**Теория**

1. **(Юля)** [Распределенные системы: основные понятия, проблемы организации](#_heading=h.1fob9te).
2. **(Егор)** [Закон Амдала и его следствия](#_heading=h.3znysh7).
3. **(Юля)** [Архитектура GPU. Основные отличия GPU и CPU. Нити, блоки, Warp. Встроенные типы и переменные](#_heading=h.2et92p0).
4. **(Егор)** [Мультипроцессорная когерентность кэш-памяти](#_heading=h.tyjcwt).
5. **(Егор)** [Классификация архитектуры кластерных систем. Топологии кластеров. Требования к аппаратному обеспечению кластеров.](#_heading=h.3dy6vkm)
6. **(Кирилл Р**)[Параллелизм как основа высокопроизводительных вычислений. Уровни параллелизма](#_heading=h.1t3h5sf).
7. **(Дима Д)** [Требования к распределенным информационным системам](#_heading=h.4d34og8).
8. **(Эля)**[Аппаратный параллелизм: идеальный параллелизм, конвейерный параллелизм, синхронный и асинхронный параллелизм](#_heading=h.2s8eyo1).
9. (**Саша**)[Архитектура GPU. Атомарные операции. Типы памяти. Оптимизация CUDA программ](#_heading=h.17dp8vu).
10. **(Настя)**[Системы с массовым параллелизмом (MPP)](#_heading=h.3rdcrjn).
11. **(Юля)** [Распределенная обработка данных. Технология Map-Reduce](#_heading=h.26in1rg).
12. (**Серёга**) [Классификация параллельных вычислительных систем по Флинну](#_heading=h.lnxbz9).
13. **(Эля)**[Симметричные мультипроцессорные системы](#_heading=h.35nkun2).
14. **(Кирилл М)**[Проблема когерентности памяти при организации параллельных вычислений](#_heading=h.1ksv4uv).
15. **(Настя)**[Модели параллельных вычислений](#_heading=h.l5228thatxol).

**Задачи**

1. (**Артём**)[Поиск среднего и медианы с использованием технологии map-reduce](#_heading=h.2jxsxqh)
2. (Батя)[Реализация сортировки списка с использованием технологии CUDA.](#_heading=h.z337ya)
3. (**Рустам**) [Реализация решения СЛАУ методом простой итерации на GPU](#_heading=h.3j2qqm3).
4. (**Артём)**[Операция перемножения матриц с использованием технологии map-reduce.](#_heading=h.1y810tw)
5. **(Кирилл М)** [Операция перемножения матриц с использованием технологии CUDA](#_heading=h.4i7ojhp).
6. (**Рустам**)[Решение СЛАУ методом Гаусса на GPU. Оценка эффективности.](#_heading=h.2xcytpi)
7. **(Тимур)** [Поиск обратной матрицы на GPU](#_heading=h.1ci93xb).
8. Алгоритм поиска пересечения двух списков с использованием технологии map-reduce.
9. Определение моды и медианы с использованием технологии map-reduce.
10. Определение 25% и 75% квантилей с использованием технологии map-reduce.
11. Параллельное вычисление алгоритма обратного распространения ошибки на GPU при обучении ИНН.
12. Особенности реализации матрично-векторного умножения при различных видах аппаратного параллелизма.

## Теория

### (Юля) Распределенные системы: основные понятия, проблемы организации.

По утверждению известного специалиста в области информатики Э. Таненбаума, не существует общепринятого и в то же время строгого определения распределенной системы.

**Легкое определение:** распределенной является любая вычислительная система, где обработка данных разделена между двумя и более компьютерами.

**Более узкое определение** (основываясь на определении Э. Таненбаума): набор соединенных каналами связи независимых компьютеров, которые с точки зрения пользователя некоторого программного обеспечения выглядят единым целым.

**Параметры:**

1. **Открытость**. Все протоколы взаимодействия *компонент*  внутри распределенной системы в идеальном случае должны быть основаны на общедоступных стандартах. Это позволяет использовать для создания *компонент* различные *средства разработки* и операционные системы. Каждая компонента должна иметь точную и полную спецификацию своих сервисов. В этом случае компоненты распределенной системы могут быть созданы независимыми разработчиками. При нарушении этого требования может исчезнуть возможность создания распределенной системы, охватывающей несколько независимых организаций.

Программная компонента – *единица* программного обеспечения, исполняемая на одном компьютере в пределах одного процесса, и предоставляющая некоторый набор сервисов, которые используются через ее внешний *интерфейс* другими компонентами, как выполняющимися на этом же компьютере, так и на удаленных компьютерах.

1. **Масштабируемость**. Наиболее важный аспект – возможность добавления в распределенную систему новых компьютеров для увеличения производительности системы, что связано с понятием балансировки нагрузки ( *load balancing* ) на серверы системы. Также необходимо эффективно распределять ресурсы сервера, обслуживающего запросы клиентов.
2. **Поддержание логической целостности данных**. *Запрос* пользователя в распределенной системе должен либо корректно выполняться целиком, либо не выполняться вообще. Ситуация, когда часть *компонент* системы корректно обработали поступивший *запрос*, а часть – нет, является наихудшей.
3. **Устойчивость**. Возможность дублирования несколькими компьютерами одних и тех же функций или же возможность автоматического распределения функций внутри системы в случае выхода из строя одного из компьютеров. В идеальном случае это означает полное отсутствие уникальной точки сбоя, то есть *выход* из строя одного любого компьютера не приводит к невозможности обслужить *запрос* пользователя.
4. **Безопасность**. Каждый *компонент*, образующий распределенную систему, должен быть уверен, что его функции используются авторизированными на это компонентами или пользователями. Данные, передаваемые между компонентами, должны быть защищены как от искажения, так и от просмотра третьими сторонами.

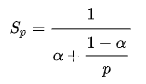
**Основные проблемы (по сравнению с традиционными системами)**:

* проблемы администрирования системы:
* проблемы ограниченности масштабируемости РС;
* проблемы переносимости программного обеспечения.

### (Егор) Закон Амдала и его следствия.

Пусть необходимо решить некоторую вычислительную задачу. Предположим, что её алгоритм таков, что доля от общего объёма вычислений может быть получена только последовательными расчётами, а, соответственно, доля 1- может быть распараллелена идеально (то есть время вычисления будет обратно пропорционально числу задействованных узлов p). 

* доля последовательных операций

р - число процессов в системе

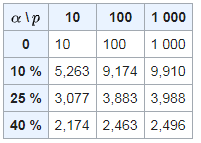
Тогда ускорение, которое может быть получено на вычислительной системе из p процессоров, по сравнению с однопроцессорным решением не будет превышать величины Sp.

Таблица показывает, во сколько раз быстрее выполнится программа с долей последовательных вычислений при использовании p процессоров.

Таким образом, только алгоритм, вовсе не содержащий последовательных вычислений (a = 0), позволяет получить линейный прирост производительности с ростом количества вычислителей в системе. Если доля последовательных вычислений в алгоритме равна 25 %, то увеличение числа процессоров до 10 дает ускорение в 3.077 раз, а увеличение числа процессоров до 1000 даст ускорение в 3.988 раз.

При доле последовательных вычислений общий прирост производительности не может превысить 1/ .Так, если половина кода — последовательная, то общий прирост никогда не превысит двух.

*«В случае, когда задача разделяется на несколько частей, суммарное время её выполнения на параллельной системе не может быть меньше времени выполнения самого длинного фрагмента». Согласно этому закону, ускорение выполнения программы за счёт распараллеливания её инструкций на множестве вычислителей ограничено временем, необходимым для выполнения её последовательных инструкций.*

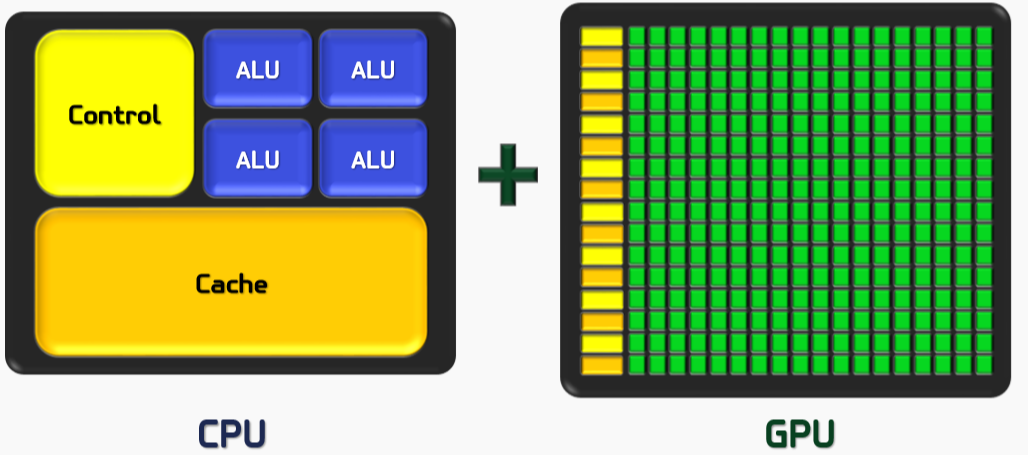
**Следствия из закона Амдала:**

1. последовательная часть накладывает строгие ограничения на ускорение, которое может быть достигнуто путем увеличения числа процессоров;
2. построение системы с большим числом процессоров неэффективно, поскольку достаточное ускорение при этом не достигается;
3. с другой стороны, большинство важных приложений, нуждающихся в распараллеливании, содержат очень незначительную долю последовательных вычислений и в этом случае системы с большим числом процессоров вполне обоснованы.

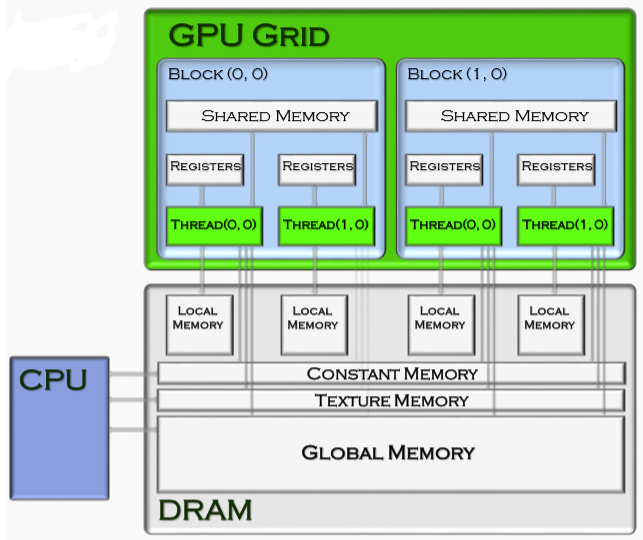
### (Юля) Архитектура GPU. Основные отличия GPU и CPU. Нити, блоки, Warp. Встроенные типы и переменные.

Видеокарта содержит Compute Units – процессорные ядра. Неразбериху вводят и производители NVidia называет свои Streaming Multiprocessor unit или SMX , а ATI – SIMD Engine или Vector Processor.

Процессорные ядра могут выполнять несколько потоков за счет того, что в каждом содержится несколько (8-16) потоковых процессоров (Stream Cores или Stream Processor). Для карт NVidia – вычисления производятся непосредственно на потоковых процессорах, но ATI ввели еще один уровень абстракции – каждый потоковый процессор, состоит из processing elements – PE (иногда называемых ALU – arithmetic and logic unit) – и вычисления происходят на них.

**Отличия GPU (SIMD архитектура):**

* очень высокая степень параллелизма
* основная часть чипа занята логикой, а не кэшем
* ограничение функциональности (для достижения высоких скоростей может выполнить лишь часть операций, которые может выполнить центральный процессор)



**Спецификаторы функций**

|  |  |  |
| --- | --- | --- |
| **Спецификатор** | **Функция выполняется на** | **Функция может вызываться из** |
| \_\_device\_\_ | device | device |
| \_\_global\_\_ | device | host, device |
| \_\_host\_\_ | host | host |

**Спецификаторы переменных**

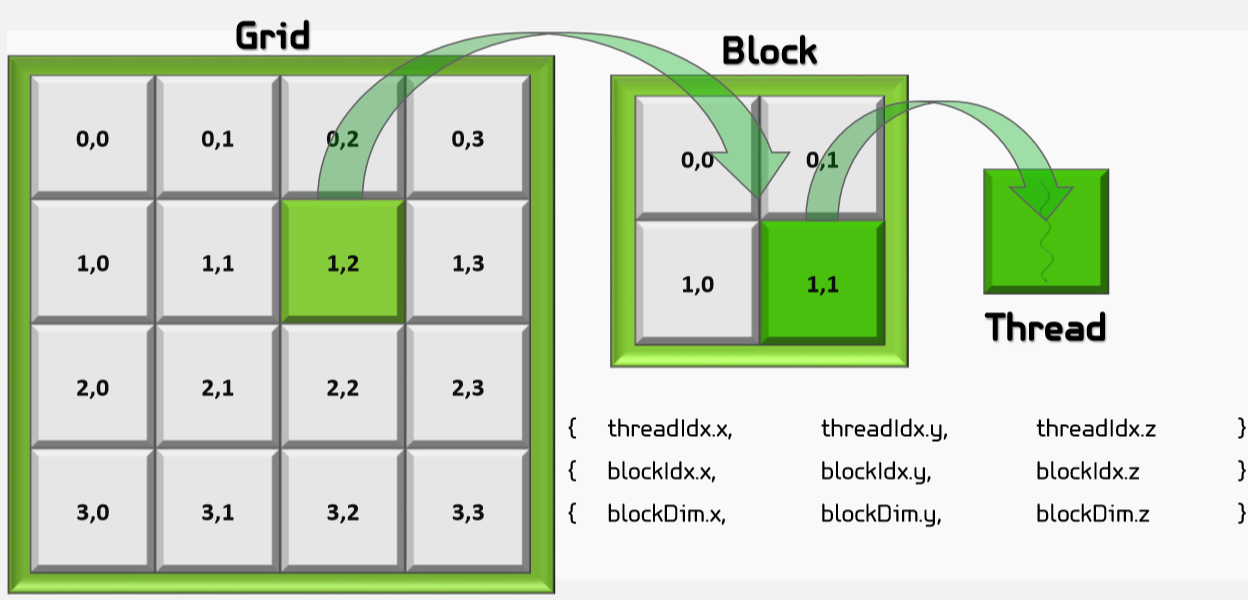
|  |  |  |  |
| --- | --- | --- | --- |
| **Спецификатор** | **Находится** | **Доступна** | **Вид доступа** |
| \_\_device\_\_ | device | device/host | R/W |
| \_\_constant\_\_ | device | device/host | R/o |
| \_\_shared\_\_ | device | block | R/W |

blockIdx - номер блока, в котором находится нить

blockDim - размер блока

**Линейный индекс нити:**

tid = threadIdx.x + blockIdx.x \* blockDim.x



Kernel – Параллельная часть алгоритма, выполняемая на Grid

Grid – Объединение блоков, которые выполняются на одном устройстве

Block – Объединение потоков, которое выполняется целиком на одном SM. Имеет свой уникальный идентификатор внутри Grid.

Thread – единица выполнения программы. Имеет свой уникальный идентификатор внутри Block.

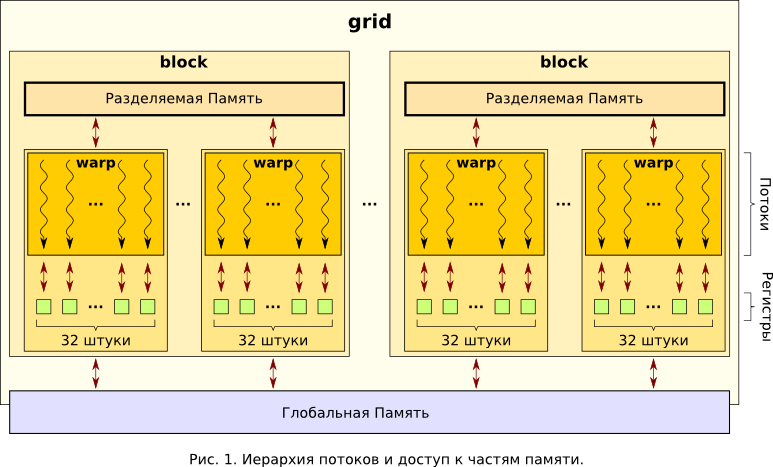
Warp – 32 последовательно идущих Thread

**Исполнение потоков**

* На каждом мультипроцессоре может обрабатываться несколько блоков потоков
* На каждом такте мультипроцессор исполняет одну и ту же инструкцию над группой потоков, называемой варп (**warp**)
* Количество потоков в варпе – warp size (32 потока)
* В случае условного ветвления, сначала исполняется одна ветвь над одной частью варпа, затем другая над оставшейся if( threadIdx.x < 8 ) { do smth } else { do smth else }

**Планировщик варпов**

* На каждом такте планировщик выбирает группу потоков и над каждым потоком из группы исполняется одна и та же команда.
* Планирование варпов осуществляется таким образом, чтобы максимально скрыть время ожидания данных из глобальной памяти: пока одни группы потоков ожидают данные, над другими исполняются инструкции.
* **Роль планировщика:** скрывать время ожидания данных одними потоками из глобальной памяти за счёт работы других потоков в это время.



* Ядро – это функция, которая исполняется над сеткой блоков потоков
* Каждый поток и блок имеет собственные координаты
* Блок потоков - ­ набор потоков, способных общаться между собой посредством
  + Разделяемой памяти
  + Точек синхронизации (барьеры)

**Вызов функции-ядра**

KernelName <<< nBlock, nThread, nShMem, nStream >>> ( args);

KernelName - название функции-ядра

nBlock - число блоков сети ( grid )

nThread - число нитей в блоке

nShMem - количество дополнительной разделяемой памяти, выделяемой на блок

nStream - номер потока из которого запускается функция-ядро

### (Егор) Мультипроцессорная когерентность кэш-памяти.

Мультипроцессорная система с совместно используемой памятью состоит из двух или более независимых процессоров, каждый из которых выполняет либо часть большой программы, либо независимую программу.

**Проблема 1**: Все процессоры обращаются к командам и данным, хранящимся в общей основной памяти (разделяемый ресурс). При обращении к ней между процессорами возникает соперничество, в результате чего средняя задержка на доступ к памяти увеличивается.

**Решение 1**: Для сокращения такой задержки каждому процессору придается локальная кэш-память, которая, обслуживая локальные обращения к памяти, во многих случаях предотвращает необходимость доступа к совместно используемой основной памяти.

**Проблема 2:** Оснащение каждого процессора локальной кэш-памятью приводит к проблеме когерентности (согласованности кэш-памяти). Система является когерентной, если каждая операция чтения по какому-либо адресу, выполненная любым из процессоров, возвращает значение, занесенное в ходе последней операции записи по этому адресу, вне зависимости от того, какой из процессоров производил запись последним.

**Пример:**

Пусть два процессора P1 и P2 связаны с общей памятью посредством шины. Сначала оба процессора читают переменную x. Копии блоков, содержащих эту переменную, пересылаются из основной памяти в локальные кэши обоих процессоров. Далее процессор P1 выполняет операцию увеличения значения переменной x на единицу. Так как копия переменной уже находится в кэш-памяти данного процессора, произойдет кэш-попадание и значение x будет изменено только в кэш-памяти 1. Если теперь процессор P2 вновь выполнит операцию чтения x, то также произойдет кэш-попадание и P2 получит хранящееся в его кэш-памяти «старое» значение x.

**Пути решения:**

1. **Программный:**

Позволяет обойтись без дополнительного оборудования или свести его к минимуму. Задача возлагается на компилятор и операционную систему.

**Плюс:** возможность устранения некогерентности еще до этапа выполнения программы, **Минус:** принятые компилятором решения могут в целом отрицательно сказаться на эффективности использования кэш-памяти.

Компилятор анализирует программный код, определяет те совместно используемые данные, которые могут причиной некогерентности, и помечает их. В процессе выполнения программы операционная система или соответствующая аппаратура предотвращают кэширование (занесение в кэш-память) помеченных данных и для доступа к ним, как при чтении, так и при записи, приходится обращаться к «медленной» основной памяти. Учитывая, что некогерентность возникает только в результате операций записи, происходящих значительно реже, чем чтение, рассмотренный прием следует признать недостаточно эффективным.

Более эффективными представляются способы, где в ходе анализа программы определяются безопасные периоды использования общих переменных, и так называемые критические периоды, где может возникнуть некогерентность. Затем компилятор вставляет в генерируемый код инструкции, позволяющие обеспечить когерентность кэш-памятей именно в такие критические периоды.

**2. Аппаратный**:

В случае мультипроцессорной системы, когда копии совместно используемых данных могут находиться сразу в нескольких кэшах, необходимо обеспечить когерентность всех копий. Ни сквозная, ни обратная запись не предусматривают такой ситуации, и для ее разрешения используются другие приемы, а именно: запись с аннулированием (write invalidate) и запись с обновлением(write update). Последняя известна также под названием записи с трансляцией(write broadcast).

**Запись с аннулированием:** если какой-либо процессор производит изменения в одном из блоков своей кэш-памяти, все имеющиеся копии этого блока в других локальных кэшах аннулируются, то есть помечаются как недостоверные. Для этого бит достоверности измененного блока (строки) во всех прочих кэшах устанавливается в 0.

**Запись с обновлением:** любая запись в локальный кэш немедленно дублируется и во всех остальных кэшах, содержащих копию измененного блока (немедленное обновление блока в основной памяти не является обязательным).

### (Егор) Классификация архитектуры кластерных систем. Топологии кластеров. Требования к аппаратному обеспечению кластеров.

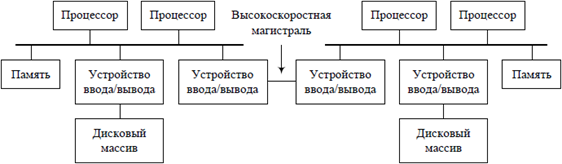
**По способу использования дисковых массивов:**

* совместно
* раздельно

**Различия**:

* в случае локальной сети узлы используют локальные дисковые массивы
* в случае выделенной линии узлы совместно используют один избыточный массив независимых жёстких дисков или так называемый RAID (RedundantArrayofIndependentDisks). RAID состоит из нескольких дисков, управляемых контроллером, взаимосвязанных скоростными каналами и воспринимаемых внешней системой как единое целое. В зависимости от типа используемого массива могут обеспечиваться различные степени отказоустойчивости и быстродействия.

**Конфигурация кластера с раздельным использованием дисковых массивов:**



**Конфигурация кластера с совместно используемыми дисками:**



|  |  |
| --- | --- |
| **Метод кластеризации** | **Описание** |
| Пассивное резервирование | Вторичный сервер при отказе первичного берет управление на себя |
| Резервирование с активным вторичным сервером | Вторичный сервер, как и первичный, используется при решении задач |
| Самостоятельные серверы | Самостоятельные серверы имеют собственные диски, а данные постоянно копируются с первичного сервера на вторичный |
| Серверы с подключением ко всем дискам | Серверы подключены к одним и тем же дискам, но каждый сервер владеет своей их частью. Если один из серверов отказывает, то управление его дисками берет на себя другой сервер. |
| Серверы с совместно используемыми дисками | Множество серверов работают в режиме коллективного доступа к дискам. |

**Топологии:**

**Топология кластерных пар** используется при организации двух- или четырехузловых кластеров. Узлы группируются попарно, дисковые массивы присоединяются к обоим узлам, входящим в состав пары, причем каждый узел пары имеет доступ ко всем дисковым массивам данной пары. Один из узлов пары используется как резервный для другого.



Четырехузловая кластерная пара представляет собой простое расширение двухузловой топологии. Обе кластерные пары с точки зрения администрирования и настройки рассматриваются как единое целое.

**Топология N + 1** позволяет создавать кластеры из двух, трех и четырех узлов. Каждый дисковый массив подключается только к двум узлам кластера. Дисковые массивы организованы по схеме RAID1 (mirroring). Один сервер имеет соединение со всеми дисковыми массивами и служит в качестве резервного для всех остальных (основных или активных) узлов. Резервный сервер может использоваться для обеспечения высокой степени готовности в паре с любым из активных узлов.



**Топология N × N** аналогично топологии N + 1 позволяет создавать кластеры из двух, трех и четырех узлов, но в отличие от нее обладает большей гибкостью и масштабируемостью. Только в этой топологии все узлы кластера имеют доступ ко всем дисковым массивам, которые, в свою очередь, строятся по схеме RAID1 (mirroring). Масштабируемость топологии проявляется в простоте добавления к кластеру дополнительных узлов и дисковых массивов без изменения соединений в системе. Топология позволяет организовать каскадную систему отказоустойчивости, при которой обработка переносится с неисправного узла на резервный, а в случае его выхода из строя на следующий резервный узел и т. д. В целом топология обладает лучшей отказоустойчивостью и гибкостью по сравнению с другими топологиями.



**Топология с полностью раздельным доступом** допускает соединение каждого дискового массива только с одним узлом кластера. Рекомендуется лишь для тех приложений, для которых характерна архитектура полностью раздельного доступа.

**Требования:**

* отказоустойчивость
* доступность функции (готовность)
* масштабируемость
* вычислительная мощность

### Параллелизм как основа высокопроизводительных вычислений. Уровни параллелизма.

В основе архитектуры большинства современных ВМ лежит представление алгоритма решения задачи в виде программы последовательных вычислений. Базовые архитектурные идеи ВМ, ориентированной на последовательное исполнение команд программы, были сформулированы Джоном фон Нейманом. В условиях постоянно возрастающих требований к производительности вычислительной техники все очевидней становятся ограничения классической фон-неймановской архитектуры, обусловленные исчерпанием всех основных идей ускорения последовательного счета. Дальнейшее развитие вычислительной техники связано с переходом к параллельным вычислениям как в рамках одной ВМ, так и путем создания многопроцессорных систем и сетей, объединяющих большое количество отдельных процессоров или отдельных вычислительных машин. Для такого подхода вместо термина «вычислительная машина» более подходит термин «вычислительная система» (ВС). Отличительной особенностью вычислительных систем является наличие в них средств, реализующих параллельную обработку, за счет построения параллельных ветвей в вычислениях, что не предусматривалось классической структурой ВМ. Идея параллелизма как средства увеличения быстродействия ЭВМ возникла очень давно — еще в XIX веке.

**Уровни параллелизма**

Методы и средства реализации параллелизма зависят от того, на каком уровне он Должен обеспечиваться. Обычно различают следующие уровни параллелизма:

■ **Уровень заданий**. Несколько независимых заданий одновременно выполняются на разных процессорах, практически не взаимодействуя друг с другом. Этот уровень реализуется на ВС с множеством процессоров в многозадачном режиме.

■ **Уровень программ**. Части одной задачи выполняются на множестве процессоров. Данный уровень достигается на параллельных ВС.

■ **Уровень команд**. Выполнение команды разделяется на фазы, а фазы нескольких последовательных команд могут быть перекрыты за счет конвейеризации. Уровень достижим на ВС с одним процессором.

■ **Уровень битов** (арифметический уровень). Биты слова обрабатываются один за другим, это называется бит - последовательной операцией. Если биты слова обрабатываются одновременно, говорят о бит - параллельной операции. Данный уровень реализуется в обычных и суперскалярных процессорах.

К понятию уровня параллелизма тесно примыкает понятие *гранулярности.* Это мера отношения объема вычислении, выполненных в параллельной задаче, к объему коммуникаций (для обмена сообщениями). Степень гранулярности варьируется от мелкозернистой до крупнозернистой. Определим понятия крупнозернистого (coarse grained), среднезернистого (medium grained) и мелкозернистого (fine grained) параллелизма.

*Крупнозернистый параллелизм:* каждое параллельное вычисление достаточно независимо от остальных, причем требуется относительно редкий обмен информацией между отдельными вычислениями. Единицами распараллеливания являются большие и независимые программы, включающие тысячи команд. Этот уровень параллелизма обеспечивается операционной системой.

*Среднезернистый параллелизм:* единицами распараллеливания являются вызываемые процедуры, включающие в себя сотни команд. Обычно организуется как программистом, так и компилятором.

*Мелкозернистый параллелизм:* каждое параллельное вычисление достаточно мало и элементарно, составляется из десятков команд. Обычно распараллеливаемыми единицами являются элементы выражения или отдельные итерации цикла, имеющие небольшие зависимости по данным. Сам термин «мелкозернистый параллелизм» говорит о простоте и быстроте любого вычислительного действия. Характерная особенность мелкозернистого параллелизма заключается в приблизительном равенстве интенсивности вычислений и обмена данными. Этот уровень параллелизма часто используется распараллеливающим (векторизирующим) компилятором.

Эффективное параллельное исполнение требует искусного баланса между степенью гранулярности программ и величиной коммуникационной задержки, возникающей между разными гранулами. В частности, если коммуникационная задержка минимальна, то наилучшую производительность обещает мелкоструктурное разбиение программы. Это тот случай, когда действует параллелизм данных. Если коммуникационная задержка велика (как в слабосвязанных системах), предпочтительней крупнозернистое разбиение программ.

### (Дима Д) Требования к распределенным информационным системам.

Основными требованиями, предъявляемыми к распределенным системам (РС), являются: прозрачность, открытость системы, безопасность, масштабируемость РС, надежность.

*Прозрачность распределенной системы*. Под прозрачностью распределенной системы понимают ее способность скрывать свою распределенную природу, а именно, распределение процессов и ресурсов по множеству компьютеров, и представляться для пользователей в виде единой централизованной компьютерной системы.

Виды прозрачности:

1. *Прозрачность местоположения*. Заключается в том, что пользователь не должен знать, где расположены необходимые ему ресурсы. Файлы могут перемещаться на различные узлы распределенной системы, например, если на узле РС произошел сбой, и данные были восстановлены на другом узле РС, но при этом, пользователь не должен замечать эти перемещения. Например, в распределенных информационных файловых системах, пользователь должен видеть лишь единое файловое пространство, притом, что данные могут располагаться физически на разных серверах.
2. *Прозрачность доступа*. Играет наиболее важную роль. Прозрачность в данном случае заключается в обеспечении сокрытия различий доступа и предоставлении данных. Вне зависимости от способов доступа к ресурсам и их внутреннего представления, обращения к локальным и удаленным ресурсам осуществляется одинаковым образом. На базовом уровне скрывается разница архитектур вычислительных платформ, а также достигается соглашение о том, как ресурсы разнородных машин, будут представляться пользователям распределенной системы единым образом. В качестве примера можно привести прикладной программный интерфейс для работы с файлами, хранящимися на множестве компьютеров различных архитектур, который предоставляет одинаковые вызовы операций как с локальными, так и с удаленными файлами.
3. *Прозрачность параллелизма (одновременного) доступа*. Позволяет нескольким пользователям (конкурирующим процессам) одновременно выполнять операции над общим, совместно используемым ресурсом без взаимного влияния друг на друга. Иначе говоря, скрывается факт использования ресурса другими пользователями. Стоит отметить, что сам ресурс должен оставаться в непротиворечивом состоянии, что может достигаться, например, с помощью механизма блокировок, когда пользователи по очереди получают исключительные права на запрашиваемый ресурс.
4. *Прозрачность репликации*. Если для повышения доступности или увеличения производительности используется несколько копий ресурса (реплик), этот факт остается скрытым от пользователя, и он полагает, что в системе присутствует только один экземпляр ресурса. Для обеспечения прозрачности репликации необходимо, чтобы все реплики имели одно и то же имя, очевидно, не зависящее от местоположения копии ресурса. Таким образом, системы, которые обеспечивают прозрачность репликации, также должны поддерживать и прозрачность местоположения.
5. *Прозрачность отказов*. Система должна пытаться скрывать частичные отказы, позволяя пользователям и приложениям выполнить свою работу вне зависимости от сбоев в аппаратных или программных компонентах распределенной системы, а также скрывать факт их последующего восстановления. В связи с тем, что любой процесс, компьютер или сетевое соединение могут отказывать независимо от других в произвольные моменты времени, каждый компонент распределенной системы должен быть готов к сбоям в других компонентах и обрабатывать подобные ситуации соответствующим образом.

*Открытость системы*. Открытая система – это система, реализующая открытые спецификации (стандарты) на интерфейсы, службы и поддерживаемые форматы данных (возможность взаимодействия с другими открытыми системами), достаточные для того, чтобы обеспечить:

1. Возможность переноса разработанного прикладного программного обеспечения на широкий диапазон систем с минимальными изменениями (мобильность приложений); (РС должны соответствовать четко определенными интерфейсами)
2. Совместную работу (взаимодействие) с другими прикладными приложениями на локальных и удаленных платформах (способность к взаимодействию); (Системы, входящие в состав РС должны легко взаимодействовать между собой)
3. Взаимодействие с пользователями в стиле, облегчающим последним переход от системы к системе (мобильность пользователя). (Системы должны обеспечивать переносимость приложений)

Открытость системы может быть достигнута с помощью: языков программирования, аппаратных платформ, программного обеспечения.

*Безопасность*. Безопасность РС является, в общем случае, совокупностью 3 факторов:

1. Обеспечение конфиденциальности данных и ресурсов;
2. Обеспечение конфиденциальности доступа к ресурсам для множества пользователей;
3. Обеспечение целостности ресурсов и данных.

Многие вопросы безопасности могут быть решены на уровне отдельных узлов РС, например, путем установки фаерволов и антивирусного ПО на отдельные узлы системы, введением политики аутентификации пользователей и другими методами. Но в силу особенности архитектуры большинства РС, данный подход не всегда является эффективным. Программное обеспечение не всегда может обеспечить необходимую конфиденциальность данных в распределенной системе. Например, программное обеспечение не всегда может полноценную защиту от DDOS атак на распределенную сеть. Зачастую методы защиты от подобных атак не всегда являются приемлемыми для узлов вычислительной сети. Важным показателем, при организации защиты РС, является уровень доступности системы. Уровень доступности распределенной системы определяется не только доступностью ресурса в момент времени t, но и принципами организации защиты РС, так как большинство программных средств, обеспечивающие защиту от атак, направлены на отказ в обслуживании.

*Надежность (Отказоустойчивость) РС*. В связи с появлением новых методов и алгоритмов, требовательных к вычислительным ресурсам и, самое главное, к ресурсам времени, необходимость в доступности распределенных систем в момент времени t становится крайне актуальным. Основным показателем, определяющим надежность всей РС, является отказоустойчивость. Отказоустойчивость - это важнейшее свойство вычислительной системы, которое заключается в возможности продолжения действий, заданных программой, после возникновения неисправностей.

Характерной чертой распределенных систем, которая отличает их от единичных компьютеров, является устойчивость к частичным отказам, т.е. система продолжает функционировать после частичных отказов. Подобная возможность достигается за счет избыточности, когда в систему добавляется дополнительное оборудование (аппаратная избыточность) или процессы (программная избыточность), которые делают возможной правильное функционирование системы при неработоспособности или некорректной работе некоторых из ее компонентов.

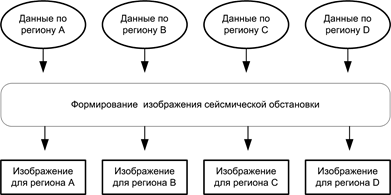
### Аппаратный параллелизм: идеальный параллелизм, конвейерный параллелизм, синхронный и асинхронный параллелизм.

Успешность распараллеливания программы для некоторой задачи зависит от природы этой задачи и архитектуры используемой вычислительной системы.

**Идеальный параллелизм** (perfect parallelism) или параллелизм уровня задания. Ключевая характеристика: множество обрабатываемых данных разбивается на подмножества и обработка каждого подмножества может выполняться полностью независимо.

Например, при составлении сейсмической карты для некоторой территории, эта территория разбивается на регионы. Соответственно разбиваются и исходные данные. Изображение для каждого участка может обрабатываться на независимых машинах, где выполняется своя копия соответствующего приложения, при этом соответствующие входные данные должны быть доступны для каждой копии приложения.

Соответствующие приложения иногда называют «тесно параллельными» (embarrassingly parallel).



**Конвейерный параллелизм** (pipeline parallelism) - данные эффективно передаются с одной стадии вычислений на другую. Ключевая характеристика: результаты однонаправлено перемещаются по конвейеру. До момента, когда данные станут доступными, на каждой ступени существует исходная стартовая задержка, то есть, общая производительность зависит от относительного числа выполняемых шагов, пока все ступени конвейера станут активными. Примером такого параллелизма может служить программа анализа структуры земной коры на основании обработки сейсмических сигналов, фиксируемых через последовательные промежутки времени.

Проблема: Если ступени конвейера вычислительно неэквивалентны, более быстрые ступени будут догонять более медленные, завершая свою работу быстрее. Одно из решений — использовать в вычислительно интенсивных ступенях более быстрые процессорные элементы (ПЭ), но точно сбалансировать вычислительную нагрузку обычно достаточно сложно. Другой путь, когда программист учитывает возможность неравной загрузки ПЭ и включает в программу проверку готовности входных данных и доступности буфера или наличия дискового пространства для хранения выходных данных. В силу упомянутых причин, конвейерный параллелизм не столь прост, как идеальный параллелизм.



Полностью **синхронный** параллелизм (fully synchronous parallelism) - каждое вычисление производится синхронно над всеми данными. Ключевая характеристика: последующие вычисления и решения зависят от результатов всех предшествующих вычислений над всеми данными. Примером может служить анализ динамики атмосферы. Анализ базируется на трехмерной модели атмосферы, где возникновение события в одной области влияет на соседние области. Со временем, эффект распространяется на все большую область во всех направлениях. Если такое приложение выполняется последовательно, то для получения текущего состояния нужно провести некоторый объем вычислений над всеми данными. Параллелизм вводится за счет множества ПЭ, участвующих в каждой итерации; каждый ПЭ обрабатывает свое подмножество данных. До перехода к следующей итерации текущая итерация должна быть полностью завершена.

Как правило, для одновременной обработки всех данных процессорных элементов в составе ВС обычно не хватает, поэтому каждый ПЭ многократно переходит от одного подмножества к другому. Если подмножества не однородны, то имеют место различия интенсивности вычислений на разных ПЭ. Следовательно, для достижения хорошей производительности полностью синхронный параллелизм требует больших усилий программиста по сравнению с параллелизмом конвейерным.

**Слабо синхронный параллелизм** (loosely synchronous parallelism) — В конце каждого временного шага ПЭ, завершившие свою работу, должны ждать пока ее закончат остальные ПЭ, с тем, чтобы промежуточные результаты могли быть использованы в следующем временном шаге. Ключевая характеристика: каждый ПЭ решает малую часть задачи, периодически обмениваясь информацией. Он наиболее труден для программиста, сочетая в себе трудности, имеющиеся в конвейерном и полностью синхронном параллелизме. Необходимость обмена информацией (в данном примере, на границе временных шагов) требует проверки, позволяющей каждому ПЭ выяснить, когда в остальных ПЭ будут готовы их данные и предотвратить перекрытие еще не использованных данных при записи новых. В данном виде параллелизма очень трудно равномерно распределить вычислительную нагрузку между ПЭ, в частности, из-за того, что объем работ варьируется как во времени, так и в пространстве.

Иллюстрируется моделью распространения загрязнения в грунтовых водах. Сначала загрязнения попадают в грунтовые воды вблизи источника этих загрязнений, но со временем они распространяются, образуя нерегулярные области концентрации. Объем требуемых вычислений пропорционален объему загрязняющего вещества и геофизической структуре, то есть он сильно меняется от области к области и от одного временного шага к другому. В последовательной программе это означает, что временные шаги будут нерегулярной (и возможно, непредсказуемой) длительности. На первых шагах каждый ПЭ выполняет вычисления для малого числа областей и длительность вычислений может быть невелика из-за малости концентраций загрязнения. Позже концентрация возрастает и возникает больше областей загрязнения, поэтому каждый ПЭ должен выполнять больше вычислений над большим числом областей.



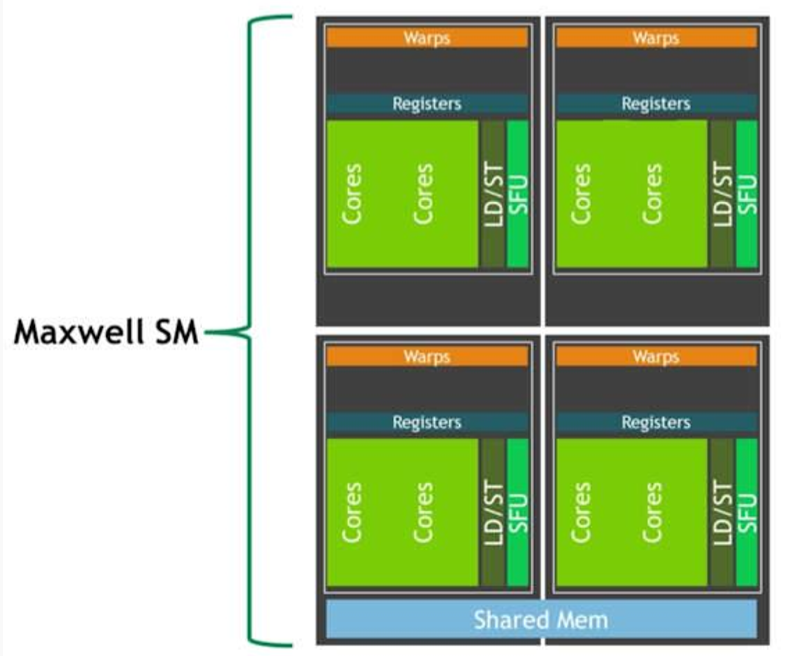
### (Саша)Архитектура GPU. Атомарные операции. Типы памяти. Оптимизация CUDA программ.

**Архитектура GPU.**

Все GPU построены на принципе SIMD - одна инструкция, много данных, что в значительной степени упрощает реализацию распараллеливания процессов.

Начиная с архитектуры Kepler. Основным изменением структуры было увеличение количества SMM блоков(streaming maxwell processor)[названия меняются с каждой архитектурой, но внутренняя структура практически не меняется] - потоковый мультипроцессов.

На примере архитектуры maxwell рассмотрим строение SM-блоков.



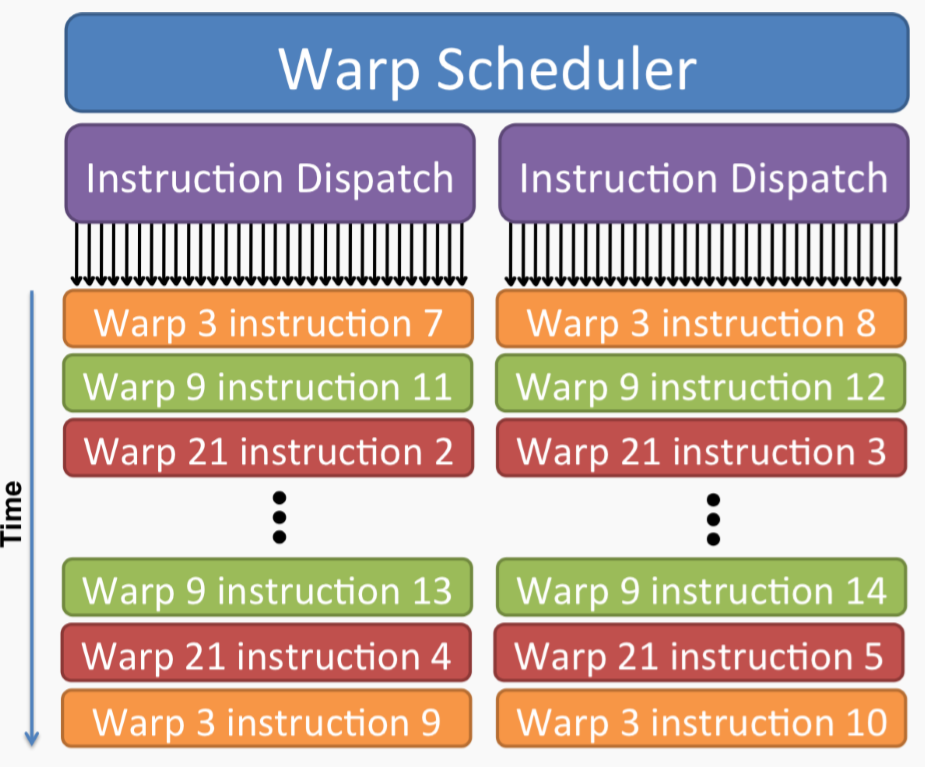
На рисунке приведена архитектура Maxwell. Принцип строение более новых архитектур отличается наличием дополнительных RTX(ray-tracing) ядер и блоков для расчетов с увеличенной точностью (Floating Point - FP32/FP64).

Каждый потоковый мультипроцессов состоит из нескольких одинаковых блоков(Warp scheduler - планировщик варпов) и общей памятью (кэш первого уровня L1).

Внутри warp’а находятся:

* Cores - 32 ядра(всего 128)
* LD/ST(Load/Store) - 8 блоков выгрузки/загрузки данных(всего 32)
* SFU(special func unit) - 8 блоков для вычислений специальных функций(всего 32)

WrapS содержит диспатчер инструкций, позволяющий синхронизировать выполнение этих инструкций в каждом блоке варпов. В случае с архитектурой Maxwell за 1 варп могут выполняться 4 инструкции(в 4 Warps-блоках).



**Атомарные операции.**

* операции, которые либо выполняются целиком, либо не выполняется вовсе.

В случае с GPU(как и в любой многопроцессорной системе) атомарные операции обеспечивают доступ(чтение/запись/CAS-сравнение с обменом) нескольких процессов(или потоков одного процесса) к разделяемыми между ними ресурсами(Shared Mem см. рис. архитектуры).

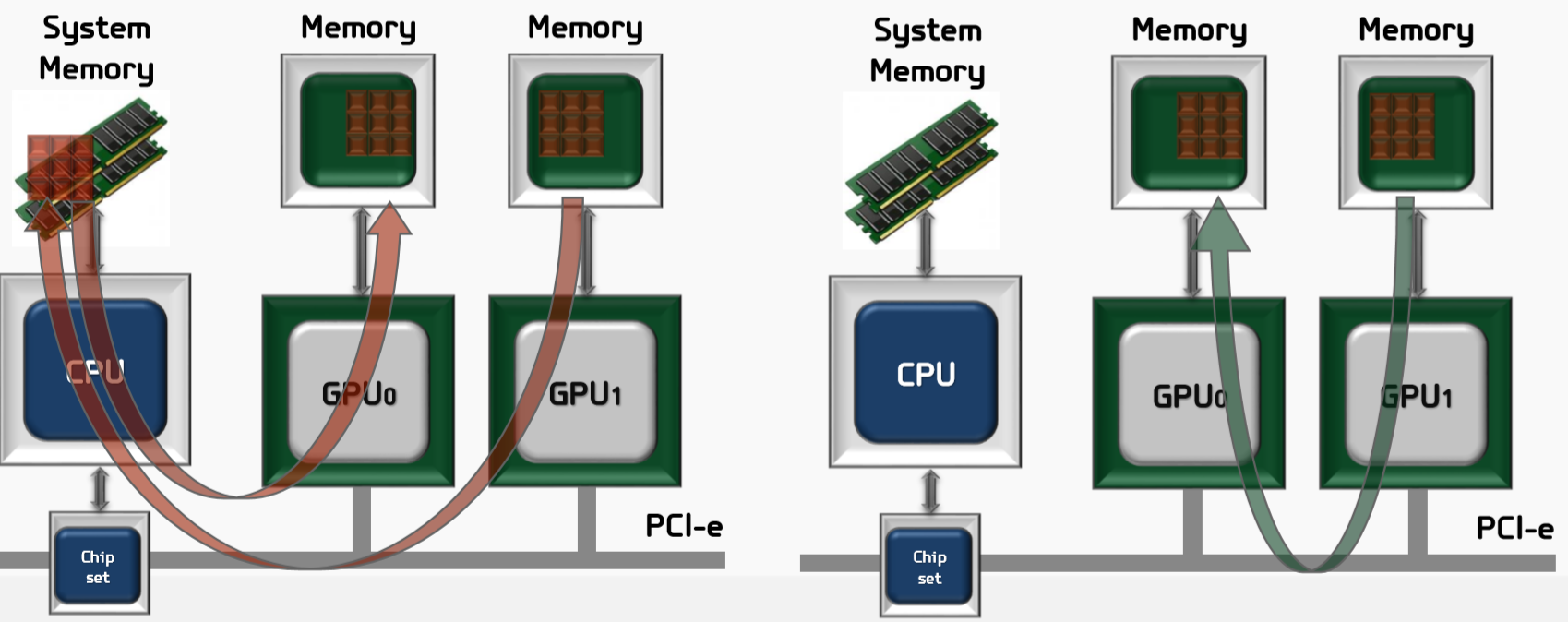
Начиная с архитектуры Maxwell атомарные операции стали встроенными.

**Типы памяти.**

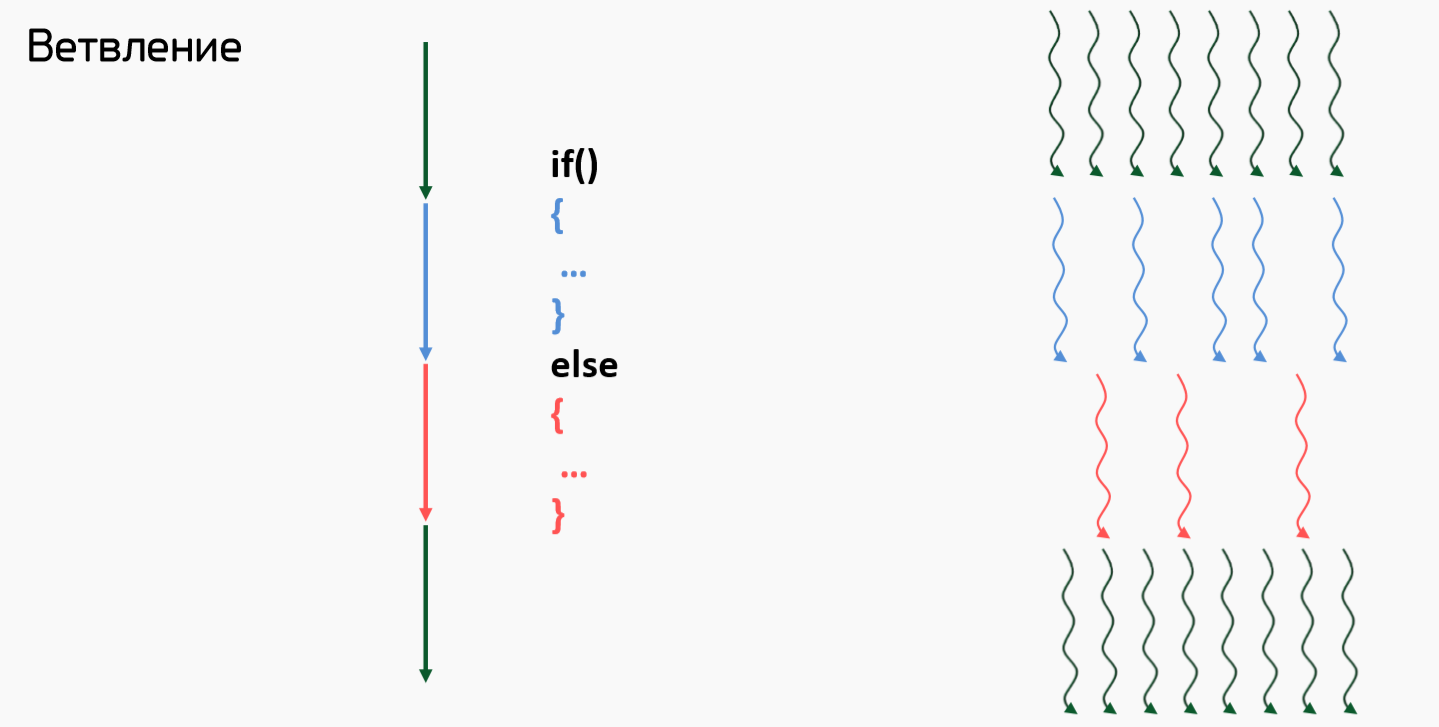
Существуют системная память(оперативка) и видео память. В связи с тем что GPU имеет SIMD принцип, видеопамять реализована проще оперативки(от 1 Ггц до 5Ггц) и имеет более высокие частоты(От 6 Ггц). Поэтому обмен данными между различными GPU осуществляется быстрее по линии PCI-Express в сравнении с оперативой, которая значительно медленнее и взаимодействие с ней происходит через процессов.

**Оптимизация CUDA программ.**

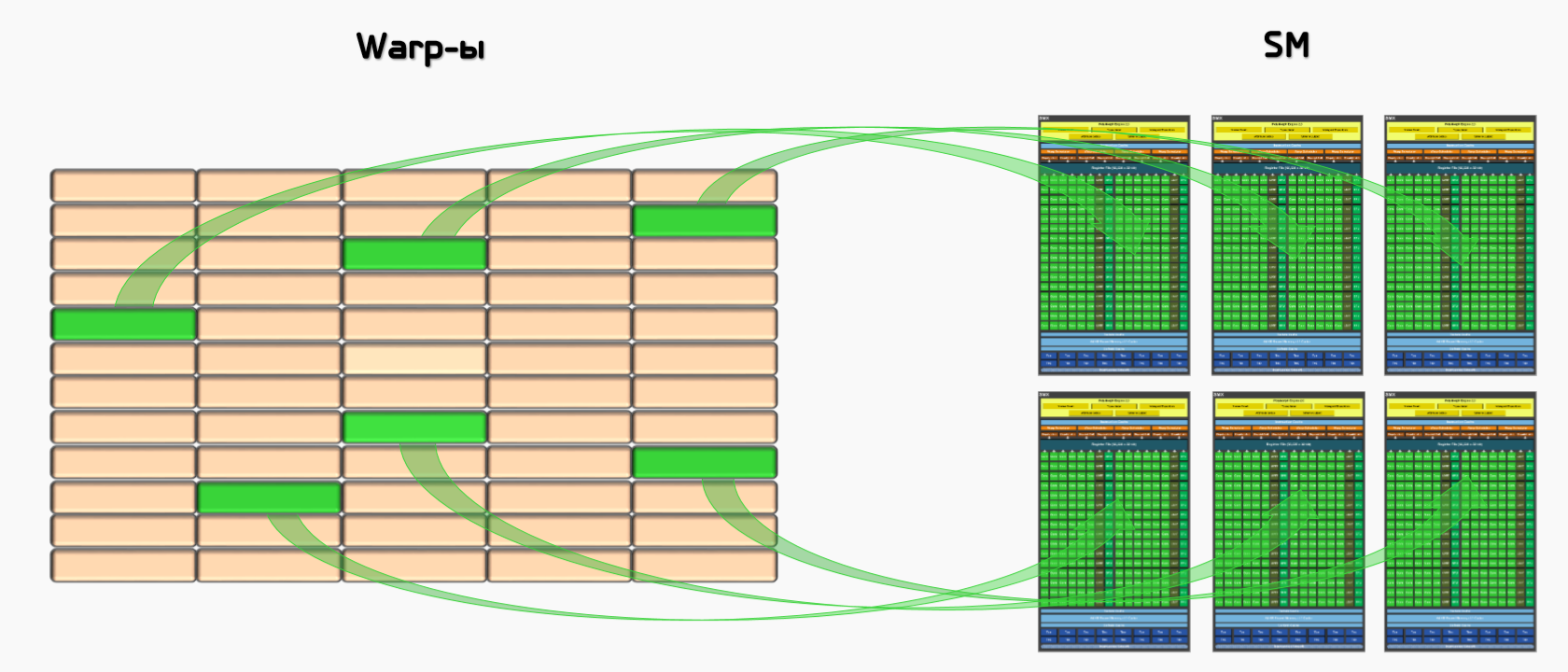
1. Хранение и изменение данных программы, которые необходимо распараллелить в программе на CUDA нужно хранить на GPU(во время выполнения программы на CUDA), поскольку доступ к памяти происходит быстрее(см предыдущий параграф), чем к оперативки.



1. По возможности нужно использовать условных выражений(if, switch и тп) тк происходит ветвление ухудшая скорость выполнения программы.



1. Необходимо правильно распределять данные по warp-ам.



### (Настя)Системы с массовым параллелизмом (MPP).

Massive Parallel Processing (Массово-параллельная архитектура) - "Shared nothing”

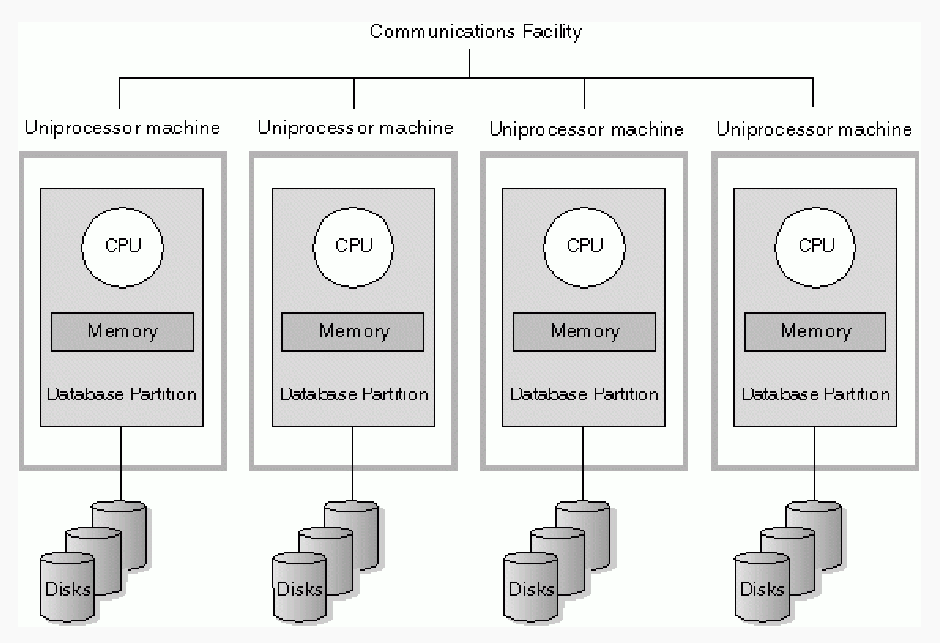
Архитектура вычислительных систем, в которой узлы не разделяют общих ресурсов, в частности процессор каждого узла имеет доступ только к памяти только своего узла и никакого другого. Такая архитектура не требует сложной синхронизации между процессорами и позволяет относительно легко масштабировать систему добавлением новых узлов (горизонтальное масштабирование).

Недостатки:

1. Сложнее обмениваться данными между процессорами.

2. Каждому процессору доступна лишь часть общей памяти системы.

В современных условиях подход, однако, хорошо себя показывает в силу того, что железо становится всё доступнее, а потому порой проще докупить новых серверов.



Массивно-параллельные системы (MPP)

|  |  |
| --- | --- |
| **Архитектура** | Система состоит из однородных *вычислительных узлов*, включающих:   * один или несколько центральных процессоров (обычно RISC), * локальную память (прямой доступ к памяти других узлов невозможен), * коммуникационный процессор или сетевой адаптер * иногда - жесткие диски (как в SP) и/или другие устройства В/В   К системе могут быть добавлены специальные узлы ввода-вывода и управляющие узлы. Узлы связаны через некоторую коммуникационную среду (высокоскоростная сеть, коммутатор и т.п.) |
| **Примеры** | IBM RS/6000 [SP2](https://parallel.ru/computers/computers.html#sp2), Intel PARAGON/ASCI Red, CRAY [T3E](https://parallel.ru/computers/computers.html#crayt3e), Hitachi [SR8000](https://parallel.ru/news/hitachi_sr8000f1.html), транспьютерные системы [Parsytec](https://parallel.ru/computers/vendors.html#parsytec). |
| **Масштабируемость** | Общее число процессоров в реальных системах достигает нескольких тысяч (ASCI Red, Blue Mountain). |
| **Операционная система** | Существуют два основных варианта:   1. Полноценная ОС работает только на управляющей машине (front-end), на каждом узле работает сильно урезанный вариант ОС, обеспечивающие только работу расположенной в нем ветви параллельного приложения. Пример: Cray T3E. 2. На каждом узле работает полноценная UNIX-подобная ОС (вариант, близкий к [кластерному](https://parallel.ru/computers/classes.html#cluster) подходу). Пример: IBM RS/6000 SP + ОС AIX, устанавливаемая отдельно на каждом узле. |
| **Модель программирования** | Программирование в рамках модели **передачи сообщений** ( [MPI](https://parallel.ru/tech/tech_dev/mpi.html), [PVM](https://parallel.ru/tech/tech_dev/ifaces.html#pvm), [BSPlib](https://parallel.ru/tech/tech_dev/ifaces.html#bsplib)) |

**Кластер представляет собой систему из нескольких компьютеров (в большинстве случаев серийно выпускаемых), имеющих общий разделяемый ресурс для хранения совместно обрабатываемых данных (обычно набор дисков или дисковых массивов) и объединенных высокоскоростной магистралью** (рис 1.1).[3]

В кластерной системе некоторое распределенное приложение параллельно на нескольких узлах обрабатывает общий набор данных, как правило, таким образом, чтобы у пользователя возникла иллюзия работы на одной машине.

Если в кластере его узлы разделяют некоторые ресурсы, то параллельные системы другого класса - системы вычислений с массовым параллелизмом (MPP) - строятся из отдельных полностью независимых компьютеров, соединенных только высокоскоростной магистралью или коммуникационными каналами (рис. 4). Это могут быть либо просто несколько серийно выпускаемых UNIX-машин, соединенных с помощью высокопроизводительной сетевой среды, либо специально сконструированная система из отдельных функциональных блоков, объединенных коммутатором.

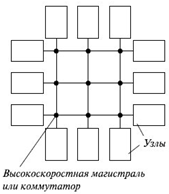


Рис. 1.2 **Структура MPP - системы**

В такой системе адресное пространство состоит из отдельных адресных пространств, которые логически не связаны между собой и доступ к которым не может быть осуществлен аппаратно другим процессором. [4]

При этом для обмена данными используется механизм передачи сообщений между процессорами. Поэтому эти машины часто называют машинами с передачей сообщений. Пользователь может определить логический номер процессора, к которому он подключен, и организовать обмен сообщениями с другими процессорами.

### (Юля) Распределенная обработка данных. Технология Map-Reduce.

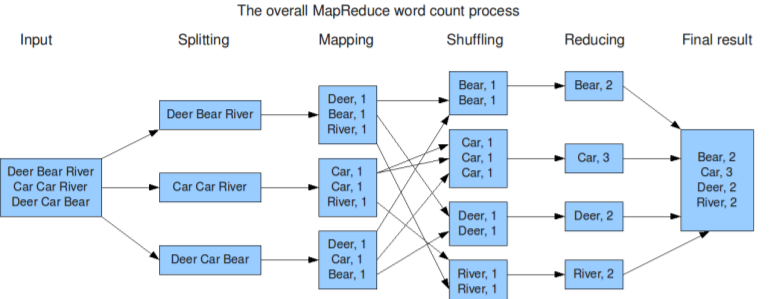
**Шаги**

1. **Map**: входные данные решаемой задачи представляют большой список значений, и на шаге Map происходит его предварительная обработка. Для этого главный узел кластера (master node) получает этот список, делит его на части и передает рабочим узлам (worker node). Каждый рабочий узел применяет функцию Map к локальным данным и записывает результат в формате «ключ-значение» во временное хранилище.
2. **Shuffle**: рабочие узлы перераспределяют данные на основе ключей (созданных функцией Map) так, что все данные принадлежащие одному ключу лежат на одном рабочем узле.
3. **Reduce**: рабочие узлы обрабатывают каждую группу результатов по порядку следования ключей. Главный узел получает промежуточные ответы от рабочих узлов и передаёт их на свободные узлы для выполнения следующего шага. Получившийся после прохождения всех необходимых шагов результат – это решение задачи, которая изначально формулировалась.

**Описание связанных функций высшего порядка**

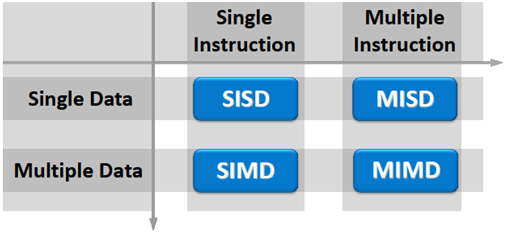
**Функция map** - принимает на вход список и некую функцию, затем применяет данную функцию к каждому элементу списка и возвращает новый список.

**Функция reduce (свёртка)** - преобразует список к единственному атомарному значению при помощи заданной функции, которой на каждой итерации передаются новый элемент списка и промежуточный результат.



### ($)Классификация параллельных вычислительных систем по Флинну.

Cамой ранней и наиболее известной является классификация архитектур вычислительных систем, предложенная в 1966 году М.Флинном. Классификация базируется на понятии потока, под которым понимается последовательность элементов, команд или данных, обрабатываемая процессором. На основе числа потоков команд и потоков данных Флинн выделяет четыре класса архитектур: SISD,MISD,SIMD,MIMD.

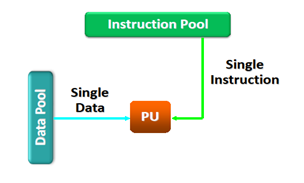


SISD-компьютеры — это обычные, «традиционные» последовательные компьютеры, в которых в каждый момент времени выполняется лишь одна операция над одним элементом данных (числовым или каким-либо другим значением). Большинство персональных ЭВМ до последнего времени, например, попадает именно в эту категорию. Иногда сюда относят и некоторые типы векторных компьютеров, это зависит от того, что понимать под потоком данных.

Типичными представителями SIMD являются векторные архитектуры. К классу MISD ряд исследователей относит конвейерные ЭВМ, однако это не нашло окончательного признания, поэтому можно считать, что реальных систем — представителей данного класса не существует. Класс MIMD включает в себя многопроцессорные системы, где процессоры обрабатывают множественные потоки данных.

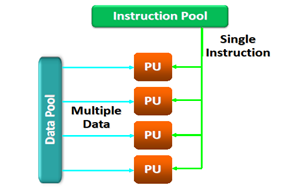
SISD – “Single Instruction / Single Data”

Скалярная обработка – каждая команда на ленте сопровождается данными для этой команды.Наличие конвейера не меняет сути



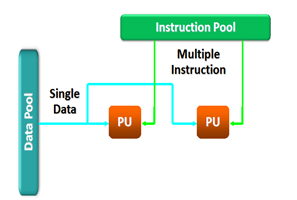
SIMD – “Single Instruction / Multiple Data”

Векторная обработка - каждая команда на ленте сопровождается множеством данных, над которыми эта команда выполняется



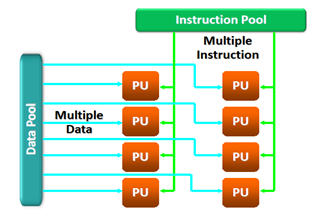
MISD – “Multiple Instruction / Single Data”

Конвейерные системы



MIMD – “Multiple Instruction / Multiple Data”

Компьютеры, способные выполнять одновременно множество команд



### Симметричные мультипроцессорные системы.

(SMP, Symmetric Multiprocessor) – это один компьютер с несколькими равноправными процессорами, но с одной памятью, подсистемой ввода/вывода и одной ОС. Каждый процессор имеет доступ ко всей памяти, может выполнять любую операцию ввода/вывода, прерывать другие процессоры и т.д., но это представление справедливо только на уровне программного обеспечения. На самом же деле в SMP имеется несколько устройств памяти.

**Архитектура:** Система состоит из нескольких однородных процессоров и массива общей памяти (обычно из нескольких независимых блоков). Все процессоры имеют доступ к любой точке памяти с одинаковой скоростью. Процессоры подключены к памяти либо с помощью общей шины (базовые 2-4 процессорные SMP-сервера), либо с помощью crossbar-коммутатора (HP 9000). Аппаратно поддерживается когерентность кэшей.

**Масштабируемость:** Наличие общей памяти сильно упрощает взаимодействие процессоров между собой, однако накладывает сильные ограничения на их число - не более 32 в реальных системах. Для построения масштабируемых систем на базе SMP используются кластерные или NUMA-архитектуры.

**Операционная система:** Вся система работает под управлением единой ОС (обычно UNIX-подобной, но для Intel-платформ поддерживается Windows NT). ОС автоматически (в процессе работы) распределяет процессы/нити по процессорам (scheduling), но иногда возможна и явная привязка.

**Модель программирования:** Программирование в модели общей памяти. (POSIX threads, OpenMP). Для SMP-систем существуют сравнительно эффективные средства автоматического распараллеливания.

Плюсы: относительная простота параллельного программирования

Такая архитектура непригодна для создания масштабных систем.

большое число конфликтов при обращении к общей шине. Остроту этой проблемы удалось частично снять разделением памяти на блоки, подключение к которым с помощью коммутаторов позволило распараллелить обращения от различных процессоров.

сложность увеличения числа процессоров (роста производительности)

виды архитектуры SMP-систем:

* с общей шиной(самая распространённая);
* с коммутатором типа «кроссбар»;
* с многопортовой памятью;
* с централизованным устройством управления.

В архитектуре *с общей шиной* физический интерфейс, логика адресации, арбитража и разделения времени остаются теми же, что и в однопроцессорных системах. Общая шина позволяет легко расширять систему путем подключения большего числа процессоров. Кроме того, поскольку шина является пассивной средой, отказ подключенного устройства не распространяется на другие устройства системы.

Недостаток - невысокая производительность (скорость системы ограничена временем цикла шины). Для компенсации недостатка каждому процессору придается собственная кэш-память (в итоге уменьшается число обращений к шине). Но, появляется новая проблема: когерентность кэш-модулей (обычное решение - применение протокола наблюдения). Все эти факторы существенно ограничивают число процессоров в ВС.

Архитектура *с коммутатором* типа «кроссбар» ориентирована на блочное построение общей памяти и призвана разрешить проблему ограниченной пропускной способности систем с общей шиной.

Коммутатор обеспечивает множественность путей между процессорами и банками памяти, причем топология связей может быть как двухмерной, так и трехмерной. Результатом становится более высокая полоса пропускания, что позволяет строить SMP-системы, содержащие больше процессоров, чем в случае общей шины. Типичное число процессоров в SMP-системах на базе матричного коммутатора составляет 32 или 64. Выигрыш в производительности достигается лишь когда разные процессоры обращаются к разным банкам памяти.

В основе архитектуры *с многопортовой памятью* лежит использование многопортовых запоминающих устройств (ЗУ). Многопортовая организация ЗУ обеспечивает любому процессору (модулю ввода/вывода) непосредственный доступ к банкам основной памяти. Такой подход требует существенного усложнения логики управления ЗУ, хотя и обеспечивает подъем производительности. Другое преимущество многопортовой организации - возможность назначать отдельные банки памяти в качестве локальной памяти конкретного процессора. В результате улучшается защита данных от несанкционированного доступа со стороны других процессоров.

В архитектуре *с централизованным устройством управления* (ЦУУ) устройство управления выполняет следующие функции: трассирует потоки данных между процессорами, памятью, устройствами ввода/вывода; буферизирует запросы; выполняет синхронизацию и арбитраж; отслеживает состояние процессоров и их кэш-памяти. Недостаток архитектуры заключается в сложности ЦУУ, которое может ограничивать производительность.

### (Кирилл М) Проблема когерентности памяти при организации параллельных вычислений.

**Когерентность памяти** (англ. memory coherence) — свойство компьютерных систем, содержащих более одного процессора или ядра, имеющих доступ к одной области памяти, заключающееся в том, что измененная одним ядром/процессором ячейка памяти принимает новое значение для остальных ядер/процессоров.

Общий доступ к данным может быть обеспечен и при физически распределенной памяти (при этом, естественно, длительность доступа уже не будет одинаковой для всех элементов памяти). Такой подход именуется как неоднородный доступ к памяти (non-uniform memory access or NUMA). Среди систем с таким типом памяти выделяют:

* Системы, в которых для представления данных используется только локальная кэш память имеющихся процессоров (cache-only memory architecture or COMA); примерами таких систем являются, например, KSR-1 и DDM;
* Системы, в которых обеспечивается однозначность (когерентность) локальных кэш памяти разных процессоров (cache-coherent NUMA or CC-NUMA); среди систем данного типа SGI Origin2000, Sun HPC 10000, IBM/Sequent NUMA-Q 2000;
* Системы, в которых обеспечивается общий доступ к локальной памяти разных процессоров без поддержки на аппаратном уровне когерентности кэша (non-cache coherent NUMA or NCC-NUMA); к данному типу относится, например, система Cray T3E.

Проблема кэша данных сопряжена с проблемой когерентности — проблемой коррекции общих данных, оказавшихся в момент их обработки размноженными и распределенными между кэшами разных процессоров. Данные, находящиеся в кэшах разных процессоров, могут быть изменены одним процессором, что, в соответствии с алгоритмическим замыслом, должно или не должно стать предметом коррекции кэшей других процессоров.

Особую актуальность эта проблема имеет в технологии SPMD (single program, multiple data), где максимальное внимание уделяется именно тому, чтобы на всех процессорах выполнялась одна и та же программа при совместной обработке общих массивов данных. Примером может служить одновременный запуск СУБД на разных процессорах при организации многоканального доступа к базе данных. При этом неизбежна синхронизация обработки отдельных общих данных или их подмассивов. Для такой синхронизации применяются **принципы data flow**, которые при конкретной реализации могут использовать механизм семафоров, но могут воспользоваться и более "быстрым" механизмом закрытия адресов. Он отмечает те адреса, или непосредственно регистры, по которым запрещено считывание до поступления в них данных. И если механизм семафоров эффективен на уровне программ (закрытый семафор вызывает прерывание и обращение процессора к очереди заданий), то механизм закрытия адресов ориентирован на уровень команд: команда пытается считать значение, пока оно не поступит в регистр с "закрытым" адресом.



**Схема обеспечения когерентности**



**Таблица закрытых адресов**

Источники: <https://www.intuit.ru/studies/courses/80/80/lecture/2455?page=3>

<http://www.hpcc.unn.ru/files/HTML_Version/part1.html>

### (Настя)Модели параллельных вычислений.

Параллельное программирование представляет дополнительные источники сложности - необходимо явно управлять работой тысяч процессоров, координировать миллионы межпроцессорных взаимодействий. Для того решить задачу на параллельном компьютере, необходимо распределить вычисления между процессорами системы, так чтобы каждый процессор был занят решением части задачи. Кроме того, желательно, чтобы как можно меньший объем данных пересылался между процессорами, поскольку коммуникации значительно больше медленные операции, чем вычисления. Часто, возникают конфликты между степенью распараллеливания и объемом коммуникаций, то есть чем между большим числом процессоров распределена задача, тем больший объем данных необходимо пересылать между ними. Среда параллельного программирования должна обеспечивать адекватное управление распределением и коммуникациями данных.

Из-за сложности параллельных компьютеров и их существенного отличия от традиционных однопроцессорных компьютеров нельзя просто воспользоваться традиционными языками программирования и ожидать получения хорошей производительности. Рассмотрим основные модели параллельного программирования, их абстракции адекватные и полезные в параллельном программировании:

***Процесс/канал*** (*Process/Channel*)

В этой модели программы состоят из одного или более процессов, распределенных по процессорам. Процессы выполняются одновременно, их число может измениться в течение времени выполнения программы. Процессы обмениваются данными через каналы, которые представляют собой однонаправленные коммуникационные линии, соединяющие только два процесса. Каналы можно создавать и удалять.

***Обмен сообщениями*** (*Message Passing*)

В этой модели программы, возможно различные, написанные на традиционном последовательном языке исполняются процессорами компьютера. Каждая программа имеют доступ к памяти исполняющего е§ процессора. Программы обмениваются между собой данными, используя подпрограммы приема/передачи данных некоторой коммуникационной системы. Программы, использующие обмен сообщениями, могут выполняться только на MIMD компьютерах.

***Параллелизм данных*** (*Data Parallel*)

В этой модели единственная программа задает распределение данных между всеми процессорами компьютера и операции над ними. Распределяемыми данными обычно являются массивы. Как правило, языки программирования, поддерживающие данную модель, допускают операции над массивами, позволяют использовать в выражениях целые массивы, вырезки из массивов. Распараллеливание операций над массивами, циклов обработки массивов позволяет увеличить производительность программы. Компилятор отвечает за генерацию кода, осуществляющего распределение элементов массивов и вычислений между процессорами. Каждый процессор отвечает за то подмножество элементов массива, которое расположено в его локальной памяти. Программы с параллелизмом данных могут быть оттранслированы и исполнены как на MIMD, так и на SIMD компьютерах.

***Общей памяти*** (*Shared Memory*)

В этой модели все процессы совместно используют общее адресное пространство. Процессы асинхронно обращаются к общей памяти как с запросами на чтение, так и с запросами на запись, что создает проблемы при выборе момента, когда можно будет поместить данные в память, когда можно будет удалить их. Для управления доступом к общей памяти используются стандартные механизмы синхронизации - семафоры и блокировки процессов.

Модель *процесс/канал* характеризуется следующими свойствами:

1. Параллельное вычисление состоит из одного или более одновременно исполняющихся процессов, число которых может изменяться в течение времени выполнения программы.
2. Процесс - это последовательная программа с локальными данными. Процесс имеет входные и выходные порты, которые служит интерфейсом к среде процесса.
3. В дополнение к обычным операциям процесс может выполнять следующие действия: послать сообщение через выходной порт, получить сообщение из входного порта, создать новый процесс и завершить процесс.
4. Посылающаяся операция асинхронная - она завершается сразу, не ожидая того, когда данные будут получены. Получающаяся операция синхронная: она блокирует процесс до момента поступления сообщения.
5. Пары из входного и выходного портов соединяются очередями сообщений, называемыми ***каналами*** (*channels*). Каналы можно создавать и удалять. Ссылки на каналы (порты) можно включать в сообщения, так что связность может измениться динамически.
6. Процессы можно распределять по физическим процессорам произвольным способами, причем используемое отображение (распределение) не воздействует на семантику программы. В частности, множество процессов можно отобразить на одиночный процессор.

Понятие процесса позволяет говорить о местоположении данных: данные, содержащихся в локальной памяти процесса - расположены ``близко ", другие данные ``удалены". Понятие канала обеспечивает механизм для указания того, что для того, чтобы продолжить вычисление одному процессу требуются данные другого процесса (зависимость по данным).

На сегодняшний деньмодель ***обмен сообщениями*** (*message passing*) является наиболее широко используемой моделью параллельного программирования. Программы этой модели, подобно программам модели *процесс/канал*, создают множество процессов, с каждым из которых ассоциированы локальные данные. Каждый процесс идентифицируется уникальным именем. Процессы взаимодействуют, посылая и получая сообщения. В этом отношение модель *обмен сообщениями* является разновидностью модели *процесс/канал* и отличается только механизмом, используемым при передаче данных. Например, вместо отправки сообщения в канал "channel 2" можно послать сообщение процессу "process 3".

Модель *обмен сообщениями* не накладывает ограничений ни на динамическое создание процессов, ни на выполнение нескольких процессов одним процессором, ни на использование разных программ для разных процессов. Просто, формальные описания систем *обмена сообщениями* не рассматривают вопросы, связанные с манипулированием процессами, Однако, при реализации таких систем приходится принимать какое-либо решение в этом отношении. На практике сложилось так, что большинство систем *обмена сообщениями* при запуске параллельной программы создает фиксированное число идентичных процессов и не позволяет создавать и разрушать процессы в течение работы программы.

В таких системах каждый процесс выполняет одну и ту же программу (параметризованную относительно идентификатора либо процесса, либо процессора), но работает с разными данными, поэтому о таких системах говорят, что они реализуют ***SPMD***(*single program multiple* data - одна программа много данных) модель программирования. SPMD модель приемлема и достаточно удобна для широкого диапазона приложений параллельного программирования, но она затрудняет разработку некоторых типов параллельных алгоритмов.

Модель ***параллелизм данных*** также является часто используемой моделью параллельного программирования. Название модели происходит оттого, что она эксплуатирует параллелизм, который заключается в применении одной и той же операции к множеству элементов структур данных. Например, "умножить все элементы массива *M* на значение *x"*, или "снизить цену автомобилей со сроком эксплуатации более 5-ти лет". Программа с *параллелизмом данных* состоит из последовательностей подобных операций. Поскольку операции над каждым элементом данных можно рассматривать как независимые процессы, то степень детализации таких вычислений очень велика, а понятие "локальности" (распределения по процессам) данных отсутствует. Следовательно, компиляторы языков с параллелизмом данных часто требуют, чтобы программист предоставил информацию относительно того, как данные должны быть распределены между процессорами, другими словами, как программа должны быть разбита на процессы. Компилятор транслирует программу с *параллелизмом данных* в SPMD программу, генерируя коммуникационный код автоматически.

В модели программирования с ***общей памяти***, все процессы совместно используют общее адресное пространство, к которому они асинхронно обращаются с запросами на чтение и запись. В таких моделях для управления доступом к общей памяти используются всевозможные механизмы синхронизации типа семафоров и блокировок процессов. Преимущество этой модели с точки зрения программирования состоит в том, что понятие собственности данных (монопольного владения данными) отсутствует, следовательно, не нужно явно задавать обмен данными между производителями и потребителями. Эта модель, с одной стороны, упрощает разработку программы, но, с другой стороны, затрудняет понимание и управление локальностью данных, написание детерминированных программ. В основном эта модель используется при программировании для архитектур с общедоступной памятью.

## Задачи

### Поиск среднего и медианы с использованием технологии map-reduce

Основная идея решения задачи нахождения среднего значения заключается в том, чтобы передавать вычислительным узлам батчи по, к примеру, 1000 значений и запускать процедуру маппинга над каждым батчем. По итогу получится n множеств из пар **{(число) : (сколько раз оно встречается)}**. Полученные множества «склеиваются» в единое и далее, простой итерацией по результирующему множеству, находится значение avg = sum / count, где sum – сумма по данным на текущей итерации, а count – количество «просмотренных» данных.

Формула для sum: sum = sum + Num \* NumCount,

Формула для count: count = count + NumCount,

где Num – текущее рассматриваемое число, NumCount – количество его вхождений в входных данных.

Задача по нахождению медианы решается следующим образом – вычислительным узлам, как и в первой задаче, передаются батчи по 1000 значений и запускается процедура маппинга на каждом узле. По итогу получится множество из пар **{(число) : (сколько раз оно встречается)}**. Далее, итерированием по множеству, полученному путём объединения полученных на каждом из узлов, находится общее количество элементов, через которое можно найти индекс среднего элемента. В завершении, в зависимости от размерности множества (чётное число элементов или нечетное), медиана вычисляется либо, как конкретный элемент на полученной позиции, либо, как полусумма двух соседних значений.

(Размер батча задается произвольно)

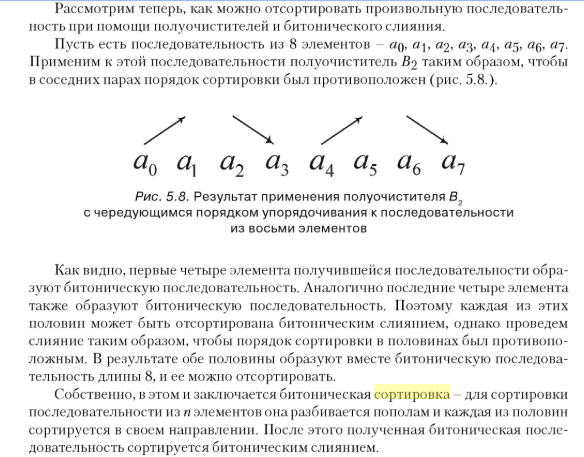
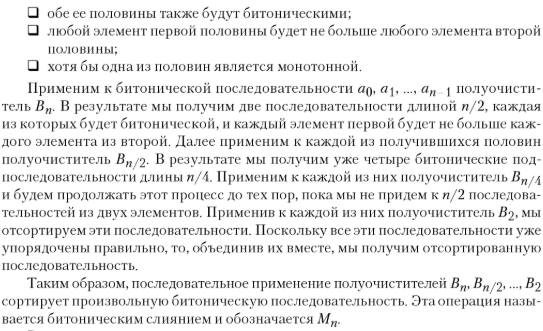
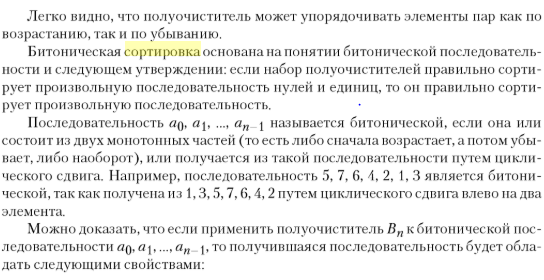
### Реализация сортировки списка с использованием технологии CUDA.

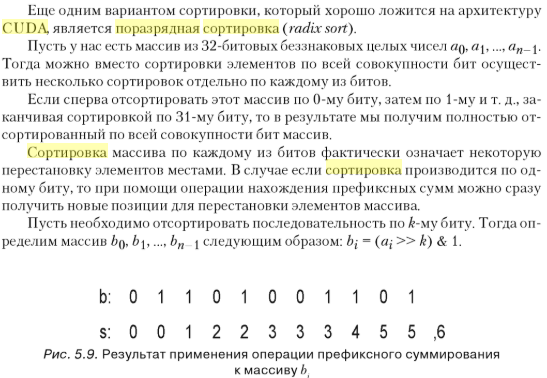
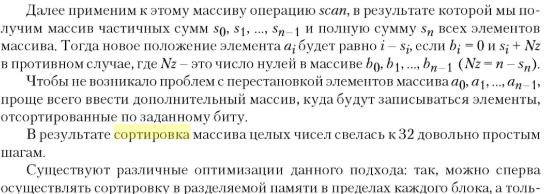
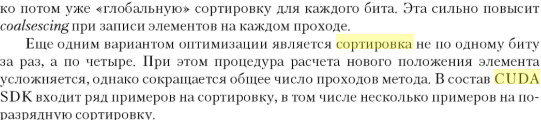
Во многих случаях при рендеринге объектов (не говоря уже о GPGPU) возникает необходимость сортировки объектов (это может быть сортировка частиц в системе частиц, сортировка полупрозрачных граней и т.п.).

Однако традиционные методы сортировки к GPU неприменимы, а передавать данные на CPU для их сортировки крайне нежелательно.

Поэтому возникает необходимость в методах сортировки данных, хорошо ложащихся на архитектуру современных GPU. Существенной особенностью таких алгоритмов является их многопроходность и параллельный характер. Т.е. при помощи фрагментного шейдера и рендеринга в текстуру строится последовательность текстур с данными. На последнем проходе мы получаем на выходе уже полностью отсортированный массив (текстуру).

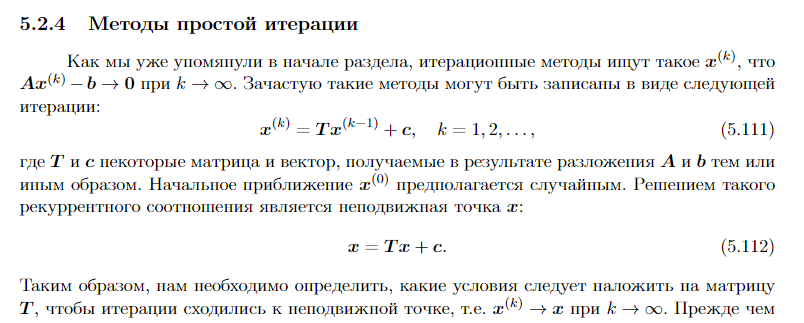


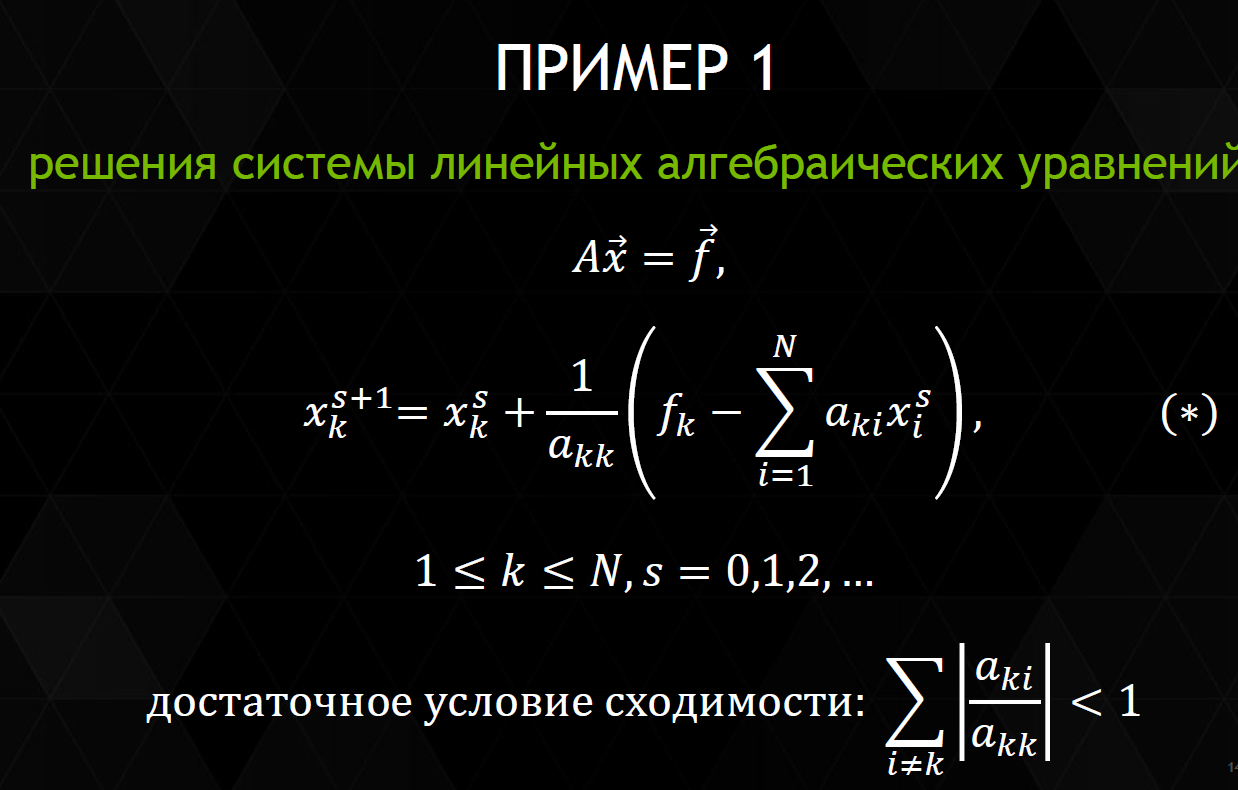


### Реализация решения СЛАУ методом простой итерации на GPU.

Теория по методу простой итерации:



Формулы:

Код:

Функция-ядро вычисления :

\_\_global\_\_ void Solve(float \*dA, float \*dF, float \*dX0, float \*dX1, long long unsigned int N)

{

float aa, sum = 0.;

long long unsigned int t = blockIdx.x \* blockDim.x + threadIdx.x;

for (long long unsigned int j = 0; j < N; j++)

{

sum += dA[j + t \* N] \* dX0[j];

}

aa = dA[t + t \* N];

dX1[t] = dX0[t] + (dF[t] - sum) / aa;

}

float \*dA - матрица A, хранящаяся как одномерный массив;

float \*dF - вектор f;

float \*dX0 - вектор x на предыдущей итерации;

float \*dX1 - вектор x на текущей итерации;

long long unsigned int N - число блоков CUDA.

Функция-ядро вычисление погрешности:

\_\_global\_\_ void Eps(float \*dX0, float \*dX1, float \*delta, long long unsigned int N)

{

long long unsigned int i = blockIdx.x \* blockDim.x + threadIdx.x;

delta[i] = fabs(dX0[i] - dX1[i]);

dX0[i] = dX1[i];

}

Итерационный процесс:

while (eps > EPS)

{

k++;

Solve<<<Blocks, Threads>>>(dA, dF, dX0, dX1, nodes\_num);

Eps<<<Blocks, Threads>>>(dX0, dX1, dDelta, nodes\_num);

cudaMemcpy(hX1, dX1, nodes\_num \* sizeof(float), cudaMemcpyDeviceToHost);

cudaMemcpy(hDelta, dDelta, nodes\_num \* sizeof(float), cudaMemcpyDeviceToHost);

eps = 0.;

for (j = 0; j < nodes\_num; j++)

{

eps += hDelta[j];

}

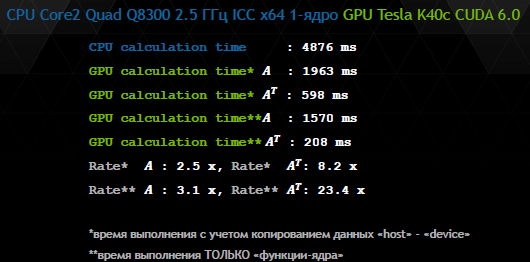
eps = eps / nodes\_num;

// printf("\n Eps[%i]=%e ", k, eps);

// cudaMemcpy(dX0, dX1, nodes\_num \* sizeof(float), cudaMemcpyDeviceToDevice);

}

Смысл в том, что каждая нить решает своё уравнение из СЛАУ(функция *Solve*). Метод выполняется до тех пор, пока погрешность не станет меньше EPS.



Основные вычисления, выполняемые в соответствии с методом, состоят в умножении матрицы A на вектор x.

### Операция перемножения матриц с использованием технологии map-reduce.

Для решения задачи перемножения квадратных матриц с использованием технологии map-reduce требуется решить две подзадачи:

1. Разделить обе входные матрицы на прямоугольники для того, чтобы найти частичные суммы (зеленый прямоугольник на рисунке 4.1).

2. Сложить частичные суммы для получения результата.

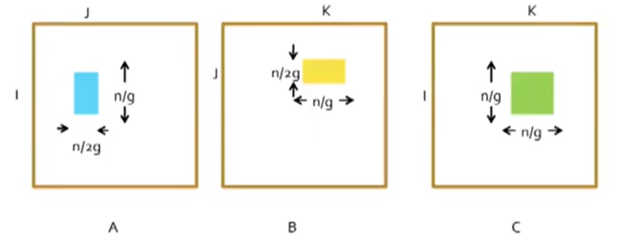


Рис. 4.1. Пример разделения матриц на группы

**Решение первой подзадачи:**

Общая суть подзадачи состоит в том, чтобы разделить строки матрицы А на g групп, каждая из которых состоит из n/g строк (n – размерность матрицы), и столбцы матрицы А на 2g групп, каждая из которых состоит из n/2g столбцов соответственно. Аналогичные действия применяются в отношении матрицы В, только каждая группа включает в себя 2g строк и g столбцов.

**Важно!** Необходимо, чтобы индексы столбцов A и строк B совпадали (как показано на рисунке 4.1 – синий и желтый прямоугольники).

Введем следующие обозначения:

· i – индекс строки в матрице А;

· j – индекс столбца в матрице А и строки в матрице В соответственно;

· k – индекс столбца в матрице В;

· I – группа строк матрицы А, к которой принадлежит i;

· J – группа столбцов в матрице А и строк в матрице В, к которой принадлежит j;

· K – группа столбцов в матрице В, к которой принадлежит k.

Таким образом, получаем, что необходимо два типа мапперов: работающий с матрицей А и работающий с матрицей B.

Первый тип мапперов формирует пару {(I, J, K) : (A, i, j, Aij)}, где К – любая;

второй – {(I, J, K) : (B, j, k, Bjk)}, где I – любая.

Редьюсер на данном этапе выполняет вычисление частичной суммы по формуле:



**Решение второй подзадачи:**

Общее описание – через частичные суммы рассчитывается окончательное значение элемента в матрице C (рисунок 4.1).

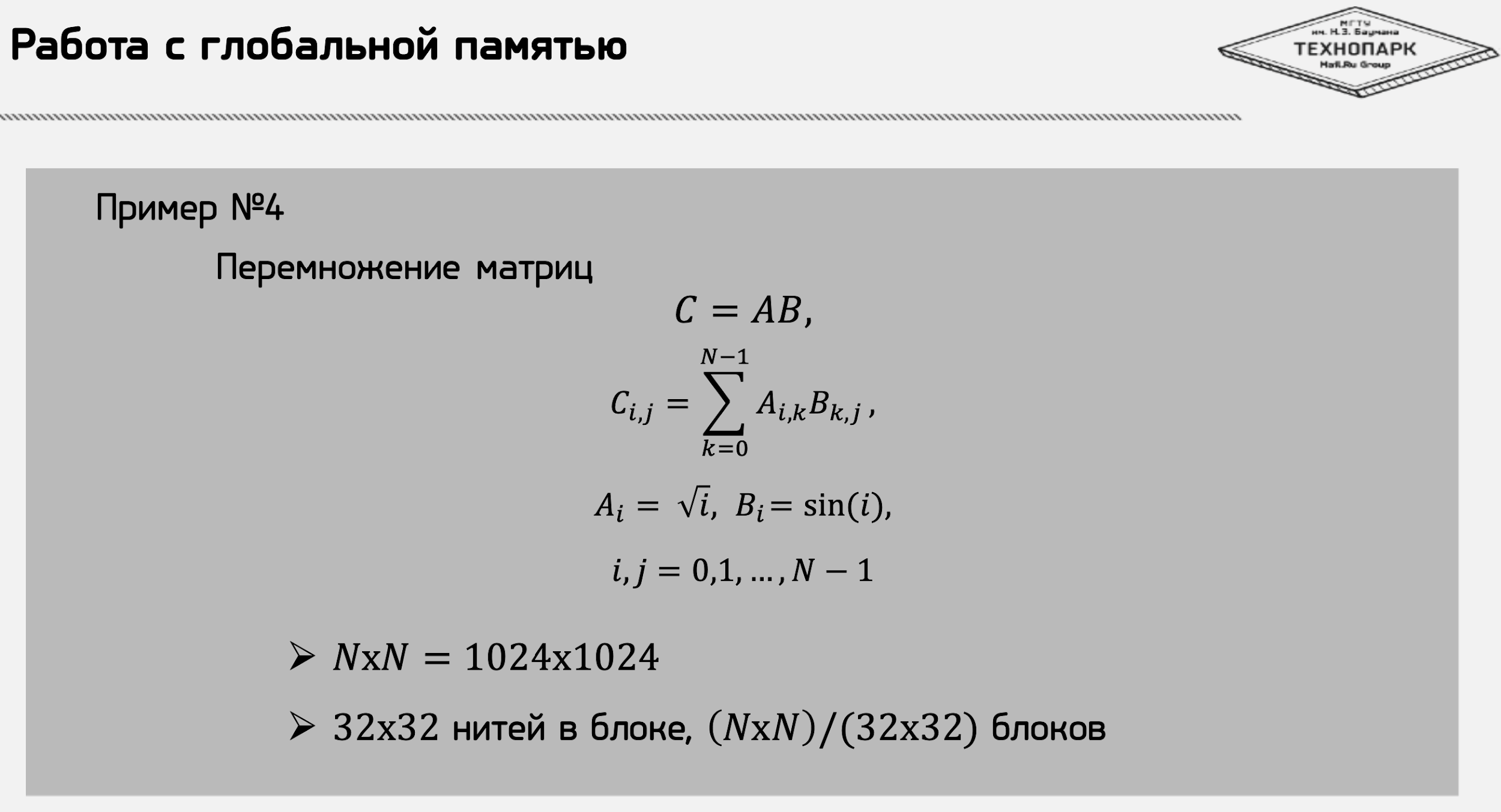
Здесь применяется один тип мапперов, который по частичной сумме, полученной из первой задачи формирует пару: {(i, k) : ()}.

Редьюсер для каждого ключа (i, k) вычисляет , где – элемент матрицы C.

### (Кирилл М) Операция перемножения матриц с использованием технологии CUDA.

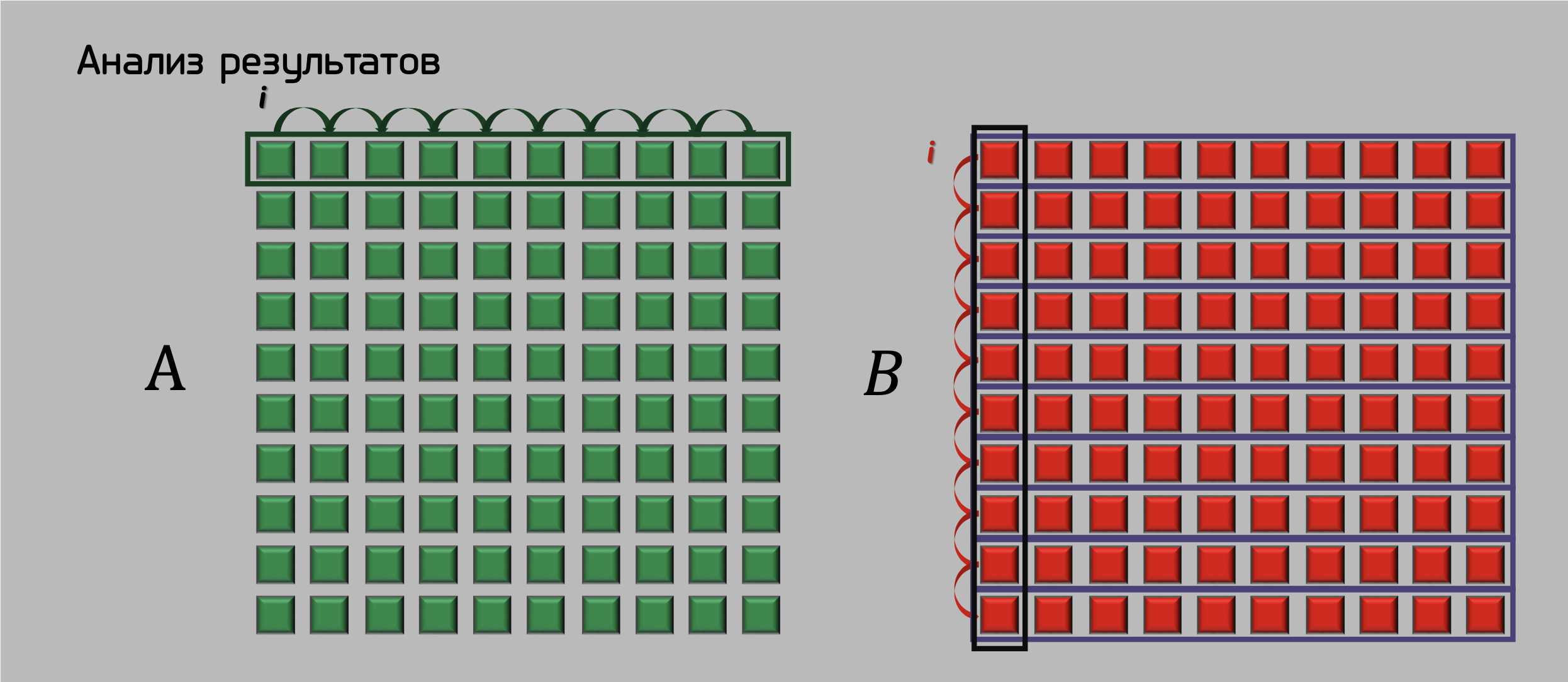
Есть 2 варианта – с использованием глобальной памяти и разделяемой памяти.

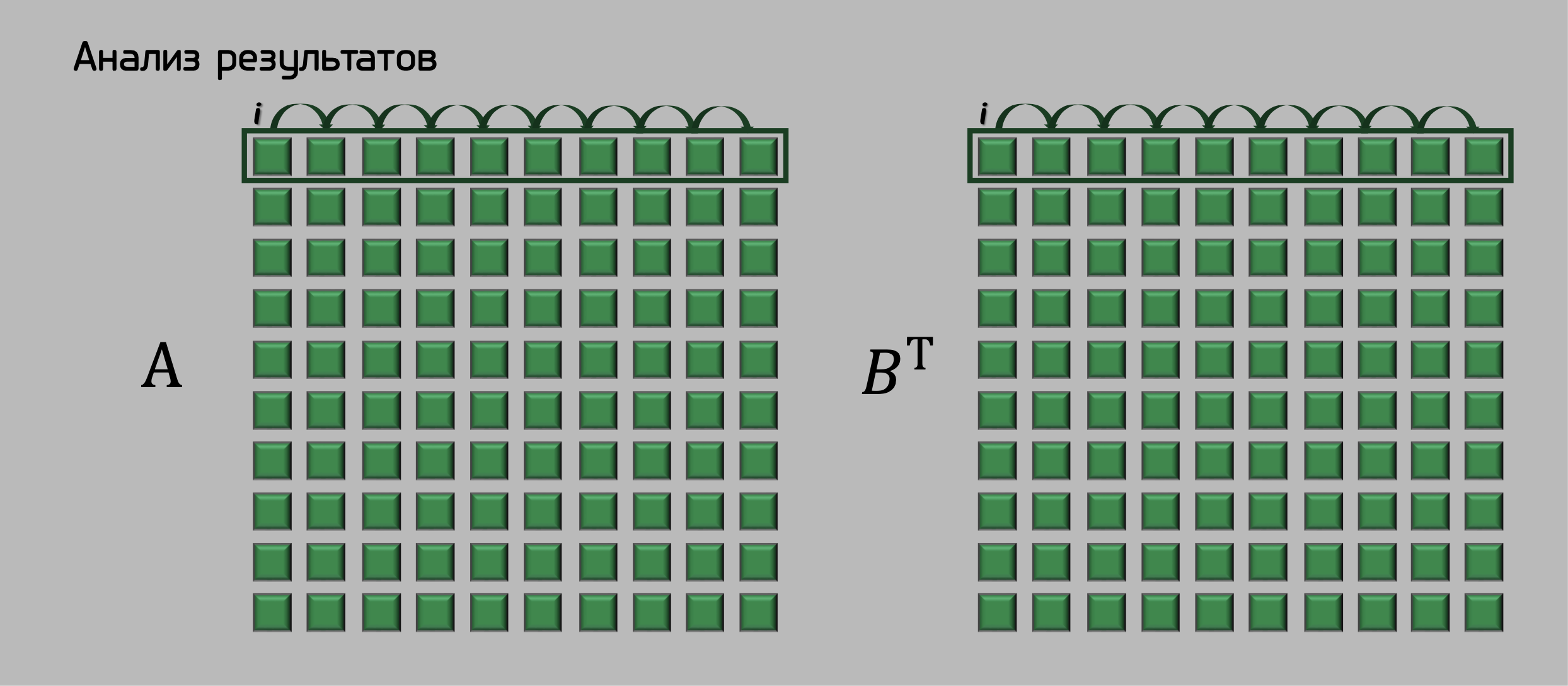
**Пример в Лекции 2**



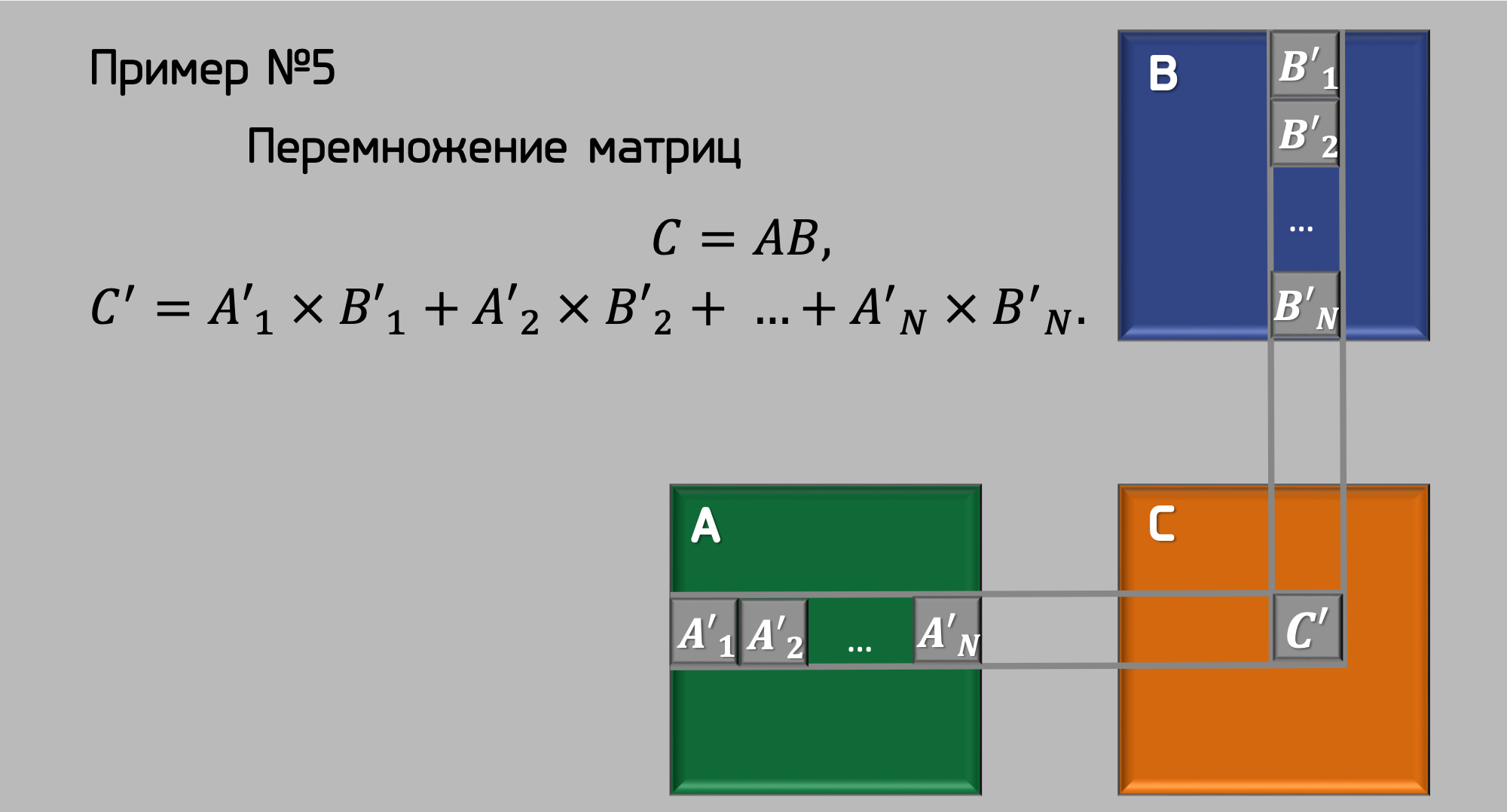
Основные моменты:

* Каждая нить считает произведение соответствующих ее номеру строки и столбца матриц A и B,
* Для увеличения скорости вычисления можно транспонировать матрицу B, тогда элементы будут считываться последовательно (рисунки ниже)





При использовании разделяемой памяти перемножение строки на столбец происходит блочно – строка и столбец бьются на блоки, каждый блок вычисляется несколькими тредами совместно, а значения, используемые в пределах блока находятся в разделяемой памяти. **(??? это бы уточнить)**



### Решение СЛАУ методом Гаусса на GPU. Оценка эффективности.

Реализаций может быть несколько, выбор конкретной зависит от типа матрицы, её размерности.

Проблема в том, что когда мы избавляемся от первого столбца , по алгоритму, переход на следующий столбец будет произведен тогда когда закончится обработка первого столбца. Поэтому метод Гаусса плохо параллелится.

Самое очевидное решение:

1. Создаём нитей в количестве, равному количеству элементов в строке, либо в N раз меньше
2. Создаем блоки в количестве строк, либо в M раз меньше

Передаём коэффициент отношения a(k)(0)/a(0)(0), где k [1, число строк - 1] всем блокам, в каждом блоке производим итерацию прямого метода Гаусса.

Аналогично для следующих итераций.

Появляется много вопросов:

Как этот коэффициент разбрасывать на блоки?

Насколько такой разброс будет тормозить?

С каждой итерацией в прямом ходе, в строках появляются нолики. Что делать потокам у которых нолики? Молотить эти нули? Или простаивать?

Как синхронизировать момент перехода к следующей итерации?

По поводу обратного хода Гаусса.

Здесь тоже встаёт проблема передачи в другие потоки посчитанных значений.

### Поиск обратной матрицы на GPU.

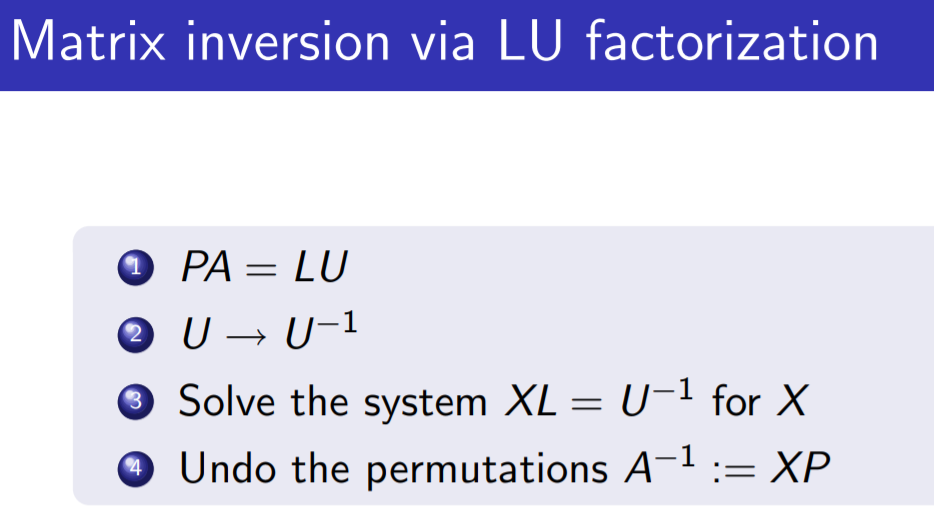


Рисунок 7.1

**Использование библиотеки cuBLAS**

[**https://docs.nvidia.com/cuda/cublas/index.html#cublas-lt-t-gt-getribatched**](https://docs.nvidia.com/cuda/cublas/index.html#cublas-lt-t-gt-getribatched)

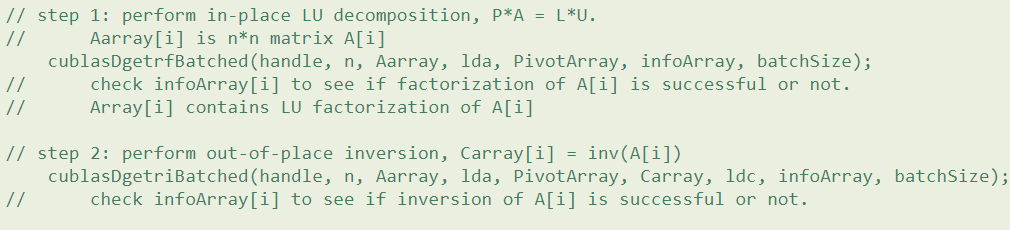
Aarray and Carray are arrays of pointers to matrices stored in column-major format with dimensions n\*n and leading dimension lda and ldc respectively.

This function performs the inversion of matrices A[i] for i = 0, ..., batchSize-1.

Prior to calling cublas<t>getriBatched, the matrix A[i] must be factorized first using the routine cublas<t>getrfBatched. After the call of cublas<t>getrfBatched, the matrix pointing by Aarray[i] will contain the LU factors of the matrix A[i] and the vector pointing by (PivotArray+i) will contain the pivoting sequence.

Following the LU factorization, cublas<t>getriBatched uses forward and backward triangular solvers to complete inversion of matrices A[i] for i = 0, ..., batchSize-1. The inversion is out-of-place, so memory space of Carray[i] cannot overlap memory space of Array[i].

Typically all parameters in cublas<t>getrfBatched would be passed into cublas<t>getriBatched. For example,

getrf - Computes the LU factorization of a general m-by-n matrix.

getri - inverse matrix

**1-й шаг** LU разложение (шаг 1 на рис. 7.1) каждой из матриц в массиве указателей на матрицы (Aarray). Функция cublasDgetrfBatched модифицирует исходную матрицу.

**2-й шаг** Вычисление обратных матриц для каждой из массива указателей Aarray. (шаги 2 - 4 на рис. 7.1). Функция cublasDgetriBatched не изменяет Aarray, а записывает результат в Carray(массив обратных матриц).

**Использование библиотеки Magma**

<https://developer.nvidia.com/sites/default/files/akamai/cuda/files/Misc/mygpu.pdf>

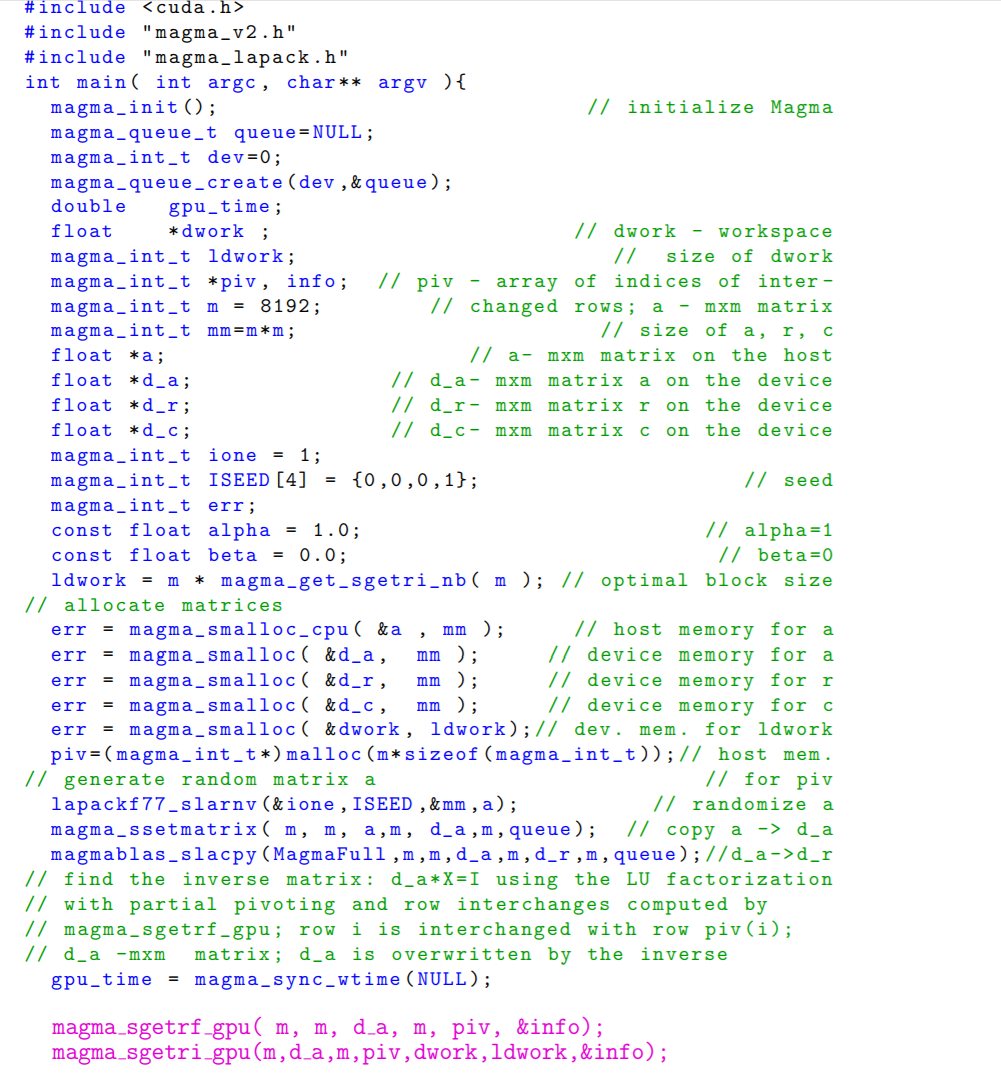
Алгоритм аналогичный. Изменяются только названия функций.

Часть кода:

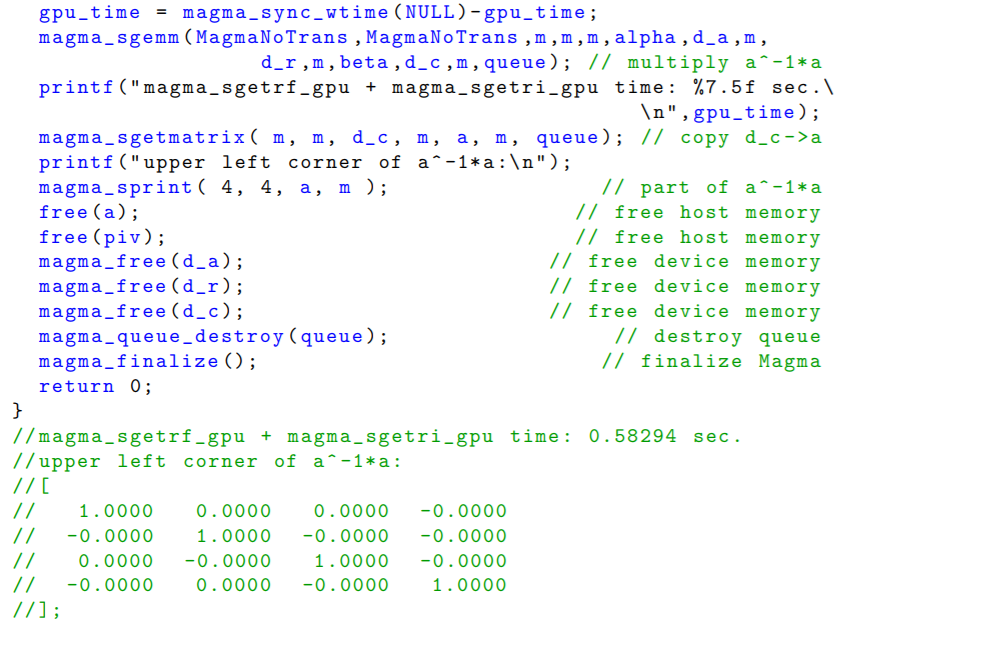
magma\_sgetrf\_gpu( m, m, d a, m, piv, &info);

magma\_sgetri\_gpu(m,d a,m,piv,dwork,ldwork,&info);

Полный код:



to be continued...



### Алгоритм поиска пересечения двух списков с использованием технологии map-reduce.

### Определение моды и медианы с использованием технологии map-reduce.

### Определение 25% и 75% квантилей с использованием технологии map-reduce.

### Параллельное вычисление алгоритма обратного распространения ошибки на GPU при обучении ИНН.

### Особенности реализации матрично-векторного умножения при различных видах аппаратного параллелизма.