

МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
Кафедра МО ЭВМ

ОТЧЕТ
по лабораторной работе №4
по дисциплине «Параллельные алгоритмы»
Тема: Алгоритмы GPGPU

Студент гр. 3343

Малиновский А.А.

Преподаватель

Сергеева Е.И.

Санкт-Петербург

2025

Цель работы

Целью данной лабораторной работы является реализация и исследование алгоритмов GPGPU, а также их сравнения с алгоритмами на CPU.

Задание

Реализовать алгоритм для исполнения на GPU. Реализацию выполнить на OpenCL/CUDA (выбор фреймворка исходя из доступного аппаратного обеспечения).

4.1 Блочное умножение матриц. Выполнить тестирование и анализ производительности. Оценить ускорение по сравнению с `lr1`. Баллы: 0 -- 8

4.2 Сортировка (алгоритм на выбор из класса "быстрых" сортировок, например merge sort) Выполнить тестирование (сравнение с `сри` реализацией) и анализ производительности. Баллы: 0 -- 10

доп. баллы: защита в срок: 2

Блочное умножение матриц на GPU

Алгоритм блочного умножения матриц на GPU реализован с использованием оптимизированного подхода с локальной памятью для минимизации глобальных обращений к памяти. Основная идея заключается в разделении матриц на небольшие блоки, которые помещаются в быструю локальную память work-group. Каждая work-group размером 16x16 потоков обрабатывает один блок результирующей матрицы. Алгоритм работает итерационно, обрабатывая внутреннюю размерность K по частям размером 16 элементов. На каждой итерации потоки work-group загружают соответствующие блоки из матриц A и B в локальную память, синхронизируются с помощью барьеров, затем выполняют матричное умножение блоков в быстрой локальной памяти. Это позволяет значительно сократить количество дорогостоящих обращений к глобальной памяти, поскольку каждый элемент матриц A и B загружается только один раз в локальную память и затем многократно используется в вычислениях. Кернел использует атрибут `reqd_work_group_size` для гарантии оптимального размера work-group и два барьера синхронизации для корректной работы с разделяемой памятью.

Функция `multiplyMatricesBlocked` выполняет блочное умножение матриц на GPU с использованием оптимизированного кернела OpenCL. Сначала функция проверяет инициализацию GPU и корректность размеров входных матриц. Затем она создает буферы в памяти GPU для матриц A , B и результирующей матрицы C , используя флаги `CL_MEM_READ_ONLY` для входных данных и `CL_MEM_WRITE_ONLY` для результата. После этого устанавливаются аргументы кернела: указатели на буферы матриц и размерности w (ширина результата), h (высота результата) и k (внутренняя размерность). Функция вычисляет размеры `NDRange`, выравнивая глобальные размеры по границам work-group для обеспечения корректной работы кернела. Проверяется поддержка выбранного размера work-group устройством OpenCL. Затем кернел запускается в очереди команд с указанными глобальными и локальными размерами. После завершения вычислений результат копируется из буфера GPU

в host-память с помощью блокирующей операции чтения. В случае ошибок на любом этапе функция генерирует исключения с подробными сообщениями.

Функция `initialize` инициализирует окружение OpenCL для выполнения операций на GPU. Она начинается с получения устройства по умолчанию через `getDefaultDevice`, который сначала ищет GPU устройства, а при их отсутствии fallback на CPU. Создается контекст OpenCL и очередь команд для управления выполнением кернелов. Затем загружается исходный код кернелов из файла `matrix_multiply.cl` и компилируется в программу OpenCL. Если компиляция fails, функция выводит лог сборки для диагностики ошибок. После успешной компиляции создаются объекты кернелов для простого и блочного умножения матриц. Функция обрабатывает исключения на каждом этапе и возвращает `false` при любой ошибке инициализации.

Функция `multiplyMatrices` выполняет простое умножение матриц на GPU без использования локальной памяти. Она создает буферы для входных и выходных матриц, устанавливает аргументы кернела `matrix_multiply_simple`, который вычисляет каждый элемент результирующей матрицы независимо через скалярное произведение соответствующей строки A и столбца B. Кернел запускается с глобальным размером, соответствующим размерности результирующей матрицы, и минимальным локальным размером 1x1. После выполнения результат копируется обратно в host-память.

Функция `readKernelFile` читает содержимое файла с исходным кодом кернелов OpenCL используя итераторы потока для эффективного чтения всего файла в строку. Функция `getDefaultDevice` реализует логику выбора устройства OpenCL, отдавая приоритет GPU над CPU и генерируя исключение при отсутствии доступных устройств.

Сортировка слиянием на GPU

Алгоритм сортировки слиянием на GPU реализован с использованием итеративного подхода, где на каждой итерации происходит слияние упорядоченных блоков возрастающего размера. Основная идея заключается в том, что каждый поток GPU обрабатывает один элемент массива и определяет его окончательную позицию в слитом блоке. На нулевой итерации рассматриваются блоки размером 1 (каждый элемент уже отсортирован), затем на каждой следующей итерации размер блоков удваивается. Алгоритм использует стратегию "odd-even merging", где нечетные блоки сливаются с предыдущими четными блоками. Для определения позиции элемента в сливаемом блоке используется бинарный поиск (функция `upper_bound_cmp`), который эффективно находит позицию вставки в отсортированной последовательности. Каждый поток независимо вычисляет позицию своего элемента в результирующем массиве, что обеспечивает высокую степень параллелизма. Алгоритм использует двойную буферизацию - данные читаются из одного буфера и записываются в другой, затем буферы меняются местами на следующей итерации. Это позволяет избежать конфликтов при чтении-записи и обеспечивает корректность параллельного выполнения.

Функция `sort` выполняет сортировку массива целых чисел на GPU с использованием алгоритма merge sort. Сначала функция вычисляет необходимое количество итераций как логарифм от размера массива по основанию 2. Затем создаются два буфера в GPU памяти для реализации двойной буферизации. Исходный массив копируется в первый буфер. Далее выполняется цикл по всем итерациям слияния, где на каждой итерации размер сливаемых блоков удваивается. Для каждой итерации устанавливаются аргументы ядра: исходный и целевой буферы, размер массива и текущая степень (определяющая размер блоков). Ядро запускается с глобальным размером, выровненным по размеру `work-group`, что обеспечивает оптимальное использование вычислительных ресурсов GPU. После каждой итерации буферы меняются местами, и окончательный результат копируется обратно в `host`-память.

Функция `upper_bound_cmp` реализует бинарный поиск для нахождения позиции вставки элемента в отсортированном подмассиве. Она принимает границы поиска $[l, r)$, значение для вставки и флаг, определяющий тип сравнения (строгое или нестрогое). Алгоритм работает по классической схеме бинарного поиска, уменьшая интервал поиска вдвое на каждом шаге. Функция возвращает первую позицию, куда можно вставить элемент без нарушения упорядоченности.

Кернел `merge_sort` является ядром алгоритма сортировки слиянием. Каждый поток обрабатывает один элемент массива, определяя его позицию в слитом блоке. Поток сначала вычисляет размер текущего блока (2^{row}) и определяет, к какому блоку принадлежит его элемент. Если блок нечетный, элемент ищет свою позицию в предыдущем четном блоке с помощью бинарного поиска. Если блок четный, элемент вычисляет, сколько элементов из следующего нечетного блока должны стоять перед ним. Для оптимизации доступа к памяти используется группировка потоков в `work-groups`, что улучшает локальность кэша. Результат записывается в соответствующую позицию целевого массива.

Функция `sort_with_profiling` выполняет сортировку с измерением времени выполнения. Она использует высокоточные часы для замера времени между началом и концом сортировки, возвращая результат в секундах как `double`. Это позволяет сравнивать производительность разных конфигураций.

Функции `get_max_work_group_size` и `get_preferred_work_group_size` предоставляют информацию об оптимальных размерах `work-group` для данного устройства. Они запрашивают у драйвера OpenCL максимально допустимый и предпочтительный размер `work-group`, что помогает в настройке параметров для достижения максимальной производительности.

Функция `get_device_info` собирает `comprehensive` информацию об устройстве OpenCL, включая название, максимальный размер `work-group`, предпочтительный множитель, количество вычислительных единиц и объем глобальной памяти. Эта информация полезна для диагностики и оптимизации.

Кернел `copy_array` обеспечивает простую операцию копирования данных между буферами в GPU памяти, что используется при инициализации и финализации алгоритма сортировки.

Тестирование корректности

Тесты на корректность умножения матриц проверяют правильность реализации различных алгоритмов умножения на разных типах входных данных. Тестирование начинается с больших квадратных матриц размером 256x256, 512x512 и 1024x1024, где эталонный результат вычисляется с помощью простого последовательного алгоритма на CPU. Для каждой размерности сравниваются результаты блочного умножения на CPU и двух версий GPU-умножения (простой и блочной) с эталонным значением с допустимой погрешностью $1e-3$. Отдельно тестируются прямоугольные матрицы с различными соотношениями размеров M , N , K для проверки корректности работы с несимметричными данными. Тесты согласованности проверяют, что все методы вычислений (простой CPU, блочный CPU, простой GPU, блочный GPU) дают идентичные результаты для одних и тех же входных данных. Алгоритм использует генерацию случайных матриц с равномерным распределением для создания реалистичных тестовых сценариев и измерения максимальной ошибки между результатами.

Тесты на корректность сортировки включают несколько категорий тестовых случаев для проверки алгоритмов на различных типах входных данных. Тестируются случайные массивы разных размеров от маленьких (10-32 элемента) до больших (50,000 элементов) для проверки общей функциональности. Отдельно проверяются уже отсортированные массивы для верификации стабильности алгоритма и обработки лучшего случая. Тесты с обратно отсортированными массивами позволяют проверить работу в худшем случае и эффективность слияния. Каждый тест выполняется четырьмя разными реализациями: GPU сортировка, однопоточная CPU сортировка слиянием, многопоточная CPU сортировка и стандартная библиотечная сортировка. Результаты всех реализаций сравниваются между собой, а для маленьких

массивов при обнаружении расхождений выводится детальная отладочная информация. Все результаты записываются в CSV файл с пометками об успешности каждого теста для последующего анализа.

Графики производительности алгоритмов

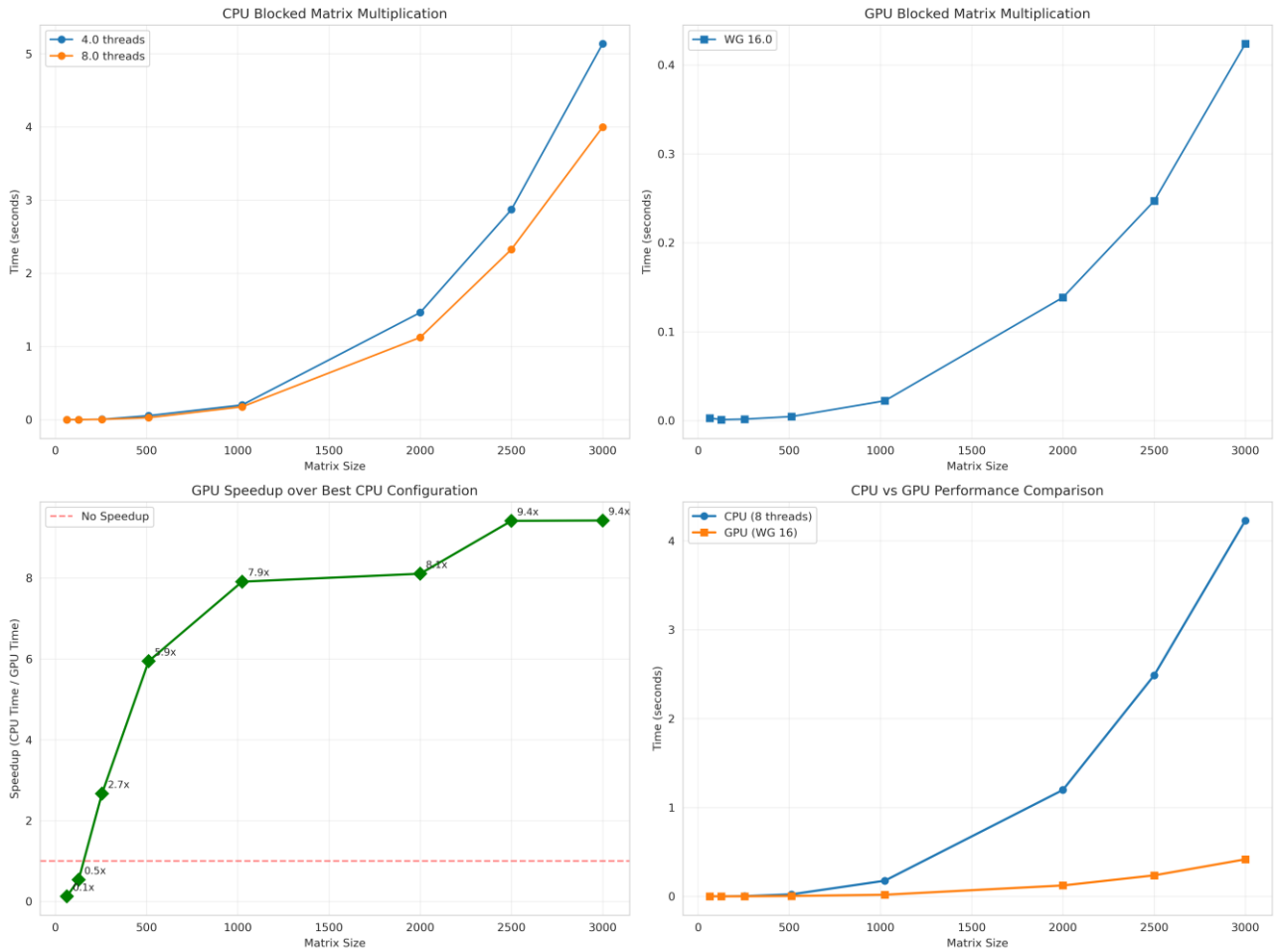


Рисунок 1 – графики блочного умножения матриц

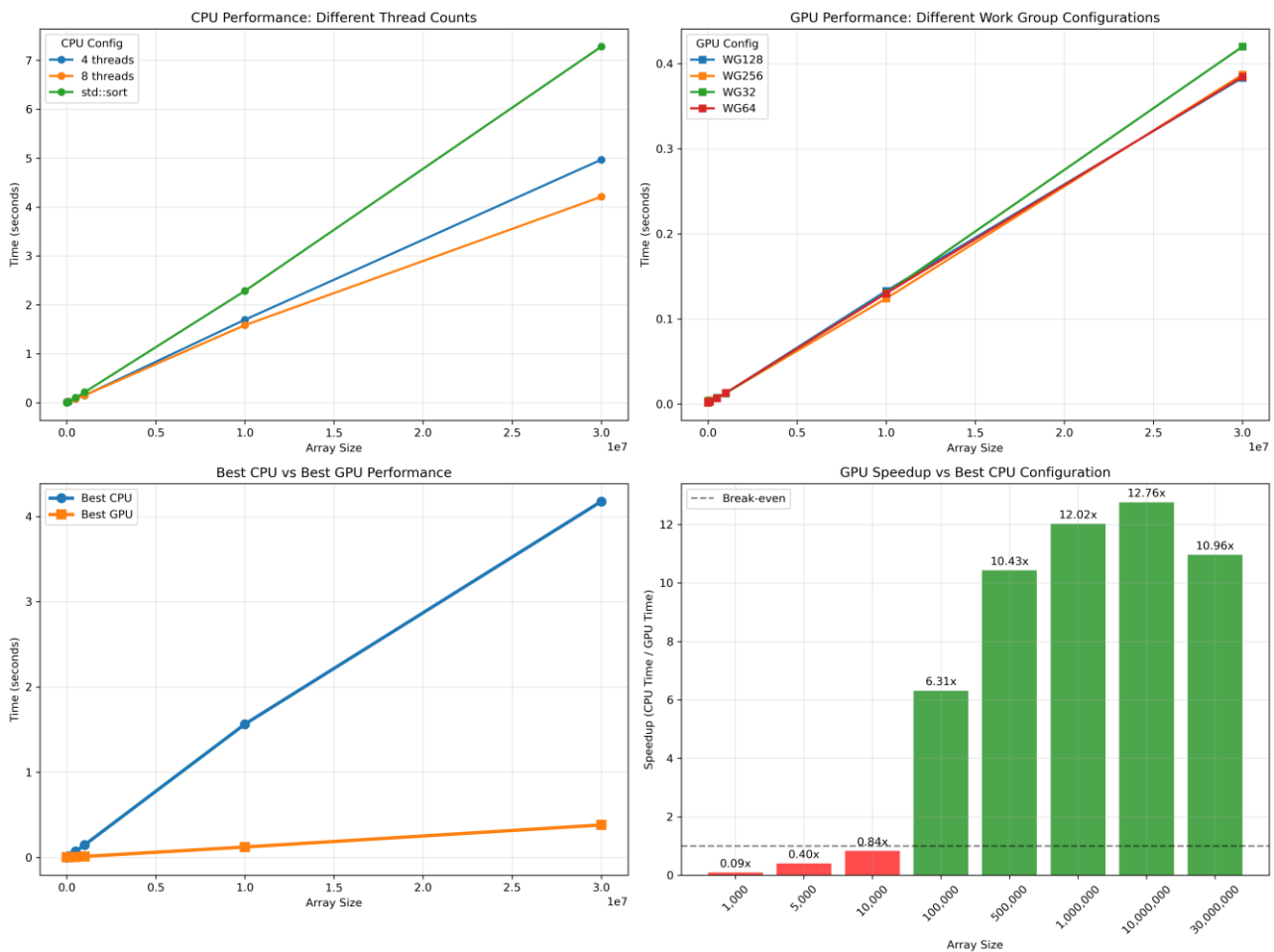


Рисунок 2 – графики сортировки слиянием

Вывод

В ходе работы успешно реализованы и протестированы алгоритмы блочного умножения матриц и сортировки слиянием с использованием технологий GPU и многопоточного CPU. Реализации демонстрируют высокую корректность вычислений на различных типах входных данных. Оптимизированные версии алгоритмов эффективно используют параллельные возможности современного оборудования. GPU реализация умножения матриц показывает значительное ускорение благодаря использованию локальной памяти и оптимизированным вычислительным ядрам. Алгоритм сортировки слиянием на GPU доказал свою эффективность для больших объемов данных. Полученные результаты подтверждают перспективность использования GPU для вычислений с регулярной структурой данных.