

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 .....              | 12 |
| 2.3 Инициализация OpenCL программы .....          | 13 |
| 2.4 Задачи на одномерных массивах .....           | 14 |
| 2.4.1 Вычисление суммы ряда .....                 | 14 |
| 2.4.2 Нахождение максимального префикса .....     | 16 |
| 2.5 Задачи на двумерных массивах .....            | 17 |
| 2.5.1 Транспонирование матрицы .....              | 17 |
| 2.5.2 Умножение матриц .....                      | 20 |
| ЗАКЛЮЧЕНИЕ .....                                  | 24 |
| СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ .....            | 25 |
| Приложение А Листинг программы .....              | 27 |
| Приложение Б Листинг сборочных файлов CMake ..... | 47 |

## ВВЕДЕНИЕ

Ранние видеокарты использовались для решения узкоспециализированных задач по отображению графических элементов на экране. До определенного момента их развитие происходило отдельно 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 в начале 2000-х закон Мура перестал действовать. Производительность было решено увеличивать за счет добавления количества 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, ставшее следствием слабой масштабируемости процессоров из-за проблем синхронизации потоков [\[1\]](#).

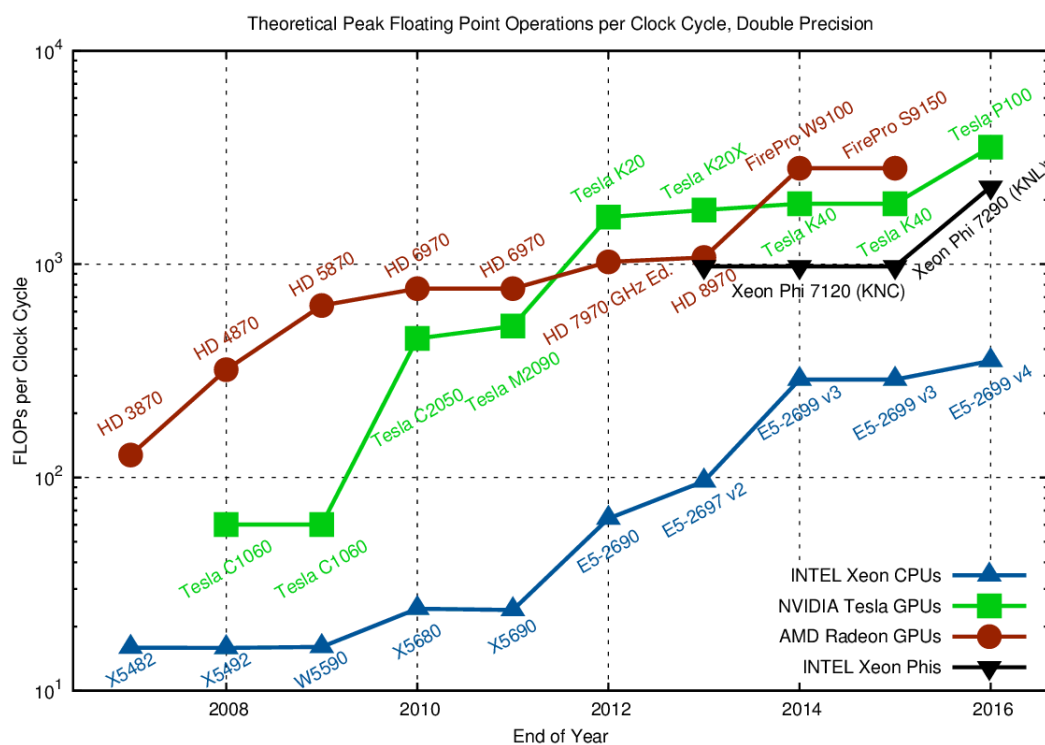


Рисунок 1 – Теоретический максимум количества операций с плавающей запятой в такт для GPU и CPU на 2016 год. График наилучшего по производительности процессора изображен снизу.

Можно заметить, что небольшое количество многофункциональных ядер процессоров не превосходит по производительности видеокарты, устроенные таким образом чтобы сотни небольших ядер работали параллельно.

В связи с тем что видеокарты созданы под решение определенного класса задач, алгоритмы должны обладать определенными свойствами чтобы решение 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;
- провести исследование производительности параллельных программ на различных видеокартах.

## 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.

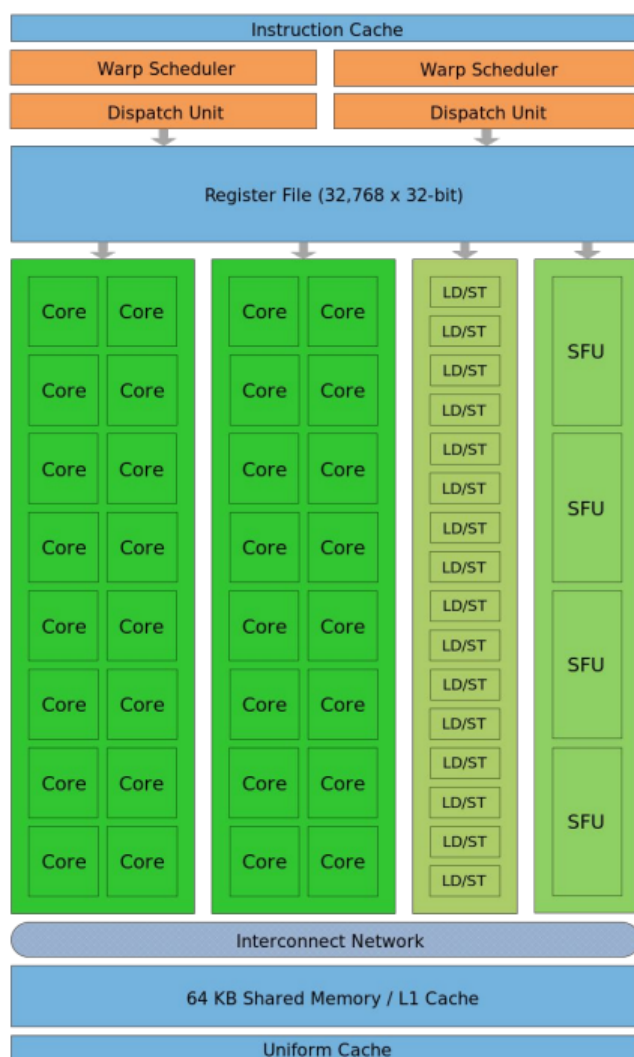


Рисунок 2 – Архитектура потокового мультипроцессора Fermi.

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

cores), каждое из которых в состоянии работы является **поток**ом. В отличие от процессора, ядра в видеокарте выполняют более узкий набор задач, что позволяет с меньшими затратами увеличить их количество в устройстве [2]. 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 сокращения задержки на загрузку памяти и быстрого переключения контекста, подробнее - в разделе 2.1.

Количество вычислительных устройств в видеокарте определяется следующим соотношением:

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 вычислительных устройств. В видеокартах AMD в одном вычислительном устройстве 64 ядра, образующие во время выполнения **wavefront**

## 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 [3].

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 представ-

лять:

- 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 потока в рабочем пространстве 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 отличие от остальной программы полностью выполняется на видеокарте [4].

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 локальный 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.

Используя приведенную выше терминологию, можно сказать что каждое ядро, выполняя заданный kernel, является work-item в некоторой рабочей группе, 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.

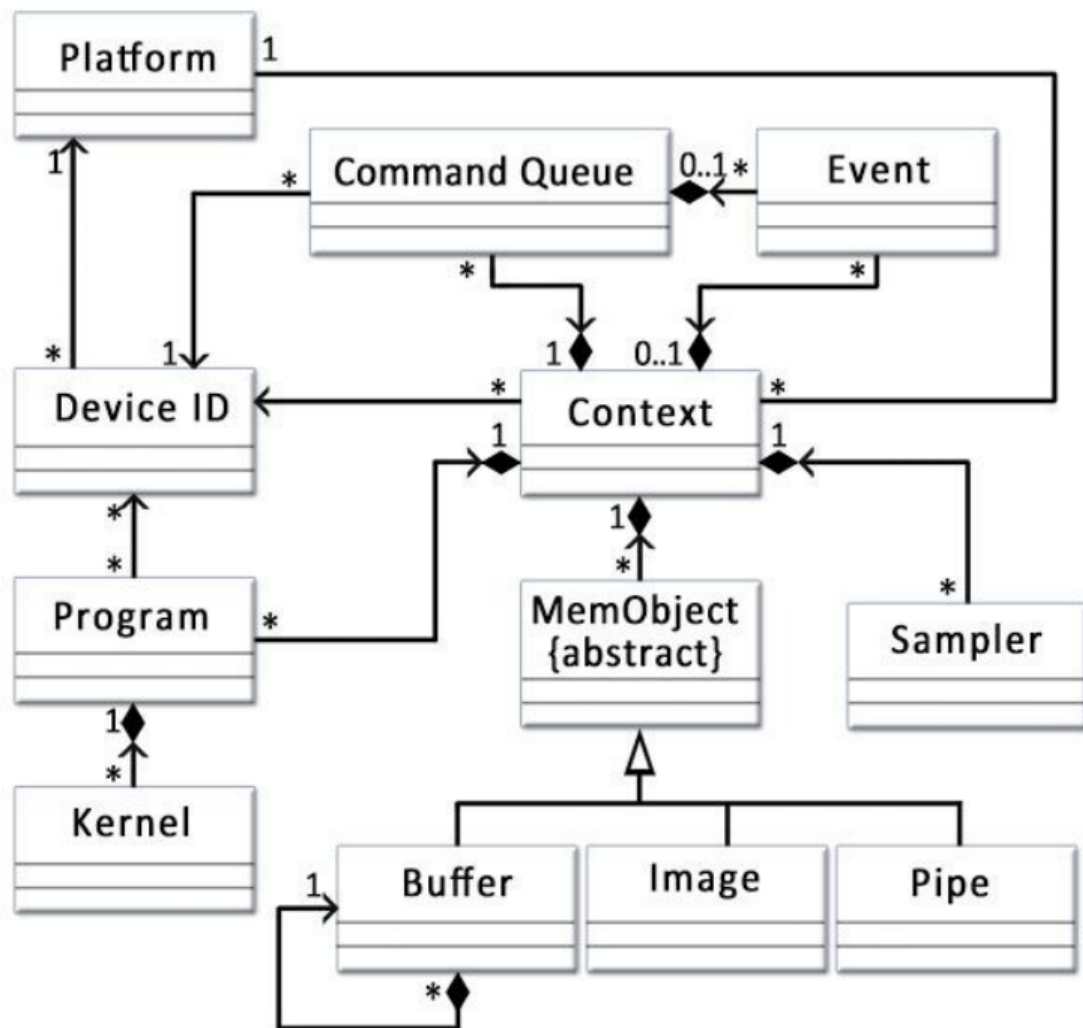


Рисунок 3 – Основные сущности в OpenCL.



## 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.
- сравнение времени выполнения параллельных программ на различных видеокартах, а также с последовательными вычислениями на процессоре.

### 2.1 Требования к алгоритмам

Любой алгоритм можно вычислить на видеокарте, но эффективность в сравнении с реализацией на центральном процессоре зависит от корректности 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 срабатывании if-части должны выполнить внутреннюю часть, однако это не совсем так, и если у потока

предикат — False, он будет скрыт от выполнения внутренней части, аналогично и для else-части. 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 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 в данной рабочей группе 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*) [5].

Другими словами, *warp* может быстро сохранить в регистровую память 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* при условии что доступ к памяти завершился, либо активируется третий *warp* и так далее. Следствие — высокая вычислительная мощность и большая про-

пуская способность видеокарты [6].

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

- количества регистров / количество используемых в warp регистров;
- количества локальной памяти / количество используемой локальной памяти;
- максимально допустимого количества warp (~10).

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

$$\text{среднее кол-во активных warp} / \text{максимальное кол-во активных warp}$$

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 сделать.

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 [5].

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 [7].

В OpenCL 2.0 появилась новая возможность, в которой устранена дан-

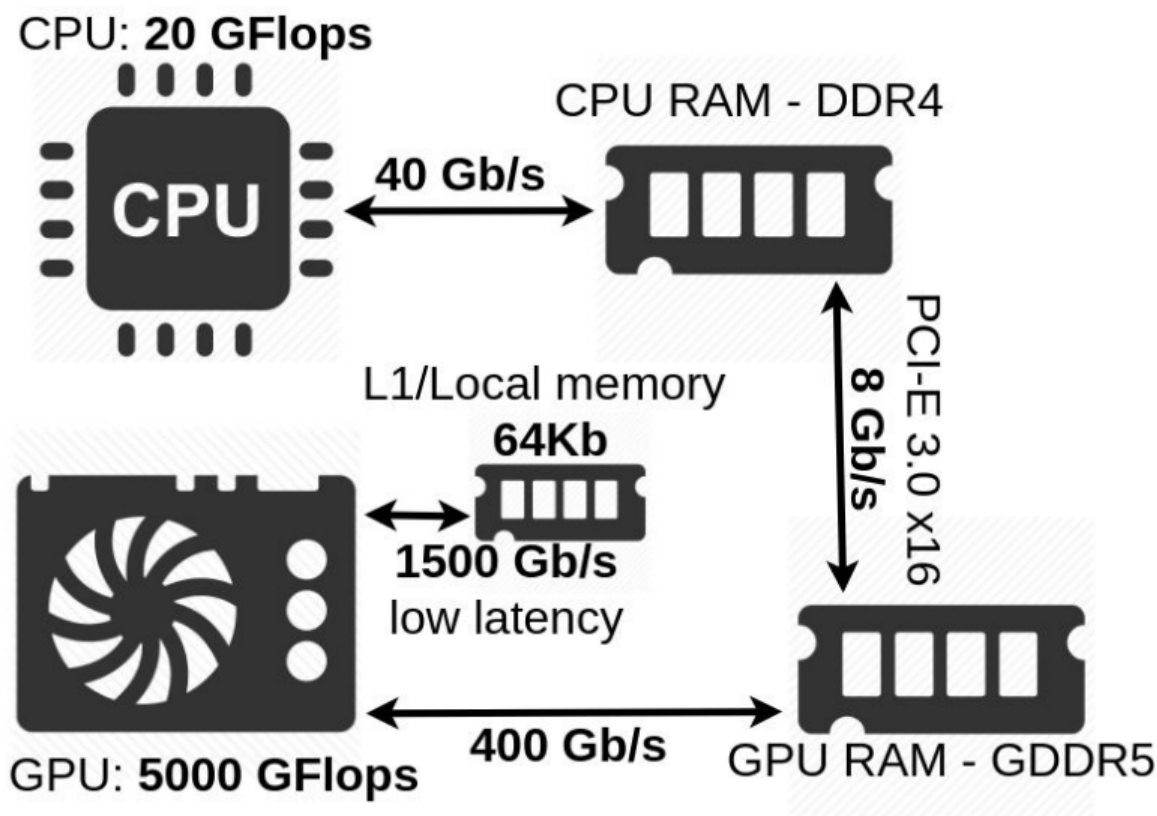


Рисунок 4 – Доступные ресурсы — память.

ная 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 можно 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.

## 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

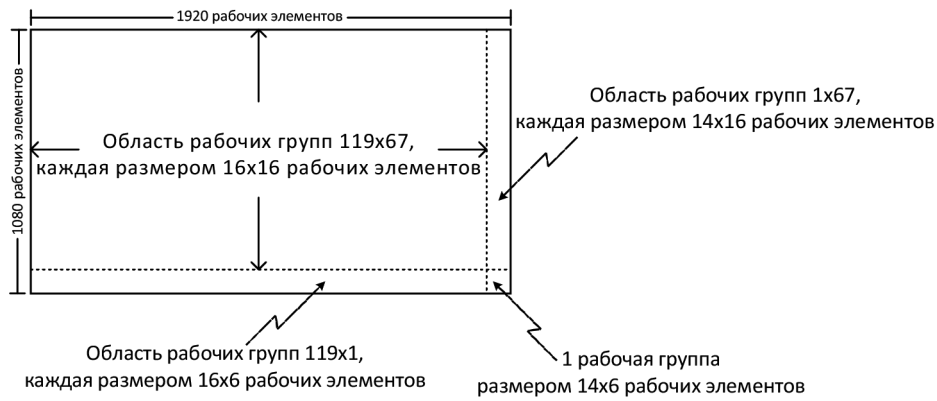


Рисунок 5 – Разделение NDRange на рабочие группы разных размеров.

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.

### 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 [\[8\]](#). С полным кодом, содержащим комментарии, переведенными на русский язык, можно ознакомиться в приложении [А](#), 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 про-

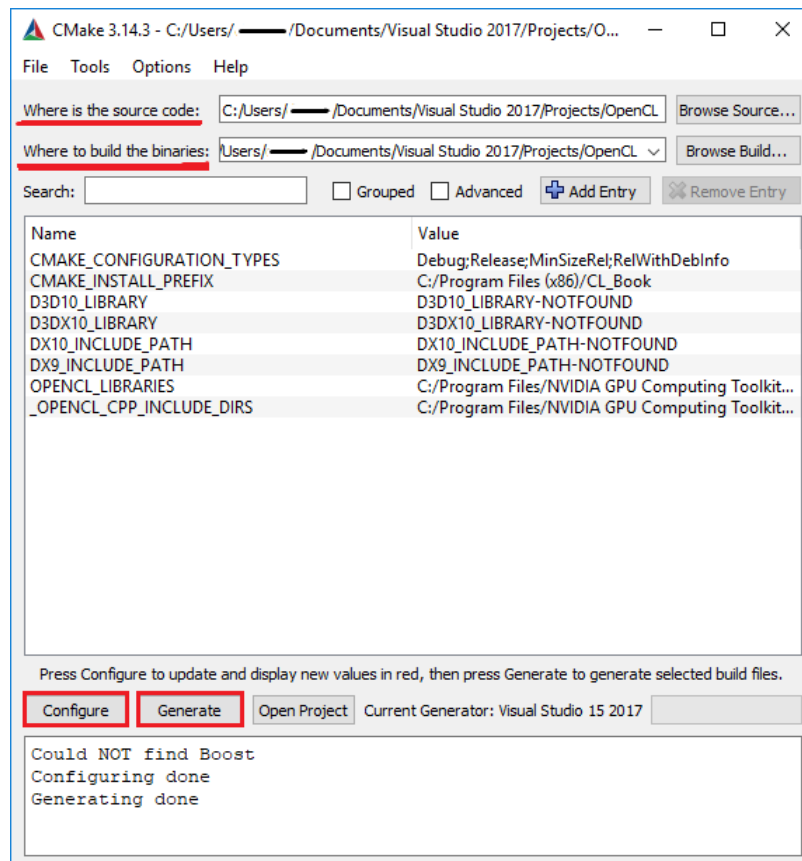


Рисунок 6 – Окно программы CMake-gui.

грамма `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 задачи на одномерных массивах, демонстрируя разные возможности OpenCL.

### 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 видеопамати 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()` [9].

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.

```

C:\WINDOWS\system32\cmd.exe
86 1089 1092 1095 1098 1101 1104 1107 1110 1113 1116 1119 1122 1125 1128 1131 1134 1137 1140 1143 1146 1149 1152 1155 1158 1161
58 1161 1164 1167 1170 1173 1176 1179 1182 1185 1188 1191 1194 1197 1200 1203 1206 1209 1212 1215 1218 1221 1224 1227 1230
30 1233 1236 1239 1242 1245 1248 1251 1254 1257 1260 1263 1266 1269 1272 1275 1278 1281 1284 1287 1290 1293 1296 1299 1302
02 1305 1308 1311 1314 1317 1320 1323 1326 1329 1332 1335 1338 1341 1344 1347 1350 1353 1356 1359 1362 1365 1368 1371 1374
74 1377 1380 1383 1386 1389 1392 1395 1398 1401 1404 1407 1410 1413 1416 1419 1422 1425 1428 1431 1434 1437 1440 1443 1446
46 1449 1452 1455 1458 1461 1464 1467 1470 1473 1476 1479 1482 1485 1488 1491 1494 1497 1500 1503 1506 1509 1512 1515 1518
18 1521 1524 1527 1530 1533 1536 1539 1542 1545 1548 1551 1554 1557 1560 1563 1566 1569 1572 1575 1578 1581 1584 1587 1590
90 1593 1596 1599 1602 1605 1608 1611 1614 1617 1620 1623 1626 1629 1632 1635 1638 1641 1644 1647 1650 1653 1656 1659 1662
62 1665 1668 1671 1674 1677 1680 1683 1686 1689 1692 1695 1698 1701 1704 1707 1710 1713 1716 1719 1722 1725 1728 1731 1734
34 1737 1740 1743 1746 1749 1752 1755 1758 1761 1764 1767 1770 1773 1776 1779 1782 1785 1788 1791 1794 1797 1800 1803 1806
06 1809 1812 1815 1818 1821 1824 1827 1830 1833 1836 1839 1842 1845 1848 1851 1854 1857 1860 1863 1866 1869 1872 1875 1878
78 1881 1884 1887 1890 1893 1896 1899 1902 1905 1908 1911 1914 1917 1920 1923 1926 1929 1932 1935 1938 1941 1944 1947 1950
50 1953 1956 1959 1962 1965 1968 1971 1974 1977 1980 1983 1986 1989 1992 1995 1998 2001 2004 2007 2010 2013 2016 2019 2022
22 2025 2028 2031 2034 2037 2040 2043 2046 2049 2052 2055 2058 2061 2064 2067 2070 2073 2076 2079 2082 2085 2088 2091 2094
94 2097 2100 2103 2106 2109 2112 2115 2118 2121 2124 2127 2130 2133 2136 2139 2142 2145 2148 2151 2154 2157 2160 2163 2166
66 2169 2172 2175 2178 2181 2184 2187 2190 2193 2196 2199 2202 2205 2208 2211 2214 2217 2220 2223 2226 2229 2232 2235 2238
38 2241 2244 2247 2250 2253 2256 2259 2262 2265 2268 2271 2274 2277 2280 2283 2286 2289 2292 2295 2298 2301 2304 2307 2310
10 2313 2316 2319 2322 2325 2328 2331 2334 2337 2340 2343 2346 2349 2352 2355 2358 2361 2364 2367 2370 2373 2376 2379 2382
82 2385 2388 2391 2394 2397 2400 2403 2406 2409 2412 2415 2418 2421 2424 2427 2430 2433 2436 2439 2442 2445 2448 2451 2454
54 2457 2460 2463 2466 2469 2472 2475 2478 2481 2484 2487 2490 2493 2496 2499 2502 2505 2508 2511 2514 2517 2520 2523 2526
26 2529 2532 2535 2538 2541 2544 2547 2550 2553 2556 2559 2562 2565 2568 2571 2574 2577 2580 2583 2586 2589 2592 2595 2598
98 2601 2604 2607 2610 2613 2616 2619 2622 2625 2628 2631 2634 2637 2640 2643 2646 2649 2652 2655 2658 2661 2664 2667 2670
70 2673 2676 2679 2682 2685 2688 2691 2694 2697 2700 2703 2706 2709 2712 2715 2718 2721 2724 2727 2730 2733 2736 2739 2742
42 2745 2748 2751 2754 2757 2760 2763 2766 2769 2772 2775 2778 2781 2784 2787 2790 2793 2796 2799 2802 2805 2808 2811 2814
14 2817 2820 2823 2826 2829 2832 2835 2838 2841 2844 2847 2850 2853 2856 2859 2862 2865 2868 2871 2874 2877 2880 2883 2886
86 2889 2892 2895 2898 2901 2904 2907 2910 2913 2916 2919 2922 2925 2928 2931 2934 2937 2940 2943 2946 2949 2952 2955 2958
58 2961 2964 2967 2970 2973 2976 2979 2982 2985 2988 2991 2994 2997 3000 3003 3006 3009 3012 3015 3018 3021 3024 3027 3030
30 3033 3036 3039 3042 3045 3048 3051 3054 3057 3060 3063 3066 3069
Executed program successfully.
Для продолжения нажмите любую клавишу . . .

```

Рисунок 7 – Результат работы программы по суммированию двух векторов.

Проблем с обращением к памяти не имеется так как потоки в рамках одной рабочей группы обращаются к последовательным данным, кэш-линиям.

Таблица 1 – Сравнение времени работы kernel суммирования векторов размерности 2097152

| Устройство             | Размер рабочей группы | Время работы, мкс |
|------------------------|-----------------------|-------------------|
| NVIDIA Geforce 1050 Ti | 1                     | 25026             |
| NVIDIA Geforce 560 Ti  | 1                     | 44150             |
| NVIDIA Geforce 1050 Ti | 128                   | 14345             |
| NVIDIA Geforce 560 Ti  | 128                   | 29097             |

#### 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 решается одним проходом по массиву в последовательных решениях, но требует значительной модификации алгоритма для параллельных вычислений на видеокарте.

Решим задачу рекурсивно. Пусть входными данными для kernel является: массив *a* с *N* заданными числами, массив *prefs*, содержащий *N* нулей. На каждом шаге найдем сумму и максимальный префикс для всех элементов на подмножестве, соответствующем некоторой рабочей группе. Все потоки в warp запрашивают данные с глобальной памяти, и после этого первый поток посчитает сумму элементов и максимальный префикс по всей рабочей группе, и записав результат в массивы *sums* и *prefs* размерностью меньше в *WORK\_GROUP\_SIZE* раз. В результате выполнения kernel получим сумму всех элементов и максимальный префикс в данной рабочей группе. Используя полученные данные, запустим данный kernel, в котором массив *a* это массив *sums* на предыдущем шаге. Повторим процедуру, пока в массиве *prefs* не останется 1 элемент — максимальный префикс на всем массиве.

Код kernel для решения данной задачи:

```

1 __kernel void maxprefix(__global const float a,
2                         __global float sums,
3                         __global float prefs) {
4     int work_group_size = get_local_size(0);
5     __local float al[work_group_size];
6
7     int i = get_global_id(0);
8     int local_i = get_local_id(0);

```



```

9      al[local_i] = a[i];
10
11      barrier(CLK_GLOBAL_MEM_FENCE);
12      if (local_i == 0) {
13          g_id = get_group_id(0);
14          for (int ind = 0; ind < work_group_size; ++ind) {
15              sum[g_id] += al[ind];
16              if (sum[g_id] > prefs[g_id]) {
17                  prefs[g_id] = sum[g_id];
18              }
19          }
20      }
21 }

```

В данном фрагменте использовались функции идентификации рабочей группы, позволяющие узнать номер данной рабочей группы и ее размер, а также номер элемента, соответствующего потоку в данной рабочей группе.

## 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 реализации 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 }

```

```

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 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 элементов меньше 10 добавлены незначащие нули

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 получают 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 (плитка), которые создаются в локальной памяти, доступ к которой происходит быстро [10].

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.

Код оптимизированного kernel:

```

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);

```

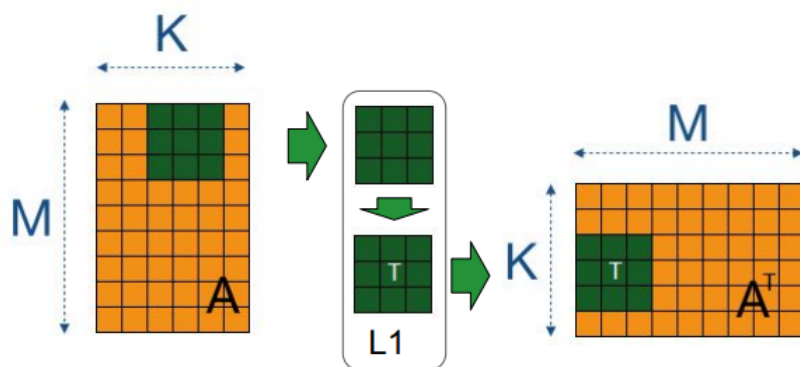


Рисунок 9 – Транспонирование матрицы с использованием «плиток».

```

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 }

```

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

Таблица 2 – Сравнение времени работы kernel транспонирования матрицы на разных устройствах для входных данных размерности 4096x2048, рабочие группы 32x32

| Устройство             | Время работы, мкс |
|------------------------|-------------------|
| NVIDIA Geforce 1050 Ti | 31648             |
| NVIDIA Geforce 560 Ti  | 75519             |

### 2.5.2 Умножение матриц

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

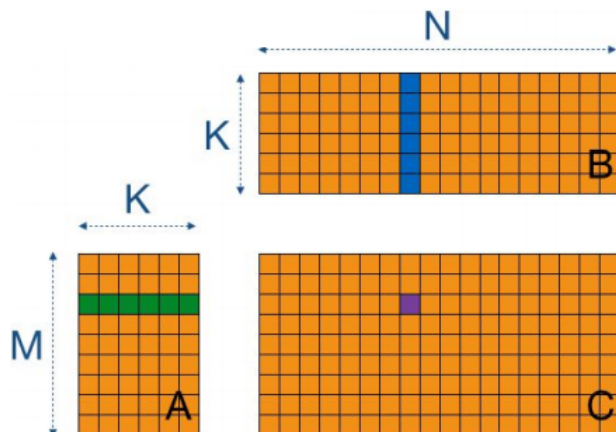


Рисунок 10 – Умножение матриц  $A (M \times K)$  и  $B (K \times N)$ .

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

[10].

```
C:\WINDOWS\system32\cmd.exe
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
Execution time: 766

136 272 408 544 680 816 952 1088 1224 1360 1496 1632 1768 1904 2040 2176
136 272 408 544 680 816 952 1088 1224 1360 1496 1632 1768 1904 2040 2176
136 272 408 544 680 816 952 1088 1224 1360 1496 1632 1768 1904 2040 2176
136 272 408 544 680 816 952 1088 1224 1360 1496 1632 1768 1904 2040 2176
136 272 408 544 680 816 952 1088 1224 1360 1496 1632 1768 1904 2040 2176
136 272 408 544 680 816 952 1088 1224 1360 1496 1632 1768 1904 2040 2176
136 272 408 544 680 816 952 1088 1224 1360 1496 1632 1768 1904 2040 2176
136 272 408 544 680 816 952 1088 1224 1360 1496 1632 1768 1904 2040 2176
136 272 408 544 680 816 952 1088 1224 1360 1496 1632 1768 1904 2040 2176
136 272 408 544 680 816 952 1088 1224 1360 1496 1632 1768 1904 2040 2176
136 272 408 544 680 816 952 1088 1224 1360 1496 1632 1768 1904 2040 2176
136 272 408 544 680 816 952 1088 1224 1360 1496 1632 1768 1904 2040 2176
136 272 408 544 680 816 952 1088 1224 1360 1496 1632 1768 1904 2040 2176
136 272 408 544 680 816 952 1088 1224 1360 1496 1632 1768 1904 2040 2176
136 272 408 544 680 816 952 1088 1224 1360 1496 1632 1768 1904 2040 2176
Executed program succesfully.
Для продолжения нажмите любую клавишу . . .
```

## выполнения

1024 × 1024

| Устройство             | Размер рабочей группы | Время работы, мкс |
|------------------------|-----------------------|-------------------|
| NVIDIA Geforce 1050 Ti | $1 \times 1$          | 577230            |
| NVIDIA Geforce 560 Ti  | $1 \times 1$          | 1274058           |
| NVIDIA Geforce 1050 Ti | $32 \times 16$        | 27290             |
| NVIDIA Geforce 560 Ti  | $32 \times 16$        | 49821             |

ные для вычисления одной ячейки, а остальные потоки не работают. В случае

рабочей группы  $32 \times 16$  в каждом вычислительном устройстве активны все 32 ядра, и во время выполнения создается 16 warp для каждой рабочей группы, тем самым возможно переключение контекста для сокращения задержки к памяти.

Оптимизируем алгоритм, используя модифицированную версию подхода, представленную в предыдущей задаче. Разобьем задачу на фрагменты в локальной памяти, используемые потоками одной рабочей группы 12.

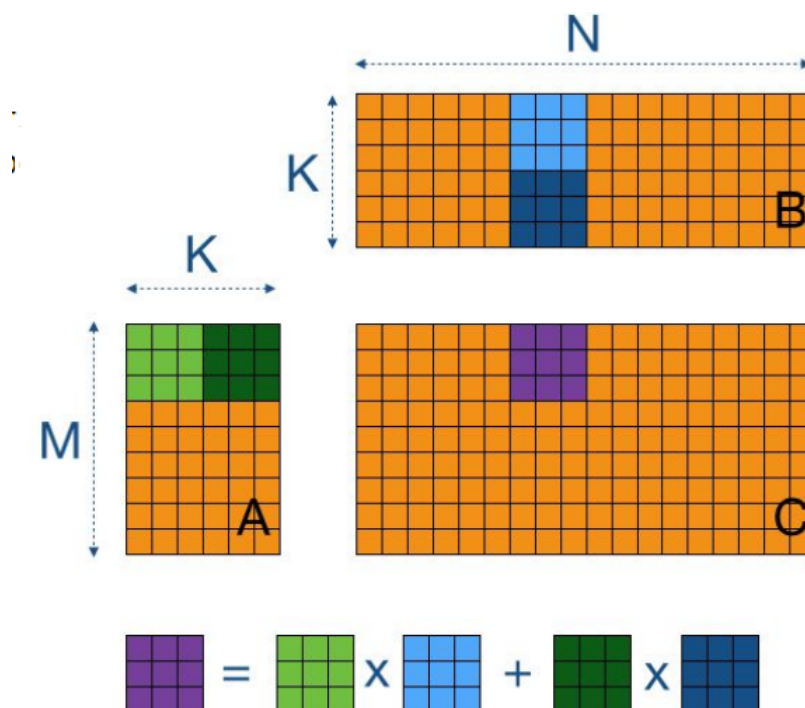


Рисунок 12 – Использование локальной памяти при умножении матриц.

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).

Таблица 4 – Сравнение времени работы оптимизированного kernel умножения матриц размерности  $1024 \times 1024$

| Устройство             | Размер рабочей группы | Время работы, мкс |
|------------------------|-----------------------|-------------------|
| NVIDIA Geforce 1050 Ti | $32 \times 16$        | 13845             |
| NVIDIA Geforce 560 Ti  | $32 \times 16$        | 30107             |

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);
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 данными разных размерностей путем написания высокопроизводительных OpenCL программ, состоящих из хостовых и kernel частей. Проведено сравнение времени выполнения алгоритма на различных конфигурациях OpenCL программы и моделях видеокарт.



## СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

- 1 [FLOPs per Cycle for CPUs, GPUs and Xeon Phis]. — URL: <https://www.karlrupp.net/2016/08/flops-per-cycle-for-cpus-gpus-and-xeon-phis/> (Дата обращения 12.05.2019). Загл. с экр. Яз. англ.
- 2 [NVIDIA's Next Generation CUDA Compute Architecture: Fermi]. — URL: [https://www.nvidia.com/content/PDF/fermi\\_white\\_papers/NVIDIA\\_Fermi\\_Compute\\_Architecture\\_Whitepaper.pdf](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.
- 3 [Введение | OpenCL]. — URL: <http://opencl.ru/node/8> (Дата обращения 12.05.2019). Загл. с экр. Яз. рус.
- 4 [The OpenCL Specification]. — URL: <https://www.khronos.org/registry/OpenCL/specs/> (Дата обращения 12.05.2019). Загл. с экр. Яз. англ.
- 5 [Введение в OpenCL. Архитектура видеокарты]. — URL: [https://compscicenter.ru/courses/video\\_cards\\_computation/2018-autumn/classes/3980/](https://compscicenter.ru/courses/video_cards_computation/2018-autumn/classes/3980/) (Дата обращения 12.05.2019). Загл. с экр. Яз. рус.
- 6 [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.
- 7 [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.
- 8 *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.
- 9 [clEnqueueNDRangeKernel]. — URL: <https://www.khronos.org/registry/OpenCL/sdk/1.0/docs/man/xhtml/clEnqueueNDRangeKernel.html> (Дата обращения 12.05.2019). Загл. с экр. Яз. англ.

- 10 [Умножение матриц | Вычисления на видеокартах].— URL: [https://compscicenter.ru/courses/video\\_cards\\_computation/2018-autumn/classes/4194/](https://compscicenter.ru/courses/video_cards_computation/2018-autumn/classes/4194/) (Дата обращения 12.05.2019). Загл. с экр. Яз. рус.

## ПРИЛОЖЕНИЕ А

### Листинг программы

Код хостовой части программы для задач с входными данными в виде одномерного массива.

```
1 // HelloWorld.cpp
2 //
3 //     В данном примере продемонстрирована базовая установка и использование OpenCL
4 //
5
6 #include <iostream>
7 #include <fstream>
8 #include <sstream>
9
10 #ifdef __APPLE__
11 #include <OpenCL/cl.h>
12 #else
13 #include <CL/cl.h>
14 #endif
15
16 ///
17 //     Константы
18 //
19 const int ARRAY_SIZE = 1000;
20
21 ///
22 //     Создание OpenCL контекста на основе доступной платформы,
23 //     использующей GPU (в приоритете) или CPU
24 //
25 cl_context CreateContext()
26 {
27     cl_int errNum;
28     cl_uint numPlatforms;
29     cl_platform_id firstPlatformId;
30     cl_context context = NULL;
31
32     // Выберем OpenCL платформу, на которой будет запущен код.
33     // В данном примере выберем первую доступную платформу.
34     errNum = clGetPlatformIDs(1, &firstPlatformId, &numPlatforms);
35     if (errNum != CL_SUCCESS || numPlatforms <= 0)
36     {
```

```

37         std::cerr << "Failed to find any OpenCL platforms." << std::endl;
38         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 очереди 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* fileName
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 задачи вычисления суммы двух векторов.

```

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 задачи вычисления максимального префикса в массиве.

```

1 __kernel void maxprefix(__global const float a,
2                           __global float sums,
3                           __global float prefs) {
4     int work_group_size = get_local_size(0);
5     __local float al[work_group_size];
6
7     int i = get_global_id(0);
8     int local_i = get_local_id(0);
9     al[local_i] = a[i];
10
11     barrier(CLK_GLOBAL_MEM_FENCE);
12     if (local_i == 0) {
13         g_id = get_group_id(0);
14         for (int ind = 0; ind < work_group_size; ++ind) {
15             sum[g_id] += al[ind];
16             if (sum[g_id] > prefs[g_id]) {
17                 prefs[g_id] = sum[g_id];
18             }
19         }
20     }
21 }

```

Код хостовой части программы для задач с входными данными в виде двумерного массива.

```

1 // HelloWorld.cpp
2 //
3 //     В данном примере продемонстрирована базовая установка и использование OpenCL
4 //
5
6 #include <iostream>
7 #include <fstream>

```

```

8 #include <sstream>
9 #include <chrono>
10
11 #ifdef __APPLE__
12 #include <OpenCL/cl.h>
13 #else
14 #include <CL/cl.h>
15 #endif
16
17 ///
18 //  Константы
19 //
20 //const int ARRAY_SIZE = 4096;
21 //const int ROWS_COUNT = 2048;
22 const int ARRAY_SIZE = 4096;
23 const int ROWS_COUNT = 2048;
24 #pragma comment(linker, "/STACK:100000000")
25
26 ///
27 //  Создание OpenCL контекста на основе доступной платформы,
28 //  использующей GPU (в приоритете) или CPU
29 //
30 cl_context CreateContext()
31 {
32     cl_int errNum;
33     cl_uint numPlatforms;
34     cl_platform_id firstPlatformId;
35     cl_context context = NULL;
36
37     // Выберем OpenCL платформу, на которой будет запущен код.
38     // В данном примере выберем первую доступную платформу.
39     errNum = clGetPlatformIDs(1, &firstPlatformId, &numPlatforms);
40     if (errNum != CL_SUCCESS || numPlatforms <= 0)
41     {
42         std::cerr << "Failed to find any OpenCL platforms." << std::endl;
43         return NULL;
44     }
45
46     // Создадим OpenCL контекст на заданной платформе.
47     // Попробуем создать основанный на GPU контекст и в случае
48     // неудача попробуем создать основанный на CPU контекст

```

```

49     cl_context_properties contextProperties[] =
50     {
51         CL_CONTEXT_PLATFORM,
52         (cl_context_properties)firstPlatformId,
53         0
54     };
55     context = clCreateContextFromType(contextProperties, CL_DEVICE_TYPE_GPU,
56                                     NULL, NULL, &errNum);
57     if (errNum != CL_SUCCESS)
58     {
59         std::cout << "Could not create GPU context, trying CPU..." << std::endl;
60         context = clCreateContextFromType(contextProperties, CL_DEVICE_TYPE_CPU,
61                                     NULL, NULL, &errNum);
62         if (errNum != CL_SUCCESS)
63         {
64             std::cerr << "Failed to create an OpenCL GPU or CPU context." << std::endl;
65             return NULL;
66         }
67     }
68
69     return context;
70 }
71
72 ///
73 //  Создание командной очередь для первого доступного
74 //  устройства из контекста
75 //
76 cl_command_queue CreateCommandQueue(cl_context context, cl_device_id *device)
77 {
78     cl_int errNum;
79     cl_device_id *devices;
80     cl_command_queue commandQueue = NULL;
81     size_t deviceBufferSize = -1;
82
83     // Получить размер буфера устройства
84     errNum = clGetContextInfo(context, CL_CONTEXT_DEVICES, 0, NULL, &deviceBufferSize);
85     if (errNum != CL_SUCCESS)
86     {
87         std::cerr << "Failed call to clGetContextInfo(...,CL_CONTEXT_DEVICES,...)";
88         return NULL;
89     }

```

```

90
91     if (deviceBufferSize <= 0)
92     {
93         std::cerr << "No devices available.";
94         return NULL;
95     }
96
97     // Выделить память под буфер устройства
98     devices = new cl_device_id[deviceBufferSize / sizeof(cl_device_id)];
99     errNum = clGetContextInfo(context, CL_CONTEXT_DEVICES, deviceBufferSize, devices,
100     if (errNum != CL_SUCCESS)
101     {
102         delete [] devices;
103         std::cerr << "Failed to get device IDs";
104         return NULL;
105     }
106
107     // Выбор первого доступного устройства
108     commandQueue = clCreateCommandQueue(context, devices[0], 0, NULL);
109     if (commandQueue == NULL)
110     {
111         delete [] devices;
112         std::cerr << "Failed to create commandQueue for device 0";
113         return NULL;
114     }
115
116     *device = devices[0];
117     delete [] devices;
118     return commandQueue;
119 }
120
121 ///
122 // Создание OpenCL программы из файла-kernel
123 //
124 cl_program CreateProgram(cl_context context, cl_device_id device, const char* fileName)
125 {
126     cl_int errNum;
127     cl_program program;
128
129     std::ifstream kernelFile(fileName, std::ios::in);
130     if (!kernelFile.is_open())

```

```

131     {
132         std::cerr << "Failed to open file for reading: " << fileName << std::endl;
133         return NULL;
134     }
135
136     std::ostringstream oss;
137     oss << kernelFile.rdbuf();
138
139     std::string srcStdStr = oss.str();
140     const char *srcStr = srcStdStr.c_str();
141     program = clCreateProgramWithSource(context, 1,
142                                         (const char**)&srcStr,
143                                         NULL, NULL);
144     if (program == NULL)
145     {
146         std::cerr << "Failed to create CL program from source." << std::endl;
147         return NULL;
148     }
149
150     errNum = clBuildProgram(program, 0, NULL, NULL, NULL, NULL);
151     if (errNum != CL_SUCCESS)
152     {
153         char buildLog[16384];
154         clGetProgramBuildInfo(program, device, CL_PROGRAM_BUILD_LOG,
155                               sizeof(buildLog), buildLog, NULL);
156
157         std::cerr << "Error in kernel: " << std::endl;
158         std::cerr << buildLog;
159         clReleaseProgram(program);
160         return NULL;
161     }
162
163     return program;
164 }
165
166 ///
167 //  Создание объектов-памяти для kernel
168 //  Kernel принимает 3 аргумента: result (вывод), a (ввод),
169 //  and b (ввод)
170 //
171 bool CreateMemObjects(cl_context context, cl_mem memObjects[2], float *a)

```

```

172 {
173     memObjects[0] = clCreateBuffer(context, CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
174                                     sizeof(float) * ARRAY_SIZE * ROWS_COUNT, a, NULL);
175     memObjects[1] = clCreateBuffer(context, CL_MEM_READ_WRITE,
176                                     sizeof(float) * ARRAY_SIZE * ROWS_COUNT, NULL, NULL);
177
178     if (memObjects[0] == NULL || memObjects[1] == NULL)
179     {
180         std::cerr << "Error creating memory objects." << std::endl;
181         return false;
182     }
183
184     return true;
185 }
186
187 ///
188 //  Очистка от созданных OpenCL ресурсов
189 //
190 void Cleanup(cl_context context, cl_command_queue commandQueue,
191             cl_program program, cl_kernel kernel, cl_mem memObjects[3])
192 {
193     for (int i = 0; i < 3; i++)
194     {
195         if (memObjects[i] != 0)
196             clReleaseMemObject(memObjects[i]);
197     }
198     if (commandQueue != 0)
199         clReleaseCommandQueue(commandQueue);
200
201     if (kernel != 0)
202         clReleaseKernel(kernel);
203
204     if (program != 0)
205         clReleaseProgram(program);
206
207     if (context != 0)
208         clReleaseContext(context);
209
210 }
211
212 ///

```



```

213 //      main() для HelloWorld
214 //
215 int main(int argc, char** argv)
216 {
217     cl_context context = 0;
218     cl_command_queue commandQueue = 0;
219     cl_program program = 0;
220     cl_device_id device = 0;
221     cl_kernel kernel = 0;
222     cl_mem memObjects[2] = { 0, 0 };
223     cl_int errNum;
224
225     // Создание OpenCL контекста для первой доступной платформы
226     context = CreateContext();
227     if (context == NULL)
228     {
229         std::cerr << "Failed to create OpenCL context." << std::endl;
230         return 1;
231     }
232
233     // Создание очереди команд для первого доступного устройства
234     // в заданном контексте
235     commandQueue = CreateCommandQueue(context, &device);
236     if (commandQueue == NULL)
237     {
238         Cleanup(context, commandQueue, program, kernel, memObjects);
239         return 1;
240     }
241
242     // Создание OpenCL программы из файла исходного кода HelloWorld.cl для kernel
243     program = CreateProgram(context, device, "Transpose.cl");
244     if (program == NULL)
245     {
246         Cleanup(context, commandQueue, program, kernel, memObjects);
247         return 1;
248     }
249
250     // Create OpenCL kernel
251     cl_int ciErrNum;
252     kernel = clCreateKernel(program, "transpose1", &ciErrNum);
253     if (kernel == NULL)

```

```

254 {
255     std::cerr << "Failed to create kernel" << std::endl;
256     Cleanup(context, commandQueue, program, kernel, memObjects);
257     return 1;
258 }
259
260 // Создание объектов памяти, используемых kernel.
261 // Сначала создаются объекты памяти, содержащие данные
262 // для аргументов kernel
263 float a[ARRAY_SIZE * ROWS_COUNT];
264 float at[ARRAY_SIZE * ROWS_COUNT];
265 for (int i = 0; i < ROWS_COUNT; i++)
266 {
267     for (int j = 0; j < ARRAY_SIZE; j++)
268     {
269         a[i * ARRAY_SIZE + j] = (float)j;
270         if (a[i * ARRAY_SIZE + j] < 10) {
271             //std::cout << 0;
272         }
273         //std::cout << a[i * ARRAY_SIZE + j] << " ";
274     }
275     //std::cout << std::endl;
276 }
277
278 // Измеряем время работы алгоритма
279 auto t1 = std::chrono::high_resolution_clock::now();
280
281 if (!CreateMemObjects(context, memObjects, a))
282 {
283     Cleanup(context, commandQueue, program, kernel, memObjects);
284     return 1;
285 }
286
287 // Задание аргументов kernel (a, b, result).
288 errNum = clSetKernelArg(kernel, 0, sizeof(cl_mem), &memObjects[0]);
289 errNum |= clSetKernelArg(kernel, 1, sizeof(cl_mem), &memObjects[1]);
290 errNum |= clSetKernelArg(kernel, 2, sizeof(int), &ARRAY_SIZE);
291 errNum |= clSetKernelArg(kernel, 3, sizeof(int), &ROWS_COUNT);
292 if (errNum != CL_SUCCESS)
293 {
294     std::cerr << "Error setting kernel arguments." << std::endl;

```

```

295     Cleanup(context, commandQueue, program, kernel, memObjects);
296     return 1;
297 }
298
299 size_t globalWorkSize[2] = { ARRAY_SIZE, ROWS_COUNT };
300 size_t localWorkSize[2] = { 32, 32 };
301
302 // Поставить kernel в очередь на исполнение
303 errNum = clEnqueueNDRangeKernel(commandQueue, kernel, 2, NULL,
304                                 globalWorkSize, localWorkSize,
305                                 0, NULL, NULL);
306 if (errNum != CL_SUCCESS)
307 {
308     std::cerr << "Error queuing kernel for execution." << std::endl;
309     Cleanup(context, commandQueue, program, kernel, memObjects);
310     return 1;
311 }
312
313 // Считать выходной буфер в основную программу
314 errNum = clEnqueueReadBuffer(commandQueue, memObjects[1], CL_TRUE,
315                              0, ARRAY_SIZE * ROWS_COUNT * sizeof(float), at,
316                              0, NULL, NULL);
317 if (errNum != CL_SUCCESS)
318 {
319     std::cerr << "Error reading result buffer." << std::endl;
320     Cleanup(context, commandQueue, program, kernel, memObjects);
321     return 1;
322 }
323
324 auto t2 = std::chrono::high_resolution_clock::now();
325 std::cout << "Execution time: "
326           << std::chrono::duration_cast<std::chrono::microseconds>(t2 - t1).count()
327           << std::endl;
328
329 // Вывод результирующего буфера
330 //std::cout << std::endl;
331 //for (int i = 0; i < ROWS_COUNT; i++)
332 //{
333 //    for (int j = 0; j < ARRAY_SIZE; j++) {
334 //        if (at[i * ARRAY_SIZE + j] < 10) {
335 //            std::cout << 0;

```

```

336     //      }
337     //      std::cout << at[i * ARRAY_SIZE + j] << " ";
338     //      }
339     //      std::cout << std::endl;
340     //}
341     std::cout << std::endl;
342     std::cout << "Executed program succesfully." << std::endl;
343     Cleanup(context, commandQueue, program, kernel, memObjects);
344
345     return 0;
346 }

```

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 }

```

Kernel для «наивной» реализации задачи умножения матриц.

```

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 }

```

Kernel для эффективной реализации задачи 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);
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 }

```

## ПРИЛОЖЕНИЕ Б

### Листинг сборочных файлов CMake

Код сборочного файла CMakeLists.txt

```
1 # This is an example project to show and test the usage of the FindOpenCL
2 # script.
3
4 cmake_minimum_required( VERSION 2.6 )
5 project( CL_Book )
6
7 set(CMAKE_MODULE_PATH "${CMAKE_SOURCE_DIR}/cmake")
8
9 find_package( OpenCL REQUIRED )
10
11 include_directories( ${OPENCL_INCLUDE_DIRS} )
12 include_directories( "${CMAKE_SOURCE_DIR}/khronos" )
13
14 SUBDIRS(src)
15
```

Пример сборочного файла CMakeLists.txt для проекта с транспонированием матрицы

```
1 cmake_minimum_required(VERSION 3.14)
2 add_executable( Transpose Transpose.cpp )
3 target_link_libraries( Transpose ${OPENCL_LIBRARIES} )
4
5 configure_file(Transpose.cl ${CMAKE_CURRENT_BINARY_DIR}/Transpose.cl COPYONLY)
```