ИНФОРМАЦИОННЫЕ ТЕХНОЛОГИИ И ТЕЛЕКОММУНИКАЦИИ
INFORMATION TECHNOLOGY AND TELECOMMUNICATIONS

УДК 004.021, 004.42

DOI: 10.21822/2073-6185-2023-50-4-59-74 Оригинальная статья /Original article

Исследование эффективности методов оптимизации программ для параллельных вычислительных систем с GPU

А.Ю. Безрученко, В.А. Егунов
Волгоградский государственный технический университет,
400005, г. Волгоград, проспект им. Ленина, 28, Россия

Резюме. Цель. В работе определена актуальность задачи повышения эффективности программного обеспечения, под которой в данном случае понимается сокращение времени работы проектируемого программного обеспечения в процессе решения вычислительно-сложных задач. **Метод.** В качестве примера подобной задачи используется выполнение сингулярного разложения методом Якоби. Данная задача находит свое применение в различных областях от обработки сигналов и изображений до систем искусственного интеллекта. В качестве целевой вычислительной архитектуры выбраны параллельные вычислительные системы, оснащенные GPU. В работе рассматриваются методы повышения эффективности программного обеспечения для целевых вычислительных архитектур с использованием CUDA. **Результат.** Описаны существующие аналитические модели оценки эффективности компьютерных программ. Рассматривается влияние различных оптимизаций, таких как оптимизация пересылок данных, использования системы unified memory, количества потоков, паттерна доступа к памяти и ряда других на эффективность получаемого программного обеспечения. Описывается процесс оптимизации программной реализации сингулярного разложения, приводятся результаты вычислительных экспериментов. **Вывод.** При увеличении числа потоков производительность может вырасти больше, чем количество потоков. Влияние паттерна доступа к памяти: при оптимальной последовательности доступа к памяти производительность заметно повышается. Настройка доли памяти, используемой для L1 кеша и для shared memory не оказывает существенного влияния на производительность.

Ключевые слова: эффективность программ, оценка эффективности, производительность, многопоточность, SVD, CUDA, GPGPU, NVidia, unified memory

Для цитирования: А.Ю. Безрученко, В.А. Егунов. Исследование эффективности методов оптимизации программ для параллельных вычислительных систем с GPU. Вестник Дагестанского государственного технического университета. Технические науки. 2023; 50(4):59-74. DOI:10.21822/2073-6185-2023-50-4-59-74

Investigation of the Effectiveness of Programs Optimization Methods for Parallel Computing Systems with GPU

A.Y. Bezruchenko, V.A. Egunov
Volgograd State Technical University,
28 Lenin Ave., Volgograd 400005, Russia

Abstract. Objective. The paper defines the relevance of the task of increasing the efficiency of software, which in this case is understood as reducing the operating time of the designed software in the process of solving computationally complex problems. **Method.** As an example of such a task, the implementation of the singular value decomposition by the Jacobi method is used. This task finds its application in various fields from signal and image processing to artificial intelligence systems. Parallel computing systems equipped with GPU are chosen as the target computing architecture. The paper discusses methods for improving the efficiency of software for target computing

architectures using CUDA. **Result.** The existing analytical models for evaluating the effectiveness of computer programs are described. The influence of various optimizations, such as optimization of data transfers, use of the unified memory system, the number of threads, memory access patterns, and a number of others on the efficiency of the resulting software is considered. The process of optimizing the SVD implementation program is described, the results of computational experiments are presented. **Conclusion.** As the number of threads increases, performance may increase more than the number of threads. Impact of memory access pattern: When the memory access sequence is optimal, performance improves noticeably. Adjusting the share of memory used for L1 cache and shared memory does not have a significant impact on performance

Keywords: program efficiency, efficiency evaluation, performance, multithreading, SVD, CUDA, GPGPU, NVidia, unified memory

For citation: A.Y. Bezruchenko, V.A. Egunov. Investigation of the Effectiveness of Programs Optimization Methods for Parallel Computing Systems with GPU. Herald of Daghestan State Technical University. Technical Sciences. 2023; 50(4):59-74. DOI:10.21822/2073-6185-2023-50-4-59-74

Введение. С ростом объёмов данных и сложности вычислений оптимизация программ для параллельных систем становится всё более важной задачей. Параллельные системы позволяют эффективно использовать ресурсы вычислительных устройств, обрабатывать большие объемы данных и сокращать время вычислений. Сингулярное разложение (Singular Value Decomposition, SVD) является дорогостоящим в вычислительном отношении преобразованием линейной алгебры, которое находит свое применение в различных областях: от сжатия данных до решения систем уравнений [1]. SVD часто используется при обработке сигналов и изображений, в различных областях аналитики больших данных: анализ временных рядов [2], тензорная декомпозиция [3]. Его можно использовать в методе главных компонент [4], используемом для уменьшения размерности данных, он используется в статистике [5], во многих других отраслях. В последние годы его начали активно использовать в машинном обучении [6].

В данной работе рассматриваются методы оптимизации программ для графических сопроцессоров (Graphics Processing Unit, GPU) на примере выполнения SVD с использованием метода Якоби. Исследуется влияние различных способов оптимизации, таких как использование unified memory, настройка количества потоков и паттерна доступа к памяти, с целью повышения эффективности вычислений.

Постановка задачи. Задача повышения эффективности программного обеспечения является актуальной задачей, привлекающей внимание исследователей. Для достижения данной цели используются различные подходы, которые достаточно условно можно разделить на теоретические и практические методы. К практическим методам можно отнести использование различных отладчиков, профилировщиков и других инструментальных средств, позволяющих улучшать характеристики получаемого программного обеспечения. С другой стороны, разработчики используют различные технологии программирования, оптимизированные алгоритмы и другие практические методы повышения эффективности компьютерных программ. Важную роль в данном случае играют различные математические модели исполнения программного обеспечения, которые позволяют выявить узкие места и повысить итоговую производительность вычислительной системы в целом. Данные модели разрабатываются для разных классов вычислительных систем, далее рассмотрим некоторые подобные модели.

Методы исследования. В работе [7] представляется Roofline Model — интуитивная визуальная модель для оценки производительности программного обеспечения на конкретной вычислительной системе. В соответствии с данной моделью строится график, где по оси X откладывается “Operational Intensity (Интенсивность операций) (FLOP/Byte)”, по оси Y – “Attainable (Достижимые) GFLOP/s”. Результат представляется линией (roof), которая сначала с ростом интенсивности операций линейно растёт, пока не достигнет максимума и не останется на нём. Пример такого графика показан на рис.1.

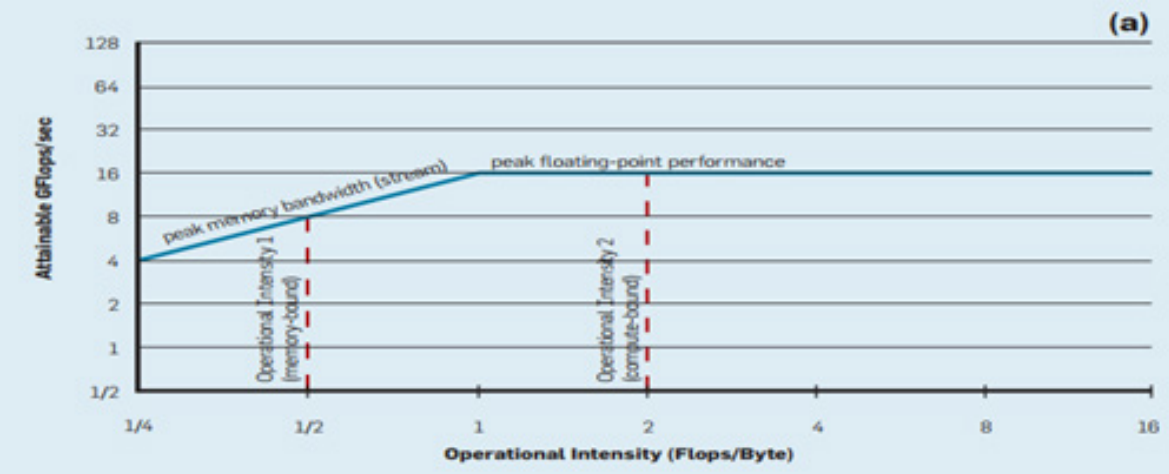

Рис. 1. Пример roofline model
Fig. 1. Example roofline model

Формула вычисления производительности при этом выглядит следующим образом:

$$attainable\ GFlops = \min \begin{Bmatrix} \max\_flops \\ \max\_throughput \times op\_intensity \end{Bmatrix}, \quad (1)$$

где max_flops — максимальная производительность при работе с числами;
max_throughput — максимальная пропускная способность памяти;
op_intensity — интенсивность операций.

Данный метод не предназначен для определения времени выполнения программы, хотя он и используется для оценки верхней планки производительности вычислительной системы. Он довольно неточен, но при этом достаточно прост в использовании и позволяет оценить, чем именно ограничивается скорость выполнения программы: пропускной способностью памяти или вычислительной мощностью процессоров.

Хотя в данной работе roofline model не используется, приведем советы по повышению производительности в многоядерных системах (на примере Opteron X2), рассмотренные в той же статье [7]: увеличение параллелизма уровня инструкций, применение SIMD; балансировка операций с плавающей запятой (чтобы уравнять количество сложений и умножений); переписывание циклов на использование шага, равного количеству ядер; использование локальной для ядра память; использование prefetching;

Конкретный вид графика зависит от используемого оборудования. В статье [8] проведен анализ двух видеокарт: AMD HD5850 и NVIDIA C2050,в статье [9] такой же анализ проводится для Xeon Phi и OpenMP. Получить данный график для конкретной вычислительной системы можно с помощью утилиты для бенчмарков Likwid [10]. Хотя изначально данная утилита разрабатывалась под CPU, она была позже адаптирована для работы с GPU от Nvidia [11]. Кроме того, существует также профилировщик Nsight от Nvidia, который также может строить такой график.

Работа [12] представляет модель с упором на уменьшение эффектов расхождения ветвей (branch divergence). Архитектура GPU от NVIDIA содержит несколько потоковых мультипроцессоров (Streaming multiprocessor, SM), каждый из которых состоит из многих потоковых процессоров (Streaming processor, SP). SM координируют выполнение программы на большом количестве потоков, разделяемых на группы по 32 (для Nvidia) потока, называемые варпами (warp).

Потоки внутри варпа начинают исполнение с одного и того же адреса; у каждого потока есть свой счётчик команд и своё состояние регистров. Если все потоки исполняют одни и те же инструкции (имеют одно и то же значение счётчика команд), то их можно исполнять параллельно. Однако в разных потоках могут выполняться разные ветки условных выражений. Когда пути исполнения в разных потоках внутри варпа расходятся, потоки, следующие по разным путям исполнения, выполняются последовательно, отдельно друг

от друга, и затем снова объединяются. В худшем случае каждый поток в варпе выполняется в своей ветке условия; в таком случае в каждый отдельный момент времени исполняется только один поток, и параллелизм на самом деле отсутствует. Данный эффект может заметно влиять на производительность приложений GPU [12].

Различные способы уменьшения расхождения исследовались в статьях [13] и [14], где предлагается динамически разделять варпы на уровне устройства, а также в статье [15], где предлагалось переназначение памяти на уровне программного обеспечения.

CUDA (Compute Unified Device Architecture) представляет собой программно-аппаратную архитектуру параллельных вычислений, которая позволяет существенно увеличить производительность вычислительной системы благодаря использованию графических процессоров фирмы Nvidia. Описание CUDA доступно в официальных руководствах [16]. В данной архитектуре GPU состоит из массива потоковых мультипроцессоров (Streaming Multiprocessor, SM). В процессе выполнения программа разбивается на блоки потоков, каждый блок исполняется независимо друг от друга. Поэтому GPU с большим числом SM в общем случае будет исполнять программы быстрее, чем GPU с меньшим числом SM.

CUDA позволяет задавать специальные функции, ядра (kernel), которые исполняются параллельно на нескольких CUDA потоках. Количество потоков в блоке ограничено (не более 1024), однако ядро может выполняться на нескольких блоках. Предполагается, что блоки выполняются независимо друг от друга, в любом порядке. Потоки внутри блока могут кооперироваться друг с другом, делясь данными через общую память.

На GPU с Compute Capability 9.0 есть дополнительный опциональный уровень иерархии — кластеры блоков потоков. Так же, как потоки в блоке гарантированно выполняются на одном мультипроцессоре, блоки в кластере выполняются на одном обрабатывающем кластере (GPU Processing Cluster, GPC). Также есть функция для синхронизации блоков в кластере. Модель памяти CUDA представлена на рис. 2.

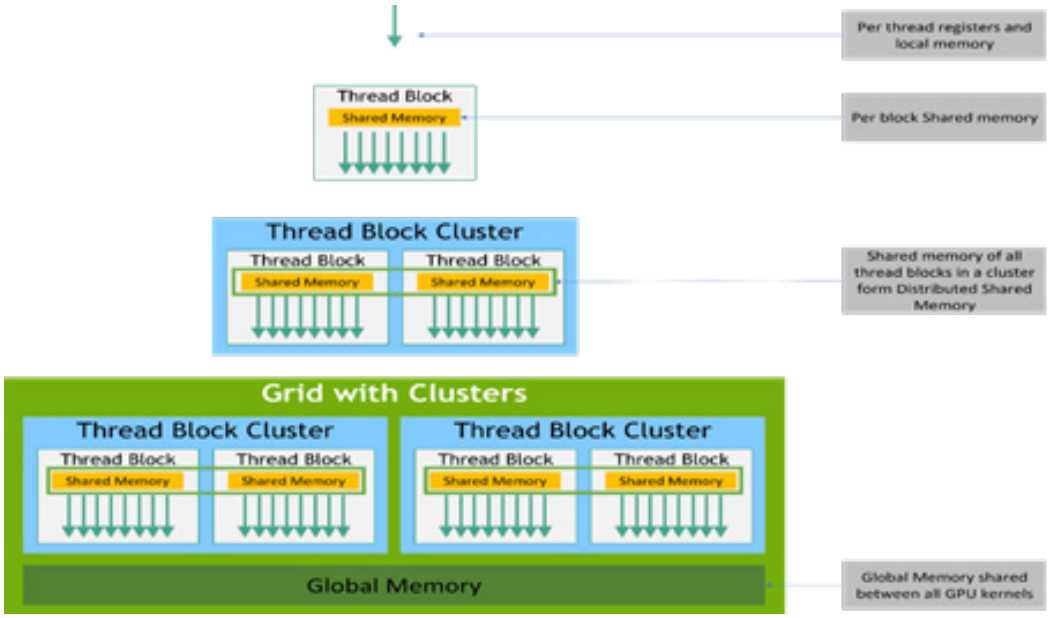

Рис. 2. Модель памяти CUDA
Fig. 2. CUDA memory model

Каждый поток имеет свою локальную память. При этом у каждого блока есть общая (shared) память, доступная всем потокам в блоке и имеющая такое же время жизни, что и время жизни блока. Доступ к shared memory своего блока есть у всех потоков. Кроме того, есть два пространства памяти только для чтения: константная и текстурная память. Глобальная, константная и текстурная памяти сохраняют свои значения между запусками одного и того же приложения.

Начиная с Compute Capability 9.0, появилась распределённая общая память (distributed shared memory). Эта система позволяет блокам внутри кластера обращаться к shared memory других блоков того же кластера. Поддерживается чтение, запись и некоторые атомарные операции над shared memory других блоков, а также синхронизация блоков внутри кластера.

Ранее предполагалось, что устройство и хост имеют разную память, при этом попытка обратиться к неправильному виду памяти вызывала ошибку. Начиная с Compute Capability 3 появилась объединённая память (unified memory). Она доступна как из CPU, так и из GPU по одному и тому же адресу. Данной памятью можно воспользоваться либо выделив её с помощью функции cudaMallocManaged(), либо с помощью глобальных переменных с модификаторами __device__ и __managed__.

Начиная с версии CUDA 8.0 и Compute Capability 6.0, unified memory используется по умолчанию; память, выделенная с помощью malloc или new, доступна как на GPU, так на CPU, так что отпадает необходимость вручную пересылать данные между устройствами. Более подробно о производительности unified memory в CUDA можно почитать в статьях [17] (общий обзор unified memory) и [18] (обзор приёмов по улучшению производительности).

Данные в unified memory пересылаются на устройство, использующее их. Например, если код, исполняемый на GPU, обращается к участку памяти, который в текущий момент находится в оперативной памяти компьютера, то данные пересылается на GPU, и программа на GPU может работать с ними так, будто память расположена на самом GPU. Когерентность в данном случае ставится выше производительности: рассинхронизации памяти нет, т.к. при запросе памяти, ранее использовавшейся другим вычислителем (где под вычислителем подразумевается, например, CPU или GPU), она пересылается на необходимый вычислитель. При этом частые пересылки данных могут вредить производительности. Вне зависимости от физического расположения памяти на текущий момент, указатели на неё остаются валидными и могут быть использованы на каждом вычислителе. На GPU с Compute Capability менее 6.x при запросе памяти пересылается вся память целиком, в 6.x введён механизм page fault, так что память пересылается страницами по запросу. Начиная с Compute Capability 6.0, память при вызове cudaMallocManaged() физически выделяется не сразу, а при первом обращении к выделенному массиву. Также появилась возможность запросить больше виртуальной памяти, чем есть физической на устройстве. До версии 6.0 такой возможности не было, и вся запрошенная память сразу же выделяется, либо выдаётся ошибка о недостаточном количестве памяти.

Стоит отметить, что многие возможности управляемой памяти, добавленные с Compute Capability 6, не работают в Windows. Когда варп выполняет запрос к памяти, его исполнение приостанавливается и начинает исполняться другой варп до тех пор, пока память не будет готова. Если несколько потоков в одном варпе запрашивают память в последовательно расположенных участках памяти, то эти запросы могут быть объединены в одну или несколько транзакций [16], ускоряя доступ к памяти. Из-за этого в циклах, обрабатывающих массив элементов, рекомендуется шаг цикла сделать равным количеству используемых потоков, а начальное значение — номеру потока; так потоки будут запрашивать последовательно расположенные адреса памяти [19]. Shared память в устройствах с Compute Capability 7.x может делить одну и ту же физическую память с L1 кешем [16]. Это же касается и Compute Capability 8.x и 9.x [16]. Если shared память не используется, можно использовать больше памяти под L1 кеш.

Обсуждение результатов. В качестве базовой задачи было выбрано выполнение SVD методом Якоби. На первом этапе была реализована версия для CPU. Профилирование программы показало, что перемножение матриц при повороте занимает 80-85% всего времени выполнения, поэтому в первую очередь необходимо оптимизировать выполнение данной операции. На основе версии для CPU была реализована версия программы для GPU, в которой перемножение матриц при повороте вынесено на GPU.

Сравним время исполнения реализаций для CPU и для GPU. Для сравнения

использовались: процессор — Ryzen 5 2600, видеокарта — Nvidia GTX 1650 (архитектура Turing, compute capability 7.5), ОС — Windows 10. Замер производился на квадратных матрицах, размеры используемых матриц приведены в табл.1. Вычисления производились несколько раз, в табл.1 приведено среднее значение времени исполнения. Для замера времени использовалась функция std::chrono::steady_clock::now(), которая в использованной Visual Studio 2019 является обёрткой над QueryPerformanceCounter, которая способна обеспечить точность до микросекунды [20]; измерения осуществлялись с точностью до миллисекунды. Сравнение времени исполнения реализации на CPU, на GPU, а также реализации этого же алгоритма в cusolver, показано в табл. 1.

Таблица 1. Сравнение производительности в процессе выполнения SVD
Table 1. Comparison of performance during SVD execution

Размер матриц Matrix size	Время выполнения на CPU, с Lead time	Время выполнения на GPU, с Lead time	Время выполнения Cusolver, с Lead time
25	0.002	0.456	0.001
50	0.021	1.31	0.001
75	0.081	3.44	0.001
100	0.201	7.599	0.001
125	0.439	11.796	0.001

Как видно, реализованная первоначальная версия работает необычно медленно. Разберёмся в причинах. Проанализируем, каким образом выполняются различные части алгоритма. В листингах комментарием «CPU» отмечены участки, выполняющиеся на CPU (и при этом запрашивающие память, физически в данный момент расположенную на GPU), комментарием «GPU» — выполняющиеся на GPU, «GPU/CPU» — смешанные.

В листинге 1 показана функция jacobi_svd. Она лишь выделяет память (alloc_matrix_gpu внутри себя вызывает cudaMallocManaged()) и копирует её из изначального массива, вся логика расположена в функции jacobi_svd_gpu. Стоит отметить, что cudaMallocManaged() выделяет память, которая будет управляться системой unified memory. К указателю на эту память можно обращаться как с хоста, так и с устройства, система сама пересылает данные туда, где они используются, однако использование этой системы может в некоторых случаях замедлять программу.

Листинг 1. Функция jacobi_svd

```
void jacobi_svd(double* A, int m, double* s, double* u, double* vt)
{
    double* cuda_A = alloc_matrix_gpu(m);
    ……
    double* buff5 = alloc_matrix_gpu(m);
    cudaMemcpy(cuda_A, A, m * m * sizeof(double), cudaMemcpyDefault);
    jacobi_svd_gpu(cuda_A, m, cuda_s, cuda_u, cuda_v, buff1, buff2, buff3, buff4, buff5);
    cudaDeviceSynchronize();
    cudaMemcpy(s, cuda_s, m * sizeof(double), cudaMemcpyDefault);
    cudaMemcpy(u, cuda_u, m * m * sizeof(double), cudaMemcpyDefault);
    cudaMemcpy(vt, cuda_v, m * m * sizeof(double), cudaMemcpyDefault);
    cudaFree(buff1);
    ……
    cudaFree(cuda_v);
}
```

В листинге 2 показана функция jacobi_svd_gpu.

Листинг 2. Анализ функции jacobi_svd_gpu

```
void jacobi_svd_gpu(double* A, int m, double* s, double* u, double* vt, double* buff1,
double* buff2, double* buff3, double* buff4, double* buff5)
{    fill_as_identity(vt, m); // CPU
    fill_as_identity(u, m); // CPU
    while (not_converged(A, m)) // CPU
    {
        for (int j = 0; j < m - 1; j++)
```

```
            for (int k = j + 1; k < m; k++)
                jacobi_rotation(A, m, j, k, buff1, buff2, buff3, buff4, buff5, u, vt); //
CPU/GPU
    }
    for (int i = 0; i < m; i++)
    {
        s[i] = getSum(A, m, i); // CPU
    }
}
```

В ней можно выделить следующие действия: формирование единичных матриц u и vt; вызов jacobi_rotation в цикле; формирование сумм элементов столбцов полученной матрицы. Инициализация u и vt происходит в функции fill_as_identity (листинг 3).

В ней происходит обращение к матрице из функции хоста. Проверка условия останова (листинг 4) и сложение элементов в столбце (листинг 5) также выполняются на хосте и тоже обращаются к матрице. Позже, в функции jacobi_rotation, к этой матрице будет обращаться код, исполняемый на GPU, и потребуется пересылка данных.

Листинг 3. Функция fill_as_identity

```
void fill_as_identity(double* matrix, int m)
{
    for (int i = 0; i < m; i++)
        for (int j = 0; j < m; j++)
            matrix[IDX2C(i, j, m)] = (i == j ? 1 : 0);
}
```

Листинг 4. Функция not_converged

```
bool not_converged(double* A, int m)
{
    for (int i = 0; i < m; i++)
        for (int j = 0; j < m; j++)
            if (i != j && fabs(A[IDX2C(i, j, m)]) > offdiagonal_threshold)
                return true;
    return false;
}
```

Листинг 5. Функция getSum

```
double getSum(double* matrix, int m, int column)
{
    double sum = 0;
    for (int i = 0; i < m; i++)
    {
        sum += matrix[IDX2C(column, i, m)];
    }
    return sum;
}
```

Проанализируем функцию jacobi_rotation (листинг 6). В ней выполняются следующие действия: проверяется, чтобы текущий элемент был больше порогового значения для останова формируется матрицу поворотов R и транспонируем ее RT вычисляетс A = R × A × RT, V = R × V, U = U × RT (здесь используется GPU)

Листинг 6. Анализ функции jacobi_rotation

```
void jacobi_rotation(double* A, int m, int j, int k, double* buff1, double* buff2,
double* buff3, double* buff_r, double* buff_rt, double*& U, double*& V)
{
    if (fabs(A[IDX2C(j, k, m)]) > offdiagonal_threshold) // CPU
    {
        double tau = (A[IDX2C(j, j, m)] - A[IDX2C(k, k, m)]) / (2 * A[IDX2C(j, k,
m)]); // CPU
        double t = sign(tau) / (abs(tau) + sqrt(1 + tau * tau));
        double c = 1 / sqrt(1 + t * t);
        double s = c * t;
        fill_as_identity(buff_r, m); // CPU
```

```
            buff_r[IDX2C(j, j, m)] = get_R(j, j, j, k, c, s); // CPU
            buff_r[IDX2C(j, k, m)] = get_R(j, k, j, k, c, s); // CPU
            buff_r[IDX2C(k, j, m)] = get_R(k, j, j, k, c, s); // CPU
            buff_r[IDX2C(k, k, m)] = get_R(k, k, j, k, c, s); // CPU
            fill_as_identity(buff_rt, m); // CPU
            buff_rt[IDX2C(j, j, m)] = get_RT(j, j, j, k, c, s); // CPU
            buff_rt[IDX2C(j, k, m)] = get_RT(j, k, j, k, c, s); // CPU
            buff_rt[IDX2C(k, j, m)] = get_RT(k, j, j, k, c, s); // CPU
            buff_rt[IDX2C(k, k, m)] = get_RT(k, k, j, k, c, s); // CPU
            // GPU
            multiply_matrices_only_jk(buff_r, A, buff1, m, j, k, false);
            // GPU
            multiply_matrices_only_jk(buff_r, V, buff2, m, j, k, false);
            // GPU
            multiply_matrices_only_jk(U, buff_rt, buff3, m, j, k, true);
            cudaDeviceSynchronize();
            // GPU
            multiply_matrices_only_jk(buff1, buff_rt, A, m, j, k, true);
            // GPU
            cudaMemcpy(V, buff2, m * m * sizeof(double), cudaMemcpyDefault);
            // GPU
            cudaMemcpy(U, buff3, m * m * sizeof(double), cudaMemcpyDefault);
            cudaDeviceSynchronize();
        }
    }
```

Функции формирования матрицы поворотов и проверки условия останова выполняются на хосте и обращаются к памяти устройства.

При этом перемножение матриц поворота (multiply_matrices_only_jk, листинг7) уже запускает ядро (multiply_matrices_only_jk_gpu, листинг 8), которое обращается к функциям на GPU (multiply_matrices_only_jk_gpu_row и multiply_matrices_only_jk_gpu_column, листинг 9). Выбор начального индекса и шага в цикле будет объяснён позднее, когда будет рассматриваться влияние паттерна доступа к памяти на производительность; также он объяснён в [21]. Кроме того, стоит отметить группировку потоков по операциям (см. листинг 8): потоки в блоке распределяются на 4 подблока так, чтобы номера потоков в каждом подблоке шли последовательно. Если доступно больше одного варпа, то желательно, чтобы все потоки внутри варпа были на одной и той же ветви условного выражения. Если это не так, то возникает проблема расхождения ветвей [12].

Листинг 7. Функция multiply_matrices_only_jk

```
void multiply_matrices_only_jk(double* first, double* second, double* res, int size,
int j, int k, bool keepFirst)
{
    if (keepFirst)
    cudaMemcpy(res, first, size * size * sizeof(double), cudaMemcpyDefault);
    else
    cudaMemcpy(res, second, size * size * sizeof(double), cudaMemcpyDefault);
    const int num_warps = 1;
    const int num_threads = num_warps * 32;
    multiply_matrices_only_jk_gpu << <1, num_threads >> > (first, second, res, size,
j, k);

}
```

Листинг 8. Функция multiply_matrices_only_jk_gpu

```
__global__ void multiply_matrices_only_jk_gpu(double* first, double* second, double*
res, int size, int j, int k)
{
    int subblockSize = blockDim.x / 4;
    int operation = threadIdx.x / subblockSize;
```

```
    int start_index = threadIdx.x % subblockSize;
    switch (operation)
    {
    case 0:
        multiply_matrices_only_jk_gpu_row(first, second, res, size, j, start_index,
subblockSize);
        break;
    case 1:
        multiply_matrices_only_jk_gpu_row(first, second, res, size, k, start_index,
subblockSize);
        break;
    case 2:
        multiply_matrices_only_jk_gpu_column(first, second, res, size, j, k, j, start_
index, subblockSize);
        break;
    case 3:
        multiply_matrices_only_jk_gpu_column(first, second, res, size, j, k, k, start_
index, subblockSize);
        break;
    }
}
```

Листинг 9. Функции умножения строки и столбца

```
__device__ void multiply_matrices_only_jk_gpu_row(double* first, double* second,
double* res, int size, int row, int start_index, int stride)
{
    for (int b_ = start_index; b_ < size; b_ += stride)
    {
        double sum = 0;
        for (int c_ = 0; c_ < size; c_++)
            sum += first[IDX2C(row, c_, size)] * second[IDX2C(c_, b_, size)];
        res[IDX2C(row, b_, size)] = sum;
    }
}
__device__ void multiply_matrices_only_jk_gpu_column(double* first, double* second,
double* res, int size, int ignore_row_1, int ignore_row_2,
    int column, int start_index, int stride)
{
    for (int a_ = start_index; a_ < size; a_ += stride)
    {
        if (a_ != ignore_row_1 && a_ != ignore_row_2)
        {
            double sum = 0;
            for (int c_ = 0; c_ < size; c_++)
                sum += first[IDX2C(a_, c_, size)] * second[IDX2C(c_, column, size)];
            res[IDX2C(a_, column, size)] = sum;
        }
    }
}
```

Как видно в листинге 6, участки кода, обращающиеся к памяти с хоста, находятся в одной функции с участками кода, обращающимися к памяти с устройства. Как видно в листинге 2, эта функция вызывается в цикле множество раз, и на каждой итерации необходимо переслать данные на устройство и обратно. Это оказывает сильное негативное влияние на производительность.

Вынесем всю работу с матрицами на GPU. Функция jacobi_svd остаётся без изменений. Функция jacobi_svd_gpu также почти не изменилась (листинг 10), единственное отличие — вынос получения сумм по столбцам в функцию; однако при этом из-за изменений вызываемых функций все действия выполняются на GPU, избавляя от необходимости пересылать данные в оперативную память.

Листинг 10. Изменённая функция jacobi_svd_gpu

```
void jacobi_svd_gpu(double* A, int m, double* s, double* u, double* vt, double* buff1,
double* buff2, double* buff3, double* buff4, double* buff5)
{
    fill_as_identity(vt, m); // GPU
    fill_as_identity(u, m); // GPU
    while (!is_converged(A, m)) // GPU
    {
        for (int j = 0; j < m - 1; j++)
            for (int k = j + 1; k < m; k++)
                jacobi_rotation(A, m, j, k, buff1, buff2, buff3, buff4, buff5, u, vt); //
GPU
    }
    dim3 block_dim(num_threads, 1, 1);
    dim3 grid_dim(num_blocks, 1, 1);
    getSums << <grid_dim, block_dim >> > (A, s, m); // GPU
}
```

В листинге 11 показана изменённая функция формирования единичной матрицы. Как видно, теперь изменение памяти, выделенной на устройстве, происходит в ядре, выполняемом на самом устройстве. Аналогично работа с памятью устройства была перенесена на устройство и в проверке условия останова (листинг 12), на хост пересылается только результат. Также вынесение вычислений на GPU было проведено в функции getSums; оно достаточно простое, поэтому не будем его здесь рассматривать.

Листинг 11. Изменённая функция fill_as_identity

```
void fill_as_identity(double* matrix, int m)
{
    dim3 block_dim(num_threads, 1, 1);
    dim3 grid_dim(num_blocks, 1, 1);
    fill_as_identity_gpu << <grid_dim, block_dim >> > (matrix, m);
}
```

Листинг 12. Изменённая функция is_converged

```
__global__ void is_converged_gpu(double* A, int m, bool* is_converged)
{
    for (int i = blockIdx.x * blockDim.x + threadIdx.x; i < m; i += blockDim.x *
gridDim.x)
    {
        if (fabs(A[i]) > offdiagonal_threshold && (i % m != i / m))
        {
            *is_converged = false;
            return;
        }
    }
}
     bool is_converged(double* A, int m)
{
    bool is_converged = true;
    bool* is_converged_d;
    cudaMalloc(&is_converged_d, sizeof(bool));
    cudaMemcpy(is_converged_d, &is_converged, sizeof(is_converged),
cudaMemcpyDefault);
    is_converged_gpu << <1, num_threads >> > (A, m, is_converged_d);
    cudaMemcpy(&is_converged, is_converged_d, sizeof(is_converged),
cudaMemcpyDefault);
    cudaFree(is_converged_d);
    return is_converged;
}
```

Проанализируем функцию jacobi_rotation после изменения (листинг 13). Пересылка

элемента в память GPU сделана явной (добавлена функция cudaMemcpy).

Листинг 13. Изменённая функция jacobi_rotation

```
void jacobi_rotation(double* A, int m, int j, int k,
    double* buff1, double* buff2, double* buff3, double* buff_r, double* buff_rt, double*&
U, double*& V)
{
    double a_jk;
    cudaMemcpy(&a_jk, &A[IDX2C(j, k, m)], sizeof(double), cudaMemcpyDefault); // GPU
    if (fabs(a_jk) > offdiagonal_threshold_cpu)
    {
        double t = get_t(A, m, j, k);  // GPU
        double c = 1 / sqrt(1 + t * t);
        double s = c * t;
        fill_as_r(buff_r, m, j, k, c, s);  // GPU
        fill_as_rt(buff_rt, m, j, k, c, s);  // GPU
        // GPU
        multiply_matrices_only_jk(buff_r, A, buff1, m, j, k, false);
        // GPU
        multiply_matrices_only_jk(buff_r, V, buff2, m, j, k, false);
        // GPU
        multiply_matrices_only_jk(U, buff_rt, buff3, m, j, k, true);
        cudaDeviceSynchronize();
        // GPU
        multiply_matrices_only_jk(buff1, buff_rt, A, m, j, k, true);
        // GPU
        cudaMemcpy(V, buff2, m * m * sizeof(double), cudaMemcpyDefault);
        // GPU
        cudaMemcpy(U, buff3, m * m * sizeof(double), cudaMemcpyDefault);
        cudaDeviceSynchronize();
    }
}
```

Вычисление тангенса угла поворота вынесено в отдельную функцию. Для вычислений необходимы некоторые значения из памяти устройства, их также копируем явно (листинг 14).

Листинг 14. Функция get_t

```
double get_t(double* A, int m, int j, int k)
{
    double jj, kk, jk;
    cudaMemcpy(&jj, &A[IDX2C(j, j, m)], sizeof(double), cudaMemcpyDefault);
    cudaMemcpy(&kk, &A[IDX2C(k, k, m)], sizeof(double), cudaMemcpyDefault);
    cudaMemcpy(&jk, &A[IDX2C(j, k, m)], sizeof(double), cudaMemcpyDefault);
    double tau = (jj - kk) / (2 * jk);
    double t = sign(tau) / (abs(tau) + sqrt(1 + tau * tau));
    return t;
}
```

Аналогично было сделано заполнение матриц поворота (листинг 15).

Листинг 15. Изменённое заполнение матриц поворота

```
void fill_as_r(double* matrix, int m, int j, int k, double c, double s)
{
    fill_as_identity(matrix, m);
    double jj = get_R(j, j, j, k, c, s);
    double jk = get_R(j, k, j, k, c, s);
    double kj = get_R(k, j, j, k, c, s);
    double kk = get_R(k, k, j, k, c, s);
 cudaMemcpy(&matrix[IDX2C(j, j, m)], &jj, sizeof(double), cudaMemcpyDefault);
 cudaMemcpy(&matrix[IDX2C(j, k, m)], &jk, sizeof(double), cudaMemcpyDefault);
 cudaMemcpy(&matrix[IDX2C(k, j, m)], &kj, sizeof(double), cudaMemcpyDefault);
 cudaMemcpy(&matrix[IDX2C(k, k, m)], &kk, sizeof(double), cudaMemcpyDefault);
}
```

```
void fill_as_rt(double* matrix, int m, int j, int k, double c, double s)
{
    fill_as_identity(matrix, m);
    double jj = get_RT(j, j, j, k, c, s);
    double jk = get_RT(j, k, j, k, c, s);
    double kj = get_RT(k, j, j, k, c, s);
    double kk = get_RT(k, k, j, k, c, s);
 cudaMemcpy(&matrix[IDX2C(j, j, m)], &jj, sizeof(double), cudaMemcpyDefault);
 cudaMemcpy(&matrix[IDX2C(j, k, m)], &jk, sizeof(double), cudaMemcpyDefault);
 cudaMemcpy(&matrix[IDX2C(k, j, m)], &kj, sizeof(double), cudaMemcpyDefault);
 cudaMemcpy(&matrix[IDX2C(k, k, m)], &kk, sizeof(double), cudaMemcpyDefault);
}
```

Сравним производительность до и после этого изменения, а также сравним с производительностью программы, написанной без использования unified memory (эта версия будет отличаться тем, что при выделении памяти будет использоваться функция cudaMalloc(), а не cudaMallocManaged()). Сравнение приведено в табл. 2. Все замеры производились на 32 потоках.

Таблица 2. Сравнение времени исполнения на GPU до изменений и после
Table 2. Comparison of execution time on the GPU before and after changes

Размер Size	До Before	После, unified	После, не unified
25	0.478	0.371	0.285
50	1.319	1.086	0.819
75	3.385	2.241	1.837
100	6.687	4.038	3.468

Как видно, уменьшение пересылок памяти заметно ускорило программу; отказ от unified memory также заметно помог. Можно сделать вывод, что на GTX 1650 использование unified memory в данном случае только вредит производительности, причём не только напрямую, но и косвенно, позволяя не думать о пересылке данных и, как следствие, поощряя программистов разрабатывать неоптимальный код.

Помимо этого, мы можем повысить производительность, увеличив количество используемых потоков. Хотя количество потоков можно задать произвольным, оно в идеале должно быть кратно 32, потому что потоки выделяются варпами — группами по 32 (стоит отметить, что на GPU AMD или в будущем число потоков в варпе может быть другим).

В табл. 3 и на рис. 3 указано время исполнения при разных размерах матрицы и разном количестве блоков и потоков, замеры времени исполнения осуществлялось аналогично предыдущим опытам. В первом столбце указано количество блоков и количество потоков в каждом блоке, далее в столбцах указаны размеры матрицы.

Как видно из табл. 3, увеличение числа потоков ожидаемо ускоряет выполнение программы; при этом увеличение числа используемых блоков при равном общем числе потоков также немного увеличивает производительность.

Таблица 3. Время выполнения программы при разной конфигурации
Table 3. Program execution time for different configurations

Число блоков × потоков/ Number of blocks × threads	32	64	96	128	160	192	224	256
1 × 32	0.356	1.246	3.015	6.091	10.839	18.045	28.026	41.129
1 × 64	0.319	0.934	1.895	3.282	5.23	8.075	11.8	17.019
2 × 32	0.322	0.929	1.879	3.283	5.200	7.921	11.59	16.816
1 × 128	0.318	0.866	1.633	2.657	3.993	5.692	7.804	10.397
2 × 64	0.314	0.86	1.694	2.626	3.937	5.697	7.651	10.239
4 × 32	0.309	0.853	1.599	2.598	3.891	5.447	7.427	9.875

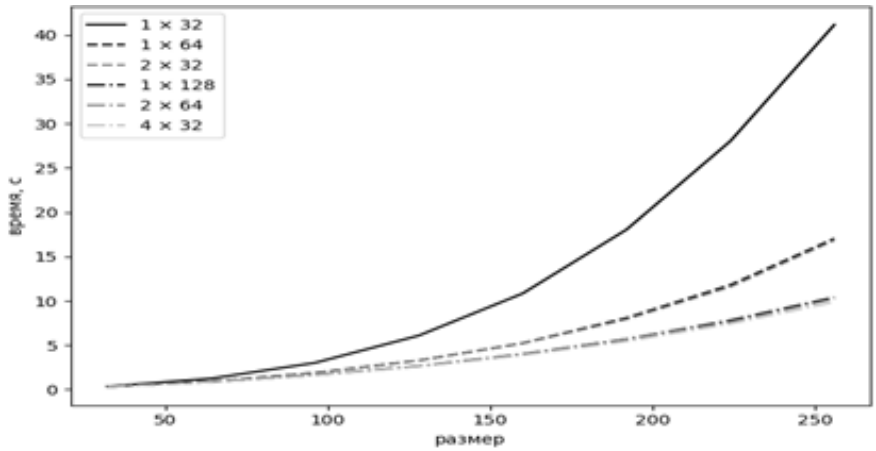

Рис. 3. Зависимость времени исполнения от конфигурации потоков
Fig. 3. Dependence of execution time on thread configuration

Интересной особенностью устройств Nvidia, начиная с Compute Capability 7.x, является то, что shared память в них может делить одну и ту же физическую память с L1 кешем [16].

Это же касается и Compute Capability 8.x и 9.x [16]. Если shared память не используется, можно использовать больше памяти под L1 кеш. Это соотношение задаётся параметром cu daFuncAttributePreferredSharedMemoryCarveout. Значению 0 соответствует максимальному размеру кеша, значению 100 — максимальный размер общей памяти.

Стоит отметить, что установка этого параметра не гарантирует задания соответствующего соотношения. Параметр указывается программистом, но драйвер может его проигнорировать, если сочтёт нужным. Эксперименты показали, что установка различных значений для данного параметра не оказали заметного влияния на производительность.

Ещё одна возможность для оптимизации — паттерн доступа к памяти. Доступ к последовательно расположенным адресам в памяти может объединяться в одну или несколько транзакций в зависимости от размера [16]. Если адреса, запрашиваемые потоками в варпе, не расположены последовательно, то требуется больше транзакций и, следовательно, больше времени уходит на пересылку данных.

Таким образом, оптимальный способ обработки элементов в цикле — взять за шаг цикла количество потоков, а за начальное значение — номер потока. В таком случае на каждой итерации цикла запрашиваемые данные будут располагаться последовательно.

Рис. 4 и 5 иллюстрирует это для случая 2 потоков, обрабатывающих 8 элементов. Элементы, запрашиваемые на указанной итерации цикла, закрашены зелёным цветом. Слева указано время, снизу под элементами указано, какой из потоков их обрабатывает (потоки обозначены как T1 и T2).

Как видно на рис. 4 и 5, при шаге цикла, равном 1, на каждой итерации запрашиваются разрозненные участки памяти. В это время при шаге цикла, равному числу потоков, на каждой итерации запрашиваются смежные участки памяти.

Рис. 4. Доступ к данным при шаге 1
Fig. 4. Accessing data in step 1

Рис. 5. Доступ к данным при шаге, равному количеству потоков
Fig. 5. Data access at a step equal to the number of threads

Проанализируем, каким образом паттерн доступа к памяти (шаг в цикле) влияет на время исполнения (табл. 4) при 256 потоках.

Таблица 4. Сравнение времени исполнения при разном шаге цикла
Table 4. Comparison of execution time at different cycle steps

Размер Size	Шаг цикла 1 Cycle step	Шаг цикла 256 Cycle step
100	1.748±0.015	1.871±0.086
200	6.125±0.022	6.007±0.154
300	13.467±0.133	12.958±0.05
400	28.458±0.136	26.36 ±0.149
500	45.637±0.183	39.18 ±0.126

Как видно, при шаге, равном количеству потоков, программа работает заметно быстрее (на большом количестве элементов).

Вывод. В данной статье были рассмотрены некоторые способы оптимизации программ под GPU. Было экспериментально определено, что использование unified memory, упрощает разработку программ, в проведенных экспериментах ухудшает итоговую производительность вычислительной системы, увеличивая время выполнения программ. Было замечено, что при увеличении числа потоков производительность может вырасти больше, чем количество потоков. Было также проверено влияние паттерна доступа к памяти на время исполнения.

Можно сделать вывод, что при оптимальной последовательности доступа производительность заметно повышается. Также было исследовано влияние настройки доли памяти, используемой для L1 кеша и для shared memory. Было показано, что данный параметр не оказывает существенного слияния на производительность.

Данные выводы были получены для реализации SVD на конкретном оборудовании (GTX 1650).

Дальнейшие исследования могут включать более подробный анализ производительности при различных параметрах программы, а также применение других методов оптимизации для дальнейшего улучшения производительности параллельных вычислительных систем.

Библиографический список:

1. A. G. Akritas and G. I. Malaschonok, “Applications of singular-value decomposition (SVD),” Mathematics and Computers in Simulation, vol. 67, pp. 15-31, 2004
2. Natarajan, Venkatanathan. “Singular Spectral Analysis (Ssa) of Solid Earth Tide (Set)-Implications to Identify Earthquake Precursors and Earthquakes in the Himalayan Region (M≥ 6) During 1991-2021.” (2022).
3. Ahmadi-Asl S. et al. Randomized algorithms for computation of Tucker decomposition and higher order SVD (HOSVD), 2021, IEEE Access, vol. 9, pp. 28684-28706
4. Wall M. E., Rechtsteiner A., Rocha L. M. Singular value decomposition and principal component analysis //A practical approach to microarray data analysis. – Boston, MA : Springer US, 2003. – С. 91-109.
5. Hammarling S. The singular value decomposition in multivariate statistics //ACM Signum Newsletter. – 1985. – Т. 20. – №. 3. – С. 2-25.

6. Amey J. L. et al. Neural network interpretation using descrambler groups //Proceedings of the National Academy of Sciences. – 2021. – Т. 118. – №. 5. – С. e2016917118.
7. S. Williams, A. Waterman, and D. Patterson, “Roofline: An Insightful Visual Performance Model for Multicore Architectures,” Comm. ACM, vol. 52, no. 4, pp. 65-76, 2009
8. H. Jia, Y. Zhang, G. Long, J. Xu, S. Yan, and Y. Li, “GPURoofline: A Model for Guiding Performance Optimizations on GPUs,” Proc. 18th Int’l Conf. Parallel Processing (Euro-Par ’12), vol. 7484, pp. 920-932, 2012.
9. T. Cramer, D. Schmidl, M. Klemm, and D. an Mey, “OpenMP Programming on Intel Xeon Phi Coprocessors: An Early Performance Comparison,” Proc. Many-Core Applications Research Community Symp. at RWTH Aachen Univ., pp. 38-44, 2012.
10. J. Treibig, G. Hager and G. Wellein: LIKWID: A lightweight performance-oriented tool suite for x86 multicore environments. Proceedings of PSTI2010, the First International Workshop on Parallel Software Tools and Tool Infrastructures, San Diego CA, September 13, 2010. DOI: 10.1109/ICPPW.2010.38 Preprint: http://arxiv.org/abs/1004.4431
11. Tutorial: Empirical Roofline Model // github.com: электронный ресурс. URL: https://github.com/RRZE-HPC/likwid/wiki/Tutorial:-Empirical-Roofline-Model (дата обращения: 18.08.2023)
12. Z. Cui, Y. Liang, K. Rupnow, and D. Chen, “An Accurate GPU Performance Model for Effective Control Flow Divergence Optimization,” Proc. IEEE 26th Int’l Parallel Distributed Processing Symp. (IPDPS), pp. 83-94, 2012
13. W. W. L. Fung, I. Sham, G. Yuan, and T. M. Aamodt, “Dynamic Warp Formation and Scheduling for Efficient GPU Control Flow,” in Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture, ser. MICRO 40, 2007, pp. 407–420.
14. J. Meng, D. Tarjan, and K. Skadron, “Dynamic warp subdivision for integrated branch and memory divergence tolerance,” in Proceedings of the 37th annual international symposium on Computer architecture, ser. ISCA ’10, 2010, pp. 235–246
15. E. Z. Zhang, Y. Jiang, Z. Guo, and X. Shen, “Streamlining GPU applications on the fly: thread divergence elimination through runtime thread-data remapping,” in Proceedings of the 24th ACM International Conference on Supercomputing, ser. ICS ’10, 2010, pp. 115–126.
16. CUDA C++ programming guide // docs.nvidia.com: электронный ресурс. URL: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html (дата обращения: 18.08.2023)
17. Yu, Qi & Childers, Bruce & Huang, Libo & Qian, Cheng & Wang, Zhiying. (2020). A quantitative evaluation of unified memory in GPUs. The Journal of Supercomputing. 76. 10.1007/s11227-019-03079-y.
18. Chien, Steven & Peng, Ivy & Markidis, Stefano. (2019). Performance Evaluation of Advanced Features in CUDA Unified Memory. 50-57. 10.1109/MCHPC49590.2019.00014. A quantitative evaluation of unified memory in GPUs Qi Yu · Bruce Childers · Libo Huang · Cheng Qian · Zhiying Wang
19. How to Access Global Memory Efficiently in CUDA C/C++ Kernels // Nvidia technical blog: электронный ресурс. URL: https://developer.nvidia.com/blog/how-access-global-memory-efficiently-cuda-c-kernels/ (дата обращения: 18.08.2023)
20. QueryPerformanceCounter function – Win32 apps // Microsoft Learn: электронный ресурс. URL: https://learn.microsoft.com/en-us/windows/win32/api/profileapi/nf-profileapi-queryperformancecounter (дата обращения: 18.08.2023)
21. CUDA Pro Tip: Write Flexible Kernels with Grid-Stride Loops // Nvidia technical blog: электронный ресурс. URL: https://developer.nvidia.com/blog/cuda-pro-tip-write-flexible-kernels-grid-stride-loops/ (дата обращения: 18.08.2023)

References

1. Akritas A. G. and G. I. Malaschonok, “Applications of singular-value decomposition (SVD),” *Mathematics and Computers in Simulation*, 2004; 67:15-31,
2. Natarajan, Venkatanathan. “Singular Spectral Analysis (Ssa) of Solid Earth Tide (Set)-Implications to Identify Earthquake Precursors and Earthquakes in the Himalayan Region (M≥ 6) During 1991-2021.” (2022).
3. Ahmadi-Asl S. et al. Randomized algorithms for computation of Tucker decomposition and higher order SVD (HOSVD), 2021; 9: 28684-28706
4. Wall M. E., Rechtsteiner A., Rocha L. M. Singular value decomposition and principal component analysis. *A practical approach to microarray data analysis.* – Boston, MA : Springer US, 2003; 91-109.
5. Hammarling S. The singular value decomposition in multivariate statistics.*ACM Signum Newsletter*. 1985;. 20(3): 2-25.
6. Amey J. L. et al. Neural network interpretation using descrambler groups. *Proceedings of the National Academy of Sciences.* 2021; 118 (5):2016917118.
7. S. Williams, A. Waterman, and D. Patterson, “Roofline: An Insightful Visual Performance Model for

Multicore Architectures," *Comm. ACM*, 2009; 52(4): 65-76,
8. H. Jia, Y. Zhang, G. Long, J. Xu, S. Yan, and Y. Li, "GPURoofline: A Model for Guiding Performance Optimizations on GPUs," Proc. 18th Int'l Conf. Parallel Processing (Euro-Par '12), 2012; 7484: 920-932.
9. T. Cramer, D. Schmidl, M. Klemm, and D. an Mey, "OpenMP Programming on Intel Xeon Phi Coprocessors: An Early Performance Comparison," Proc. Many-Core Applications Research Community Symp. at RWTH Aachen Univ., 2012; 38-44.
10. J. Treibig, G. Hager and G. Wellein: LIKWID: A lightweight performance-oriented tool suite for x86 multicore environments. Proceedings of PSTI2010, the First International Workshop on Parallel Software Tools and Tool Infrastructures, San Diego CA, September 13, 2010. DOI: 10.1109/ICPPW.2010.38 Preprint: http://arxiv.org/abs/1004.4431
11. Tutorial: Empirical Roofline Model // github.com: elektronnyj resurs. URL: https://github.com/RRZE-HPC/likwid/wiki/Tutorial:-Empirical-Roofline-Model (data obrashcheniya: 18.08.2023)
12. Z. Cui, Y. Liang, K. Rupnow, and D. Chen, "An Accurate GPU Performance Model for Effective Control Flow Divergence Optimization," Proc. IEEE 26th Int'l Parallel Distributed Processing Symp. (IPDPS), 2012; 83-94
13. W. W. L. Fung, I. Sham, G. Yuan, and T. M. Aamodt, "Dynamic Warp Formation and Scheduling for Efficient GPU Control Flow," in Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture, ser. MICRO 40, 2007, pp. 407–420.
14. J. Meng, D. Tarjan, and K. Skadron, "Dynamic warp subdivision for integrated branch and memory divergence tolerance," in Proceedings of the 37th annual international symposium on Computer architecture, ser. ISCA '10, 2010;. 235–246
15. E. Z. Zhang, Y. Jiang, Z. Guo, and X. Shen, "Streamlining GPU applications on the fly: thread divergence elimination through runtime thread-data remapping," in Proceedings of the 24th ACM International Conference on Supercomputing, ser. ICS '10, 2010; 115–126.
16. CUDA C++ programming guide // docs.nvidia.com: elektronnyj resurs.. URL: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html (data obrashcheniya: 18.08.2023)
17. Yu, Qi & Childers, Bruce & Huang, Libo & Qian, Cheng & Wang, Zhiying. (2020). A quantitative evaluation of unified memory in GPUs. The Journal of Supercomputing. 76. 10.1007/s11227-019-03079-y.
18. Chien, Steven & Peng, Ivy & Markidis, Stefano. (2019). Performance Evaluation of Advanced Features in CUDA Unified Memory. 50-57. 10.1109/MCHPC49590.2019.00014. A quantitative evaluation of unified memory in GPUs Qi Yu · Bruce Childers · Libo Huang · Cheng Qian · Zhiying Wang
19. How to Access Global Memory Efficiently in CUDA C/C++ Kernels // Nvidia technical blog: elektronnyj resurs.. URL: https://developer.nvidia.com/blog/how-access-global-memory-efficiently-cuda-c-kernels/ (data obrashcheniya: 18.08.2023)
20. QueryPerformanceCounter function – Win32 apps. Microsoft Learn: elektronnyj resurs.. URL: https://learn.microsoft.com/en-us/windows/win32/api/profileapi/nf-profileapi-queryperformancecounter (data obrashcheniya: 18.08.2023)
21. CUDA Pro Tip: Write Flexible Kernels with Grid-Stride Loops. Nvidia technical blog: elektronnyj resurs.. URL: https://developer.nvidia.com/blog/cuda-pro-tip-write-flexible-kernels-grid-stride-loops/ (data obrashcheniya: 18.08.2023)

Сведения об авторах:
Безрученко Алексей Юрьевич, аспирант, alexei.bezruchenko@yandex.ru
Егунов Виталий Алексеевич, кандидат технических наук, доцент, кафедра ЭВМ и систем; vegunov@mail.ru
Information about the authors:
Aleksei Yu. Bezruchenko, Postgraduate Student; alexei.bezruchenko@yandex.ru
Vitaly A. Egunov, Cand. Sci. (Eng.), Assoc. Prof., Computers and Systems Department; vegunov@mail.ru
Конфликт интересов/Conflict of interest.
Авторы заявляют об отсутствии конфликта интересов/The authors declare no conflict of interest.
Поступила в редакцию/ Received 08.10.2023.
Одобрена после рецензирования / Reviced 20.10.2023.
Принята в печать /Accepted for publication 20.10.2023.