std::endl;
142         return NULL;
143     }
144 
145     errNum = clBuildProgram(program, 0, NULL, NULL, NULL, NULL);
146     if (errNum != CL_SUCCESS)
147     {
148         char buildLog[16384];
149         clGetProgramBuildInfo(program, device, CL_PROGRAM_BUILD_LOG,
150                               sizeof(buildLog), buildLog, NULL);
151 
152         std::cerr << "Error in kernel: " << std::endl;
153         std::cerr << buildLog;
154         clReleaseProgram(program);
155         return NULL;
156     }
157 
158     return program;
159 }
160 
161 ///
```

```
162 //  Создание объектов-памяти для kernel
163 //  Kernel принимает 3 аргумента: result (вывод), a (ввод),
164 //  and b (ввод)
165 //
166 bool CreateMemObjects(cl_context context, cl_mem memObjects[3],
167                       float *a, float *b)
168 {
169     memObjects[0] = clCreateBuffer(context, CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
170                                    sizeof(float) * ARRAY_SIZE, a, NULL);
171     memObjects[1] = clCreateBuffer(context, CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
172                                    sizeof(float) * ARRAY_SIZE, b, NULL);
173     memObjects[2] = clCreateBuffer(context, CL_MEM_READ_WRITE,
174                                    sizeof(float) * ARRAY_SIZE, NULL, NULL);
175
176     if (memObjects[0] == NULL || memObjects[1] == NULL || memObjects[2] == NULL)
177     {
178         std::cerr << "Error creating memory objects." << std::endl;
179         return false;
180     }
181
182     return true;
183 }
184
185 ///
186 //  Очистка от созданных OpenCL ресурсов
187 //
188 void Cleanup(cl_context context, cl_command_queue commandQueue,
189              cl_program program, cl_kernel kernel, cl_mem memObjects[3])
190 {
191     for (int i = 0; i < 3; i++)
192     {
193         if (memObjects[i] != 0)
194             clReleaseMemObject(memObjects[i]);
195     }
196     if (commandQueue != 0)
197         clReleaseCommandQueue(commandQueue);
198
199     if (kernel != 0)
200         clReleaseKernel(kernel);
201
202     if (program != 0)
```

```
203         clReleaseProgram(program);
204 
205     if (context != 0)
206         clReleaseContext(context);
207 
208 }
209 
210 ///
211 //        main() для HelloWorld
212 //
213 int main(int argc, char** argv)
214 {
215     cl_context context = 0;
216     cl_command_queue commandQueue = 0;
217     cl_program program = 0;
218     cl_device_id device = 0;
219     cl_kernel kernel = 0;
220     cl_mem memObjects[3] = { 0, 0, 0 };
221     cl_int errNum;
222 
223     // Создание OpenCL контекста для первой доступной платформы
224     context = CreateContext();
225     if (context == NULL)
226     {
227         std::cerr << "Failed to create OpenCL context." << std::endl;
228         return 1;
229     }
230 
231     // Создание очереди команд для первого доступного устройства
232     // в заданном контексте
233     commandQueue = CreateCommandQueue(context, &device);
234     if (commandQueue == NULL)
235     {
236         Cleanup(context, commandQueue, program, kernel, memObjects);
237         return 1;
238     }
239 
240     // Создание OpenCL программы из файла исходного кода HelloWorld.cl для kernel
241     program = CreateProgram(context, device, "HelloWorld.cl");
242     if (program == NULL)
243     {
```

```
244         Cleanup(context, commandQueue, program, kernel, memObjects);
245         return 1;
246     }
247 
248     // Create OpenCL kernel
249     kernel = clCreateKernel(program, "hello_kernel", NULL);
250     if (kernel == NULL)
251     {
252         std::cerr << "Failed to create kernel" << std::endl;
253         Cleanup(context, commandQueue, program, kernel, memObjects);
254         return 1;
255     }
256 
257     // Создание объектов памяти, используемых kernel.
258     // Сначала создаются объекты памяти, содержащие данные
259     // для аргументов kernel
260     float result[ARRAY_SIZE];
261     float a[ARRAY_SIZE];
262     float b[ARRAY_SIZE];
263     for (int i = 0; i < ARRAY_SIZE; i++)
264     {
265         a[i] = (float)i;
266         b[i] = (float)(i * 2);
267     }
268 
269     if (!CreateMemObjects(context, memObjects, a, b))
270     {
271         Cleanup(context, commandQueue, program, kernel, memObjects);
272         return 1;
273     }
274 
275     // Задание аргументов kernel (a, b, result).
276     errNum = clSetKernelArg(kernel, 0, sizeof(cl_mem), &memObjects[0]);
277     errNum |= clSetKernelArg(kernel, 1, sizeof(cl_mem), &memObjects[1]);
278     errNum |= clSetKernelArg(kernel, 2, sizeof(cl_mem), &memObjects[2]);
279     if (errNum != CL_SUCCESS)
280     {
281         std::cerr << "Error setting kernel arguments." << std::endl;
282         Cleanup(context, commandQueue, program, kernel, memObjects);
283         return 1;
284     }
```

```
285
286     size_t globalWorkSize[1] = { ARRAY_SIZE };
287     size_t localWorkSize[1] = { 1 };
288
289     // Поставить kernel в очередь на исполнение
290     errNum = clEnqueueNDRangeKernel(commandQueue, kernel, 1, NULL,
291                                     globalWorkSize, localWorkSize,
292                                     0, NULL, NULL);
293     if (errNum != CL_SUCCESS)
294     {
295         std::cerr << "Error queuing kernel for execution." << std::endl;
296         Cleanup(context, commandQueue, program, kernel, memObjects);
297         return 1;
298     }
299
300     // Считать выходной буфер в основную программу
301     errNum = clEnqueueReadBuffer(commandQueue, memObjects[2], CL_TRUE,
302                                  0, ARRAY_SIZE * sizeof(float), result,
303                                  0, NULL, NULL);
304     if (errNum != CL_SUCCESS)
305     {
306         std::cerr << "Error reading result buffer." << std::endl;
307         Cleanup(context, commandQueue, program, kernel, memObjects);
308         return 1;
309     }
310
311     // Вывод результирующего буфера
312     for (int i = 0; i < ARRAY_SIZE; i++)
313     {
314         std::cout << result[i] << " ";
315     }
316     std::cout << std::endl;
317     std::cout << "Executed program succesfully." << std::endl;
318     Cleanup(context, commandQueue, program, kernel, memObjects);
319
320     return 0;
321 }
```

Kernel для данной задачи.

```
1
2 __kernel void hello_kernel(__global const float *a,
```

```
3                                                   __global const float *b,
4                                                   __global float *result)
5 {
6     int gid = get_global_id(0);
7
8     result[gid] = a[gid] + b[gid];
9 }
```

Kernel для «наивной» реализации задачи транспонирования матрицы.

```
1
2 __kernel void transpose1(__global float *a,
3                                                   __global float *at,
4                                                   unsigned int m, unsigned int n)
5 {
6     int i = get_global_id(0);
7         int j = get_global_id(1);
8
9         float x = a[j * k + i];
10    at[i * m + j] = x;
11 }
```

Kernel для эффективной реализации задачи транспонирования матрицы.

```
1 #define TILE_SIZE 32
2 __kernel void transpose2(__global float *a,
3                                                   __global float *at,
4                                                   unsigned int m, unsigned int n)
5 {
6         int i = get_global_id(0);
7         int j = get_global_id(1);
8
9     _local float tile[TILE_SIZE][TILE_SIZE];
10        int local_i = get_local_id(0);
11        int local_j = get_local_id(1);
12
13        tile[local_j][local_i] = a[j * k + i];
14        barrier(CLK_LOCAL_MEM_FENCE);
15
16        float tmp = tile[local_j][i];
17        tile[local_j][local_i] = tile[local_i][local_j];
18        tile[local_i][local_j] = tmp;
19        barrier(CLK_LOCAL_MEM_FENCE);
```

```
20
21          at[i * m + j] = tile[j * TILE_SIZE][i];
22 }
```

ПРИЛОЖЕНИЕ Б

Листинг сборочных файлов CMake

Код сборочного файла CMakeLists.txt

```
# This is an example project to show and test the usage of the FindOpenCL
# script.

cmake_minimum_required( VERSION 2.6 )
project( CL_Book )

set(CMAKE_MODULE_PATH "${CMAKE_SOURCE_DIR}/cmake")

find_package( OpenCL REQUIRED )

include_directories( ${OPENCL_INCLUDE_DIRS} )
include_directories( "${CMAKE_SOURCE_DIR}/khronos" )

SUBDIRS(src)

```

Пример сборочного файла CMakeLists.txt для проекта с транспонированием матрицы

```
# This is an example project to show and test the usage of the FindOpenCL
# script.

cmake_minimum_required( VERSION 2.6 )
project( CL_Book )

set(CMAKE_MODULE_PATH "${CMAKE_SOURCE_DIR}/cmake")

find_package( OpenCL REQUIRED )

include_directories( ${OPENCL_INCLUDE_DIRS} )
include_directories( "${CMAKE_SOURCE_DIR}/khronos" )

SUBDIRS(src)

```